# 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**.

#### 1) Variable Environment:

a) 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.

b) 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.

c) The variable environment also includes a reference to the outer environment, which is the variable environment of the parent scope.

#### 2) Thread of Execution:

a) The Thread of Execution is the sequence of code execution that is currently being executed.
b) It is responsible for running the code one line at a time and keeping track of where the execution is at any given moment.
c) 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:

### 1. Creation Phase:

a. During the creation phase, the JavaScript engine creates a new execution context and sets up the environment for executing the code.
b. 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."

### 2. Execution Phase:

a. During the execution phase, the JavaScript engine executes the code line by line within the thread of execution.
b. 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.

4. 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.

a. `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.

b. `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.

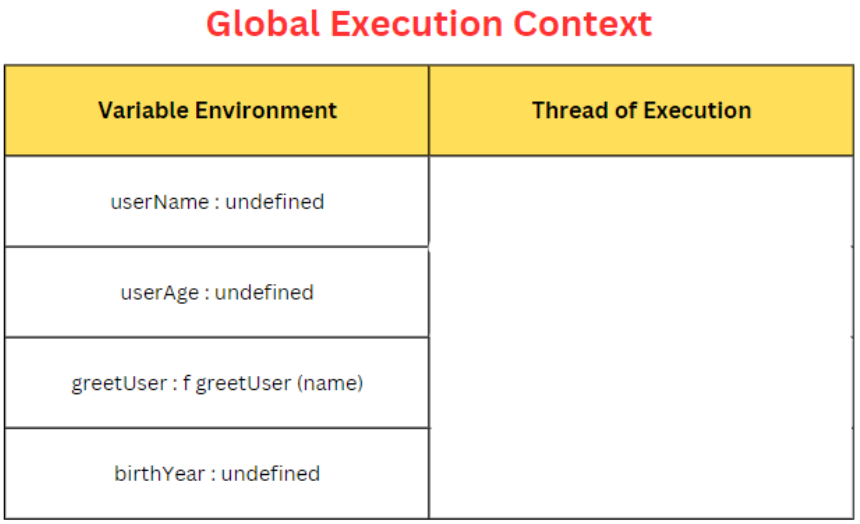
```
var userName='Tom';
var userAge=10;
console.log(`username: ${userName}`);
console.log(`userage: ${userAge}`);

function greetUser(name){
  var greet='I hope you are doing fine.';
  console.log(`hello, ${name}, ${greet}`);
  var currentYear = 2030;
  const year = currentYear - userAge;
  return `Your birthyear is ${year}`;
}
const birthYear = greetUser(userName);
console.log(birthYear);
```

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:

1. **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:




2. **Execution Phase:**
    **a.** 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:

## Global Execution Context

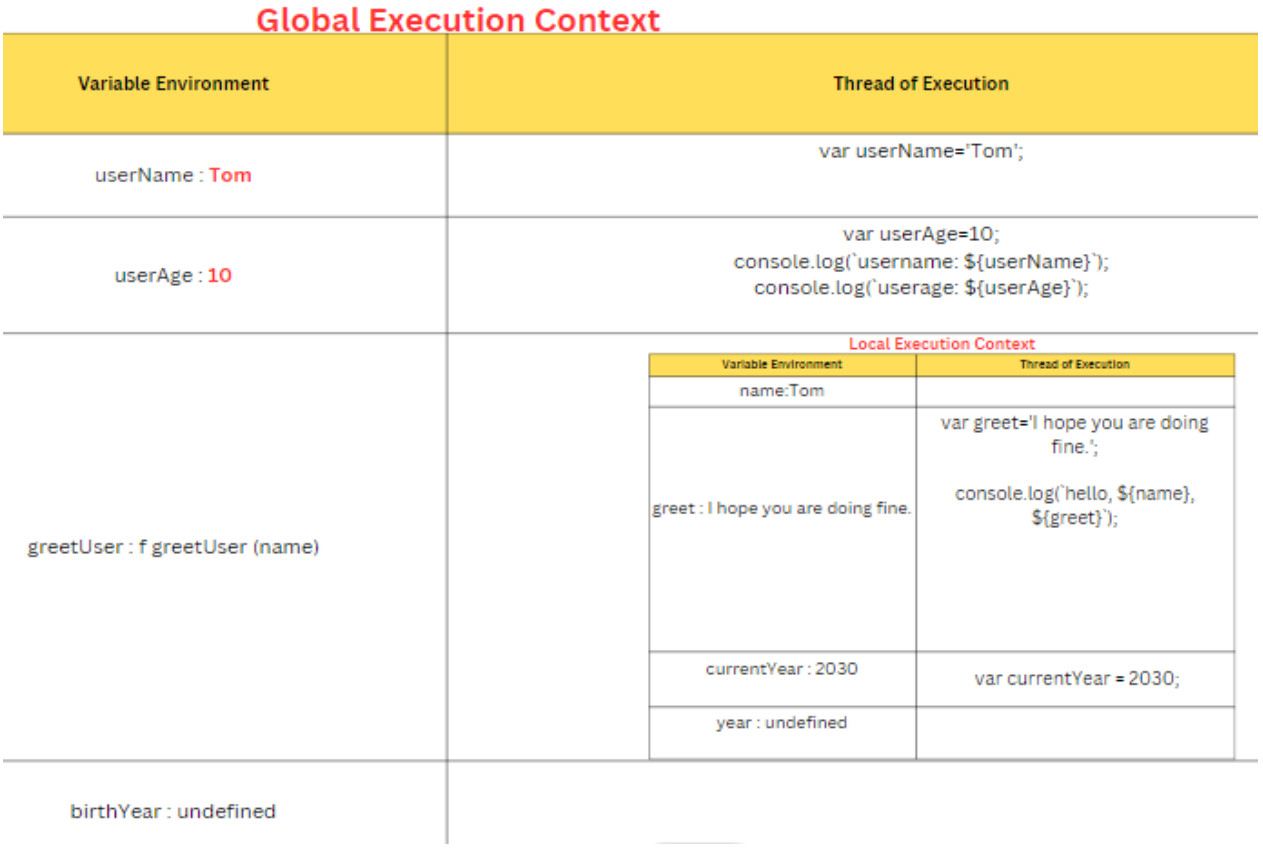
| Variable Environment | Thread of Execution |
|---|---|
| userName : Tom | var userName='Tom'; |
| userAge : 10 | var userAge=10; |
| greetUser : f greetUser (name) | |
| birthYear : undefined | |

**b.** 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.

Global Execution Context

<table>
<tr><th>Variable Environment</th><th>Thread of Execution</th></tr>
<tr><td>userName : Tom</td><td>var userName='Tom';</td></tr>
<tr><td>userAge : 10</td><td>var userAge=10;<br>console.log(`username: ${userName}`);<br>console.log(`userage: ${userAge}`);</td></tr>
<tr><td>greetUser : f greetUser (name)</td><td>Local Execution Context
<table>
<tr><th>Variable Environment</th><th>Thread of Execution</th></tr>
<tr><td>name:Tom</td><td></td></tr>
<tr><td>greet : undefined</td><td></td></tr>
<tr><td>currentYear : undefined</td><td></td></tr>
<tr><td>year : undefined</td><td></td></tr>
</table></td></tr>
<tr><td>birthYear : undefined</td><td></td></tr>
</table>

- 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 context.

## Execution Context in Dev Tools

To access and inspect the execution context in Google Chrome Dev Tools specifically, you can use the following steps:

1. Open Google Chrome browser.
2. Navigate to the webpage or web application where the JavaScript code is running.
3. Right-click on the page and select "**Inspect**" from the context menu. Alternatively, you can use the keyboard shortcut:
    a. **Windows:** Ctrl + Shift + I

b. **Mac:** Command + Option + I

Once the Dev Tools panel is open, follow these steps to access the execution context:

I. In the Dev Tools panel, locate and click on the "**Sources**" tab.
II. In the left-hand sidebar, expand the file or snippet containing the JavaScript code you want to inspect.
III. Set breakpoints at desired locations by clicking on the line number next to the code or using shortcut F9.
IV. Refresh the webpage or trigger the execution of the JavaScript code.
V. The execution will pause at the breakpoints, allowing you to inspect the execution context.
VI. Use the "**Scope**" section in the right-hand sidebar to explore the variables and their values within the execution context.
VII. You can also use the "**Call Stack**" section to see the sequence of function calls that led to the current execution context.

# 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:

```
x = 5;
console.log(x);
var x;
```

  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:

```
greet();

function greet() {
  console.log('Hello, world!');
}
```

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:

```
console.log(x);

var x = 5;
```

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.

```
var userName = 'Tom';
var userAge = 10;
console.log(`username: ${userName}`);
console.log(`userAge: ${userAge}`);
console.log(this);

greetUser(userName);

function greetUser(name) {
  console.log(`*****************************`);
  var greet = 'I hope you are doing fine.';
  console.log(`hello ${name}, ${greet}`);
  var currentYear = 2030;
  const year = birthYear(currentYear, userAge);
  console.log(`*****************************`);
  return `Your birth year is ${year}`;
}

function birthYear(year, age) {
  console.log('Calculating the birth year');
  return year - age;
}

var bYear = greetUser(userName);
console.log(bYear);

```

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).

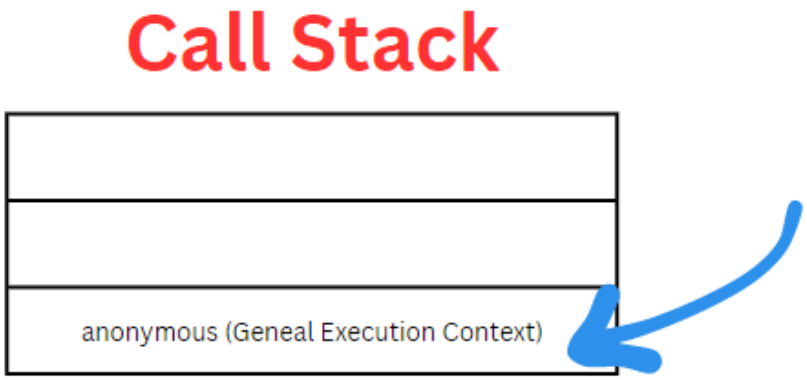



**figure- 3(a)**

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

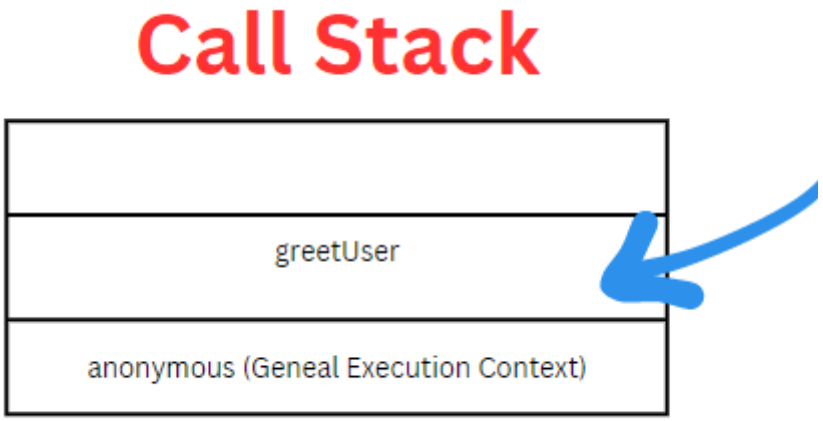



**figure-3(b)**

3. 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)

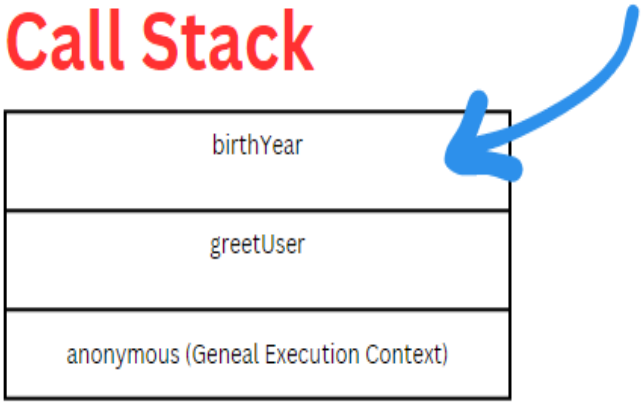



**figure-3(c)**

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

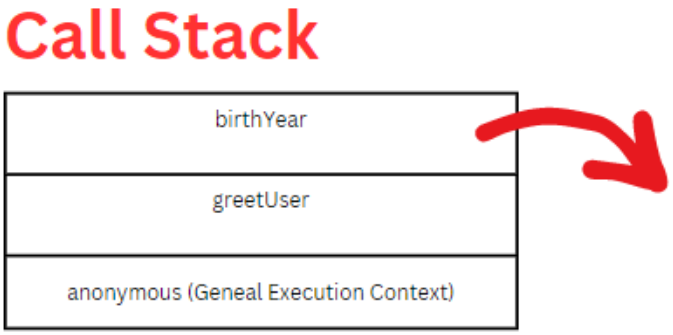



**figure-3 (d)**

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

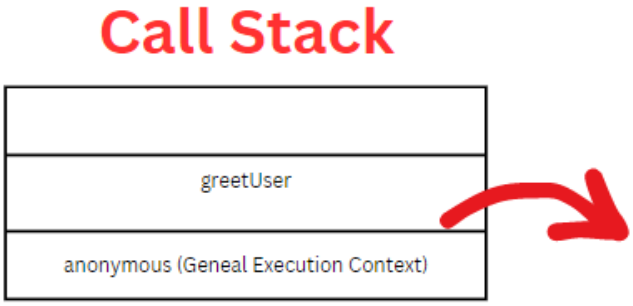



**figure-3 (e)**

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

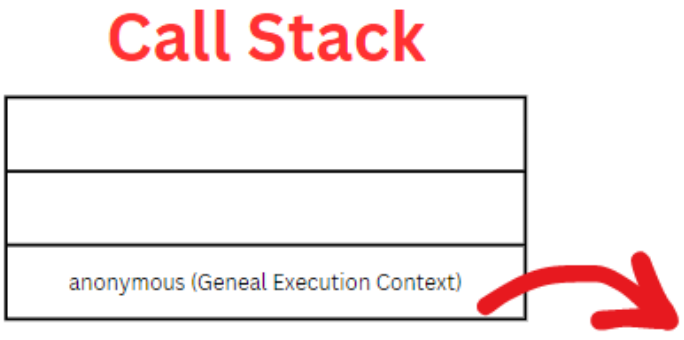



**figure-3 (f)**

# 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.

## 1. 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:**

```
// 1. Global scope
var globalVar = 'I am in the global scope';

function position() {
  console.log(globalVar); // Output: I am in the global scope
}
position();
```

## 2. 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:**

```
// 2. Local Scope
function position() {
  var localVar = 'I am in the functional scope';

  console.log(localVar); // Output: I am in the functional scope
}

position();
console.log(localVar); // Throws an error: localVar is not defined
```

### 3. 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:**

```
// 3. block scope
function foo() {
  if (true) {
    let blockVar = 'I am in the block scope';
    console.log(blockVar); // Output: I am in the block scope
  }

  console.log(blockVar); // Throws an error: blockVar is not defined
}
```

## 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.

1. **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.

2. **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.

3. **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

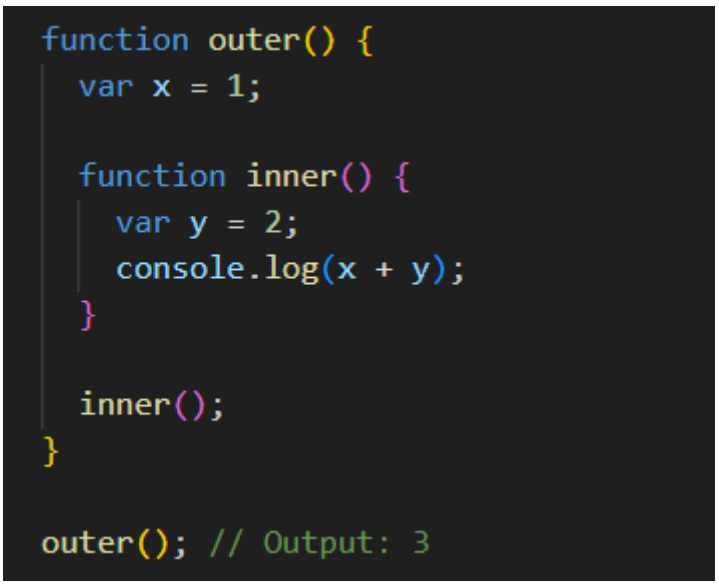
## Lexical environment

- 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:



```
function outer() {
  var x = 1;

  function inner() {
    var y = 2;
    console.log(x + y);
  }

  inner();
}

outer(); // Output: 3
```

- 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.

## Undefined vs Not Defined

In JavaScript, undefined and not-defined are two different concepts that often confuse developers who are new to the language. Understanding the difference between them is crucial in writing bug-free and robust code.

- **`undefined`** is a value that indicates the absence of a value. It is a primitive type and is assigned to a variable when it is declared but not initialized with a value or when a function does not return anything. For example:

```
let x; // x is declared but not initialized, so its value is undefined
console.log(x); // output: undefined

function func() {
  // no return statement, so the function returns undefined
}
console.log(func()); // output: undefined
```

In this example, `x` and `func()` are defined variables but have not been initialized with a value. Therefore, their value is undefined.

- **not-defined** is a state of a variable that has not been declared at all. If you try to access such a variable, JavaScript throws a `ReferenceError` exception. For example:

```
console.log(y); // ReferenceError: y is not defined
```

In this example, `y` is not defined anywhere in the code. Therefore, JavaScript throws a `ReferenceError` exception.

# Strict mode

- Strict mode in JavaScript is a feature introduced in ECMAScript 5 (ES5) that allows developers to opt into a stricter set of rules and best practices for writing JavaScript code.
- When strict mode is enabled, the JavaScript interpreter enforces a more stringent set of rules, catches common mistakes, and disables certain silent errors.

Here are some key points about strict mode:

1. **Activation:** Strict mode can be enabled for an entire script or a specific function. To enable strict mode for an entire script, add the following line at the beginning of your script:

```
'use strict';
```

To enable strict mode for a specific function, add the same line at the beginning of the function's body.

2. **Benefits:** Strict mode helps developers write more reliable and maintainable code by addressing some of the quirks and pitfalls of JavaScript. It prevents the use of undeclared variables, eliminates the automatic creation of global variables, prohibits the use of duplicate function parameter names, and disallows certain unsafe language features.
3. **Variable Declarations:** In strict mode, all variables must be explicitly declared with var, let, or const keywords. Implicitly creating global variables by omitting the var keyword is disallowed. This catches accidental global variable declarations, reducing the risk of naming conflicts.
4. **Undeclared Variables:** Accessing undeclared variables in strict mode throws a ReferenceError. In non-strict mode, accessing an undeclared variable creates an implicit global variable, which can lead to subtle bugs and unintended consequences.

Enabling strict mode is highly recommended for modern JavaScript development as it promotes better coding practices and helps catch potential errors early on. By adhering to strict mode guidelines, developers can write more robust and predictable code.

## Temporal Dead Zone

- In JavaScript, the Temporal Dead Zone (TDZ) is a behavior that occurs during the variable creation phase of the execution context.
- It refers to the period between the creation of a variable and its initialization, during which the variable cannot be accessed.
- Attempting to access a variable during the TDZ will result in a `ReferenceError`.

Let's take a look at an example to illustrate the concept of TDZ:

```
console.log(a); // ReferenceError: a is not defined
let a = 10;
```

In this example, we try to access the variable `'a'` before it is declared. This results in a `ReferenceError` because the variable is in the TDZ and cannot be accessed until it has been declared.

- It occurs only for variables declared with `let` and `const` keywords and not for variables declared with `var`. This is because `var` is function-scoped and hoisted to the top of the function, making it accessible throughout the function, even before it is declared.

  ```
  console.log(b); // undefined
  var b = 20;
  ```

  In this example, we try to access the variable `'b'` before it is declared, but we get `undefined` instead of a `ReferenceError`. This is because `var` variables are hoisted to the top of their function and initialized with undefined by default.

- To avoid TDZ errors, it's best practice to declare variables at the beginning of their scope, before they are accessed. This will ensure that they are not in the TDZ and can be accessed without throwing a `ReferenceError`.

# Closures in JS

- In JavaScript, a closure is created when a function accesses variables outside of its immediate lexical scope.
- The closure retains a reference to the environment in which it was created, allowing the function to access and manipulate variables in that environment, even after the outer function has returned.
    a. Here's an example of a closure in JavaScript:

```
function outer() {
  let count = 0;

  function inner() {
    count++;
    console.log(count);
  }

  return inner;
}

const counter = outer();
counter(); // 1
counter(); // 2
counter(); // 3
```

- In this example, the `outer` function creates a variable `count` and a nested function `inner` that increments the `count` variable and logs it to the console. The `outer` function then returns the `inner` function.
- We then assign the returned `inner` function to the counter variable. We can now call the counter function multiple times, and each time it will increment and log the `count` variable.
- The important thing to note here is that the `inner` function retains a reference to the `count` variable in the environment where it was created (inside the `outer` function). This is possible because JavaScript functions are **closures**, and they **maintain a reference to the environment in which they were created.**

**b.** Here's another example to illustrate the concept of closures:

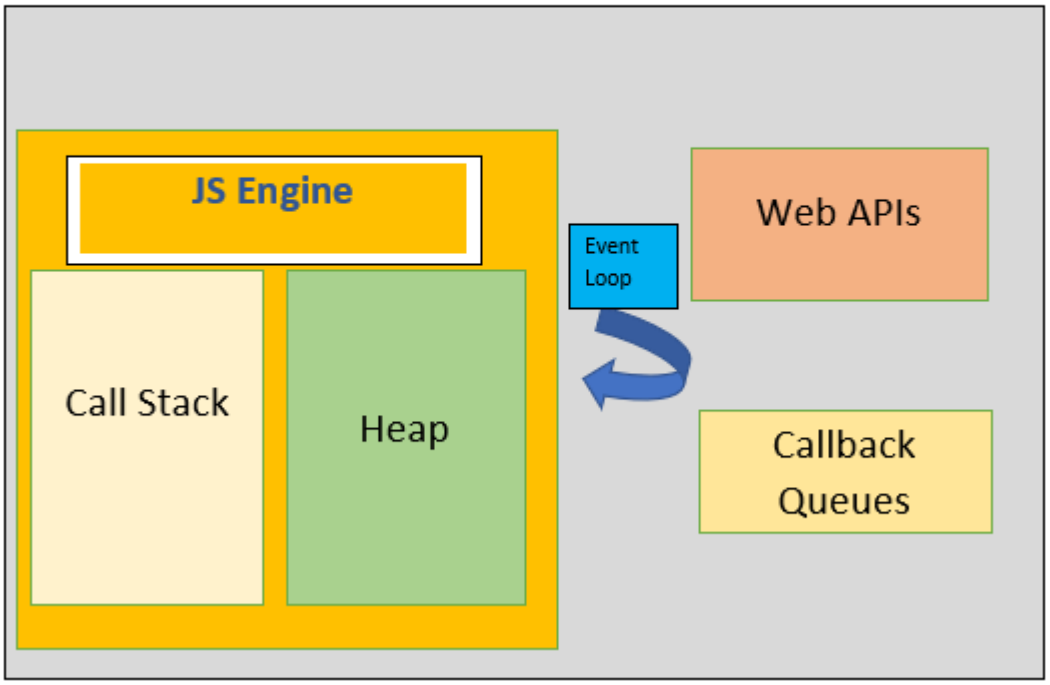
```
function createCounter() {
  let count = 0;

  function increment() {
    count++;
    console.log(count);
  }

  function decrement() {
    count--;
    console.log(count);
  }

  return { increment, decrement };
}

const counter = createCounter();
counter.increment(); // 1
counter.increment(); // 2
counter.decrement(); // 1
```

- In this example, the `createCounter` function returns an object with two methods: `increment` and `decrement`. Both methods have access to the `count` variable, which is created in the `createCounter` function's environment.
- We then assign the returned object to the `counter` variable, and we can call the `increment` and `decrement` methods on it. Each time we call a method, it will update and log the `count` variable.

## JavaScript Runtime Environment

- JavaScript runtime environment (JRE) is a term used to describe the environment in which JavaScript code is executed. It includes the JavaScript engine, call stack, heap, web APIs, and callback queues.

# JavaScript Runtime Environment (JRE)




- The JavaScript engine is the component responsible for interpreting and executing JavaScript code. Popular engines include Google's V8 engine used in Chrome and Node.js, Mozilla's SpiderMonkey, and Apple's JavaScriptCore.
- The heap is the memory space the JavaScript engine uses to store objects and values created by the code.
- Web APIs are interfaces provided by the browser environment that allows JavaScript to interact with browser features. These APIs include the `alert()`, `confirm()`, `prompt()`, and `setTimeout()` and `setInterval()` methods.
- Callback queues and the event loop are used to manage asynchronous code execution in JavaScript. We will be covering it in the upcoming lectures.

The JavaScript runtime environment is the foundation for executing JavaScript code in the browser environment. It provides a set of tools and interfaces that allow developers to create complex and interactive web applications.

# Summarizing it

Let's summarize what we have learned in this Lecture:

- Execution Context in JS
- Hoisting in JS
- Call Stack
- Scope and Scope Chaining
- Strict modes and Temporal Dead Zone
- Closures in JS
- JavaScript Runtime Environment

# References

- Closures in JS: **Link**