



# CS472 WAP

## Execution Context & Closures

# Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

**Wholeness:** A fundamental aspect of function-oriented programming in JavaScript is the use of closures to store state information associated with a function when the function is passed to other objects.

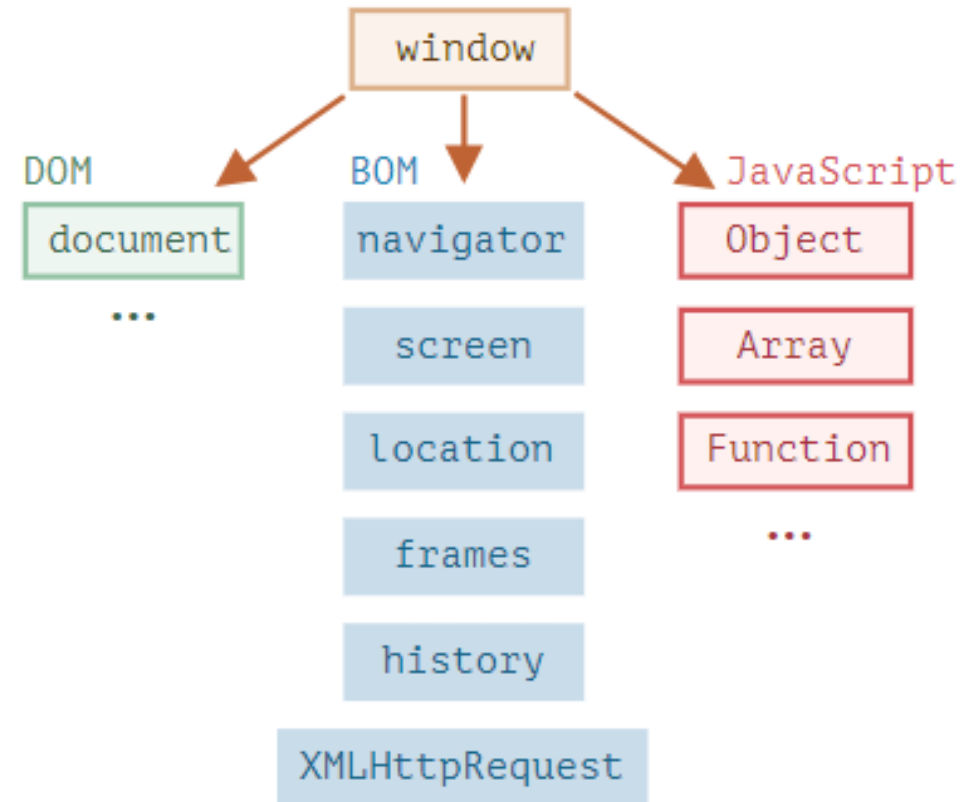
**Science of Consciousness:** Closures provide a protective wrapper for state information associated with a function. An analogy in consciousness is the supportive wrapper that transcendental consciousness provides to our own consciousness. At this level of consciousness, we are connected to the home of all the laws of nature.

# Browser environment

- The JavaScript language was initially created for web browsers. Since then, it has evolved into a language with many uses and platforms.
- when JavaScript runs in a web browser:

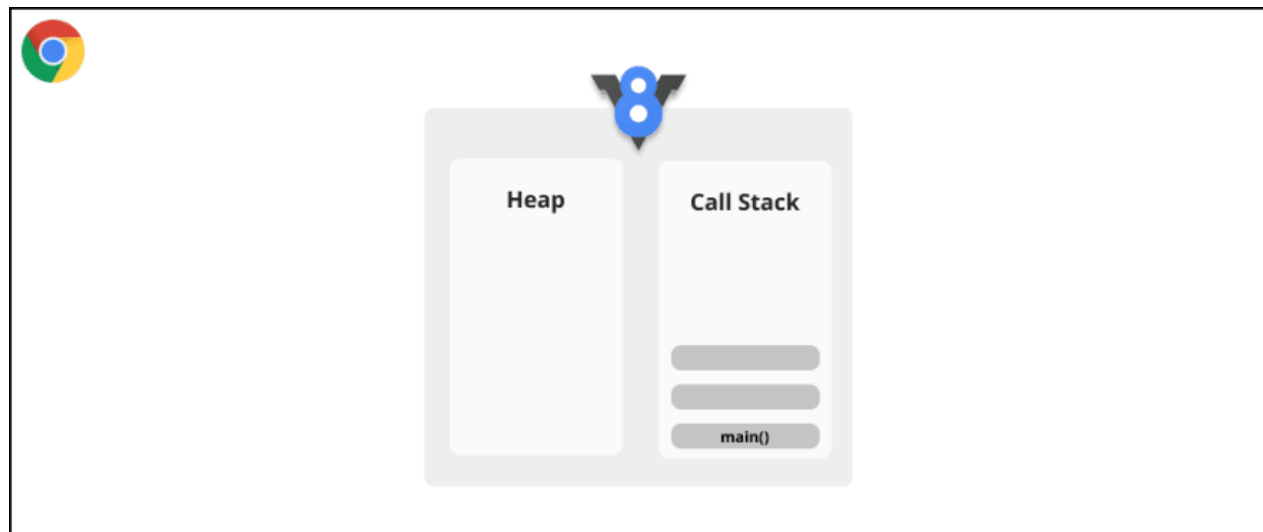
The **Document Object Model**, or **DOM** for short, represents all page content as **objects** that can be modified.

The `document` object is the main “entry point” to the page. We can change or create anything on the page using it.

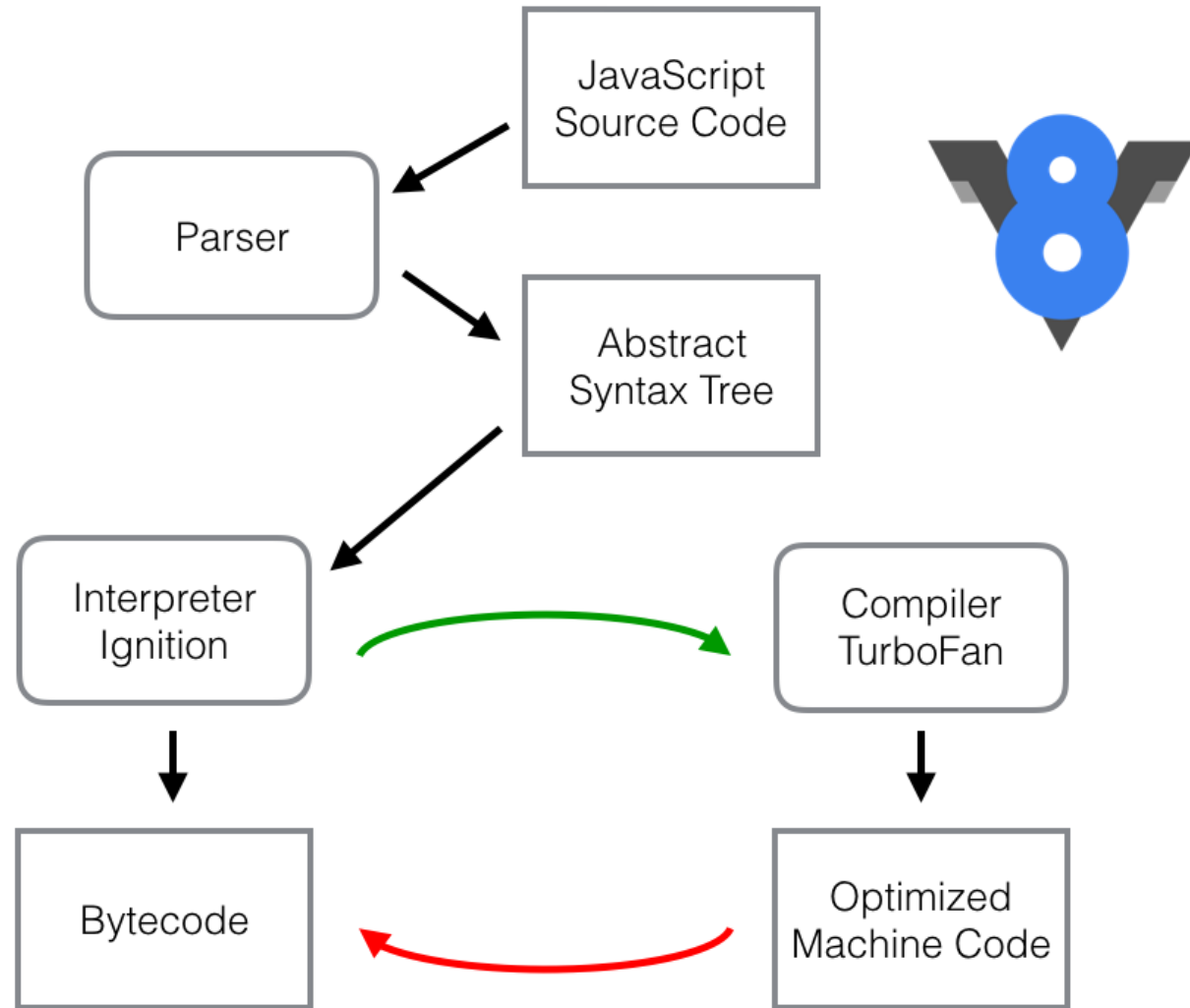


# JavaScript Engine

- The JavaScript engine is a program that executes the JavaScript code.
- A popular example of a JavaScript engine is **Google's V8** engine.
- The V8 engine is an open-source and written in C++.
- Two main components:
  - **Heap** is an unstructured memory that is used for memory allocation of the objects.
  - **Call Stack** is a LIFO data structure that is used for function calls that record where we are in the program.



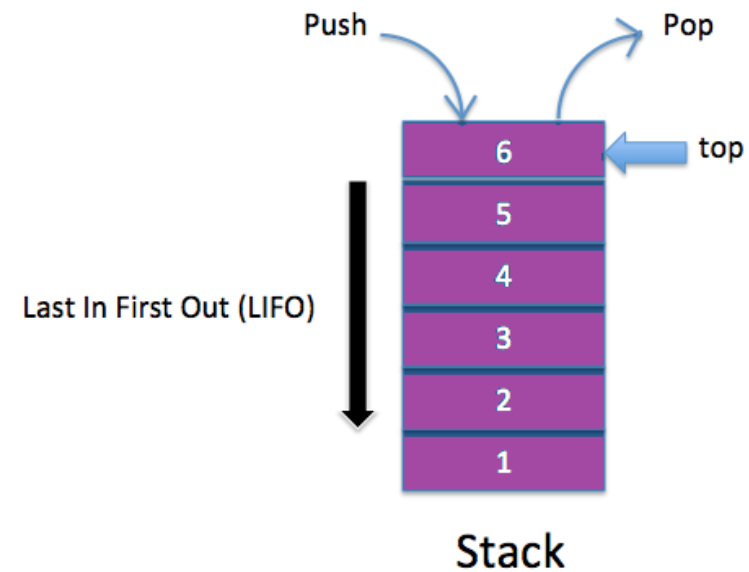
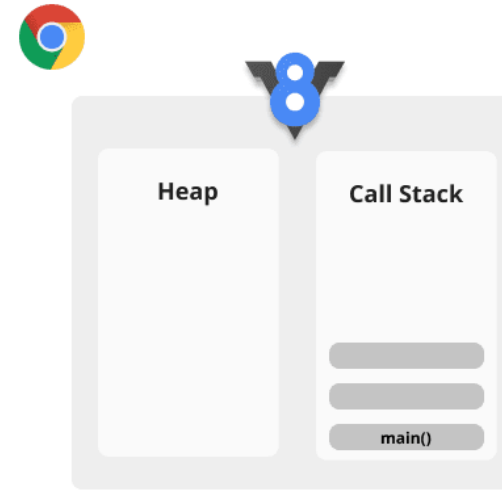
# V8



# Heap and Stack

**The Heap** is a memory space where all objects live.

**The Stack** is LIFO data structure, which contains execution contexts.

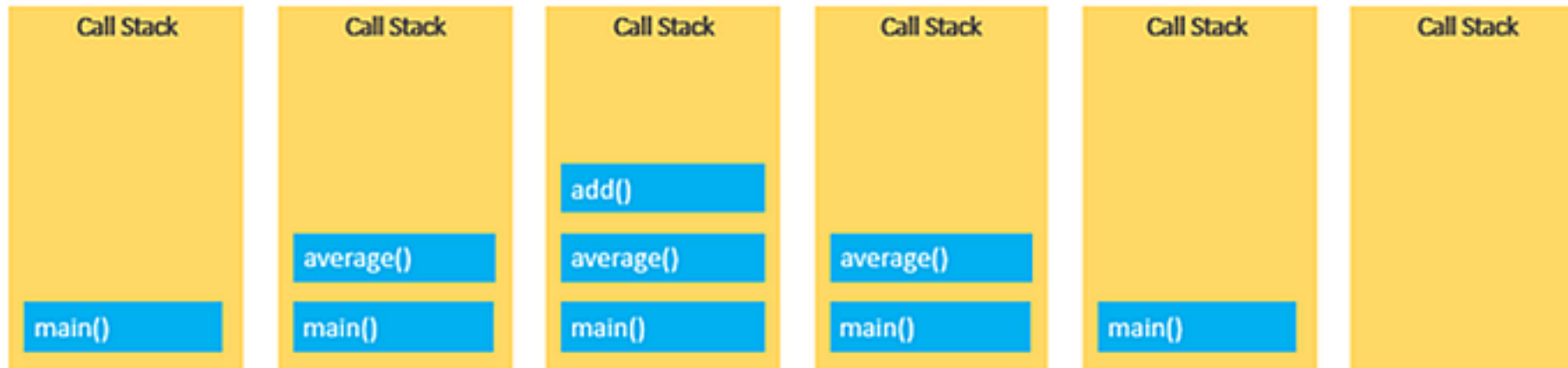


# The Execution Stack

```
const add = (a, b) => a + b;  
const average = () => add (3, 5) / 2;
```

```
const result = average();
```

```
console.log(result);
```



Each blue box (method) has its own execution context.



# Lexical Environment and Execution Context

The lexical environment is where the code is sitting physically. Your code is going to be executed based on where it's lexically located.

Every lexical environment will have its own execution context (wrapper) in which your code will be running.

**Note:** **const** and **let** variables are scoped by the nearest enclosing block

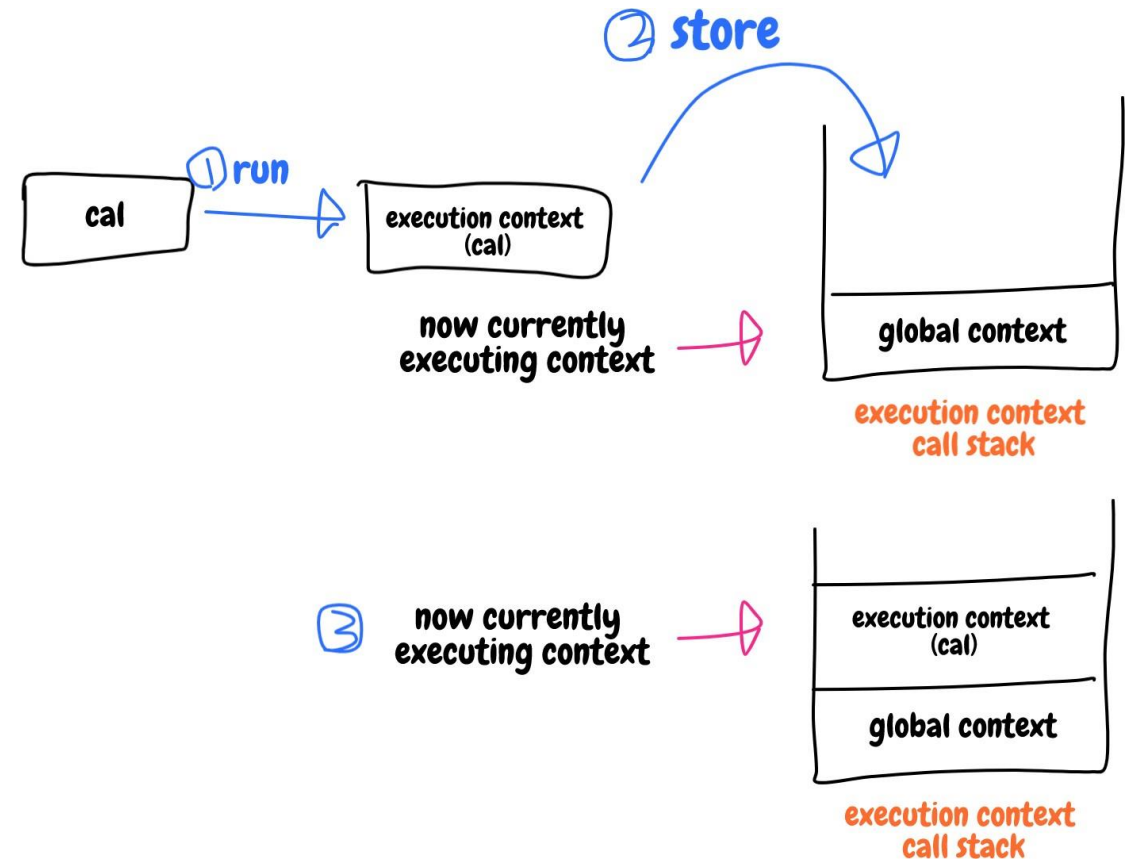
# What is Execution Context?

- **Execution context (EC)** is created when the JavaScript code is executed.
- All ECs are pushed to Execution Context Stack (**call stack**)
- Two types of EC:
  - **Global execution context (GEC):**
    - Default, loads first when run JS in browser
    - Only 1 GEC, JS is single threaded
    - Stores global objects: `window`, `document`, `history`, etc.
  - **Functional execution context (FEC):**
    - Created whenever any function is called
    - Each function has its own execution context
    - No `window` or other global objects
    - `arguments` object

# Draw Execution Context Stack

```
function cal(type, a, b) {  
  if (type === 'add') {  
    return a + b;  
  } else if (type === 'subtract') {  
    return a - b;  
  } else if (type === 'multiply') {  
    return a * b;  
  } else {  
    return a / b;  
  }  
}
```

```
let four = 4;  
let seven = 7;  
cal('add', four, seven);
```



# Two Stages Creating EC by JS Engine

## 1. Creation Phase

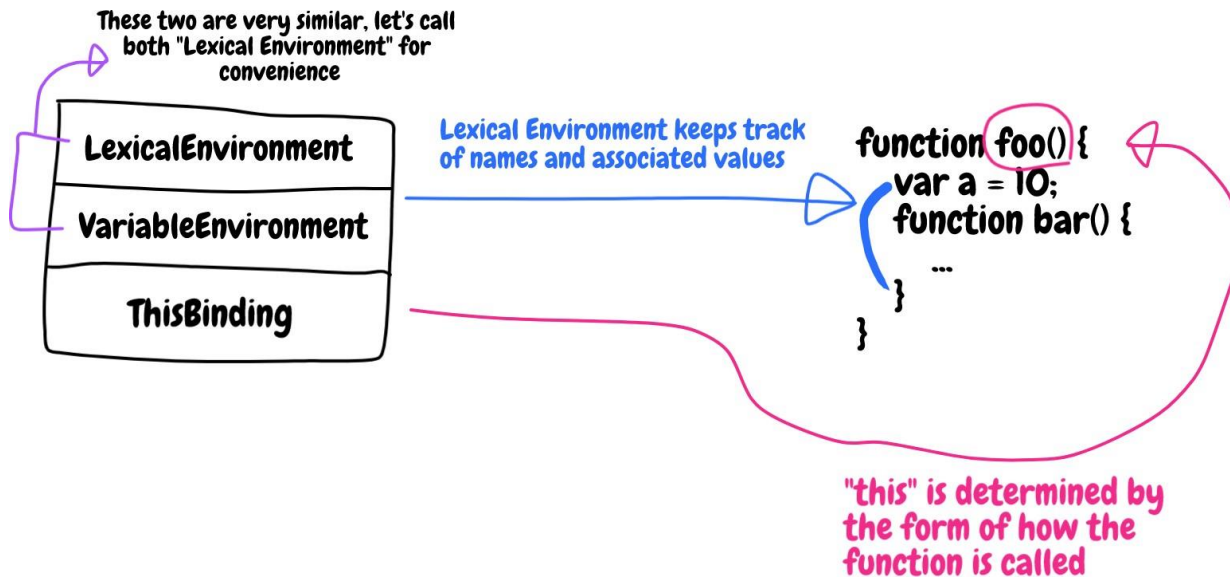
- A function is called but its execution has not started
- JS engine compile the code, doesn't execute any code
  - Records variable and functions declarations – only variable declared with `var`, not `let` and `const`

## 2. Execution Phase

- Scans through the function
- Update the variable object with the values of the variables
- Execute variables declared with `let` and `const`

# Creation Phase: Lexical Environment (LE)

- A execution context is divided into three different areas
- **LE** is to keep track of variables, function names and associated values
  - ThisBinding determines how the function is called, will explain in later lecture.



```
function foo() {  
    var a = 10;  
    function bar() {}  
}  
foo();
```

// When foo is called, a new execution environment  
// might look like this below

```
execution_environment: {  
    LexicalEnvironment: {  
        a: undefined,  
        bar: function() {}  
    },  
    ThisBinding: ...  
}
```

# Creation Phase:

## What's inside Lexical Environment?

- EnvironmentRecord
  - records declaration of variables, functions.
- Outer Lexical Environment – Scope
  - Links to parent LexicalEnvironment
  - Used when can't find a property in the current LexicalEnvironment
  - Global LexicalEnvironment doesn't have outer, `null`

# Creation Phase:

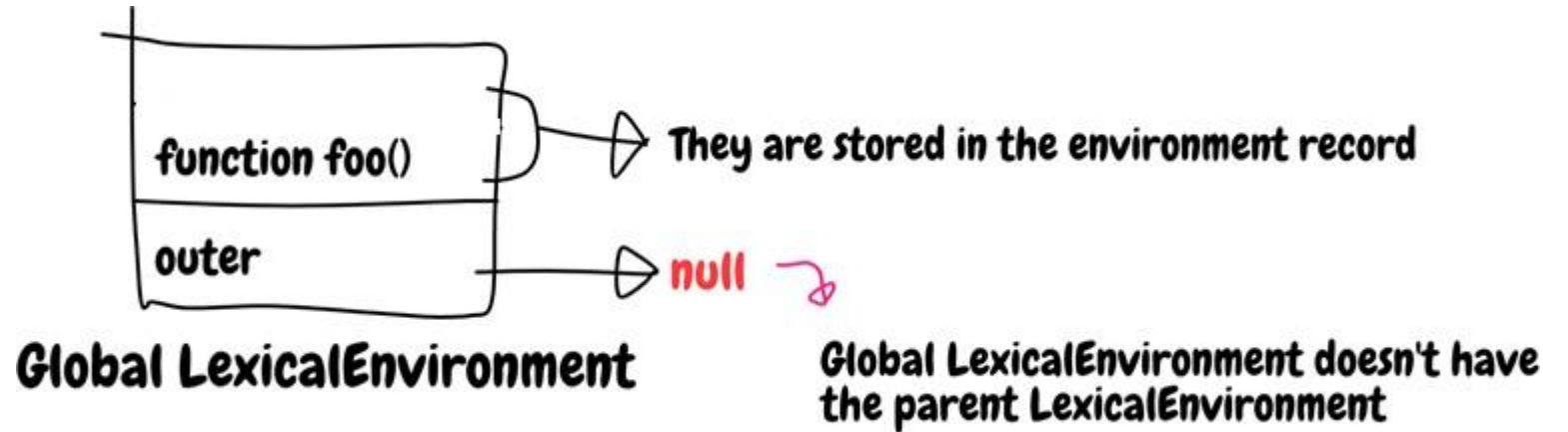
## Example: The Global Lexical Environment

```
let x = 1;

function foo() {
  let y = 2;

  function bar() {
    let z = 3;

    function baz() {
      console.log(z);
      console.log(y);
      console.log(x);
      console.log(w);
    }
    baz();
  }
  bar();
}
foo();
```

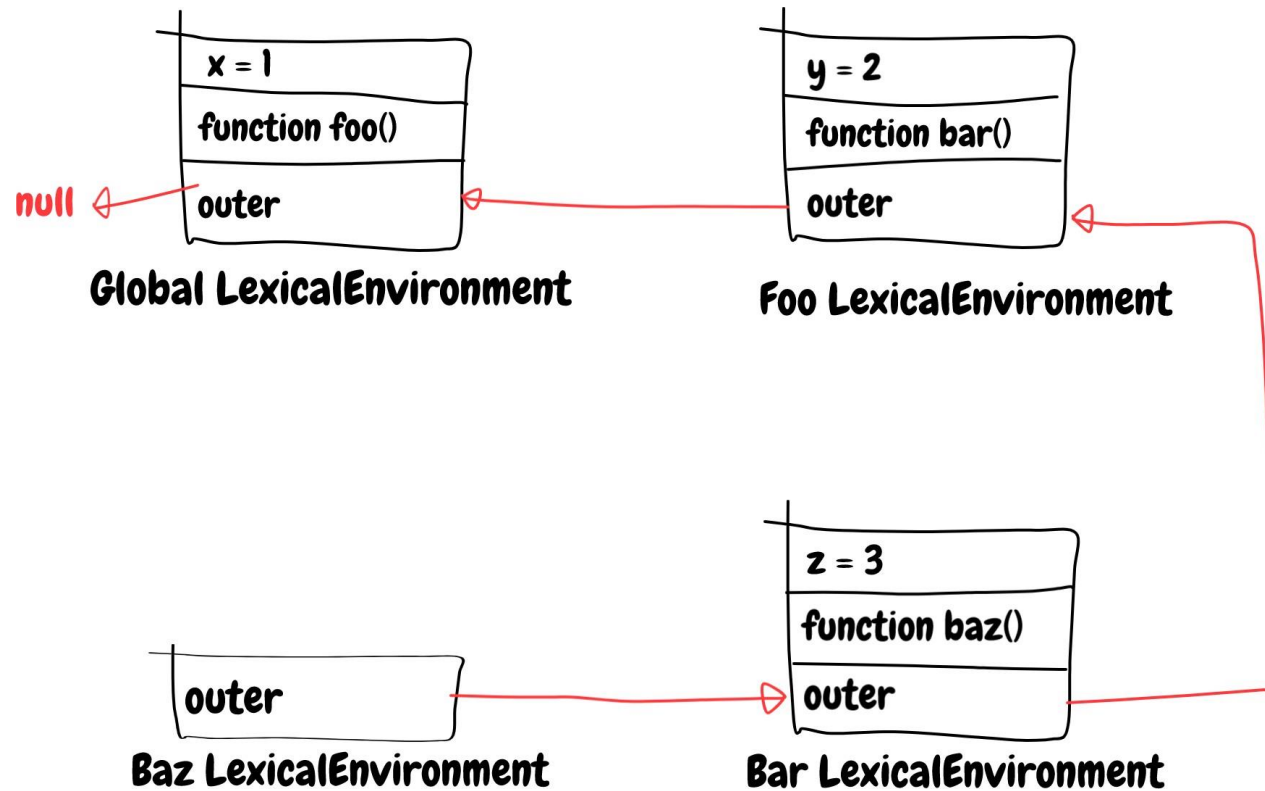


# Execution Phase: Scope Chain

- A list of all the variable objects of functions inside which the current function exists
- The entire relationship amongst LexicalEnvironments

```
let x = 1;
```

```
function foo() {  
  let y = 2;  
  
  function bar() {  
    let z = 3;  
  
    function baz() {  
      console.log(z);  
      console.log(y);  
      console.log(x);  
      console.log(w);  
    }  
    baz();  
  }  
  bar();  
}  
foo();
```





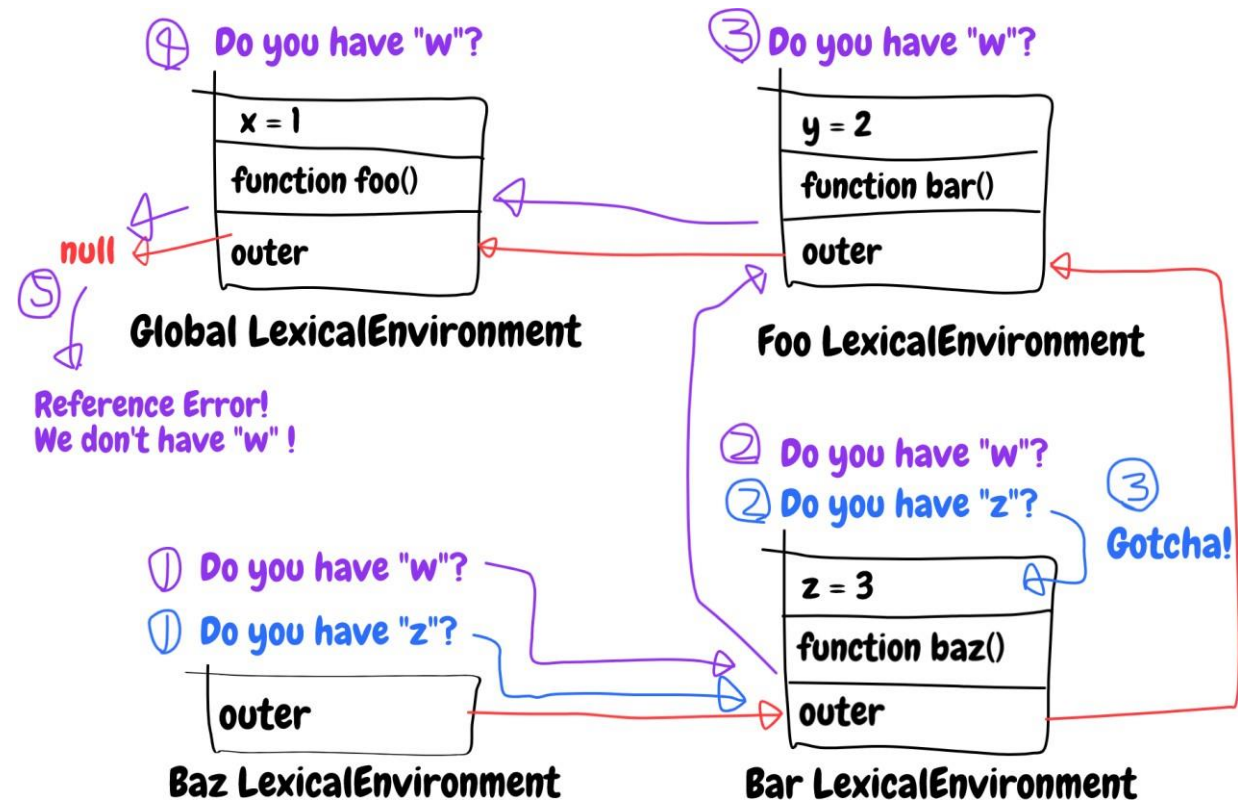
# Execution Phase:

## The workflow when `z` and `w` are looked for

- When `baz()` is called, it looks for `z`, `y`, `x`, and `w`.

```
let x = 1;
```

```
function foo() {  
  let y = 2;  
  
  function bar() {  
    let z = 3;  
  
    function baz() {  
      console.log(z);  
      console.log(y);  
      console.log(x);  
      console.log(w);  
    }  
    baz();  
  }  
  bar();  
}
```



# Nested functions

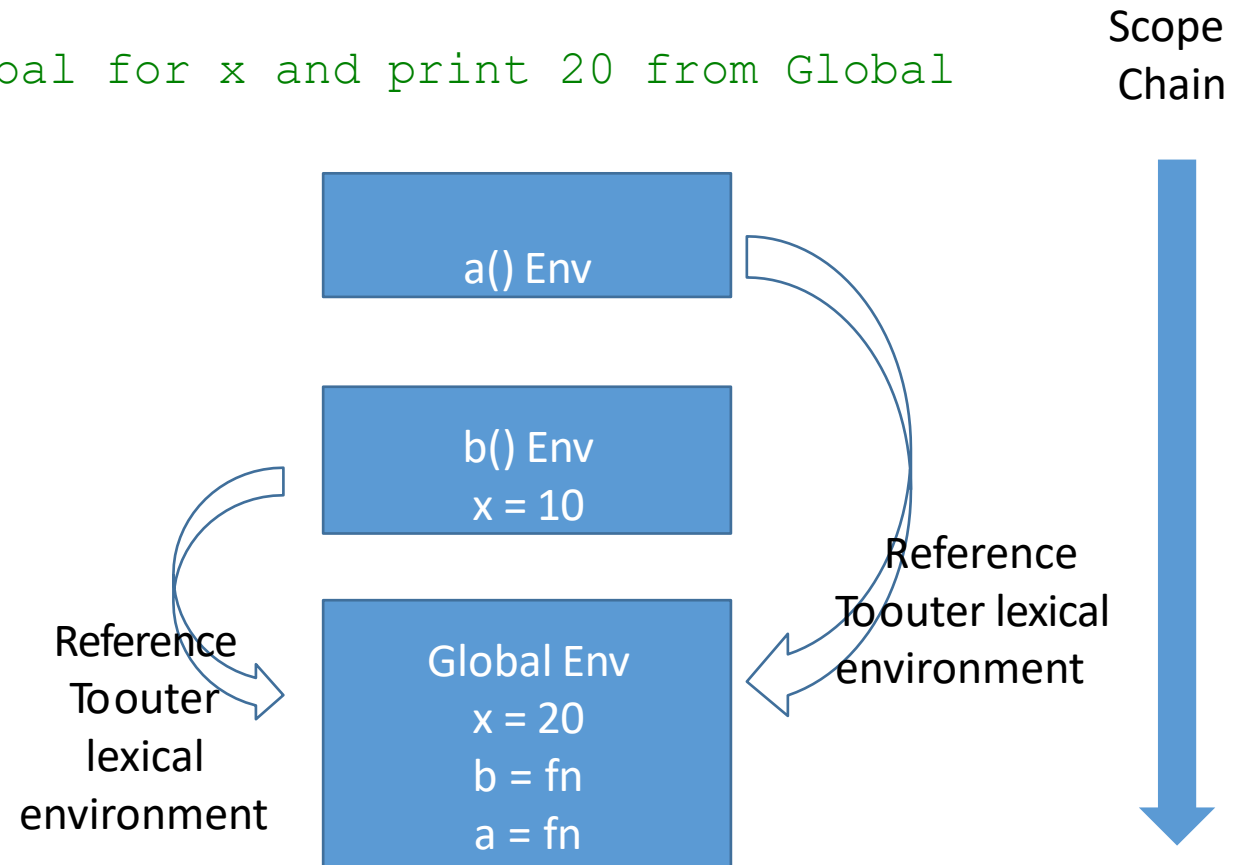
- A function is called “nested” (inner) when it is created inside another function
- Nested functions are quite common in JavaScript.
- What’s much more interesting, a nested function can be returned:
  - as a property of a new object
    - if the outer function creates an object with methods
  - as a result by itself.
    - can then be used somewhere else.
    - No matter where, it still has access to the same outer variables

# Environment exercise:

## Scope Example1



```
function a() {  
  console.log(x); // consult Global for x and print 20 from Global  
}  
function b() {  
  const x = 10;  
  a(); // consult Global for a  
}  
const x = 20;  
b();
```

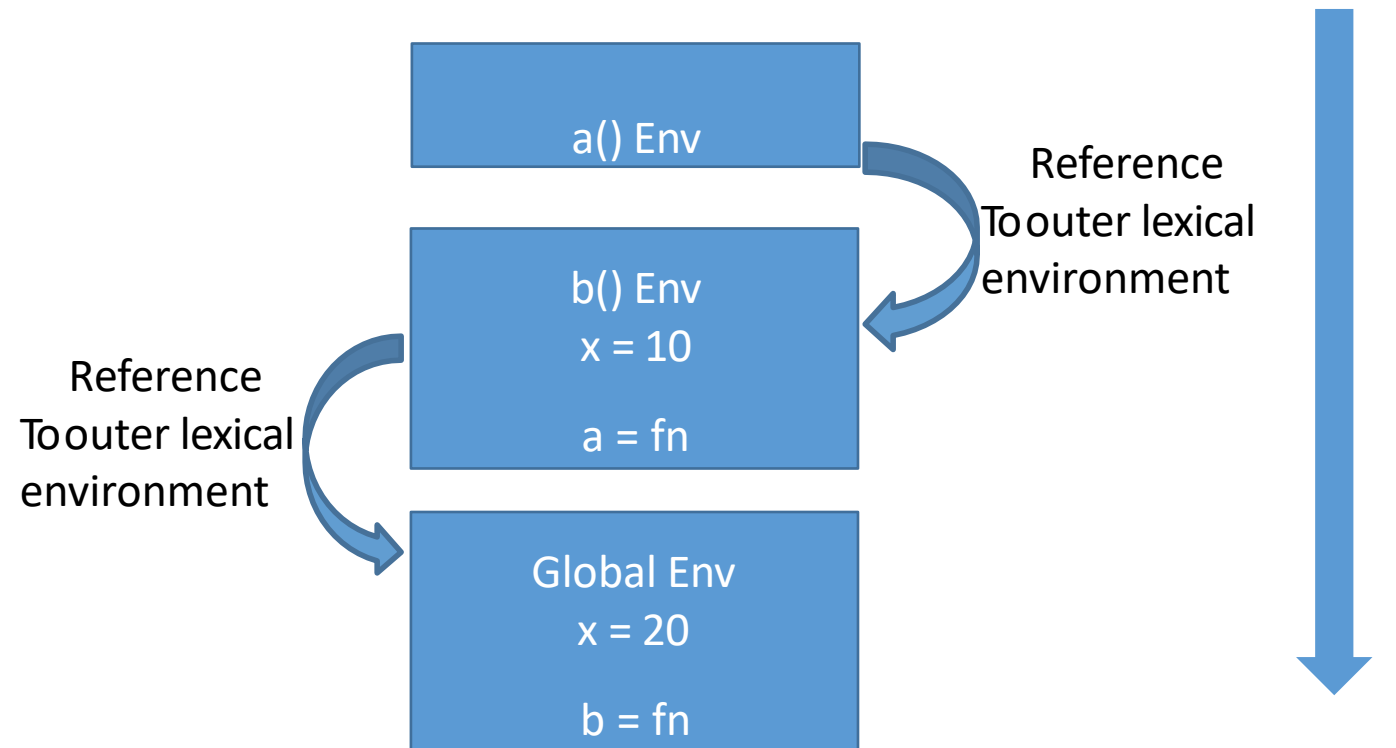


# Environment exercise:

## Scope Example 2: Inner function



```
function b() {  
  function a() {  
    console.log(x);  
  }  
  const x = 10;  
  a();  
}  
const x = 20;  
b();
```

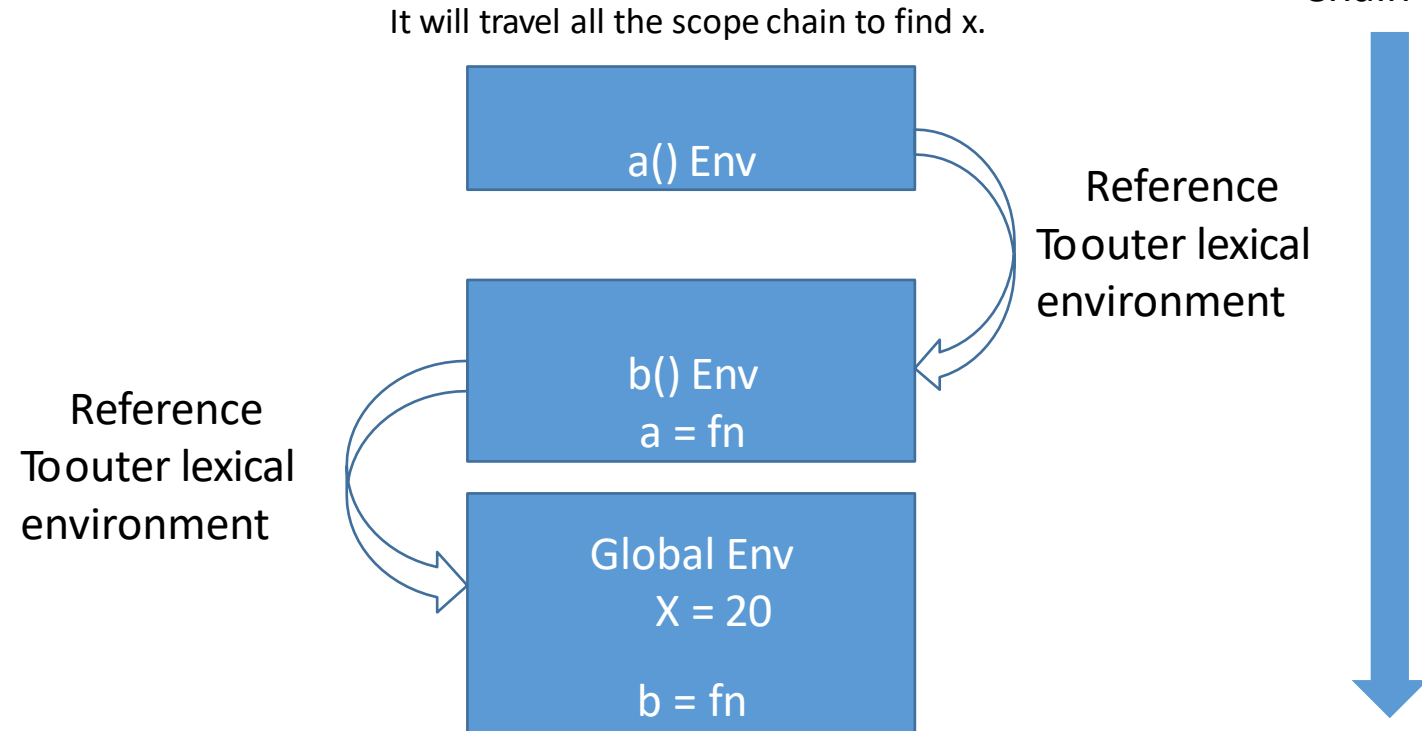


# Environment exercise:

## Scope Example 3



```
function b() {  
  function a() {  
    console.log(x);  
  }  
  a();  
}  
const x = 20;  
b();
```



# Closures

## Closure

A first-class function that binds to free variables that are defined in its execution context.

## Free variable

A variable referred to by a function that is not one of its parameters or local variables.

A closure occurs when a(n inner) function is defined and attaches itself to the free variables from the surrounding environment to "close" up those stray references.

# Closure Example

```
const x = 1;

function f() {
  let y = 2;
  const sum = function() {
    const z = 3;
    console.log(x + y + z);
  }; // inner function closes over free variables x, y
  y = 10;
  return sum;
}

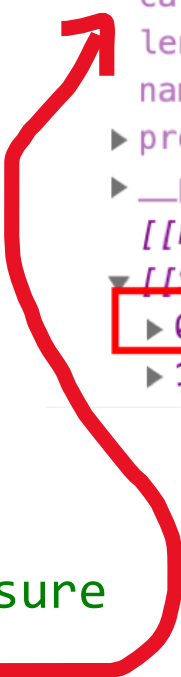
const g = f();
g(); // 14
```

# Closure Details

```
let x = 1;

function foo(y) {
  return function(z) {
    return x + y + z;
  }
}

let f = foo(2); // f is closure
console.dir(f);
```



```
▼ f anonymous(z) ⓘ
  arguments: null
  caller: null
  length: 1
  name: ""
  ► prototype: {constructor: f}
  ► __proto__: f ()
  [[FunctionLocation]]: VM1644:3
  ▼ [[Scopes]]: Scopes[2]
    ► 0: Closure (foo) {y: 2}
    ► 1: Global {parent: Window, postMessage: f, blur: f, focus: f, close:...
```

## Closure Scope

```
arguments: { 0: 2, length: 1 }
```

```
this: window
```

```
y: 2
```



# Execute Closure Details

```
let x = 1;

function foo(y) {
  return function(z) {
    return x + y + z;
  }
}

let f = foo(2); // f is closure
console.dir(f);
f(5);
```

## Closure Scope

arguments: { 0: 2, length: 1 }

this: window

y: 2

## anonymous Execution Context

Phase: Creation

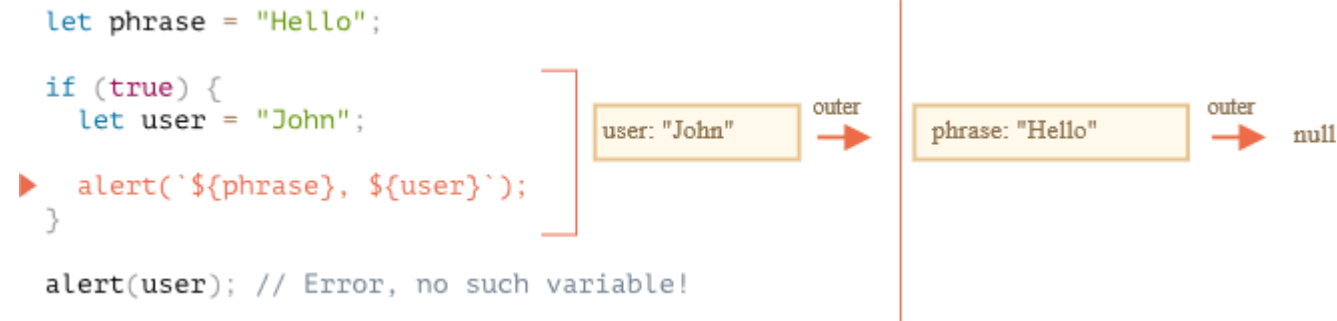
arguments: { 0: 5, length: 1 }

this: window

z: 5

# Code blocks and scope

- a Lexical Environment exists for any code block { . . . }
- created when a code block runs and contains block-local variables.



- When execution gets to the `if` block,
  - new `"if-only"` Lexical Environment is created for it
  - has the reference to the outer one, so `phrase` can be found.
  - all variables and Function Expressions declared inside `if` reside in that Lexical Environment
    - can't be seen from the outside
    - after `if` finishes, the `alert` below won't see the `user`, hence the error.

# For, while

- For a loop, every iteration has a separate Lexical Environment.
  - If a variable is declared in `for(let ...)`, then it's also in there:

```
for (let i = 0; i < 10; i++) {  
    // Each loop has its own Lexical Environment  
    // {i: value}  
}  
alert(i); // Error, no such variable
```

- `for let i` is visually outside of `{ ... }`.
  - The `for` and `while` constructs are special
  - each iteration of the loop has its own Lexical Environment with the current `i` in it.
- like `if`, after the loop `i` is not visible.

# The old “var”

- In the very first chapter about variables, we mentioned three ways of variable declaration:
  1. `let`
  2. `const`
  3. `var`
- `let` and `const` behave exactly the same way in terms of Lexical Environments.
  - `var` is a very different beast,
  - originates from very old times.
  - generally not used in modern scripts, but still lurks in the old ones.
- If you don't plan on meeting such scripts, you may even skip this chapter or postpone it
  - But is used in millions of programs – anything prior to ES6
- function scope
  - Ignores blocks
  - Are always ‘hoisted’, which means they are visible throughout the function
    - Even if defined at the end! (like a function declaration)
    - Only the declaration is hoisted, not the assignment
    - Undefined until assignment reached

# CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Actions Supported by All the Laws of Nature

1. Closures are a feature of functional programming languages that allow state information to be encapsulated with functions when they are passed among objects.
2. The scope of variables in modern JavaScript is defined by the lexical environment.

- 
3. **Transcendental consciousness.** Is the home of all the laws of nature.
  4. **Impulses within the transcendental field:** Thoughts encapsulated by this deep level of consciousness will result in actions in accord with all the laws of nature.
  5. **Wholeness moving within itself:** In unity consciousness all thoughts and perceptions are enhanced by this supportive experience.

