Working of JS



# Execution Context in JS

* Execution context is a fundamental concept in JavaScript that describes the environment in which a piece of code is executed.
* It includes variables, functions, and other elements that are necessary for the code to run.
* In other words, an execution context in JavaScript is a container that holds information about the current state of code being executed. The concept of execution context is important in understanding how JavaScript code is executed.

## Components of Execution Context

There are two important components of an execution context: the **Variable Environment** and the **Thread of Execution**.

### Variable Environment:

* 1. The Variable Environment is a fundamental component that organizes and holds all the variables, functions, and parameters accessible within a given scope. However, we will delve into the concept of scope and its significance in greater detail later on.
  2. It is created when a new function is executed and contains information about all the variables that are declared within that function and the values assigned to them.
  3. The variable environment also includes a reference to the outer environment, which is the variable environment of the parent scope.

### Thread of Execution:

* 1. The Thread of Execution is the sequence of code execution that is currently being executed.
  2. It is responsible for running the code one line at a time and keeping track of where the execution is at any given moment.
  3. When a new function is called, a new thread of execution is created, and the execution continues within that thread until the function returns.

## Phases of Execution Context

The Execution Context goes through two phases during its lifecycle:

### Creation Phase:

* 1. During the creation phase, the JavaScript engine creates a new execution context and sets up the environment for executing the code.
  2. This involves establishing a fresh variable environment, configuring the scope chain, and establishing a reference to the outer environment, which will be further explored when we cover the concept of scope chain."

### Execution Phase:

* 1. During the execution phase, the JavaScript engine executes the code line by line within the thread of execution.
  2. It uses the variable environment to look up variables and functions as needed and updates the values of variables as they are changed in the code.

## Call Stack Overview

* Before we delve into the details of execution context, it's essential to understand how a computer remembers and manages the order of execution in JavaScript. This is where the **call stack** comes into play.
* The call stack is a fundamental mechanism used by JavaScript to keep track of function calls and their corresponding execution contexts.
* It acts as a memory structure that helps the computer remember which functions are currently being executed and where to return after a function completes its execution.
* We will cover the call stack more comprehensively in the following section, exploring its inner workings and how it influences the flow of our code.

## Global vs Local Execution Context

#### Global Execution Context:

* The global execution context is the default environment in which JavaScript code is executed.
* The global execution context is created when the JavaScript program starts running and stays in memory until the program ends.
* During the creation phase, the JavaScript engine performs several steps to set up the environment for executing the code. These steps are described below:
  + 1. Defining Window Object: The engine defines the global window object, which serves as the outermost object in the environment.
    2. Creating this Variable: The this variable is created and assigned to the window object. It represents the context in which the current code is executing.
    3. Hoisting: Hoisting takes place, where variable and function declarations are moved to the top of their respective scopes. This allows you to use variables and functions before they are formally declared in the code.

We will cover the concept of hoisting in more detail in a dedicated section later on.

* + 1. Memory Allocation: After hoisting, the engine allocates memory for variables and functions, preparing them for later use during the execution phase. It's

important to note that the way variables are allocated and their initial values can vary depending on the type of declaration.

* + - 1. var Variables: When a variable declared with var is encountered during memory allocation, it is assigned the default value of undefined. This means that although the variable exists in memory, it holds the value undefined until a value is explicitly assigned to it.
      2. let and const Variables: Variables declared with let and const also go through memory allocation. However, their behavior is more nuanced and will be discussed in a later section when we cover the Temporal Dead Zone (TDZ). It's during this phase that let and const variables are assigned the

initial value of undefined within the TDZ.

#### Local Execution Context:

* A local execution context is created each time a function is called. It serves as a separate environment for the function's execution, encompassing several

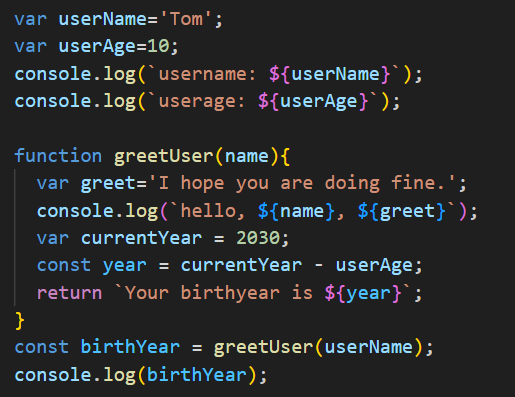
important aspects.

* Firstly, the local execution context defines the this variable. The value of this is determined based on how the function is invoked. If the function is called in the global scope or without any specific context, this will be assigned the global `window` object. However, in strict mode, this will be undefined in such cases.
* Additionally, the local execution context includes the creation of the

`**arguments`** object. This object is available within the function and contains a list of all the arguments passed to it.

* Furthermore, during the creation phase of the local execution context, memory allocation takes place similar to the one we discussed above.
* When a variable is referenced in a function, JavaScript first looks for it in the local execution context's variable environment. If the variable is not found there, it looks in the parent execution context (if any), and so on, until it reaches the global execution context.
* When a function completes its execution, its local execution context is removed from memory.

Consider the example given below to understand the workflow of Execution Context in JavaScript.



The detailed explanation of the phases of Execution Context for the given code is as follows:

* The global and local execution context for the given code snippet can be described as shown below:



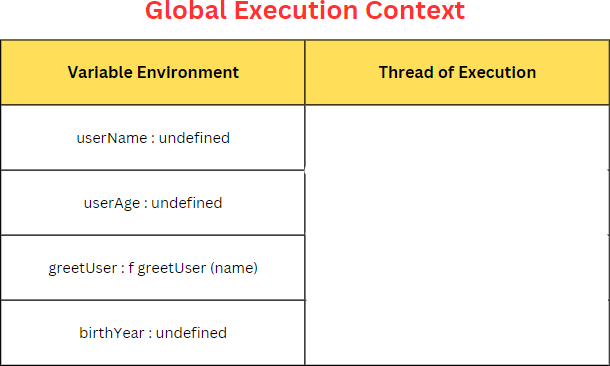
* The workflow for the execution context of this code snippet will be as follows:

#### Creation Phase:

During the creation of the global execution context, the JavaScript engine declares three variables (userName, userAge, and

birthYear) and initializes them to undefined. It also declares a function named greetUser, which is stored in memory but not executed yet.

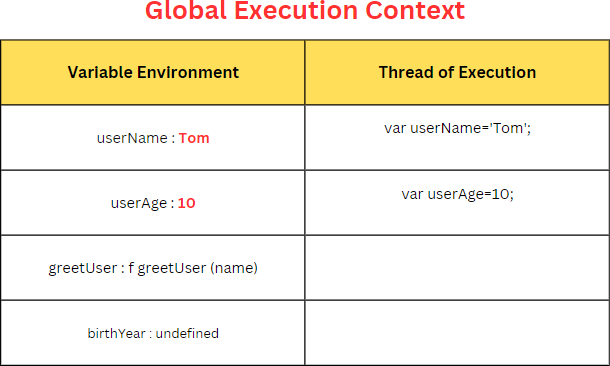
This work-flow is depicted in the figure below:



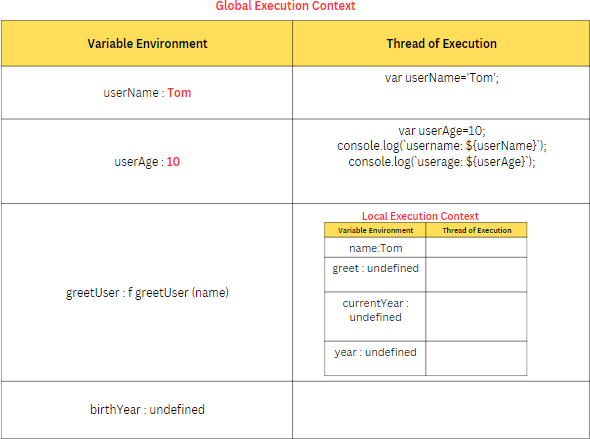
#### Execution Phase:

1. The JavaScript engine assigns 'Tom' to the userName variable and 10 to the userAge variable. It then executes console.log() twice, outputting the values of userName and userAge to the console as

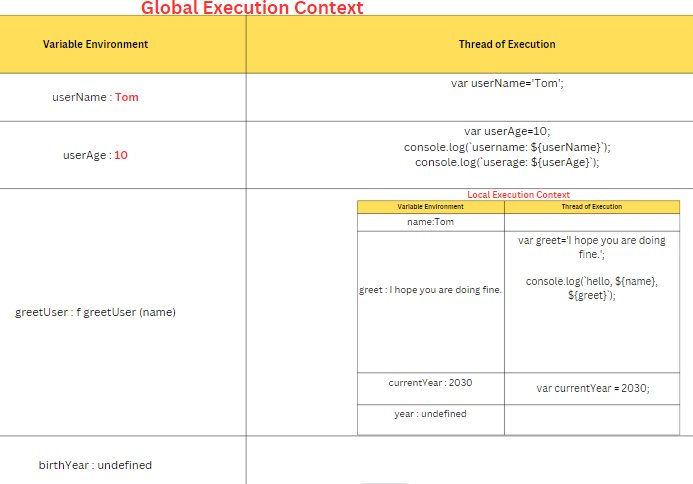
"username: Tom" and "userAge: 10" as shown below:



1. When a function is invoked, a new Execution Context is built all together to carry out the same procedures for that function call/invoke. The JavaScript engine creates a new execution context (as shown below) for the greetUser function following the same procedure discussed above.



* The further workflow of the Local Execution Context is described in the following figure, where the variables within the greetUser function are assigned values.

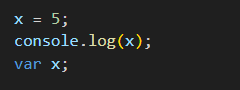


* Finally, The greetUser function returns a string containing the calculated birth year and the local execution context for the greetUser function is removed, and control is returned to the global execution contex

# Hoisting in JS

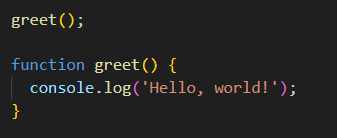
Hoisting is a behavior in JavaScript where variable and function declarations are relocated to the beginning of their code blocks during the compilation phase, no matter where they are actually written in the code. This means that they can be accessed before they are declared.

* + For example, the following code will work without any errors:



This is because the `var x;` declaration is hoisted to the top of its code block, which in this case is the global block, and is executed before the `x = 5;` assignment.

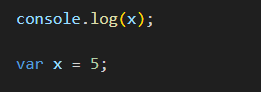
* + Function declarations are also hoisted in a similar way. For example:



This code will also work without errors because the greet() function declaration is hoisted to the top of its code block (in this case, the global block) before it is called.

* + However, it's important to note that **only the declarations themselves are hoisted**, not their assignments.

For example:



In this code, the var x; declaration is hoisted to the top of its code block, but the assignment x = 5; is not. So when console.log(x); is executed, x is still undefined.

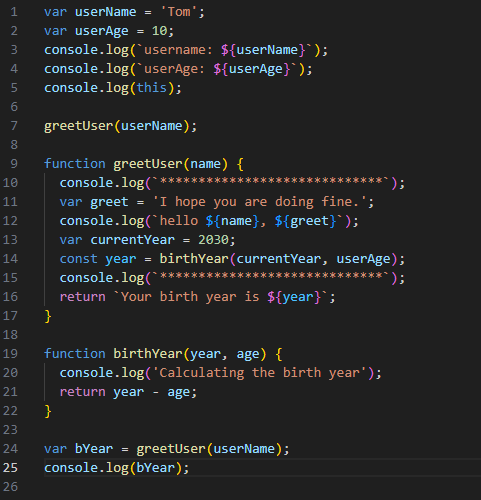
## Understanding Hoisting in JavaScript: Variables, Functions, and Declarations:

* + When using the var keyword, variable declarations are hoisted to the top of their code blocks, whether it's the global scope or a function scope. This allows variables to be accessed before they are formally declared in the code.
  + Similarly, function declarations are hoisted, enabling functions to be called before they appear in the code.
  + However, it's important to note that function expressions, where functions are assigned to variables, are not hoisted. Only the function declarations themselves are hoisted.
  + Furthermore, block-scoped variables declared with let and const do not experience hoisting behavior. They are not moved to the top of their code blocks and remain in their lexical position, ensuring that they are not accessible before their actual declaration in the code.

Generally, it's best practice to always declare variables and functions before using them to avoid unexpected behavior due to hoisting.

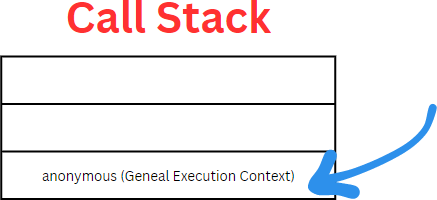
# Call Stack

* + In JavaScript, the call stack is a data structure that tracks the execution of functions during the runtime of a program.
  + Every time a function is called, a new frame is created on top of the call stack to hold information about the function call, such as its arguments and local variables.
  + The call stack operates on a "last in, first out" (LIFO) basis, meaning that the most recent function is the first to be completed and removed from the stack.
  + The call stack is essential for understanding how JavaScript executes functions, and it plays a crucial role in identifying and debugging errors that occur during program execution.



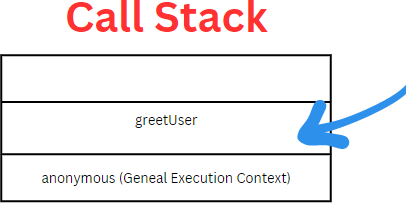
Here is how the call stack builds up for the given code:

1. The global execution context (here referred to as General Execution Context in the figures) is created in the call stack, as shown in figure-3(a).



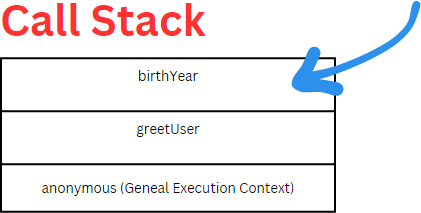
**figure- 3(a)**

1. The greetUser() function is called with userName as an argument, and a new execution context is created as shown in figure-3(b)



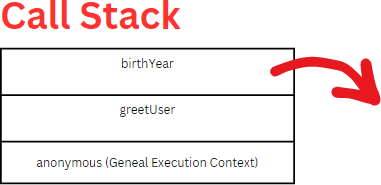
**figure-3(b)**

1. The birthYear() function is called with currentYear and userAge as arguments, and a new execution context is created on top of the current execution context for the greetUser() function as depicted in the figure-3(c)



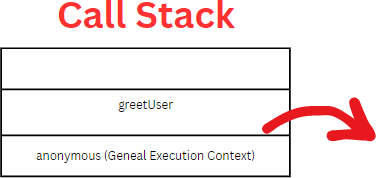
**figure-3(c)**

1. Once the operations of the birthYear function have been completed, it will be popped out of the stack as shown in figure-3(d).



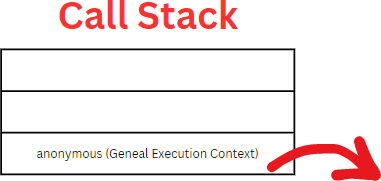
**figure-3 (d)**

1. Similarly, the greetUser function is popped out after its operation has been performed. This operation is depicted in figure-3(e).



**figure-3 (e)**

1. Finally, in the end, the Global Execution Context (General Execution Context) is popped out of the Call Stack as depicted in figure-3(f).



**figure-3 (f)**

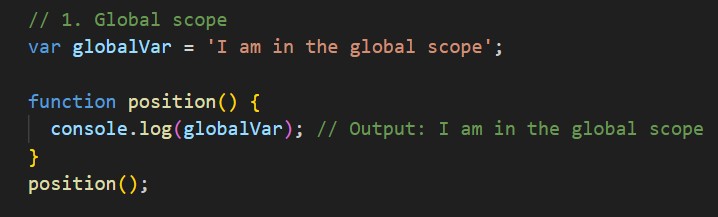
# Scope in JS

* + In JavaScript, scope refers to the accessibility of variables and functions in your code. Understanding scope is crucial for writing clean and efficient code.
  + There are three types of scopes: global scope, functional/local scope, and block scope.

### Global Scope:

Variables declared outside of any function or block are in the global scope. They can be accessed from anywhere in the code, including inside functions or blocks.

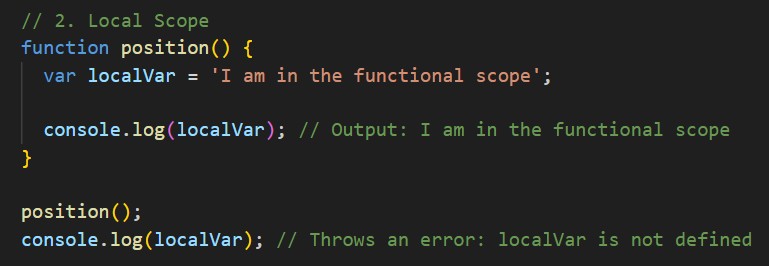
**For example:**



### Functional/Local Scope:

Variables declared inside a function or block are in the functional/local scope. They can only be accessed from within that function or block.

**For example:**



### Block Scope:

Variables declared using let or const inside a block (e.g., inside a for loop or if

statement) are in the block scope. They can only be accessed from within that block.

**For example:**



**Difference between let, var, and const**

In JavaScript, let, var, and const are used to declare variables, but they differ in terms of scoping and mutability.

#### let:

* let is used to declare block-scoped variables.
* It allows you to declare a variable inside a block and use it only within that block.
* let variables can be re-assigned but not re-declared within the same scope.
* let is more strict than var and helps avoid bugs caused by variable hoisting.

#### var:

* var is used to declare function-scoped variables.
* It can be declared and re-declared multiple times within the same scope.
* var declarations are hoisted to the top of the function or global scope, which means that they are processed before any code is executed.
* Because of hoisting, var can lead to bugs in code if not used properly.
* However, it's important to note that var has two scopes: global scope and functional scope. Variables declared with var in the global scope are accessible throughout the entire program, while variables declared with var inside a function are only accessible within that function.

#### const:

* const is used to declare read-only variables.
* It can be used to declare a variable once and cannot be re-assigned within the same scope.
* const variables are also block-scoped.
* const is useful for declaring constants that should not be changed throughout the program.

# Scope Chaining

## Lexical environment

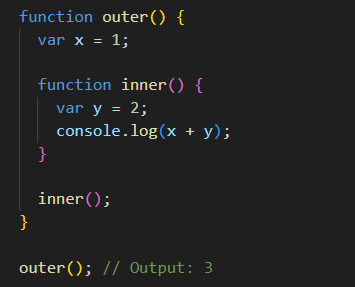
* Lexical environment is a fundamental concept in JavaScript that refers to the specific context in which code is executed.
* It encompasses variables, functions, and objects that are accessible and in scope at a particular point during the execution of code.
* A fresh lexical environment is generated whenever a function is called or

invoked in JavaScript. This lexical environment encompasses all the variables and functions that are in scope and can be accessed within that particular function call.

## Scope chaining

* Scope chaining, also known as lexical scoping, is a mechanism in JavaScript that allows a function to access variables from its outer (enclosing) lexical environment as well as from the global scope.
* This means that functions can access variables defined in their parent functions, grandparent functions, and so on, all the way up to the global scope.

Here's an example to illustrate how scope chaining works in JavaScript:



* In this example, inner() is nested inside outer(), so it has access to x, which is defined in the lexical environment of outer().
* When outer() is called, a new lexical environment is created that contains the variable x, and when inner() is called, a new lexical environment is created that contains both x and y.
* Therefore, inner() can access x from its outer lexical environment and y

from its own lexical environment, and the result of x + y is 3.

It's important to note that **scope chaining only works in one direction, from inner to outer, and not the other way around.** That means variables defined in an inner scope cannot be accessed from an outer scope.