Assignment

Assignment of Class 3

Internship Class Assignment - 3

Name:	Ali Ahasan
Position:	Intern
Tracking Number:	J2DFEEB
Team:	InnovX Team of Business Automation Limited

Task:

- Study about **Javascript execution context full cycle**.
- Describe Javascript execution context as you have understood.

Instructions:

1. Complete all the tasks as outlined above.
2. Submit the answer in .zip file through the provided Google Form.

Contents

Javascript execution context full cycle.

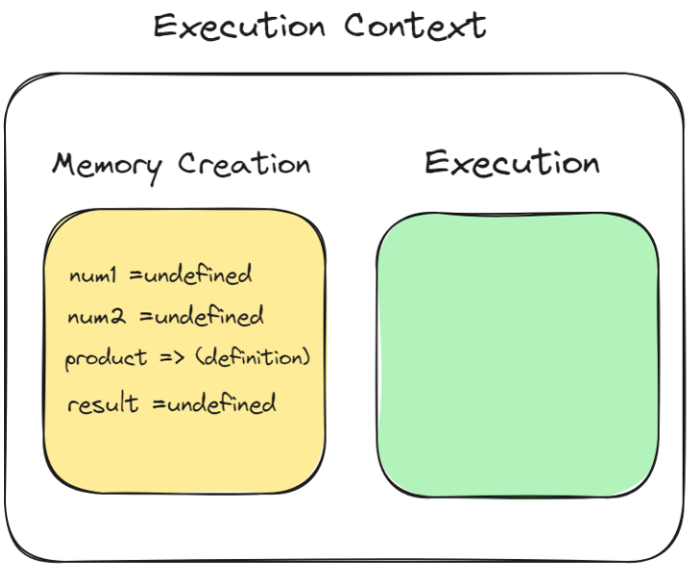
The concept of the JavaScript execution context significantly impacts. This is for how the JavaScript engine processes and runs code. It relates to the surrounding environment or context. In which JavaScript code is evaluated and run. Understanding this concept is important and mandatory. For grasping variable scoping, function behavior and the JavaScript engine's handling. Consists of the execution stack. The JavaScript execution context cycle is outlined. Below in its entirety:

1. Creation Phase

For the initial stage, the JavaScript engine performs various preparatory tasks:

- **Lexical Environment Creation:** It sets up the lexical environment. Lifting declarations of variables and functions within it. For this case, variables are assigned undefined values. Whereas functions are fully initialized.
- **Scope Chain Establishment:** Every execution context has a scope chain. This chain orders the hierarchy of variable access. Including variables from the current context and all previous execution contexts.
- **this Binding:** The execution context determines the value of this. In a worldwide perspective, this pertains to the global entity (usually window in internet browsers). In function contexts, the reference of this changes based on how the function is called.

Depiction:



Example:

Consider the following code snippet:

```
console.log(foo); // Outputs: undefined
var foo = "Hello";
console.log(foo); // Outputs: Hello
```

In the creation phase, the variable foo is hoisted also initialized as undefined. It's only assigned the value "Hello" during the execution phase.

2. Execution Phase

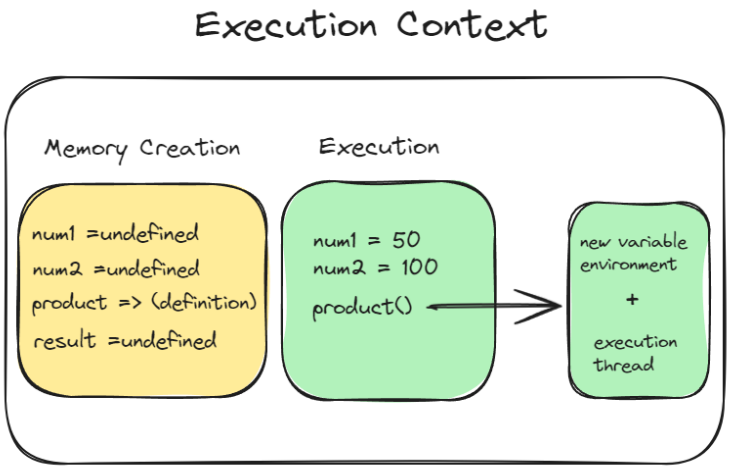
Once the environment is setup, the JavaScript engine starts running the code:

- **Variable Assignment:** In this stage, variables and functions that were hoisted in the creation phase. That are assigned their values as the engine goes through the code. Which is in order.
- **Function Execution:** When a function is called, the JavaScript engine creates a new execution context. That is tailored to that function. Going through the same creation and execution stages.

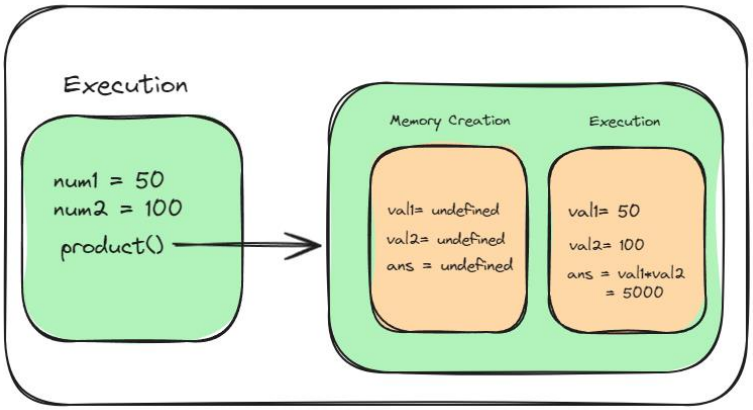
- **Stack Management:** The engine oversees a call stack in order to handle all execution contexts. The top of this stack is occupied by the active context. Whenever a function is invoked, its context is pushed onto the stack. On the other hand, when a function returns, its context is deleted from the stack.

Depiction:

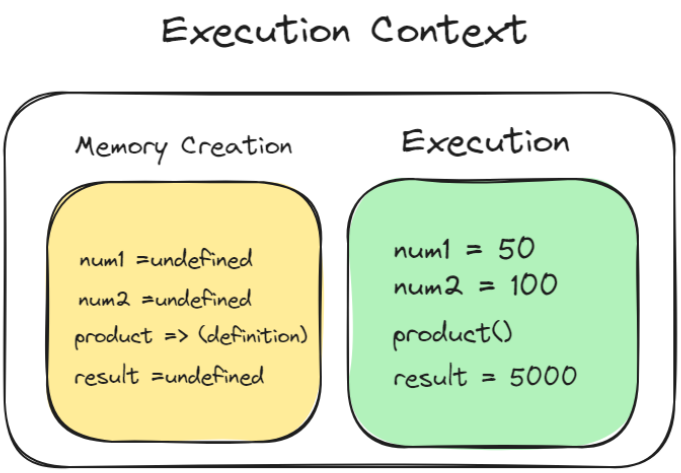
2.1



2.2



2.3



Example:

```
function sayHello() {
    var greeting = "Hello World!";
    console.log(greeting);
}
sayHello(); // Outputs: Hello World!
```

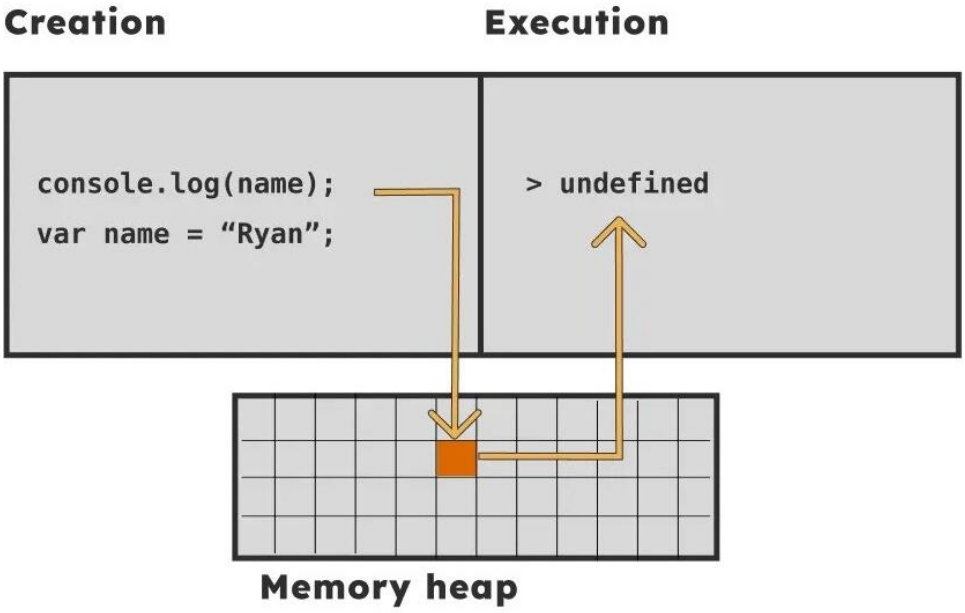
In this case, when sayHello is called, a new execution context is created for it. This is processing the function's internal variables as well as executing its code.

3. Hoisting

Declarations of variables and functions are moved to the top of their scope during the creation phase in JavaScript in a behavior known as hoisting. This happens before the code starts running:

- **Variable Hoisting:** Hoisting is when variable and function declarations are moved in JavaScript. To the top of their scope. During the creation phase before the code is executed.
- **Function Hoisting:** Function declarations can be invoked. Before they are defined in the code due to full hoisting.

Depiction:



Example:

```
console.log(num); // Outputs: undefined
var num = 6;
```

This demonstrates variable hoisting. Where the declaration (but not the initialization) is lifted to the top of their scope. This is how it is working.

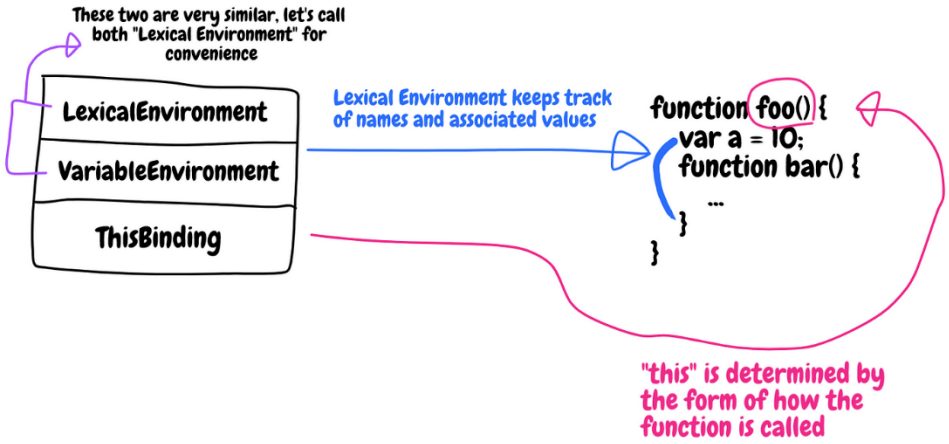
4. Closure

Closures are when functions maintain access to their original scope. No matter where they are run. This ability comes from JavaScript's closure mechanism. In which functions naturally remember the lexical environment. Where they were created:

- **Lexical Scoping:** Functions run based on the scope chain. That existed when they were defined. Not the scope chain present during their activation.

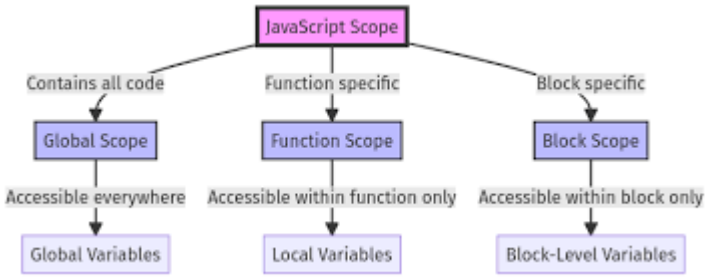
Depiction:

4.1

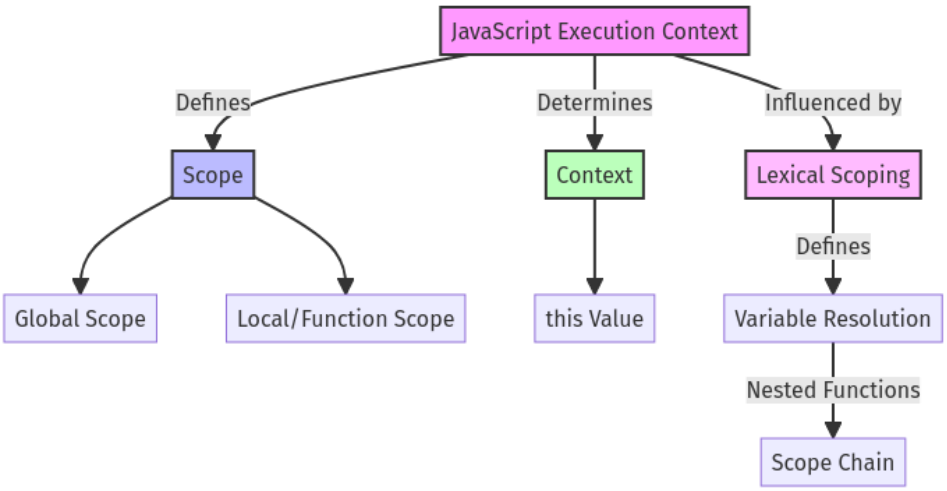


4.2

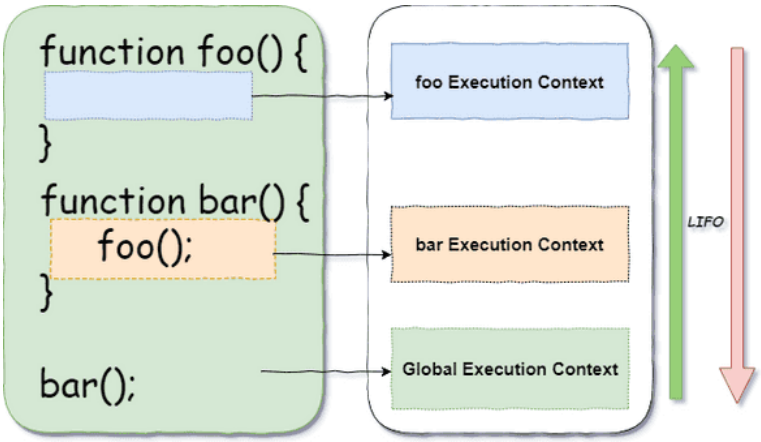
4.3



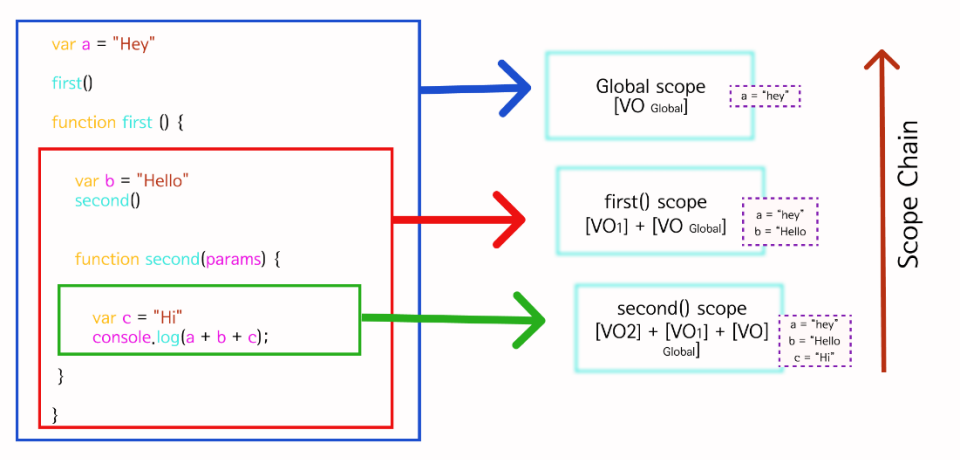
4.4


JavaScript Execution Context
Defines
Determines
Influenced by
Scope
Context
Lexical Scoping
Global Scope
Local/Function Scope
this Value
Defines
Variable Resolution
Nested Functions
Scope Chain

4.5


function foo() {
}
function bar() {
foo();
}
bar();
foo Execution Context
bar Execution Context
Global Execution Context
LIFO

4.6


var a = "Hey"
first()
function first () {
var b = "Hello"
second()
function second(params) {
var c = "Hi"
console.log(a + b + c);
}
}
Global scope
[VO Global]
a = "hey"
first() scope
[VO1] + [VO Global]
a = "hey"
b = "Hello
second() scope
[VO2] + [VO1] + [VO]
Global]
a = "hey"
b = "Hello
c = "Hi"
Scope Chain

Example:

```
function outerFunction() {
    var outer = "I'm outside!";
    function innerFunction() {
        console.log(outer);
    }
    return innerFunction;
}
var myInnerFunction = outerFunction();
myInnerFunction(); // Outputs: I'm outside!
```

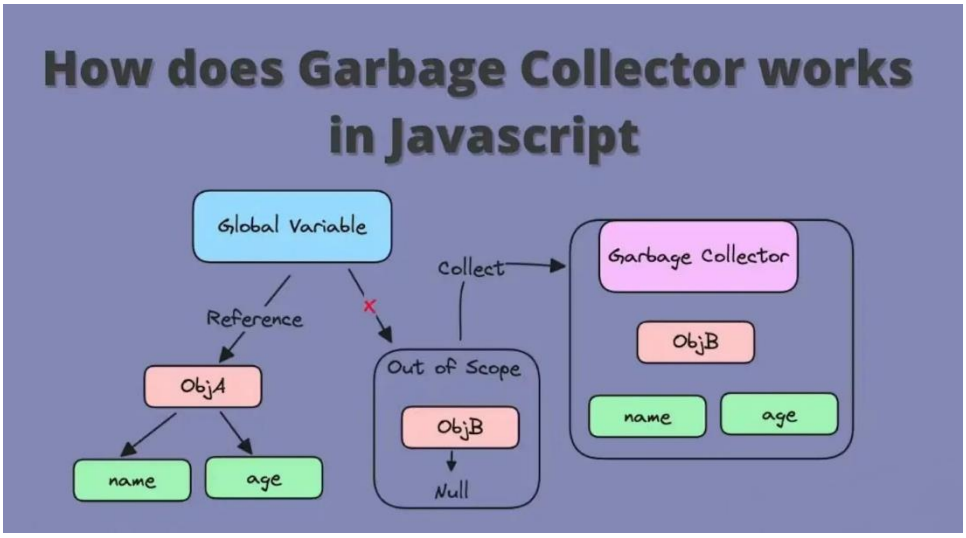
This shows a closure. Where innerFunction retains access to its lexical scope. Even after the outer function has finished its executing.

5. Garbage Collection

JavaScript handles memory automatically. This is done through by removing outdated data. This is the purging that takes place when variables and objects which go out of scope. Making them unreachable by the running code or detached from the global context.

Understanding the phases and features that are of the JavaScript execution context helps. That enables developers to create code. Which is reliable and free from mistakes.

Depiction:



Example:

```
function process() {
    var largeObject = new Array(1000).fill(new Array(1000).fill(0));
    return largeObject[0][0];
}
process();
```

Post-execution, largeObject is not accessible outside process(). Thus making it eligible for garbage collection.

References:

- MDN Web Docs: Execution Context
- JavaScript Info: The Modern JavaScript Tutorial
- You Don't Know JS: Scopes & Closures
- All about JavaScript Execution Context
- What is Hoisting in JavaScript?
- Understanding JavaScript Execution Context: Scope, Context, and Lexical Scoping
- Execution Context, Lexical Environment, and Closures in JavaScript
- Lexical Environment-the hidden part to understand Closures
- JavaScript Execution Context – How JS Works Behind The Scenes
- Garbage Collector in javascript