Inside JavaScript

Synchronous, Single-Threaded Language

Imagine JavaScript as a chef in a kitchen that can only focus on one dish at a time and doesn't have any assistants.

Why JavaScript is Synchronous:

1. **One Task at a Time:** Just like our chef can cook only one dish at a time, JavaScript can handle only one operation at a time. It finishes cooking one dish before moving on to the next.
2. **No Skipping Steps:** If a chef is busy chopping vegetables, they can't suddenly start cooking meat. Similarly, JavaScript completes one task before starting another, making it a synchronous language.

What Makes JavaScript Single-Threaded:

Scenario:

Order 1: You order a salad.
Order 2: At the same time, someone else orders a pizza.

Single-Threaded (One Chef) Behavior:

Step 1: The chef starts making your salad first because it's the first order they received.
Step 2: After finishing your salad, the chef then begins making the pizza for the second order.

Result:

1. Even though both orders came in around the same time, the chef can only work on one dish at a time due to being single-threaded. So, the pizza order has to wait until the salad is prepared.

In Short:
JavaScript is like a single chef in a kitchen, cooking one dish at a time. It can't jump between tasks, ensuring that everything happens in a specific order.

---

1. Execution Context

An execution context is like a container that holds all the necessary information for a piece of code to be executed. Every time a JavaScript code is run, it creates a global execution context. When functions are invoked, new execution contexts are created for each function call.

2. Phases of Execution Context

2.1. Memory Component:

- Variable Environment: In this phase, JavaScript engine stores variable declarations and function declarations. The variables are initialized to `undefined` during this phase.

2.2. Code Component:

- Execution Phase: After setting up the memory, the JavaScript engine starts executing the code line-by-line.

3. Call Stack

Each function call creates a new execution context, forming a stack known as the call stack. The most recently invoked function's context is at the top of the stack and is the currently executing context.

4. Function Execution & Memory Release

Once a function completes its execution, its corresponding context is removed from the top of the call stack. This process continues until all functions have been executed. As contexts are popped off, the associated memory is freed up.

Example of Memory Allocation

1. After Assignment:
   a. Variables: `x` (value: 5)
   b. Functions: `printX()`
2. Before Assignment:
   a. Variables: `x`
   b. Functions: `printX()`
3. If `x` is Not Defined:
   a. Variables: `x` (value: undefined)
   b. Functions: `printX()`
4. Debugger Console Output:
   a. Undefined Variable
   b. Function Not Defined Variable

Example for Arrow Functions

Debugger console output for arrow functions can show specific behaviors related to the lexical `this` value and memory allocation.

Example of Call Stack

The call stack can be visualized using tools like browser developer tools, showing the order in which functions are called and executed.

Example of Call Stack and Memory Allocation

1. Demonstrates how execution contexts are created and stacked for each function call.
2. Illustrates how the call stack is managed as functions complete execution and contexts are popped off, freeing up memory.

Window Object Explanation

1. Properties and Methods: The Window Object contains properties and methods related to the browser window, such as `window.location`, `window.alert()`, etc.
2. Creation and Destruction: The Window Object is created when the browser window opens and destroyed when it closes.
3. `this` Keyword: At the global level, `window` object is equivalent to `this`. Example: `(window) === (this)`.
4. Accessing Variables and Functions: Variables and functions can be accessed using `this` or `window` within the global scope.

Explanation of Undefined and Not Defined

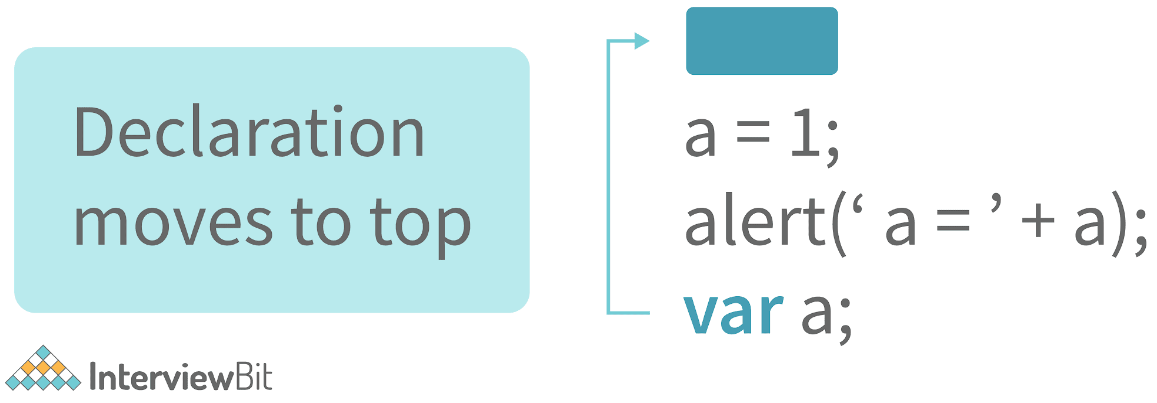
1. Undefined: Occurs when a variable is declared but not assigned a value.
2. Undefined in Memory Component: JavaScript engine assigns `undefined` as a placeholder during the memory component phase.
3. Not Defined: Occurs when a variable is referenced but not declared.

Explanation of Scopes and Environment

1. Scope Chain: It's a sequence of execution contexts that have access to the current function's variables and functions.
2. Lexical Environment: It's a part of the memory component of an execution context, containing the variable environment and the scope chain.

Hoisting in JavaScript

Hoisting is the default behaviour of javascript where all the variable and function declarations are moved on top.



1. **Var, Let, and Const:** `var` declarations are hoisted to the top of their scope, but with `let` and `const`, there is a concept of the "**temporal dead zone**," where variables are hoisted but not initialized until the actual declaration.