JavaScript

Memory Management

Memory Management

Low level languages like C, have manual memory management primitives such as malloc() and free().

In contrast, JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection) .

Big Confusion

This automaticity is a potential source of confusion. It can give developers the false impression that they don’t need to worry about memory management.

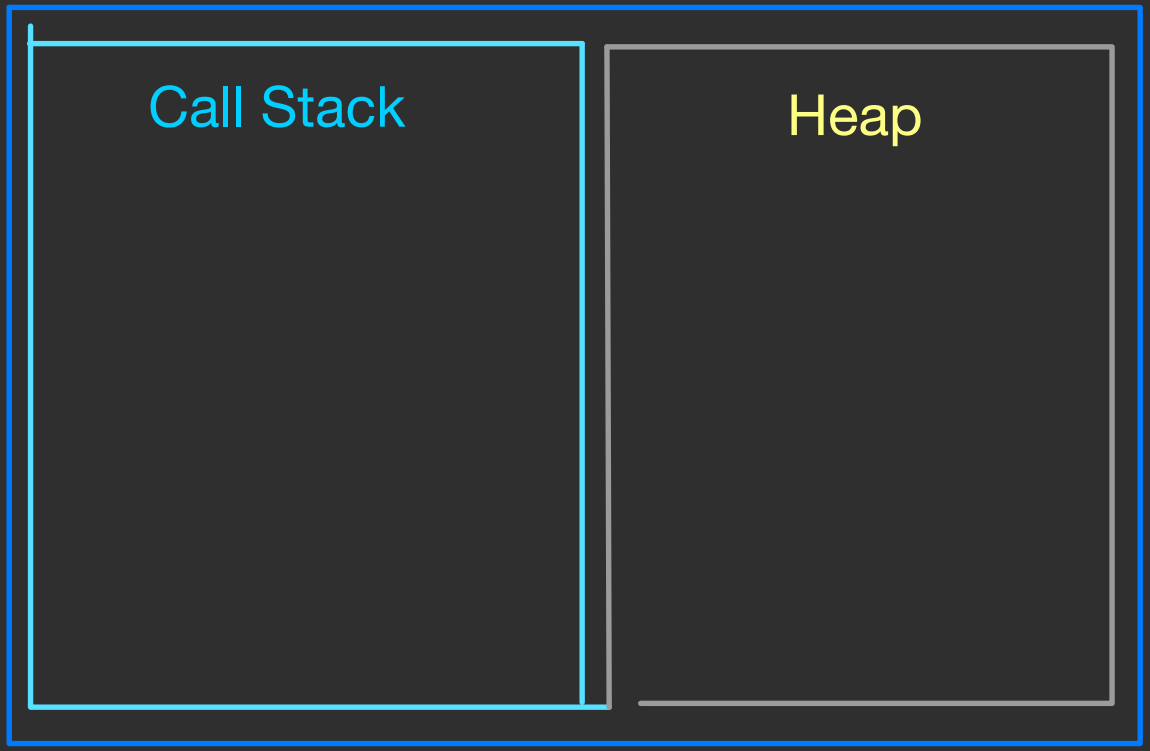
Why Memory Management?

Memory management in program execution is essential because computer memory is a finite and valuable resource.

Programs need memory to store instructions, data, and variables while they are running.

Without proper memory management, several critical issues can arise leading to unstable Slow and potentially insecure software.

Memory diagram



Execution Context

1. Global Execution Context
2. Function Execution Context

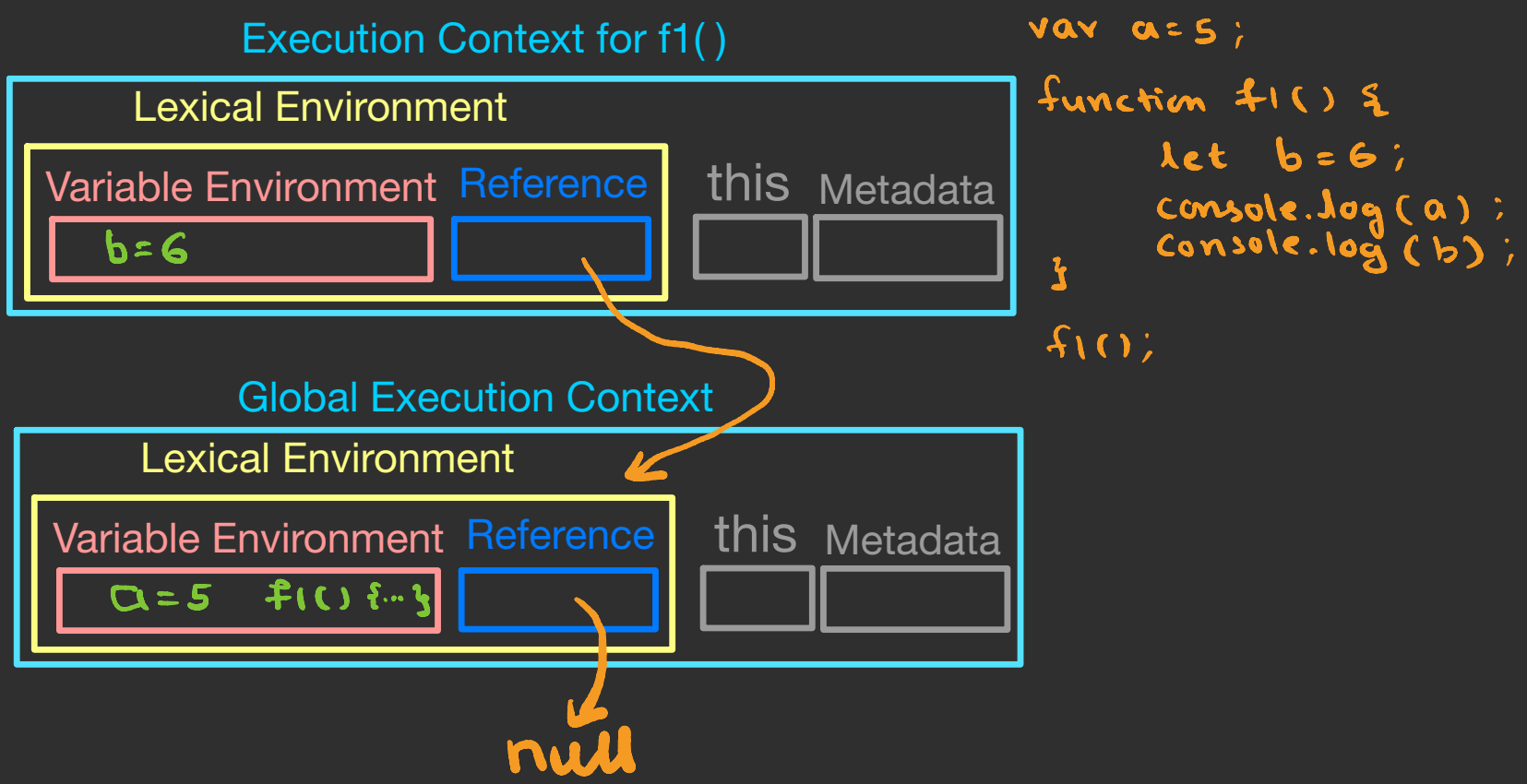
The global execution context is created when a JavaScript script first starts to run and it represents the global scope in Javascript.

A function execution context is created whenever a function is called, representing the function's, local scope .

Each execution context has

- Lexical environment
  - Variable Environment
  - Reference to the outer environment (scope chain)
- this
- Some internal metadata

Heap is not a part of execution context, rather execution context has variable environment which may have references to the non-primitive data stored in the heap


Execution Context for f1( )
Lexical Environment
Variable Environment
Reference
this
Metadata
b=6
Global Execution Context
Lexical Environment
Variable Environment
Reference
this
Metadata
a=5
f1( ) {...}
null
var a=5;
function f1( ) {
let b=6;
console.log(a);
console.log(b);
}
f1( );

Variable Environment is object like structure

Lexical environment is implemented using linked list data structure

Call Stack is an implementation of classic stack data structure

Misconceptions

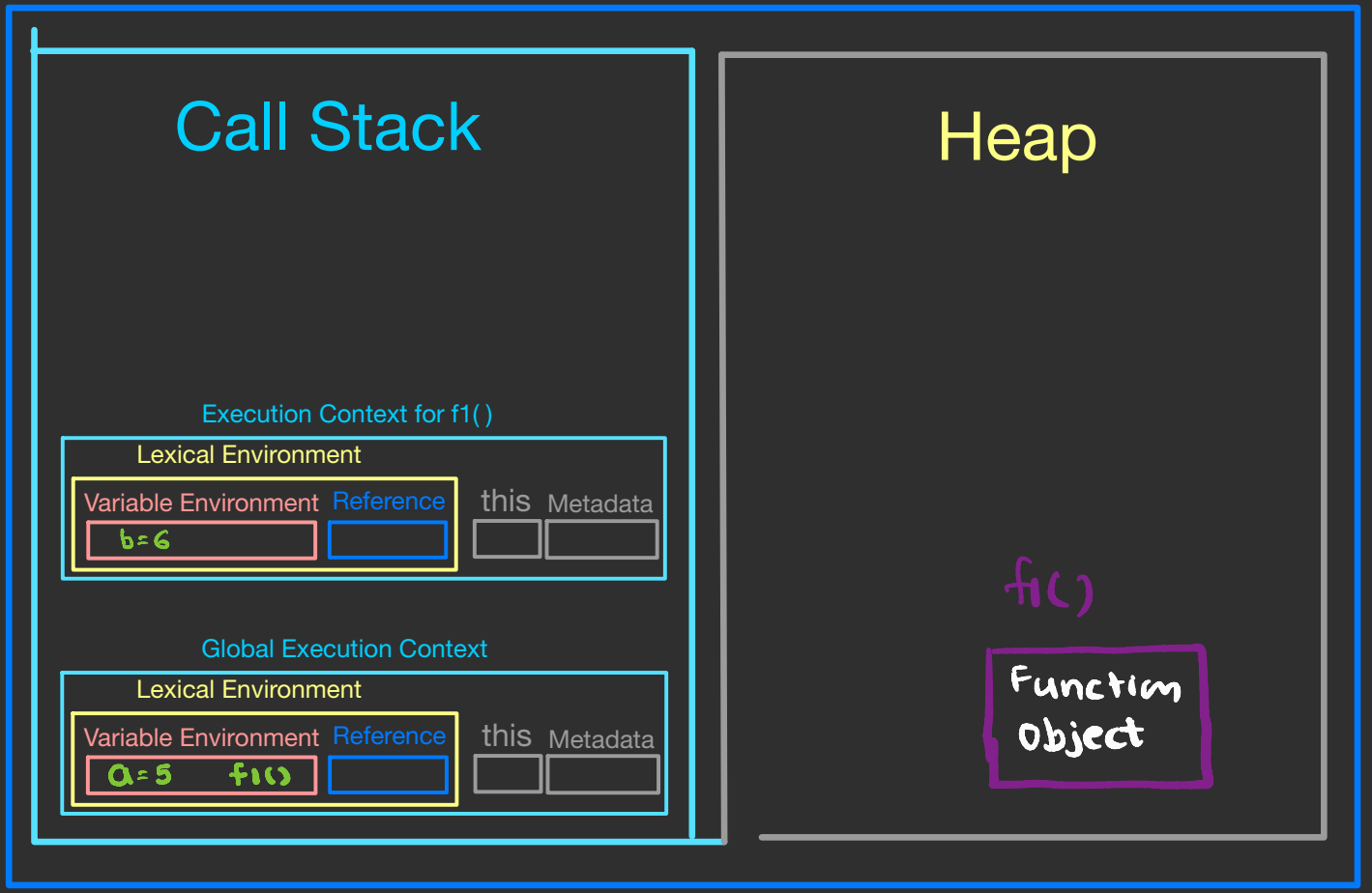
Variable environment is a stack

Heap is a part of execution context

primitive types has a fixed size and values of primitive types are stored directly in the variable environment

Values of non primitive types, and functions are stored in heap and their references are stored in the variable environment.

Memory diagram



Call Stack

Call stack keeps track of where the program is in its function execution(who called what). Think of it as the “who is currently running” list.

Call Stack is implemented using classic STACK data structure.

Code execution works in two steps

1. Memory creation
2. Code execution

Scope Chain

Scope chain deals with variable accessibility.

It decides what variables a function can see when it runs.

1. Global Scope
2. Script Scope
3. function Scope (local Scope)

Scope chain is implemented using Linked List data structure.

Global Scope and Script Scope are slightly different.

Global Scope is tied to the window object (in browsers) or to the global object (in Node.js)

Var a=10 ⟶ Global Scope

let b=20 ⟶ Script scope

a adds to the Window object
b is not a window property

a can be redeclared in the same scope
but b cannot be redeclared in the
same scope

How does a function access global variables if it’s context is on top of the stack?

Because each execution context has a reference to its outer lexical environment, which is the scope where the function was defined, not where it was called from.

```
let name = "MySirG";
function f1(){
    console.log(name)
}
f1();
```

① Global Execution context is created

name = "MySirG"

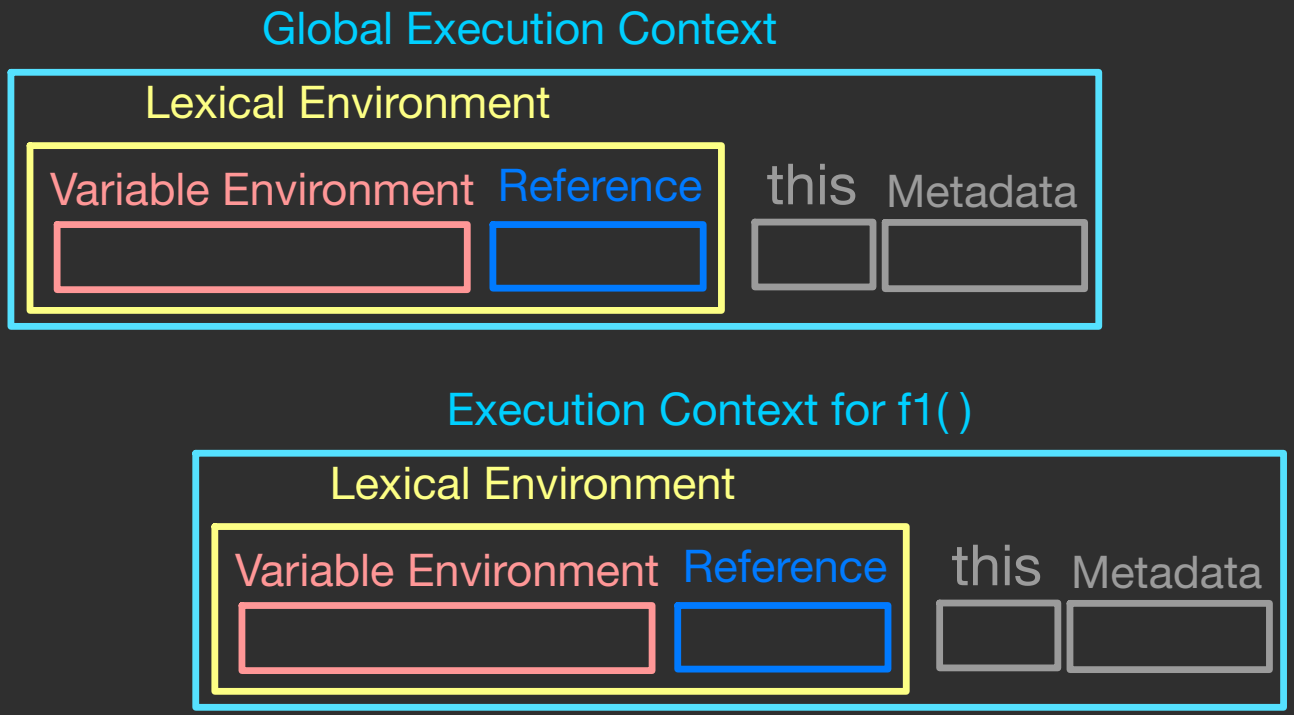
f1 → { ... }

② f1() is called → a new execution context for f1 is pushed on to the call Stack

③ Inside f1 it tries to access name

④ name is not found inside f1, so javascript looks "up the scope chain" to its outer environment (which is global)

⑤ It finds name and log it.


Global Execution Context
Lexical Environment
Variable Environment
Reference
this
Metadata
Execution Context for f1( )
Lexical Environment
Variable Environment
Reference
this
Metadata