JavaScript

```
const LOCALE = globalThis.navigator.language

const div = document.body.appendChild(document.createElement('div'))
const list = div.appendChild(document.createElement('ol'))

const dayNames = new Map()

for (let i = 0; i < 7; ++i) {
    const d   Temporal.PlainDate ro  ({
        yea   mp ra .N w pl in  ISO  .ea ,
        mon  : ,
        d   i
    })

    dayNames.set(d.dayOfWeek, d.toLocaleString(LOCALE, { weekday: 'long' }))
}

for (const num of [...dayNames.keys()].sort((a, b) => a - b)) {
    list.appendChild(Object.assign(
        document.createElement('li'),
        { textContent: dayNames.get(num) },
    ))
}
```

Agenda

- Var, let and const
- Hoisting
- Memory
- Thread and call stack
- Execution context
- Data Types

PRODEVANS

JavaScript Engines

Source

01010
01110

Machine level code

JS

Google chrome (V8)

Edge (chakra)

Mozilla (spider monkey)

Webkit (Safari)

node JS

Runtime Environment for JS

npm

A package manager for JS (dependencies)

Express JS

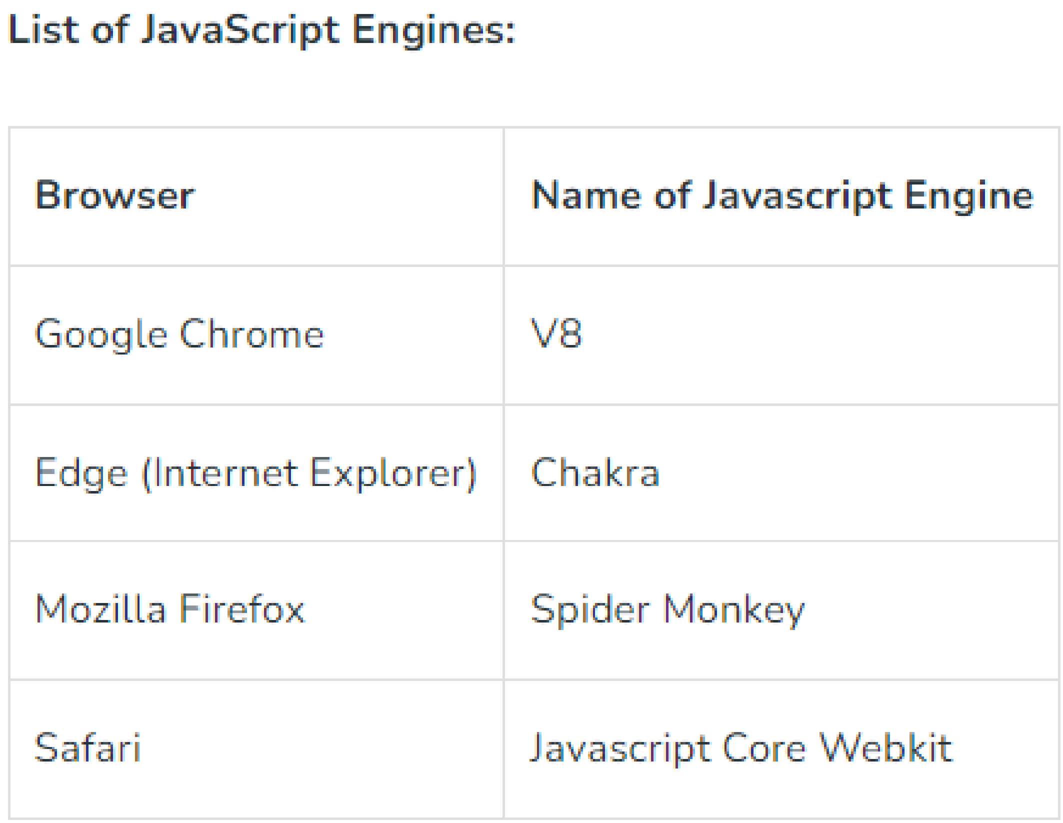
Node.js web application framework

JS

ECMAScript

Standards that JS uses.

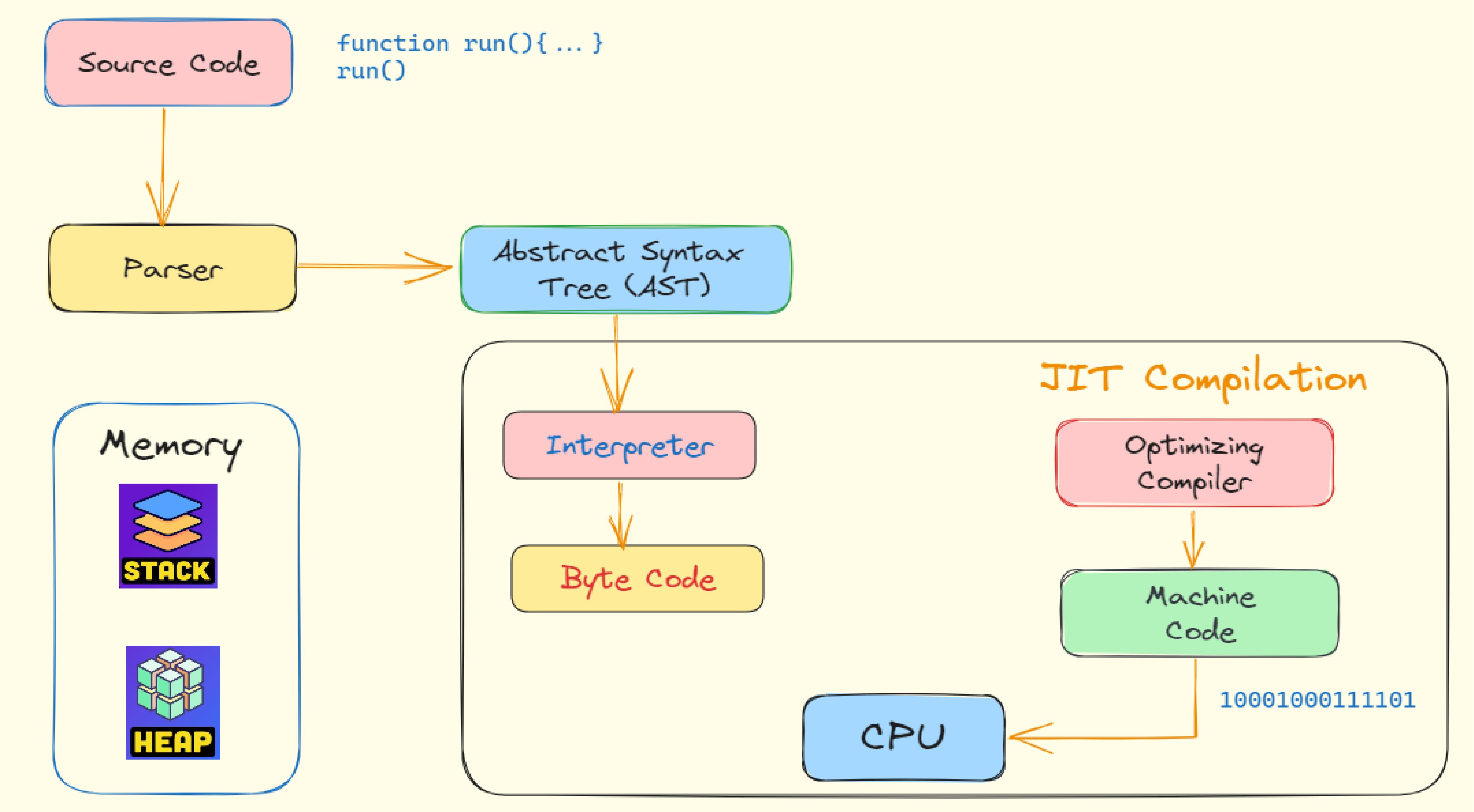
JS ES6



List of JavaScript Engines:

Browser	Name of Javascript Engine
Google Chrome	V8
Edge (Internet Explorer)	Chakra
Mozilla Firefox	Spider Monkey
Safari	Javascript Core Webkit

JavaScript Engine



```
function run(){ ... }
run()
```

Node.js Architecture

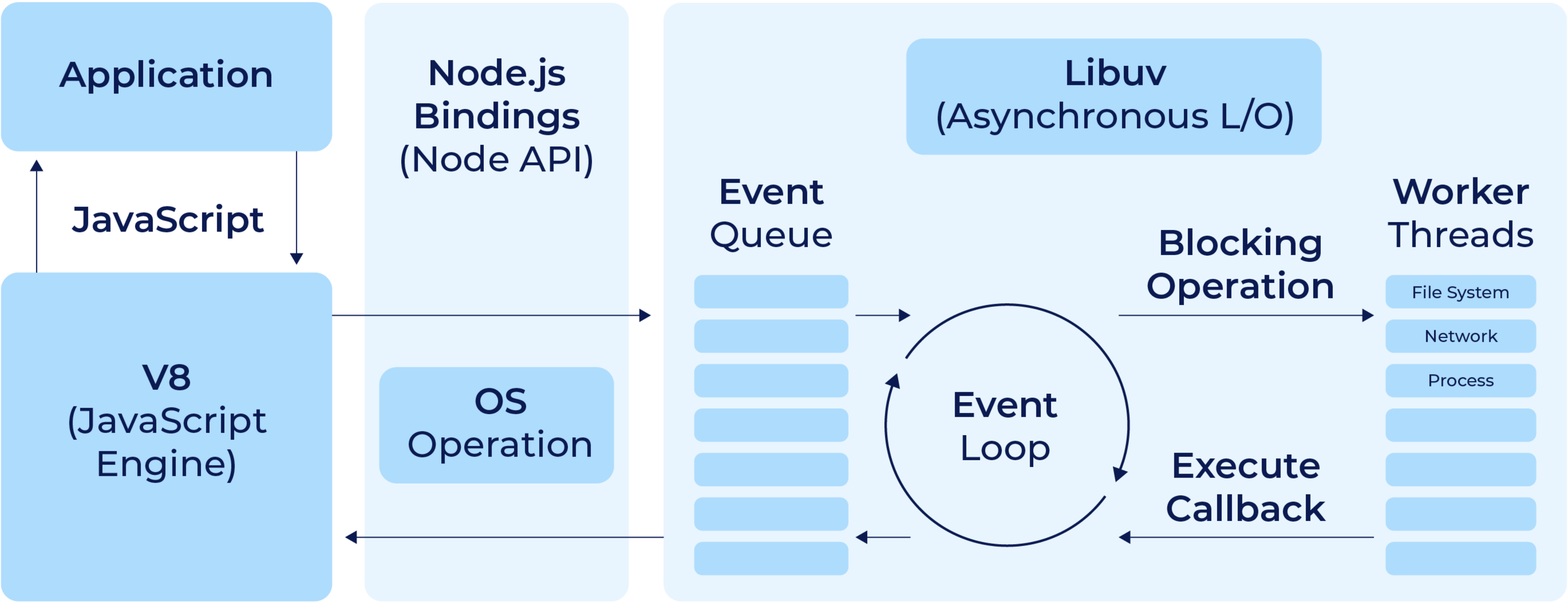
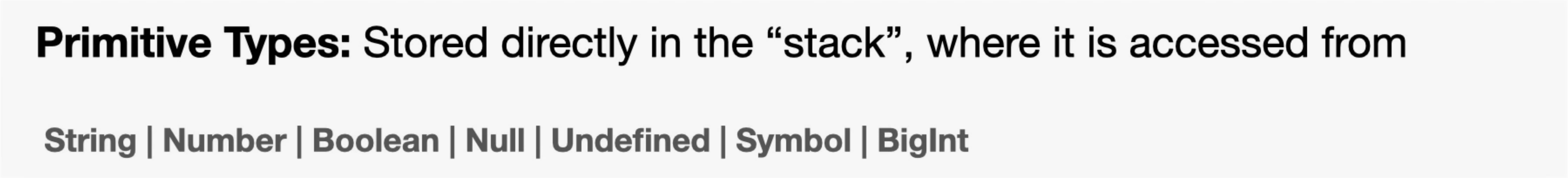


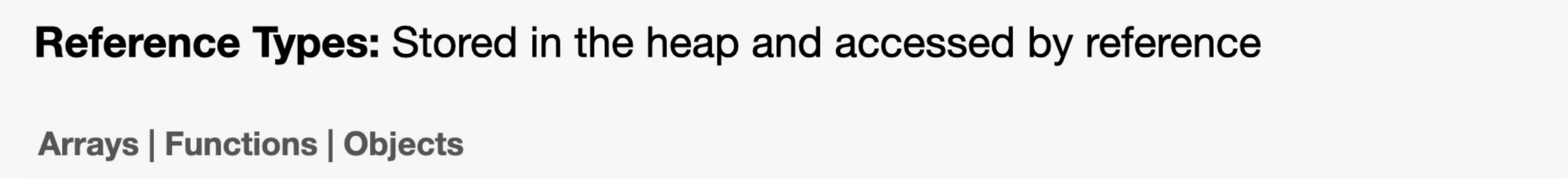
image source: https://www.turing.com/kb/understanding-the-nodejs-architecture

Data Types



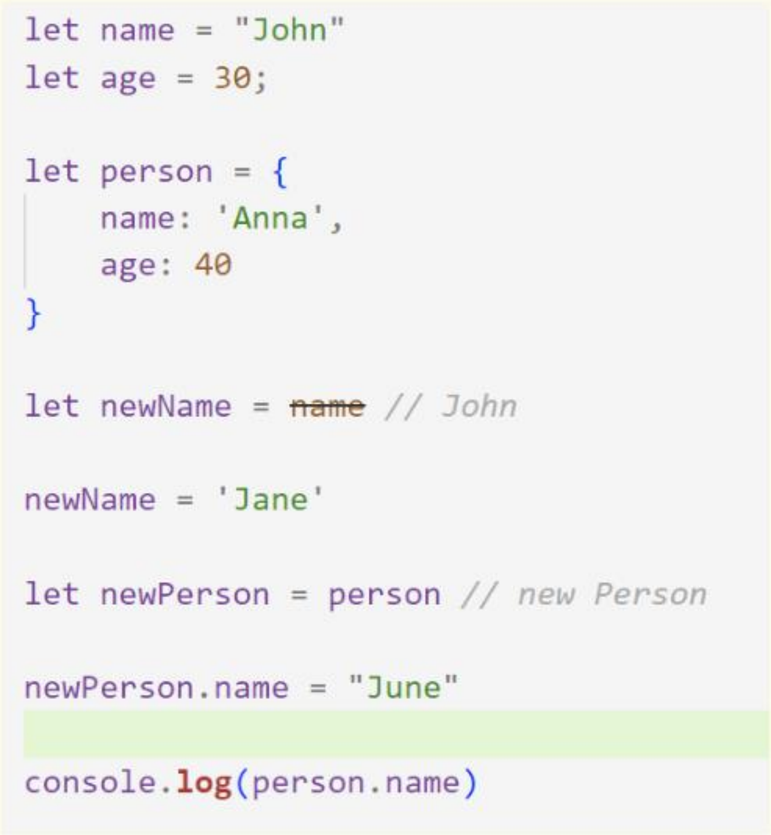
Primitive Types: Stored directly in the “stack”, where it is accessed from

String | Number | Boolean | Null | Undefined | Symbol | BigInt

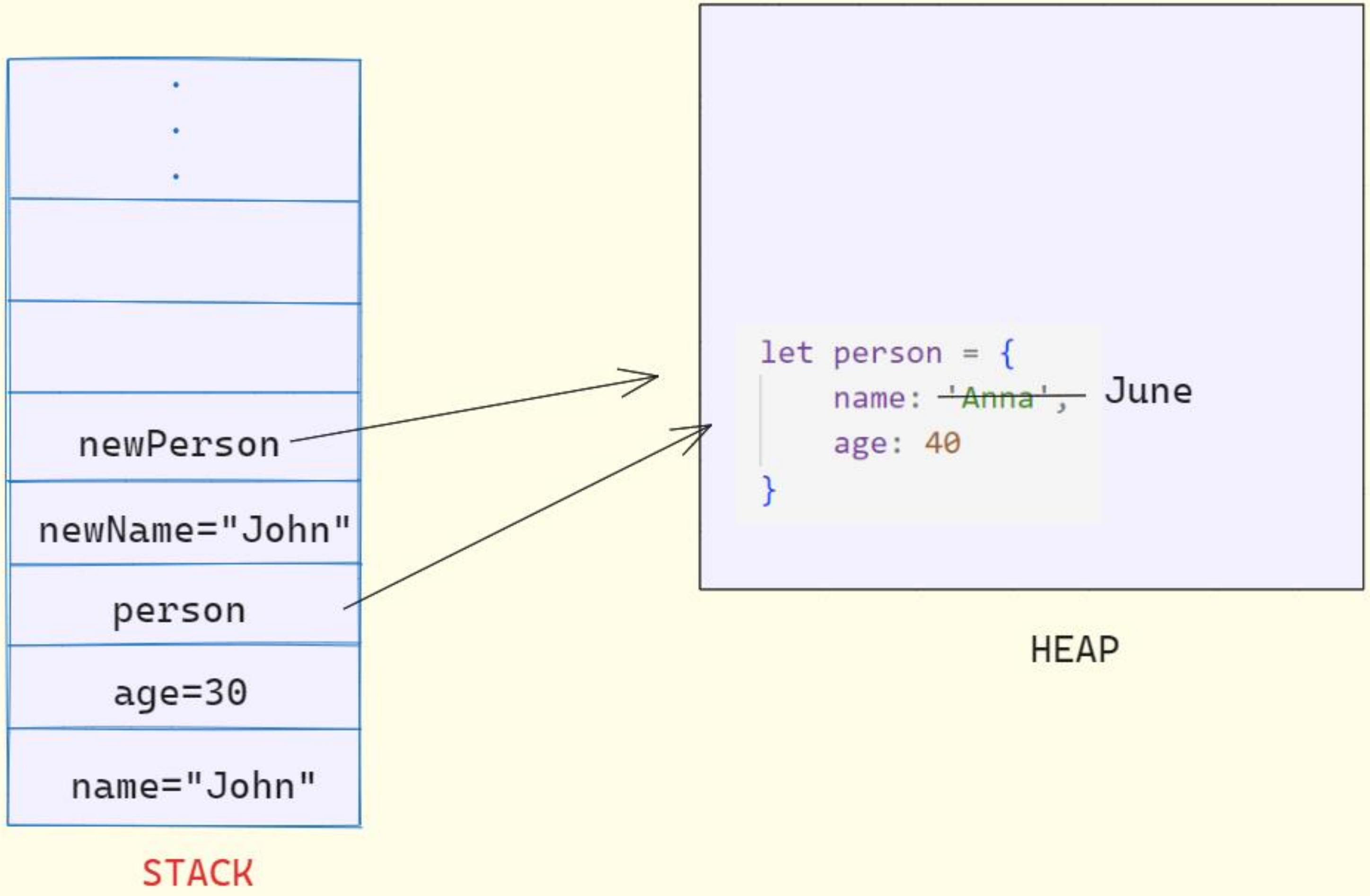


Reference Types: Stored in the heap and accessed by reference

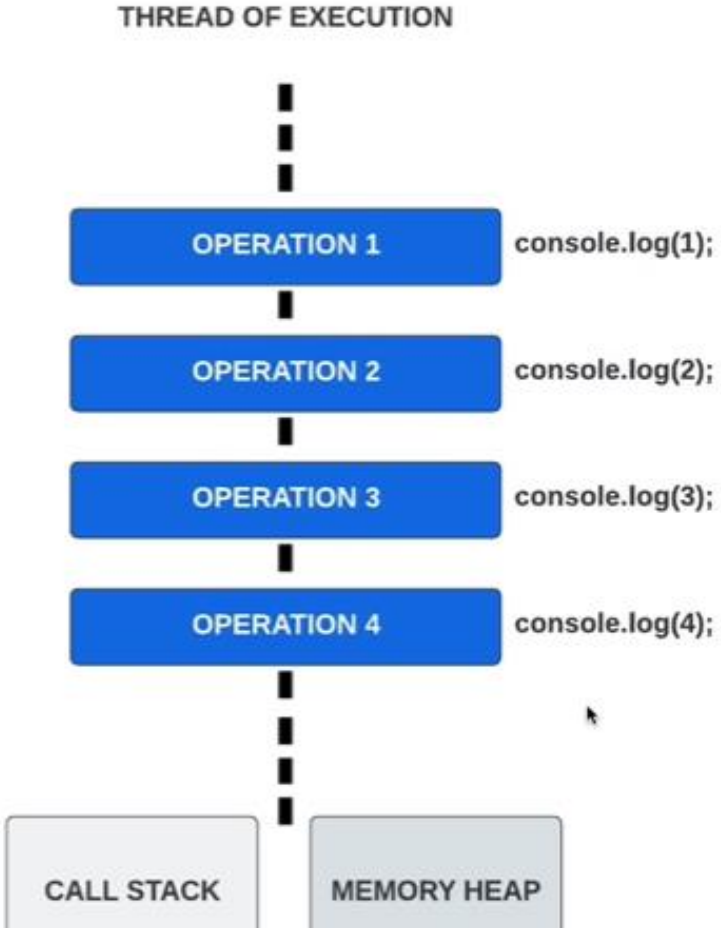
Arrays | Functions | Objects



```
let name = "John"
let age = 30;

let person = {
    name: 'Anna',
    age: 40
}

let newName = ~~name~~ // John

newName = 'Jane'

let newPerson = person // new Person

newPerson.name = "June"

console.log(person.name)
```

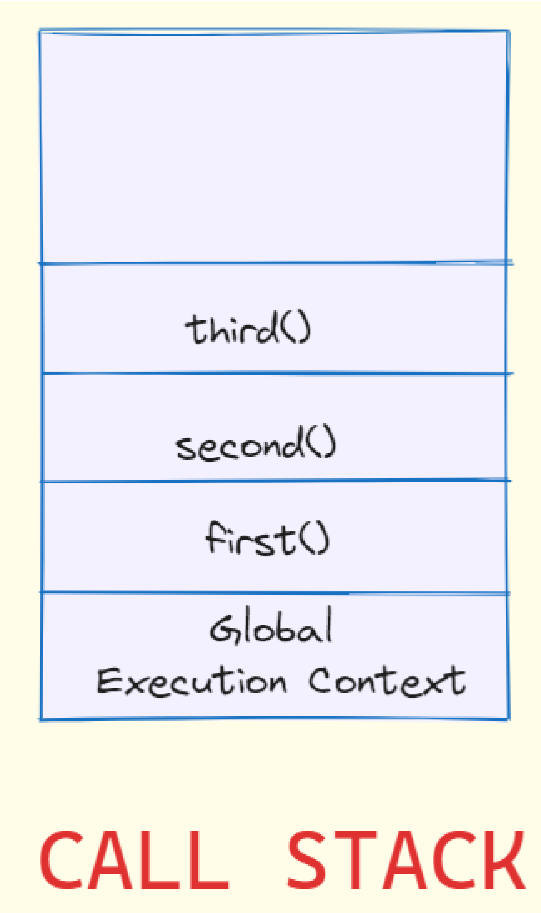


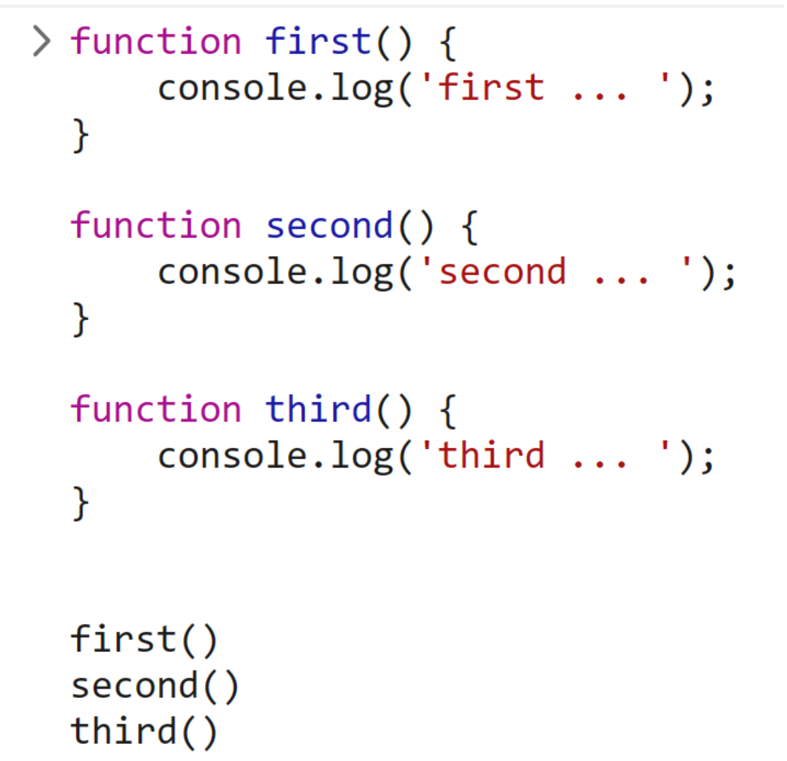
- JavaScript is a **single-threaded language**
- Single sequential flow of control
- JavaScript is a **synchronous language** with asynchronous capabilities
- A thread has a **call stack and memory**



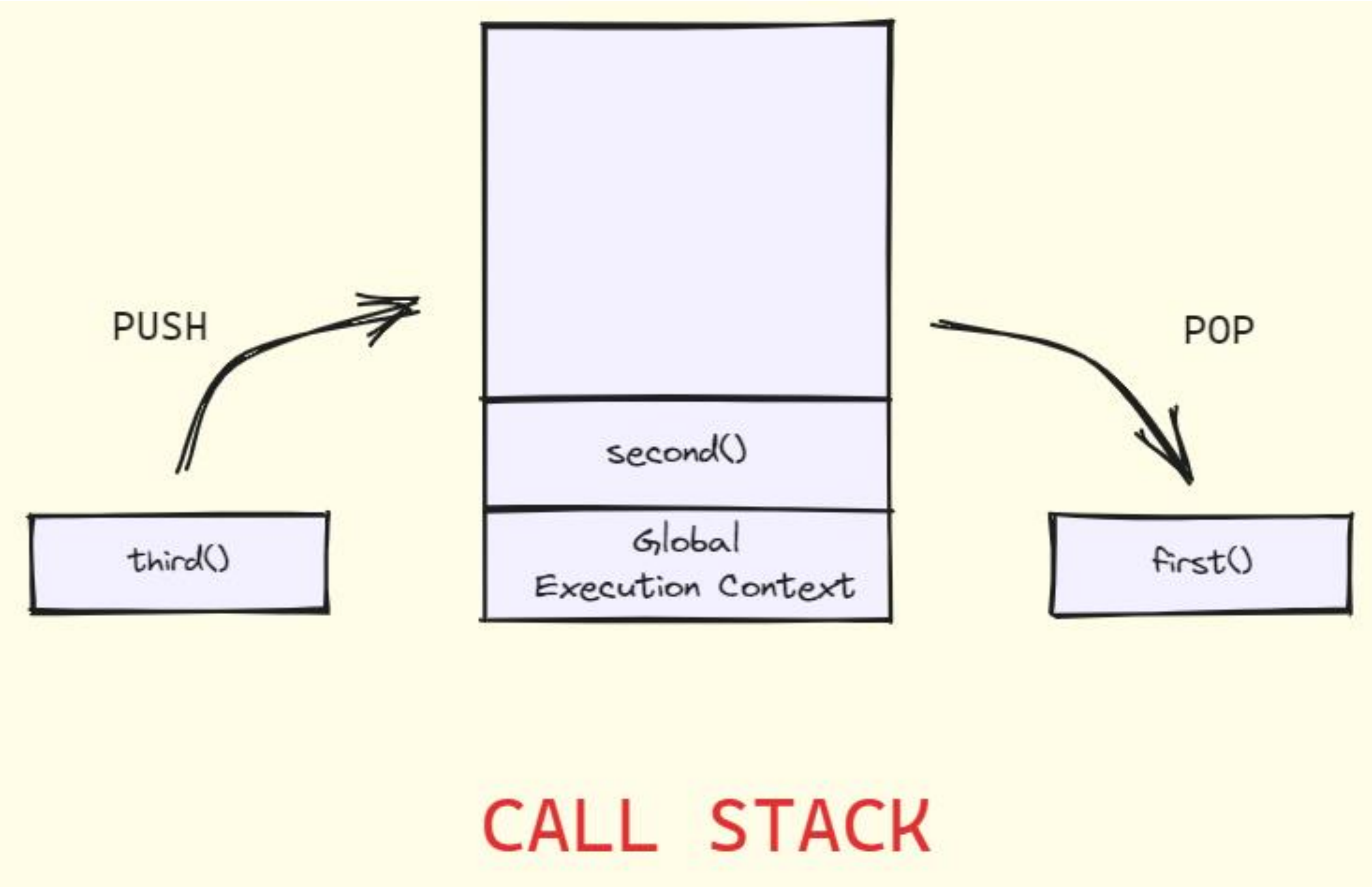
The Call Stack

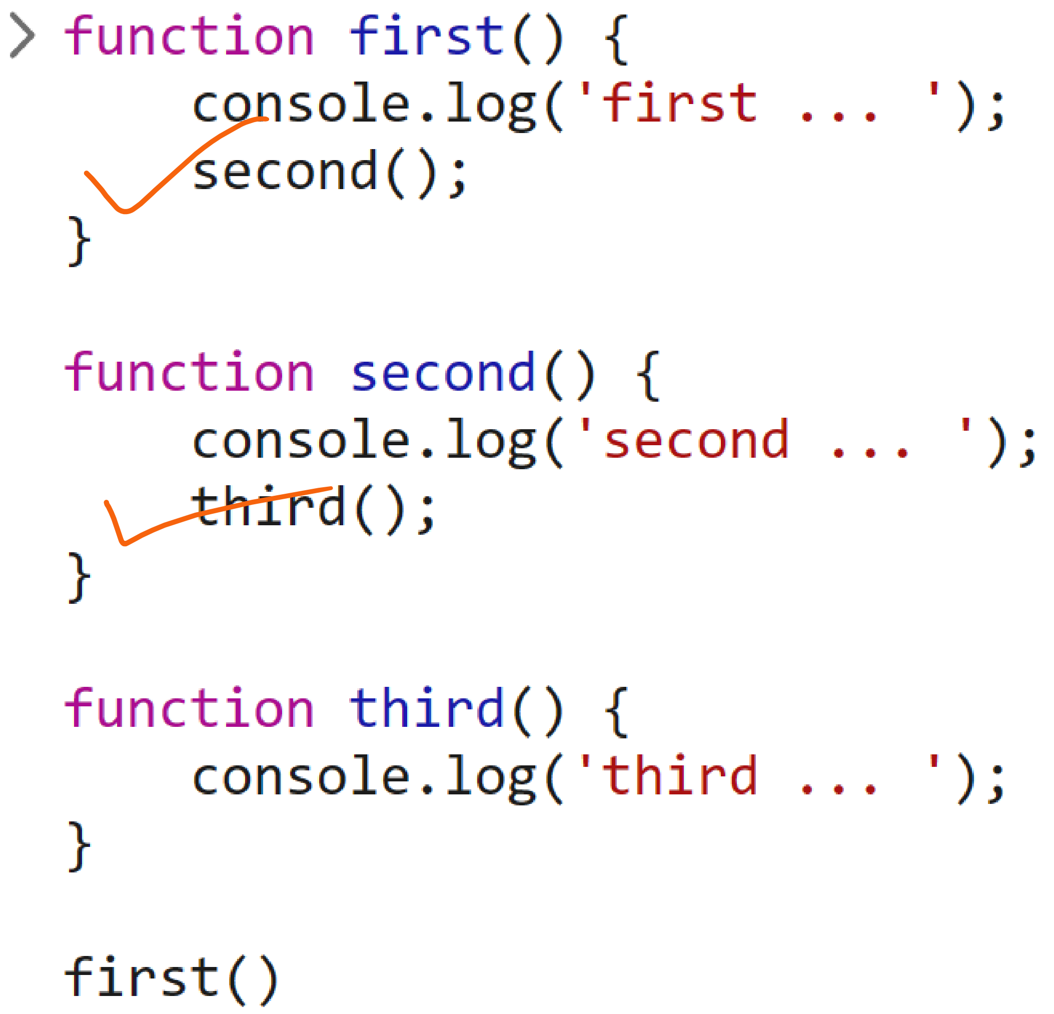
- A call stack keeps track of our functions.
- It manages what we call as **Execution Context.**
- Stacks are LIFO **last in first out**



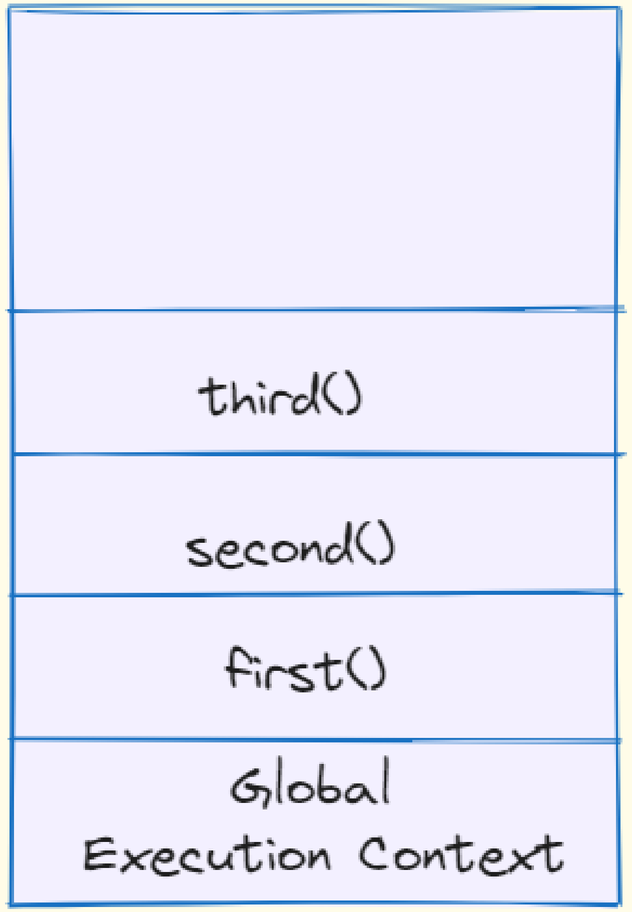


```
> function first() {
      console.log('first ... ');
  }

  function second() {
      console.log('second ... ');
  }

  function third() {
      console.log('third ... ');
  }


  first()
  second()
  third()
```





```
> function first() {
      console.log('first ... ');
      second();
  }

  function second() {
      console.log('second ... ');
      third();
  }

  function third() {
      console.log('third ... ');
  }

  first()
```



CALL STACK

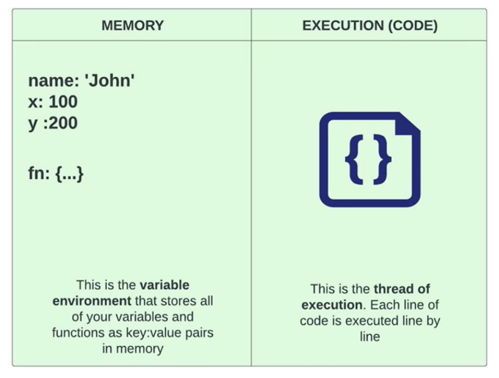
Execution Context

- Whenever we run our JavaScript code, whether in browser or in NodeJS, it creates a special environment that handle the transformation and execution of code. This is called the **execution context**. It contains the currently running code and everything that aids in its execution.

- There is a global execution context as well as a function execution context for every function invoked.

Execution Context

Global Execution Context

MEMORY	EXECUTION (CODE)
name: 'John' **x: 100** **y :200** **fn: {...}** This is the **variable environment** that stores all of your variables and functions as key:value pairs in memory	 This is the **thread of execution**. Each line of code is executed line by line

MEMORY	EXECUTION (CODE)
name: 'John' **x: 100** **y :200** **fn: {...}** This is the **variable environment** that stores all of your variables and functions as key:value pairs in memory	{} This is the **thread of execution**. Each line of code is executed line by line

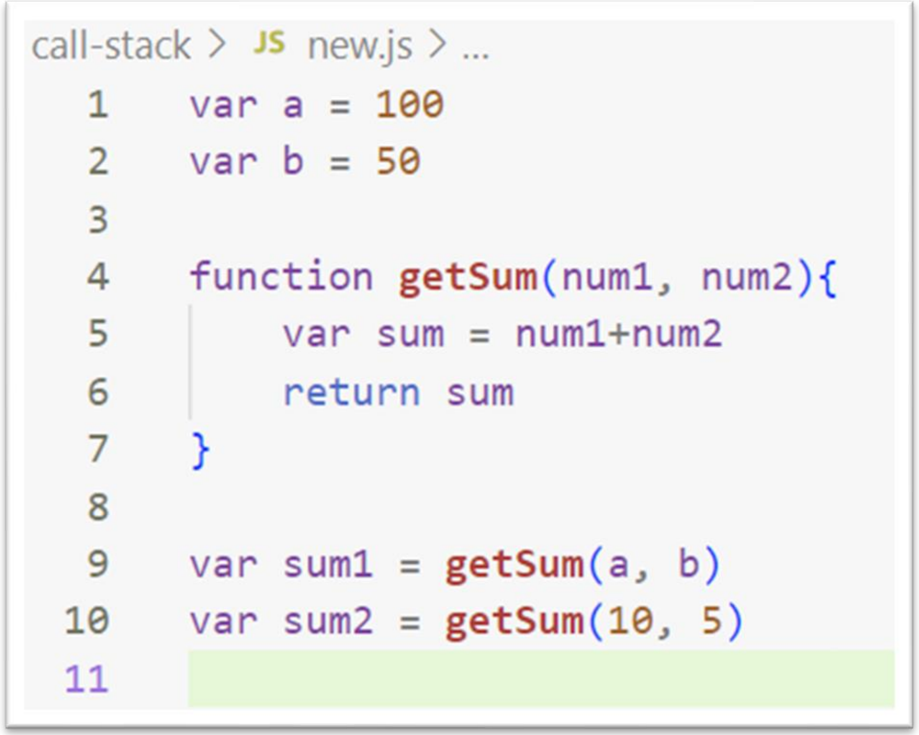
Function Execution Context

Execution Context Phases

- Memory Creation Phase:
  1. Create the global object
     - Browser = window, Node.js = global
  2. Create the **'this'** object and bind it to the global object.
  3. Setup memory heap for storing variables and function references.
  4. Store functions and variables(var) in global execution context and set it to **"undefined"**

- Execution Phase:
  1. Execute code line by line
  2. Create a new execution context for each function call.

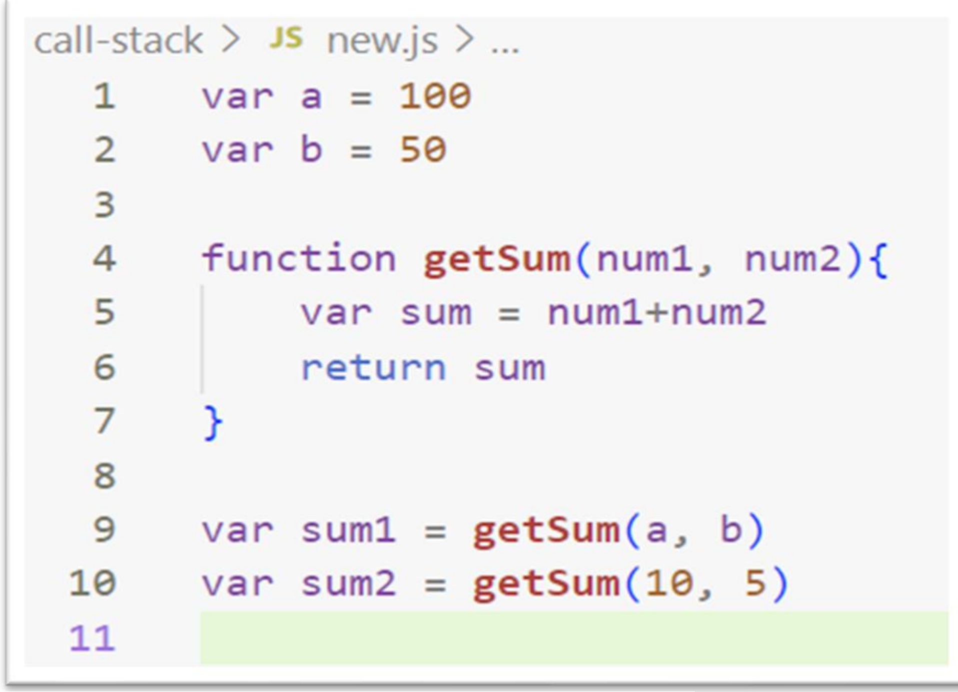
- Creation Phase:
  - **Line 1: a** variable is allocated memory and stores “undefined”
  - **Line 2: b** variable is allocated memory and stores “undefined”
  - **Line 4:** *getSum()* function is allocated memory and stores all the code.
  - **Line 9:** sum1 variable is allocated memory and stores “undefined”.
  - **Line 10:** sum2 variable is allocated memory and stores “undefined”.



```
1   var a = 100
2   var b = 50
3
4   function getSum(num1, num2){
5       var sum = num1+num2
6       return sum
7   }
8
9   var sum1 = getSum(a, b)
10  var sum2 = getSum(10, 5)
11
```

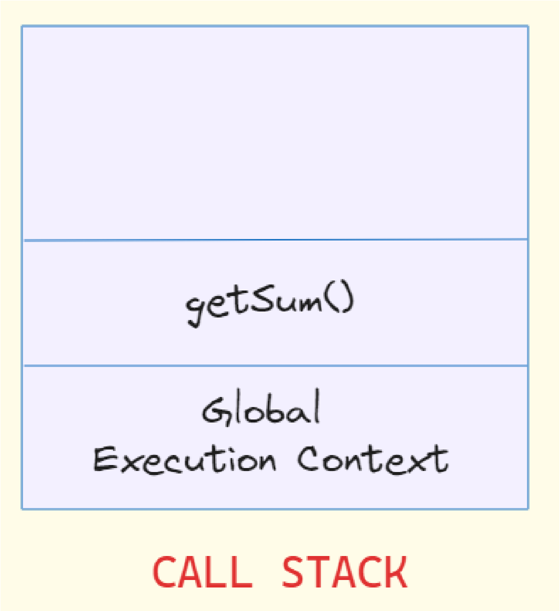
- Execution Phase:
  - **Line 1:** Places the value of 100 into the a variable.
  - **Line 2:** Places the value 50 into the b variable.
  - **Line 4:** Skips the function because there is nothing to execute.
  - **Line 9:** invokes the *getSum()* function and creates a new function execution context

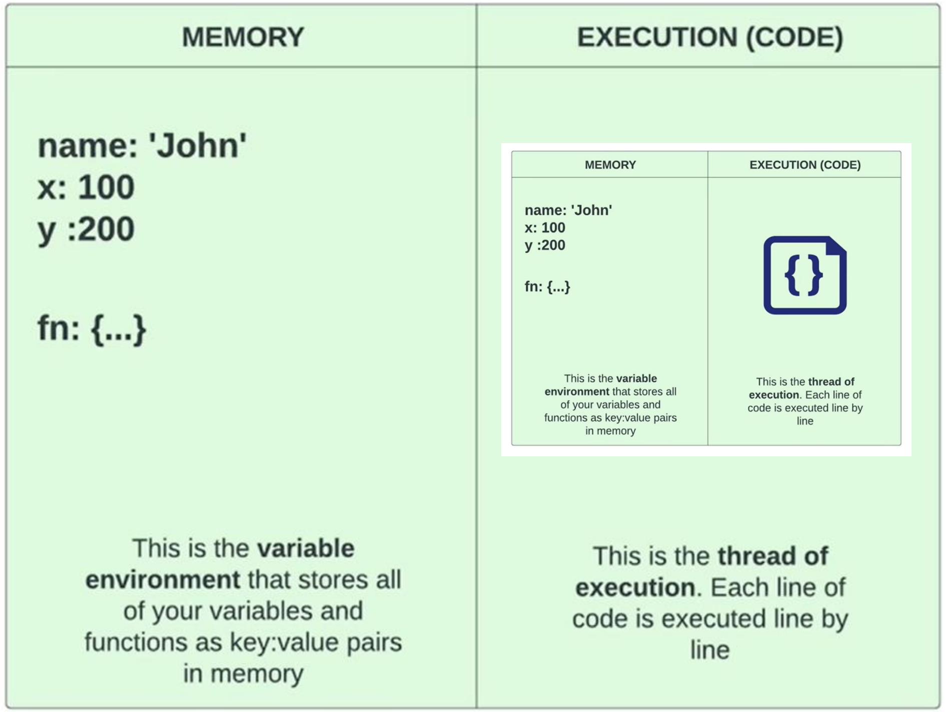
- Function EC Creation Phase:
  - **Line 4:** num1 & num2 variables are allocated memory and stores “undefined”.
  - **Line 5:** sum variable is allocated memory and stores “undefined”
- Function EC Execution Phase:
  - **Line 4:** num1 & num2 are assigned 100 and 50
  - **Line 5:** Calculation is done and 150 is put into the sum variable
  - **Line 6:** return tells the function EC to return to the global EC with value of the sum = 150
  - **Line 9:** Returned sum value is put into the sum1 variable.
  - **Line 10:** Open another function EC and do the same thing



call-stack > JS new.js > ...

```
1   var a = 100
2   var b = 50
3
4   function getSum(num1, num2){
5       var sum = num1+num2
6       return sum
7   }
8
9   var sum1 = getSum(a, b)
10  var sum2 = getSum(10, 5)
11
```





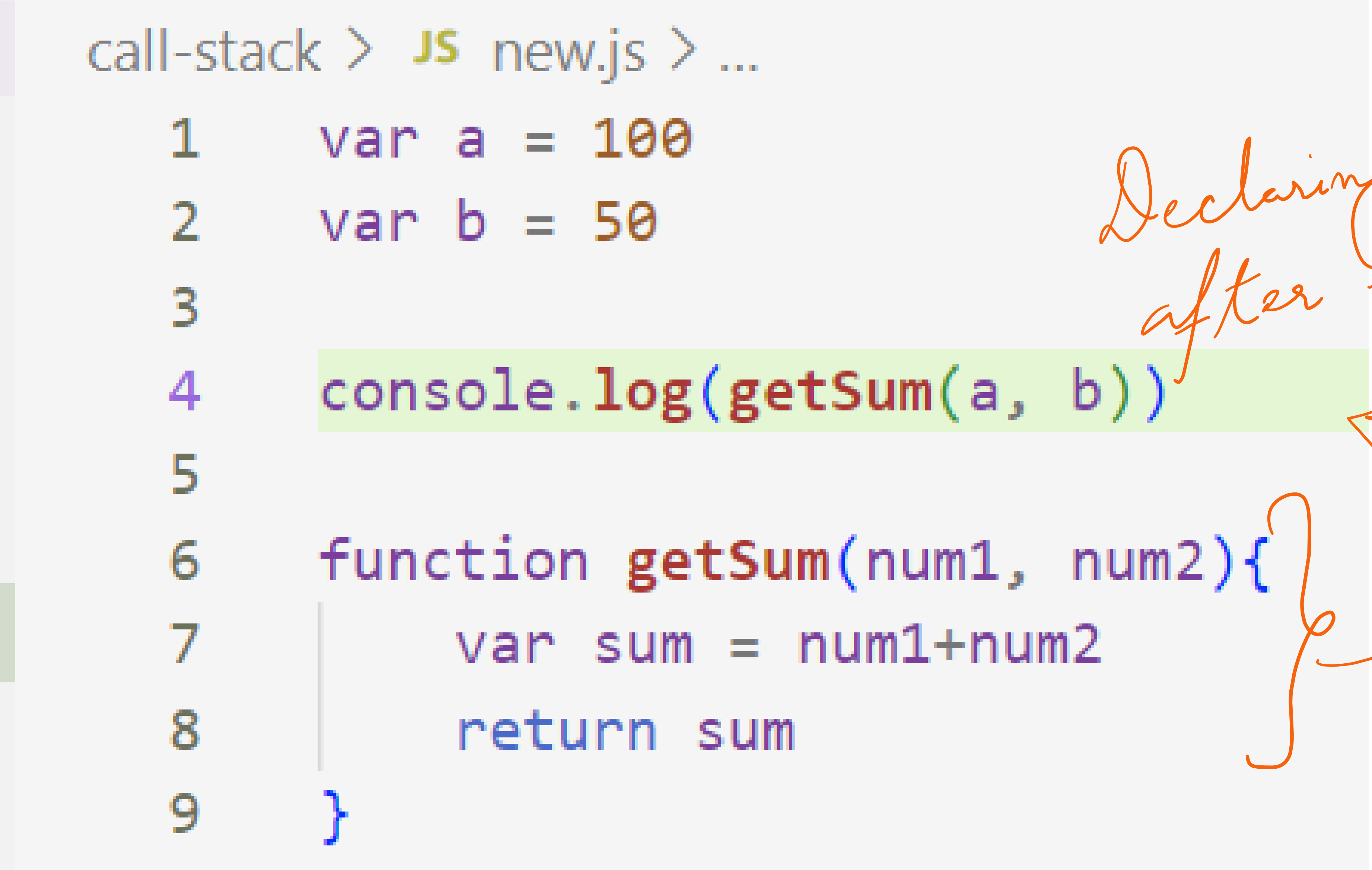
MEMORY	EXECUTION (CODE)
name: 'John' x: 100 y :200 fn: {...} This is the **variable environment** that stores all of your variables and functions as key:value pairs in memory	This is the **thread of execution**. Each line of code is executed line by line

MEMORY	EXECUTION (CODE)
name: 'John' x: 100 y :200 fn: {...} This is the **variable environment** that stores all of your variables and functions as key:value pairs in memory	This is the **thread of execution**. Each line of code is executed line by line

Hoisting

Hoisting is often referred to as the process where the interpreter appears to **move the declaration of function and variables** to the top of their scope prior to the execution of the code.

Hoisting

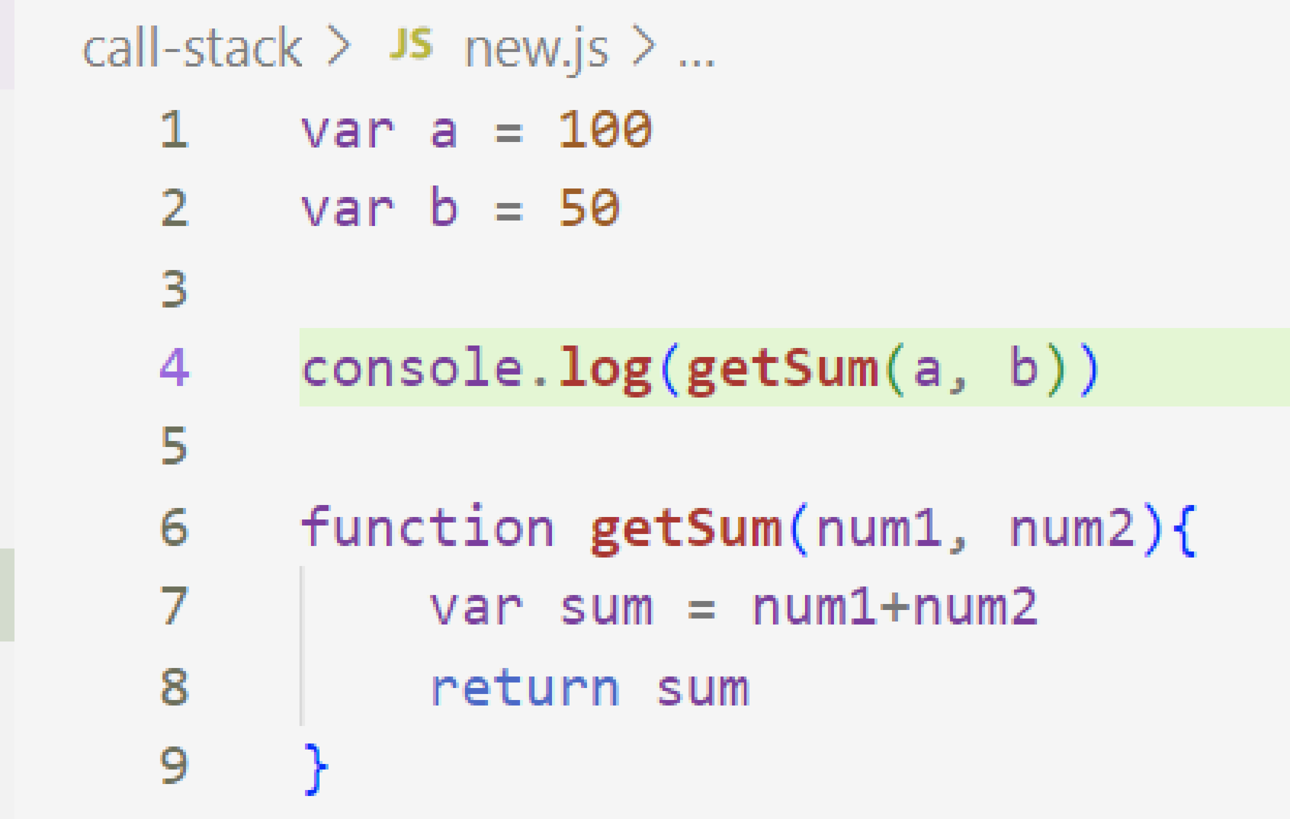


call-stack > JS new.js > ...

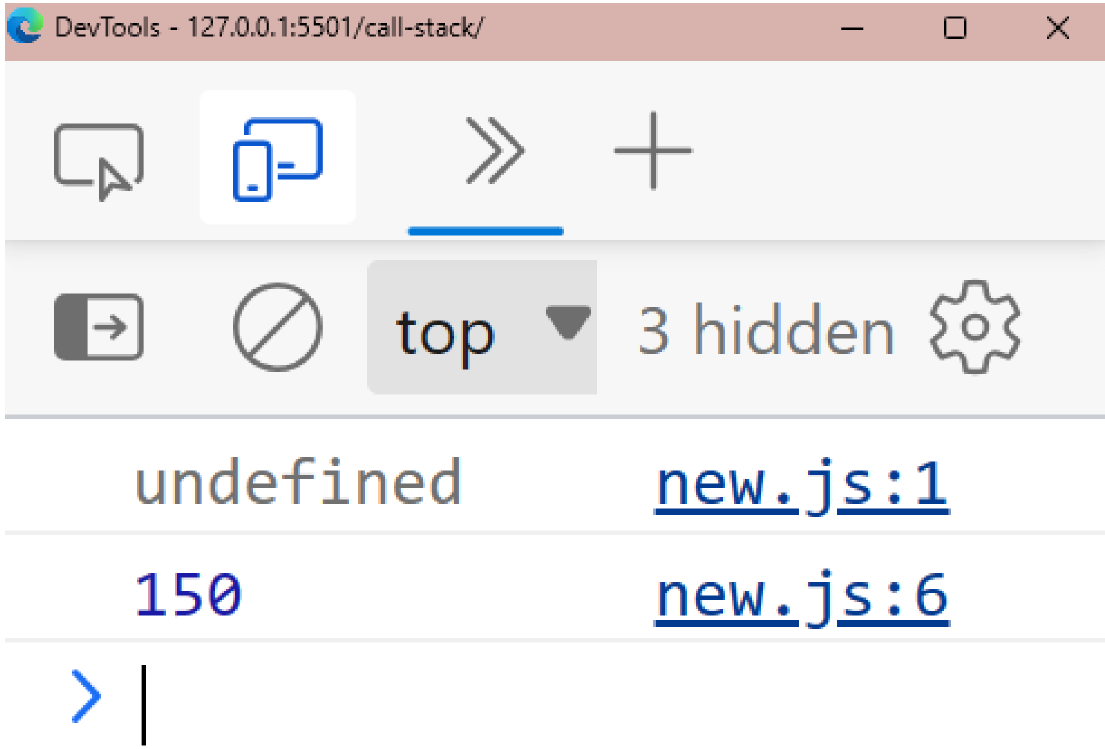
```
var a = 100
var b = 50

console.log(getSum(a, b))

function getSum(num1, num2){
    var sum = num1+num2
    return sum
}
```

Declaring method after it's called

Hoisting

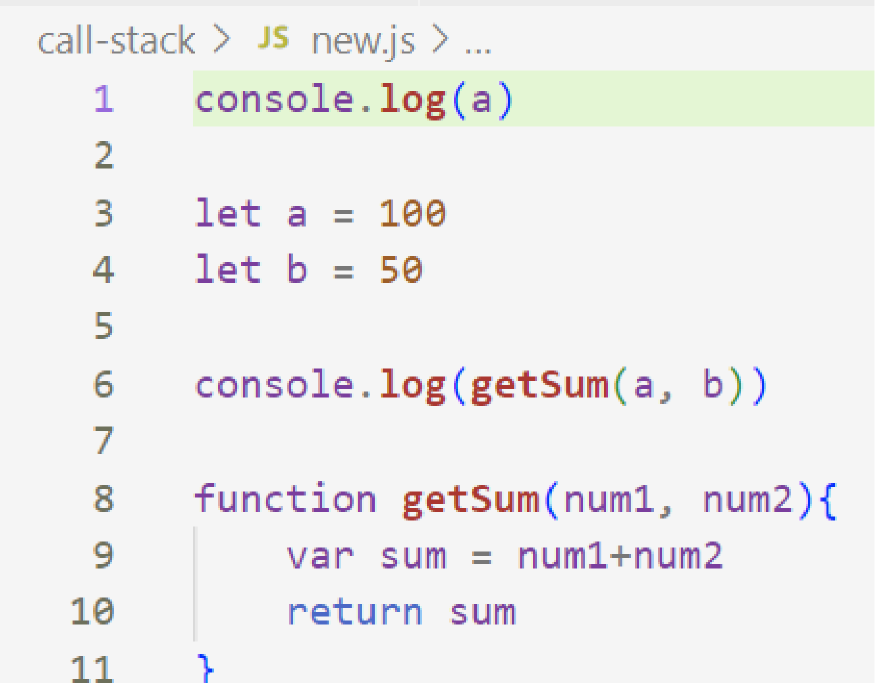


```
var a = 100
var b = 50

console.log(getSum(a, b))

function getSum(num1, num2){
    var sum = num1+num2
    return sum
}
```

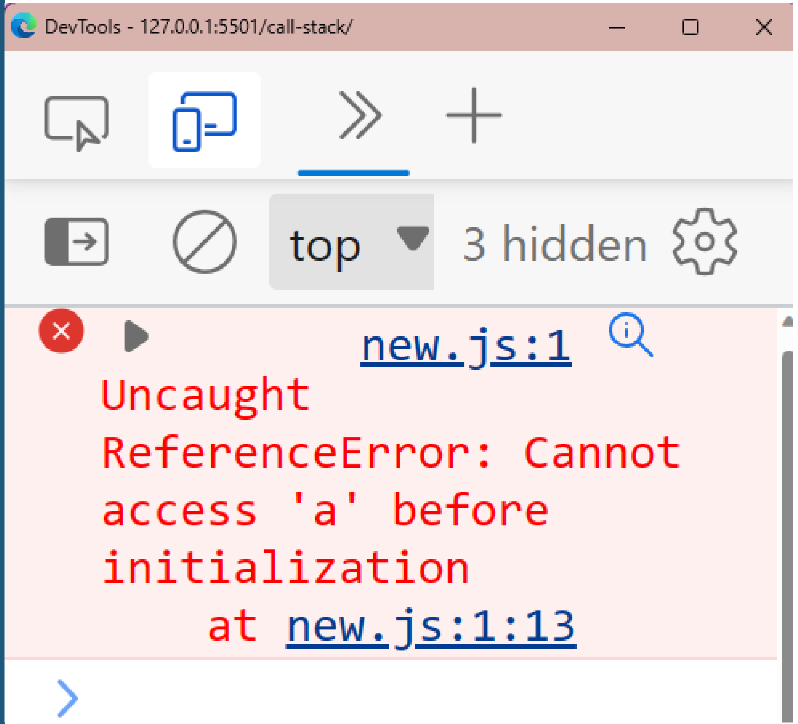


```
undefined                new.js:1
150                      new.js:6
>
```

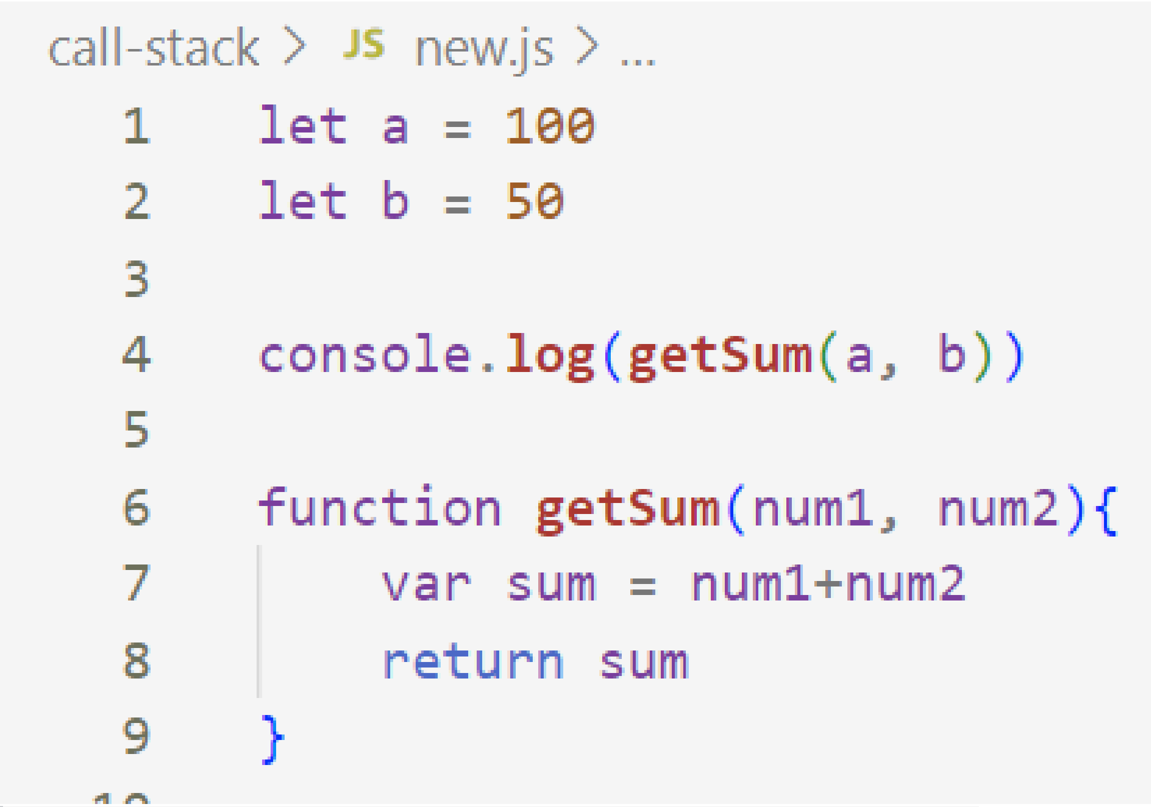
var, let and const



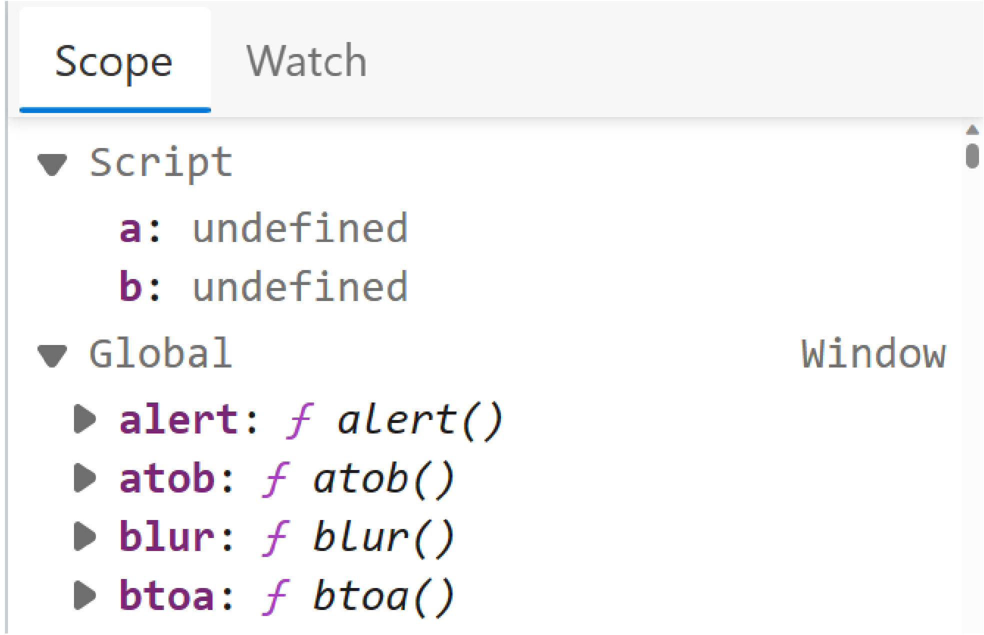
```
console.log(a)

let a = 100
let b = 50

console.log(getSum(a, b))

function getSum(num1, num2){
    var sum = num1+num2
    return sum
}
```



```
new.js:1
Uncaught
ReferenceError: Cannot
access 'a' before
initialization
    at new.js:1:13
```



```
let a = 100
let b = 50

console.log(getSum(a, b))

function getSum(num1, num2){
    var sum = num1+num2
    return sum
}
```



```
▼ Script
    a: undefined
    b: undefined
▼ Global                    Window
  ▶ alert: ƒ alert()
  ▶ atob: ƒ atob()
  ▶ blur: ƒ blur()
  ▶ btoa: ƒ btoa()
```