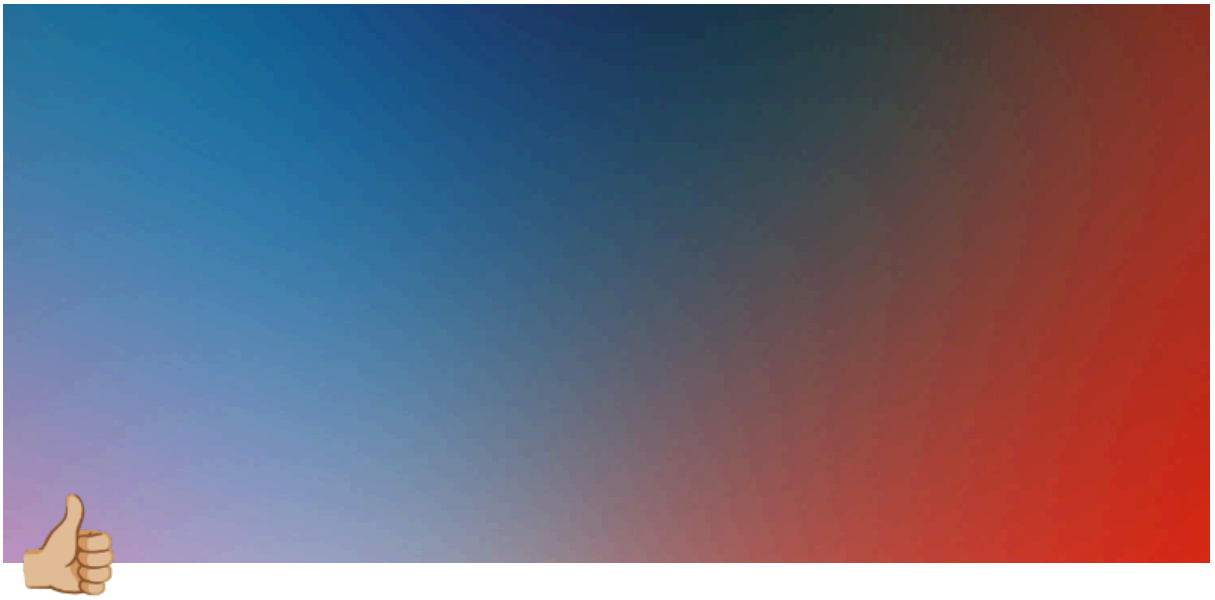


Namasthe JavaScript - Season 1

This Notes is taken from Namasthe JS Season 1 course from Akshay Saini - Youtube

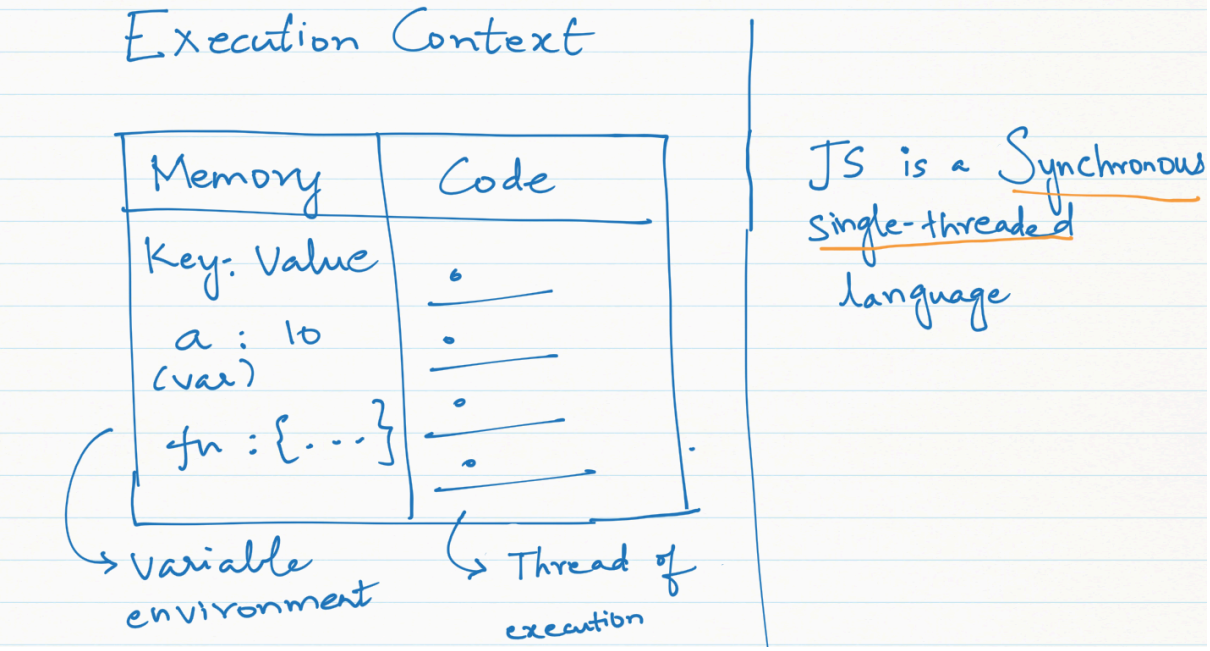
How JS Works?

1. Everything in JS happens inside the “Execution Context”.
2. There are two components of Execution Context as mentioned below:

- Memory component [variable environment] → This is the place where all variables and functions are stored as key value pairs. ex: {key: value || n:2;}.
- Code Component [Thread of execution] → This is the place where code is executed one line at a time.

1. ***JavaScript is a synchronous single-threaded language.**
2. Single Threaded means JavaScript can execute one command at a time.

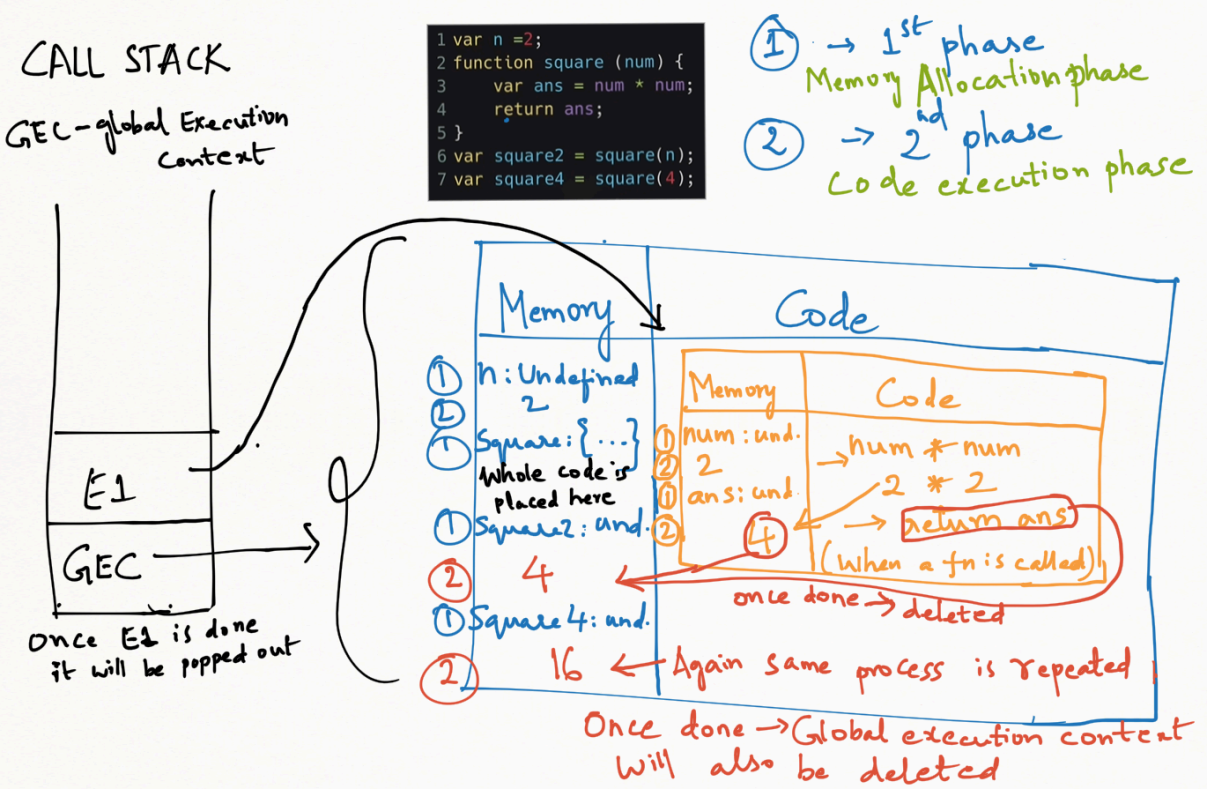
3. Synchronous Single-Threaded means JavaScript can execute one command at a time in a specific order. (One command should complete to execute the other).



How JS Code is executed?

1. When JavaScript code is executed, Execution Context is created and it is called Global Execution Context.
2. JavaScript program is executed in **TWO PHASES** inside Execution Context:
   a. MEMORY ALLOCATION PHASE - JavaScript program goes throughout the program and allocate memory of Variables and Functions declared in program.
   b. CODE EXECUTION PHASE - JavaScript program now goes throughout the code line by line and execute the code.
3. A Function is invoked when it is called and it acts as another MINI PROGRAM and creates its own Execution Context.
4. **return** keyword returns the control back to the PREVIOUS Execution-Context where the Function is called and Execution Context of the Function is DELETED.

5. **CALL STACK** maintains the ORDER of execution of Execution Contexts. It CREATES Execution Context whenever a Program starts or a Function is invoked and it pops out the Execution Context when a Function or Program ENDS.
6. Other names of CALL STATCK - Execution Context Stack, Program Stack, Control Stack, Runtime Stack, Machine Stack

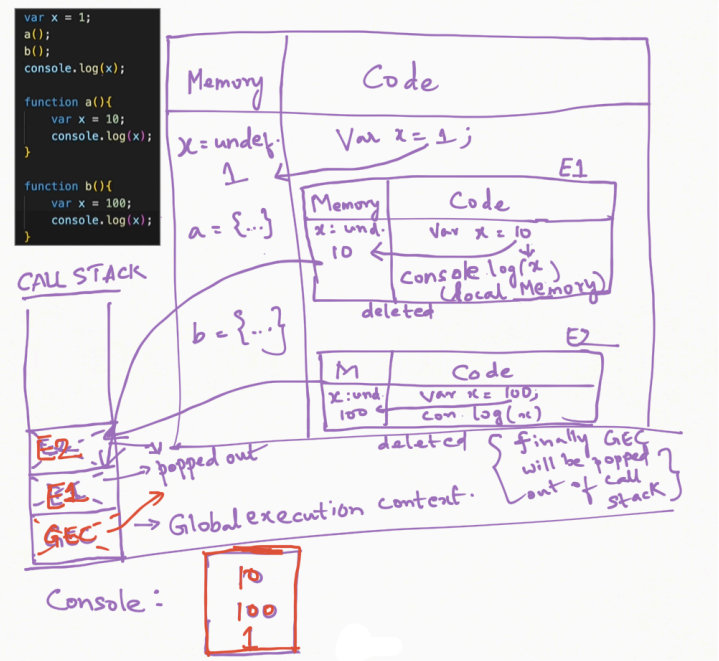


Hoisting in JS

1. Hoisting in JavaScript is a process in which all the Variables, Functions and Class defination are declared BEFORE execution of the code.
2. Variables are initialised to UNDEFINED when they are declared and Function defination is stored AS IT IS..
3. JavaScript hoists variable declarations and function definitions to the top of their scope. This means that when the code runs, these elements are already declared in memory, even if they haven't been assigned values yet..

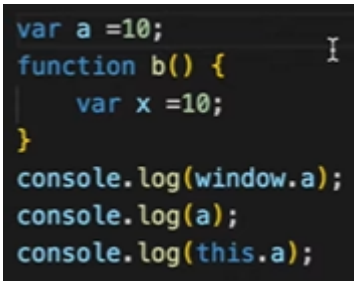
4. They are declared in Memory Allocation Phase in the Memory Component of Execution Context, so we can use them even BEFORE even code starts executing.
5. UNDEFINED means Variable has been declared but value is not ASSIGNED but NOT DEFINED means Variables is NOT DECLARED at all.
6. When we assign Variable to a Function defination, we CAN NOT call this Variable as Function BEFORE declaration as it will behave as Variable with UNDEFINED value.
   Example: var square2 in above image.
7. Hoisting will work with a variable ONLY when it is declared using the var keyword, but will not work when a variable is declared using the let keyword, which is now the standard of declaring variables in JavaScript as of the year 2024.

How Function in JS Works?



Shortest JS Program (window and this)

1. Shortest Program in JS: Empty file. Still, browsers make global EC and global space along with Window object.
2. Global Space: Anything that is not in a function, is in the global space.
3. Variables present in a global space can be accessed by a "window" object. (like window.a or even this.a)
4. In global space, (this === window) object.



```
var a =10;
function b() {
    var x =10;
}
console.log(window.a);
console.log(a);
console.log(this.a);
```

Undefined vs Not defined

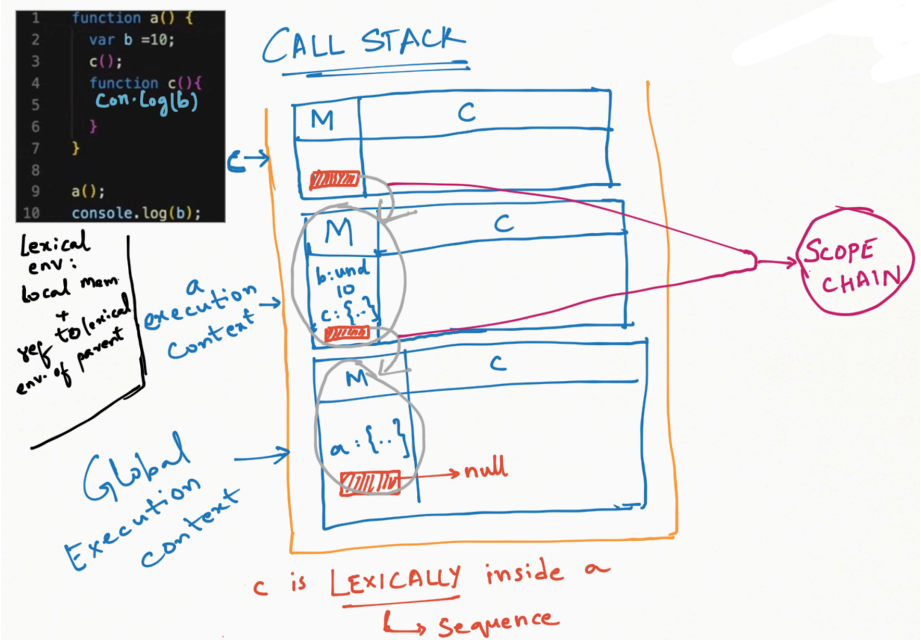
1. Undefined is a Special Placeholder which is used to reserve memory for the variables in the memory creation phase. Even before a single line of code is executed, JS engine assigns undefined to the variables. It will stay as a placeholder until a value is assigned to a variable.
2. undefined !== not defined (Not defined means Variable is not even declared and we are trying to access the value).
3. JS - weakly typed / loosely typed language which means it does not attaches its variables to specific data types like in C++ and Java.
4. Never ever assign a variable to undefined. Its a bad practice.

```
var a;
console.log(a);
a=10;
```

```
console.log(a);
a="hello JS";
console.log(a);
```

Scope Chain - Scope and Lexical Environment

1. Scope of a variable is directly dependent on the lexical environment.
2. Whenever an execution context is created, a lexical environment is created. Lexical environment is the local memory along with the lexical environment of its parent. Lexical as a term means in hierarchy or in sequence.
3. Having the reference of parent's lexical environment means, the child or the local function can access all the variables and functions defined in the memory space of its lexical parent.
4. The JS engine first searches for a variable in the current local memory space, if its not found here it searches for the variable in the lexical environment of its parent, and if its still not found, then it searches that variable in the subsequent lexical environments, and the sequence goes on until the variable is found in some lexical environment or the lexical environment becomes NULL (Global Execution Context).
5. Global execution context holds reference to null.
6. The mechanism of searching variables in the subsequent lexical environments is known as Scope Chain. If a variable is not found anywhere, then we say that the variable is not present in the scope chain.
7. Lexical environment = Execution Contexts's Local Memory + Reference to Lexical Environment of its parent.



let and const variable in JS (Temporal Dead Zone)

1. let and const are hoisted but its memory is allocated at other place than window (Global Execution Context) which cannot be accessed before initialisation.
2. Temporal Dead Zone (TDZ) exists until variable is declared and assigned a value. (let and const variables only)
3. window.variable OR this.variable will not give value of variable defined using let or const.
4. We cannot redeclare the same variable with let/const(even with using var the second time).
5. const variable declaration and initialisation must be done on the same line.
6. let variable can be declared and initialised separately.
7. Strictness: const>let>var
8. There are three types of error:

- referenceError - occurs when trying to access a variable which is not present in local memory.
- typeError - occurs when we change type of variable that is not supposed to be changed. (Like changing const to var after a variable is already declared as const).
- syntaxError - occurs when proper syntax (way of writing a statement) is not used. (Like declaring a same let variable again).

9. Use const wherever possible followed by let, Use var as little as possible (only if you have to). It helps avoid error.
10. Initialising variables at the beginning of the code is good practice which helps in shrinking TDZ to zero.

BLOCK SCOPE and Shadowing in JS

1. Block is also called as compound statement
2. { Block }
3. Block: It is used to combine multiple statement into one statement so that we can use it at those places where javascript expects to have single statement.
4. Scope: scope of a variable or a function is the place where these are accessible.
5. let & const are hoisted in a block scope. var is in global scope.
6. Block scope: The variables and function present within the scope of a block section and block follows the lexical scope chain pattern while accessing the variable.
7. Shadowing: Providing same name to the variable as of those variable which are present in outer scope.
8. Shadowing happens in functions also.
9. Examples:

```
// { } empty block is also valid code in JS
var a = 100;
{
  var a = 10;
  var b = 20;
  var c = 30;
  console.log(a); // 10
  console.log(b); // 20
  console.log(c); // 30
}

console.log(a); // 10 & if var a = 100 is not declared then also 10
// a = 10 modifies a = 100 to 10 as both are pointing to global memory.
// 10 shadows 100
console.log(b); // 20
console.log(c); // 30
```

```
let a = 100;
{
  let a = 20;
  console.log(a); // 20 (Here 20 shadows 100)
}
console.log(a); // 100 & here a does not point to global memory because let and const
// are hoisted in separate memory.

// Similar for const also
```

0. If you shadow let variable inside a block using var variable, It is illegal shadowing.
1. Block also follows lexical scope.
2. The shadow should not cross the scope of original otherwise it will give error.

Closures in JS

1. Closure is a function binded/bundled with its lexical environment.
2. Function along with its lexical scope bundled together forms a closure.
3. Examples:

```
function x() {
  var a = 7;
  function y() {
    console.log(a);
  }
  y();
}

x();
//-------------------------------------
//Console:
//7
```

```
function x(){
var a = 7;
function y(){   // Also can be → return function y() {..}
console.log(a);
}
return y;
}
var z = x();
console.log(z);
//.............
z();
//-------------------------------------
Console:
f y(){
console.log(a);
}
```

```
7
```

4. When a function is returned in Javascript, Function along with its lexical scope is returned which is a Closure.
5. The function that is returned is reference, so any change in value of that variable inside function will be reflected as such.

```
function x(){
var a = 7;
function y(){   // Also can be -> return function y() {..}
console.log(a);
}
a = 100;
return y;
}
var z = x();
console.log(z);
//.............
z();
-------------------------------------
Console:
f y(){
console.log(a);
}

100

// Here a in function y() reference points to variable a not the value 7 or 10
0.
```

```
function z() {
  var b = 900;
  function x() {
    var a = 7;
    function y() {
```

```
    console.log(a, b);
   }
   y();
  }
  x();
}
z();

// ----- Execution flow -----
// z() is called.
// Inside z, x() is called.
// Inside x, y() is called.
// y logs a (7) and b (900) to the console.
```

6. Uses of Closures:
   - Module Design Pattern
   - Currying
   - Functions like once
   - memoize
   - maintaining state in async world
   - setTimeouts
   - Iterators
   - and many more...
7. Summary: Function bundled with its lexical environment is known as a closure. Whenever function is returned, even if its vanished in execution context but still it remembers the reference it was pointing to. Its not just that function alone it returns but the entire closure.

setTimeout() + Closures Interview Questions

1. setTimeout stores the function in a different place and attaches a timer to it, when the timer is finished it rejoins the call stack and executed.

2. Here, JS creates a closure for setTimeout function and proceeds to next line of code. So, “Namasthe JavaScript” is printed in console first and after 3 seconds “10” is printed in console.

```
function x() {
 var i = 10;
 setTimeout(function () {
  console.log(i);
 }, 3000);
 console.log("Namaste JavaScript");
}
x();
```

3. Question: Print 1 after 1 Seconds, 2 After 2 seconds … until 5
Result: After seeing the console, We can now infer that i in setTimeout function is a reference to variable i in closure. So by the time setTimeout function gets triggered, for loop is executed and value of i is 6.

```
function x() {
 for (var i = 1; i <= 5; i++) {
  setTimeout(function () {
   console.log(i);
  }, i * 1000);
 }
 console.log("Namaste JavaScript");
}
x();

Console:
Namaste JavaScript
6
6
6
```

```
6
6
```

4. How to fix it?
   Solution: Use let instead of var since it is block scoped. Every time the loop is running, Each closure will have different reference to i since it is block scoped. So 1st closure in for loop refers to i with value 1 and 2nd closure in for loop refers to different i with value 2 and so on..

```
function x() {
 for (let i = 1; i <= 5; i++) {
  setTimeout(function () {
   console.log(i);
  }, i * 1000);
 }
 console.log("Namaste JavaScript");
}
x();

Console:
Namaste JavaScript
1
2
3
4
5
```

5. If you want to use only var, then?
   Solution: Using closures. Here we have enclosed setTimeout in function close. When we are calling close(i) for each loop, A closure is created with different setTimeouts referring to different i.

```
function x() {
 for (var i = 1; i <= 5; i++) {
  function close(y) {
```

```
    setTimeout(function () {
      console.log(y);
    }, y * 1000);
  }
  close(i);
 }
 console.log("Namaste JavaScript");
}
x();
```

Crazy JS Interview Questions → Ft. Closures

Summary of Closures

- A closure is the combination of a function and the lexical environment within which that function was declared. This environment consists of any variables that were in scope at the time the function was created.
- Closures enable a function to access variables from its outer scope even after the outer function has returned.
- Closures are created when a function is defined inside another function. The inner function has access to the outer function's variables.

Examples:

```
// Closures
function outer(b) {
 function inner() {
  console.log(a, b);
 }
 let a = 10;
 return inner;
}
```

```
outer("Hello")();
// or
var close = outer("Hello");
close();

// Console: 10 "Hello"
```

```
function outest() {
  var c = 20;
  function outer(b) {
    function inner() {
      console.log(a, b);
    }
    let a = 10;
    return inner;
  }
  return outer;
}

let a = 100;

var close = outest()("Hello");
close();

// inner is a closure with outer and outest lexical environments

// Console: 10 "Hello" 20

// if a = 10 is commented then 100 "Hello" 20

// if both a = 10 & a =100 are commented then reference error → a is not de
fined
```

Summary of Lexical Scope

- Lexical scoping determines the accessibility of variables based on their position within the source code.

- Inner functions have access to the variables declared in their outer functions.
- This scope is determined at compile time (or when the function is defined), not at runtime.

Understanding the Difference: Compile Time vs. Runtime

- **Compile Time (or Function Definition):**
  - In JavaScript (even though it's interpreted, the engine does a lot of work before executing), this refers to the moment when the JavaScript engine parses your code and creates the functions.
  - At this point, the engine analyzes the structure of your code and establishes the relationships between functions and the variables they can access.
  - This is when the "lexical environment" of a function is determined.
- **Runtime (or Execution):**
  - This is when the JavaScript engine actually executes your code, line by line.
  - The values of variables can change during runtime, and functions can be called in different contexts.

Why Lexical Scoping is Compile-Time/Definition-Time?

- **Static Analysis:**
  - The JavaScript engine can analyze the code's structure (where functions are nested) to determine the scope of variables.
  - It doesn't need to wait for the code to run to figure out which variables a function can access.
- **Predictable Behavior:**
  - Lexical scoping makes JavaScript code more predictable. You can look at the code and determine which variables are in scope at any given point.

- This is in contrast to dynamic scoping, where the scope of a variable depends on how the function is called (the "call stack").

- **Closure Formation:**
  - When a function is defined inside another function, the inner function "remembers" its lexical environment. This environment is captured at the time of the inner function's *definition*, not when it's eventually called.
  - That captured environment is what allows closures to work.

Example:

```
function outer() {
  let outerVar = "I am from outer";

  function inner() {
    console.log(outerVar);
  }

  return inner;
}

let myInnerFunction = outer(); // outerVar's scope is captured here.
myInnerFunction(); // Even when called later, inner() still accesses outerVar.
```

- When `outer()` is called, the `inner()` function is created.
- At that moment (definition time), `inner()` captures the lexical environment of `outer()`, which includes `outerVar`.
- Even after `outer()` finishes executing, `inner()` (now stored in `myInnerFunction`) still has access to `outerVar` because it was captured at definition time.
- If scope was determined at runtime, then once outer finished executing, inner would not have access to outerVar.

In essence, lexical scoping is about the *structure* of your code, while runtime is about the *execution* of your code.

Data Hiding and Encapsulation

- Closures facilitate data hiding by allowing the creation of private variables.
- Variables declared within a closure are only accessible from within that closure.
- This provides a mechanism for encapsulation, where the internal state of an object is protected from outside access.

Examples:

```
// Data Hiding (or) Encapsulation → Here count is accessible inside counter () only and not outide.
function counter() {
  var count = 0;
  return function incrementCounter() {
    count++;
    console.log(count);
  };
}
var counter1 = counter();
counter1();
counter1();

var counter2 = counter();
counter2();

// Console
// 1
// 2

// 1
```

```
// counter2 creates a fresh counter → advantage of closure
```

```
// Constructor function with closures
function Counter() {
  var count = 0;
  this.incrementCounter = function () {
    count++;
    console.log(count);
  };
  this.decrenentCounter = function () {
    count--;
    console.log(count);
  };
}

// Always use new keyword when using constructors and function name m
ust be starting with Capital letter
var counter1 = new Counter();
counter1.incrementCounter();
counter1.increnentCounter();
counter1.decrementCounter();

// Console:
// 1
// 2
// 1
```

Garbage Collection

- JavaScript uses garbage collection to automatically manage memory.
- Disadvantage of closures is with Garbage collection, Since they cannot be garbaged as they might be used later and consume a lot of memory.

- The garbage collector reclaims memory occupied by variables that are no longer reachable.
- Closures can sometimes lead to memory leaks if they unintentionally maintain references to variables that are no longer needed elsewhere in the program. This prevents the garbage collector from reclaiming that memory.
- It's important to be mindful of the variables captured by closures and ensure they are released when no longer required to prevent memory leaks.

Example: (Smart garbaging by modern browsers)

```
function a() {
  var x = 0, z = 10;
  return function b() {
    console.log(x);
  };
}
var y = a();

y();

// Here z is in closure with b(), but it is not used so it is smartly garbaged by modern browsers
```

FIRST CLASS FUNCTIONS ft. Anonymous Functions

1. **Function Statement (aka Function declaration) & Function Expression**
   Difference is in hoisting.
   Example:

```
a();
b();
```

```
// Function Statement
function a() {
 console.log("a called");
}

// Function Expression
var b = function () {
 console.log("b called");
};

// Console:
// a called
// uncaught type error: b is not a function;
```

2. **Anonymous functions**

   Anonymous functions are used when the functions are used as values.

3. **Named Function Expression**

```
// Function Expression
var b = function xyz() {
 console.log("b called");
};

b();
xyz();

// Console
// b called
// Uncaught error: xyz is not defined
```

4. **Parameters vs Arguments**

```
function a (parameters) {
   console.log(parameters)
}

a(arguments);
```

5. **First Class Functions (First Class Citizens)**
   a) Passing function as a argument to a function.
   b) The Ability of functions to be used as values.
   c) To return function from a function.

```
var b = function (param1) {
   console.log(param1);
}

b(function () {
})

// Console
// f () { }
```

```
var b = function (param1) {
   console.log(param1);
}

function xyz () {
}

b(xyz)

// Console
// f xyz() { }
```

```
var b = function () {
   return function () {
```

```
  }
}

console.log(b());

// Console
// f () { }
```

6. SUMMARY

- **Function Statement:** This involves using the `function` keyword followed by a name, like `function a() { console.log('call'); }`.
- **Function Expression:** This is when a function is assigned to a variable, such as `var b = function() { console.log('b called'); }`.
- **Hoisting:** Function statements are hoisted, allowing them to be called before they are defined in the code. Function expressions, however, are not hoisted in the same way. If you try to call a function expression before it's assigned, you'll encounter an error.
- **Function Declaration:** This is another term for a function statement.
- **Anonymous Function:** This is a function without a name. It cannot be used as a function statement on its own, as it will result in a syntax error. Anonymous functions are typically used where functions are treated as values, such as when assigning them to variables.
- **Named Function Expression:** This is similar to a function expression, but the function has a name. For example, `var b = function xyz() { console.log('xyz'); }`. The name `xyz` is only accessible within the function itself, not in the outer scope.
- **Parameters vs. Arguments:** Parameters are the labels or identifiers listed when defining a function (e.g., `param1`, `param2`), while arguments are the actual values passed to the function when it is called (e.g., `1`, `2`).

- **First Class Functions:** In JavaScript, functions are treated as first-class citizens. This means they can be used as values, passed as arguments to other functions, and returned as values from other functions. This ability is what defines first-class functions.
- **First Class Citizens:** The terms "first class functions" and "functions are first class citizens" mean the same thing.
- `let` **and** `const`**:** Using `let` or `const` with function expressions follows the same rules as with other variables, including temporal dead zones.

Callback Functions in JS ft. Event Listeners

1. **Summary**

- **Callback Functions**: In JavaScript, functions are treated as first-class citizens, meaning they can be passed as arguments to other functions. A callback function is a function that is passed into another function as an argument, which is then invoked inside the function to complete some kind of routine or action. These functions gives us access to the whole asynchronous world in a synchronous single threaded language.
- **Asynchronous Tasks**: Callback functions enable asynchronous operations in JavaScript. For example, `setTimeout` uses a callback function that is executed after a specified time delay.
- **Execution Flow**: JavaScript executes code line by line. When an asynchronous operation like `setTimeout` is encountered, it's registered and placed in a separate space. The main thread (call stack) continues executing the rest of the code. Once the timer expires, the callback function is moved to the call stack and executed.

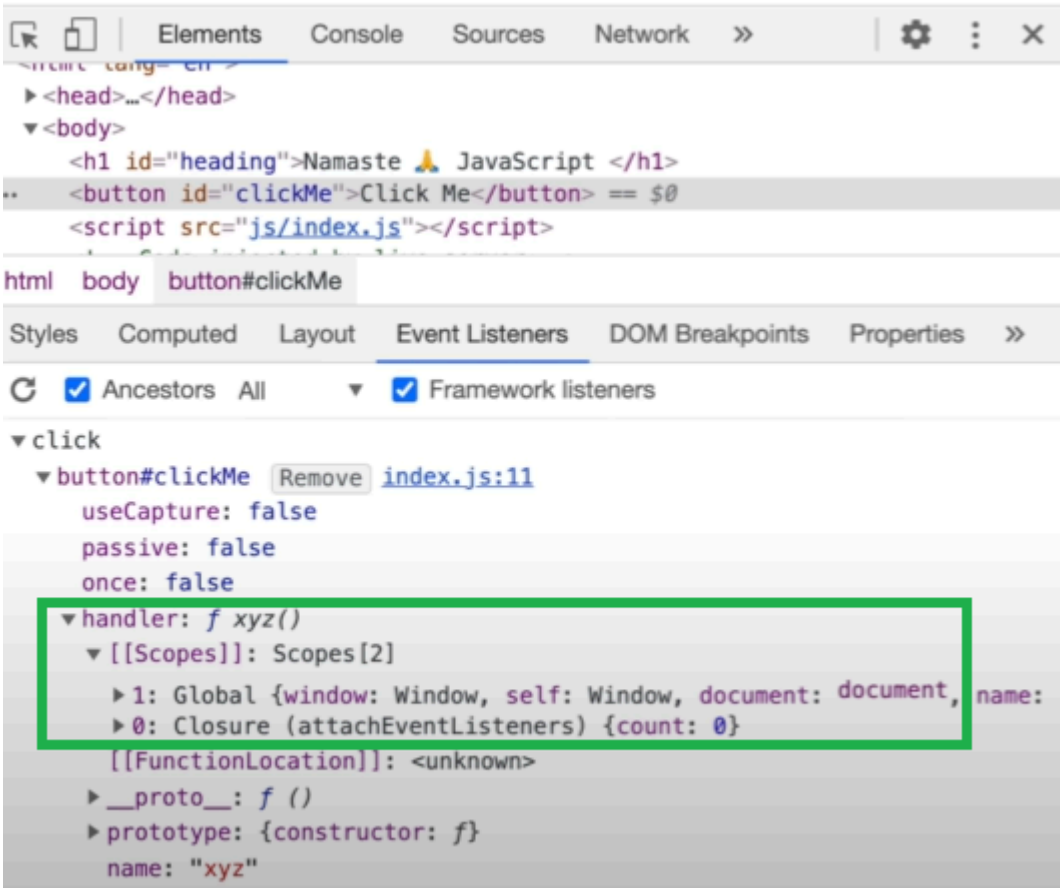
- **Event Listeners**: Event listeners attach a function to an HTML element, which will be executed when a specific event occurs. This is another use of callback functions.
- **Closures**: When an event listener is attached, it forms a closure, remembering the environment in which it was created. This allows the event handler to access variables from its parent scope even after the parent function has completed.
- **Memory Management**: Event listeners are heavy. It's important to remove event listeners when they are no longer needed to free up memory. Failing to do so can lead to memory leaks, especially in applications with many event listeners.
- **Examples:**

```
setTimeout(function () {
  console.log("timer");
}, 5000);

function x(y) {
  console.log("x");
  y();
}

x(function y() {
  console.log("y");
});

// Here function inside a setTimeout is a callback function.
// Callback function is again called to call stack after 5 seconds and is exec
uted
/*
Console:
x
y
```

```
timer (after 5 seconds)
*/
```

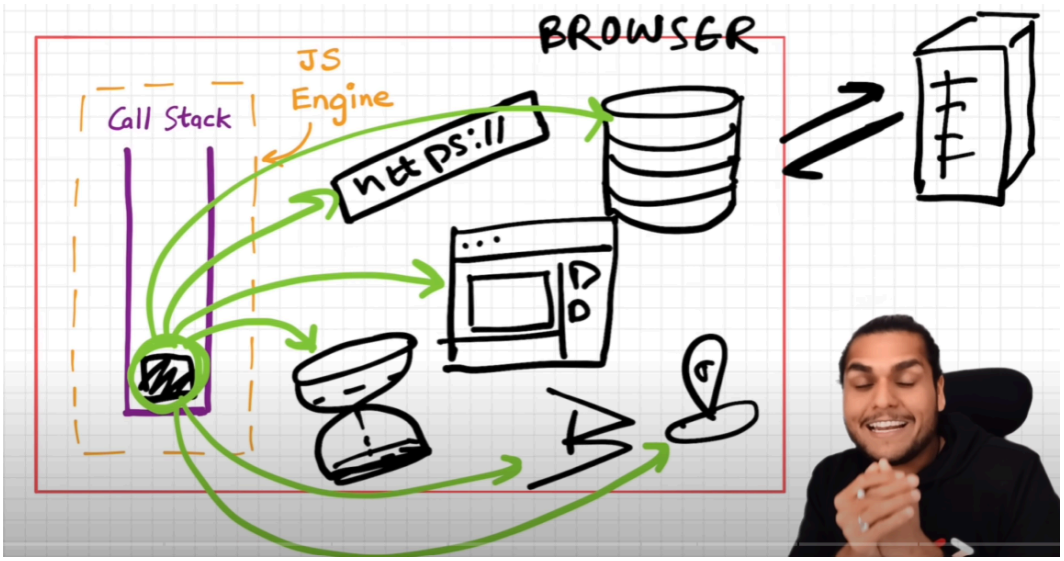
```
// Event listeners
document.getElementById("clickMe").addEventListener("click", function xyz() {
  console.log("Button Clicked");
});

// Here, When a button is clicked function xyz is called to call stack and is executed.
// xyz is a callback function
```

```
// Event Listeners with closures (** FAMOUS INTERVIEW QUESTION) -> To count the no. of times button is clicked.
function attachEventListener() {
  let count = 0;
  document.getElementById("clickMe").addEventListener("click", function xyz() {
    console.log("Button Clicked", count++);
  });
}

attachEventListener();
// Below is the image of debugging tool for this event listener
```

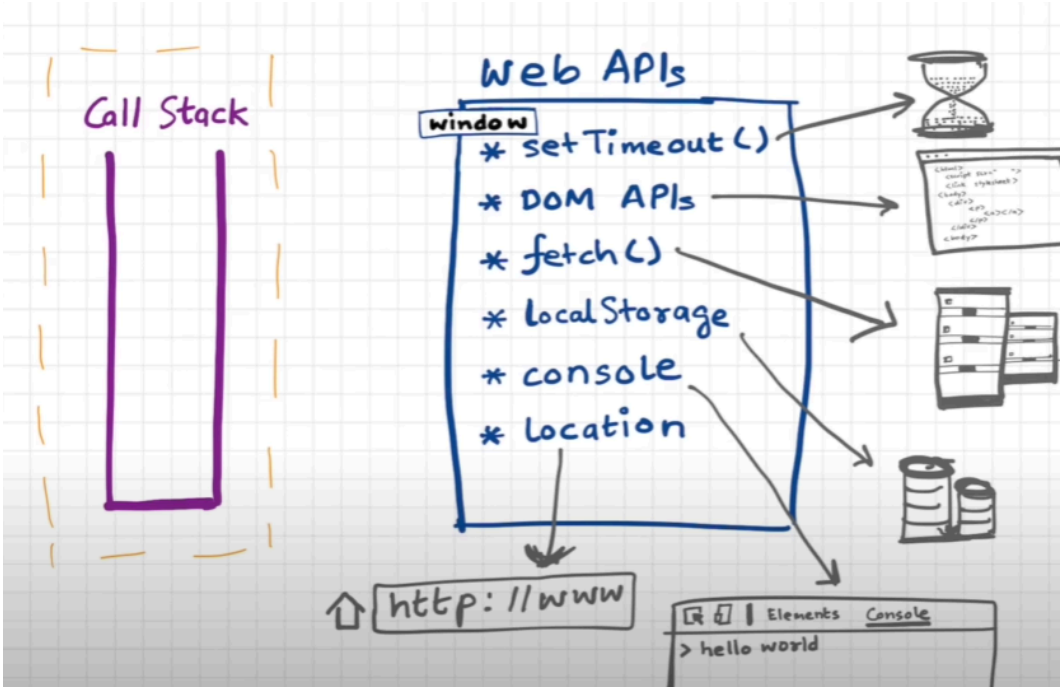


Asynchronous JavaScript & EVENT LOOP

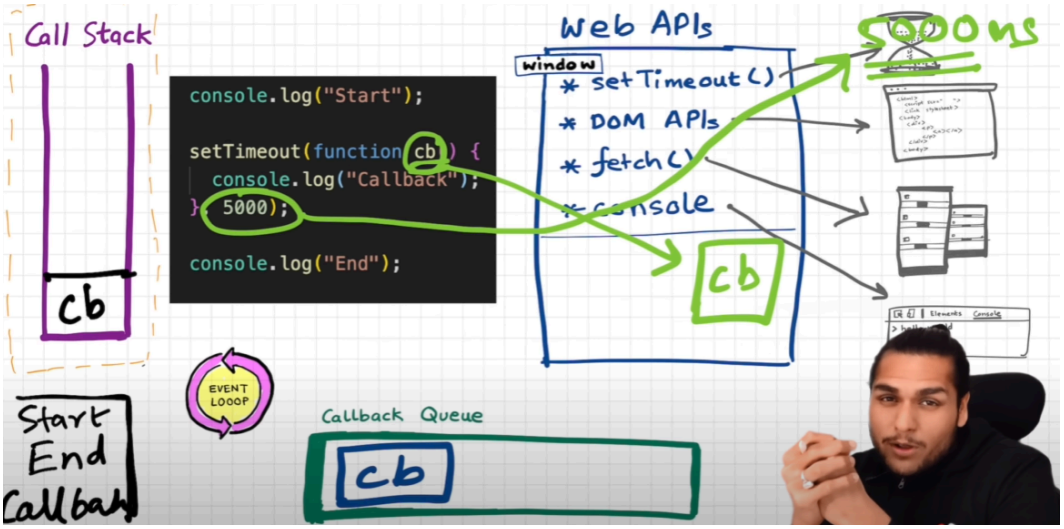
1. BROWSER AND JS ENGINE

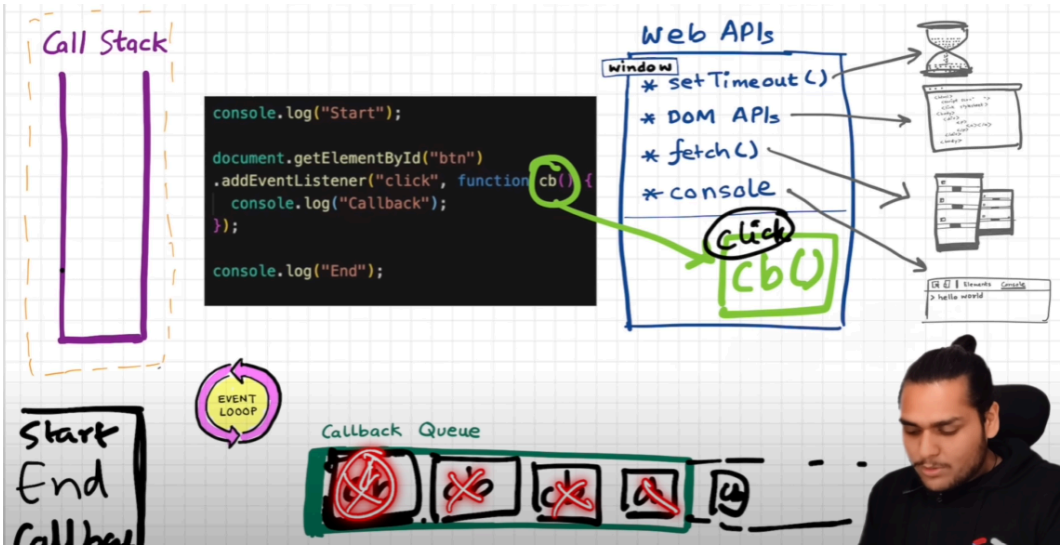


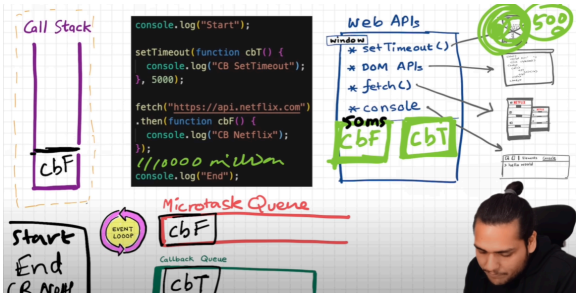
2. To access all the things from browser, Web API’s are required and they are part of browser.



3. When a timer completes, the callback function (here, 'cb') is placed into the callback queue. The event loop continuously monitors the callback queue and pushes the callback function onto the call stack when it is empty. Callback functions are registered in the Web API’s environment.







Callback functions from Promises and Mutation observers goes to the Microtask Queue which has higher priority. All others go to Callback Queue. Callback queue are also referred as Task queue.

4. **SUMMARY**

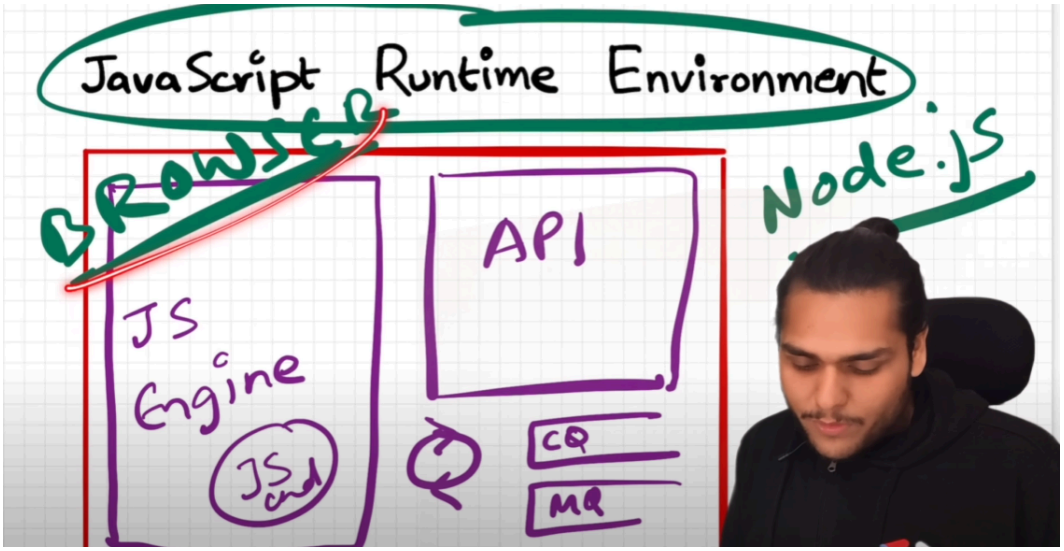
- **Call Stack:** JavaScript code is executed inside the call stack, operating on a first-in, first-out basis. When a function is invoked, an execution context is created and pushed onto the call stack.
- **Browser APIs:** Browsers provide access to features like timers, DOM manipulation, and network requests, which are essential for asynchronous operations. These APIs are accessed in JavaScript through the global `window` object.
- **Callback Queue:** When asynchronous operations like `setTimeout` or event listeners complete, their callback functions are

placed in the callback queue.

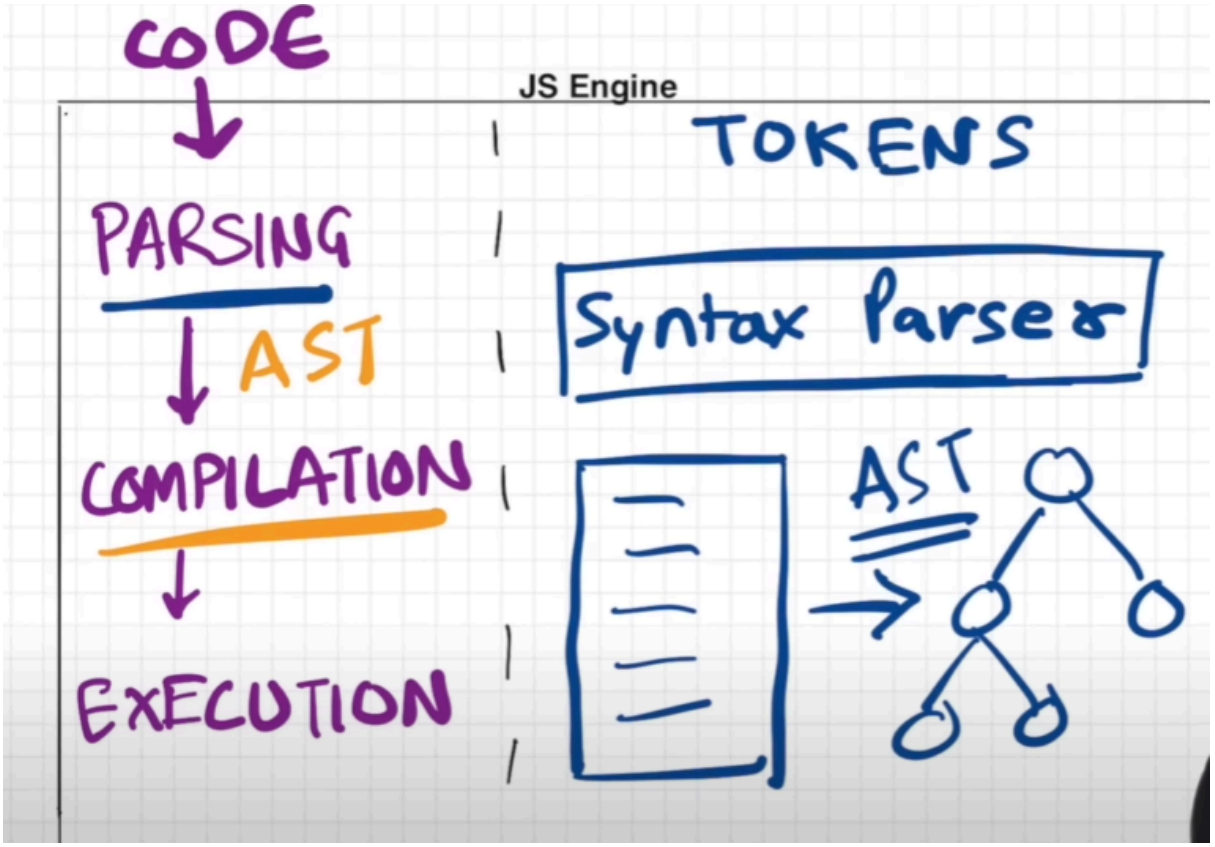
- **Event Loop:** The event loop continuously monitors the call stack and the callback queue. If the call stack is empty, the event loop moves the first callback from the queue to the call stack for execution.
- **Microtask Queue:** Callbacks from promises and Mutation observers are placed in the microtask queue, which has a higher priority than the callback queue. The event loop processes microtasks before checking the callback queue.
- **Starvation:** If the microtask queue is never empty, callbacks in the callback queue will never get a chance to execute.

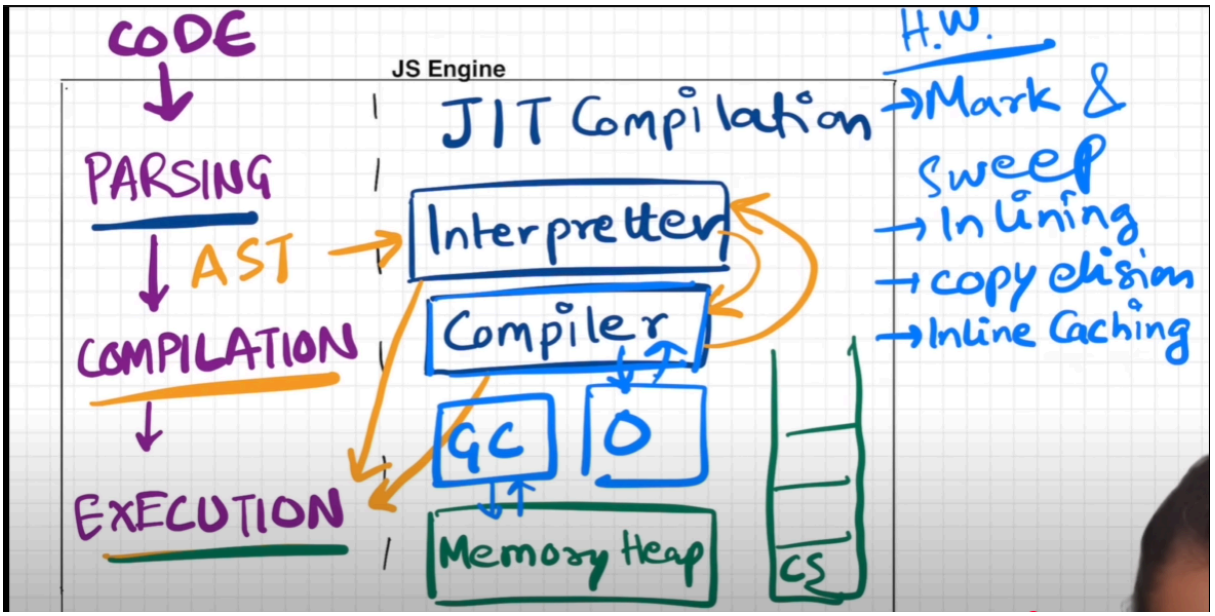
JS Engine EXPOSED

1. JS Runtime Environment

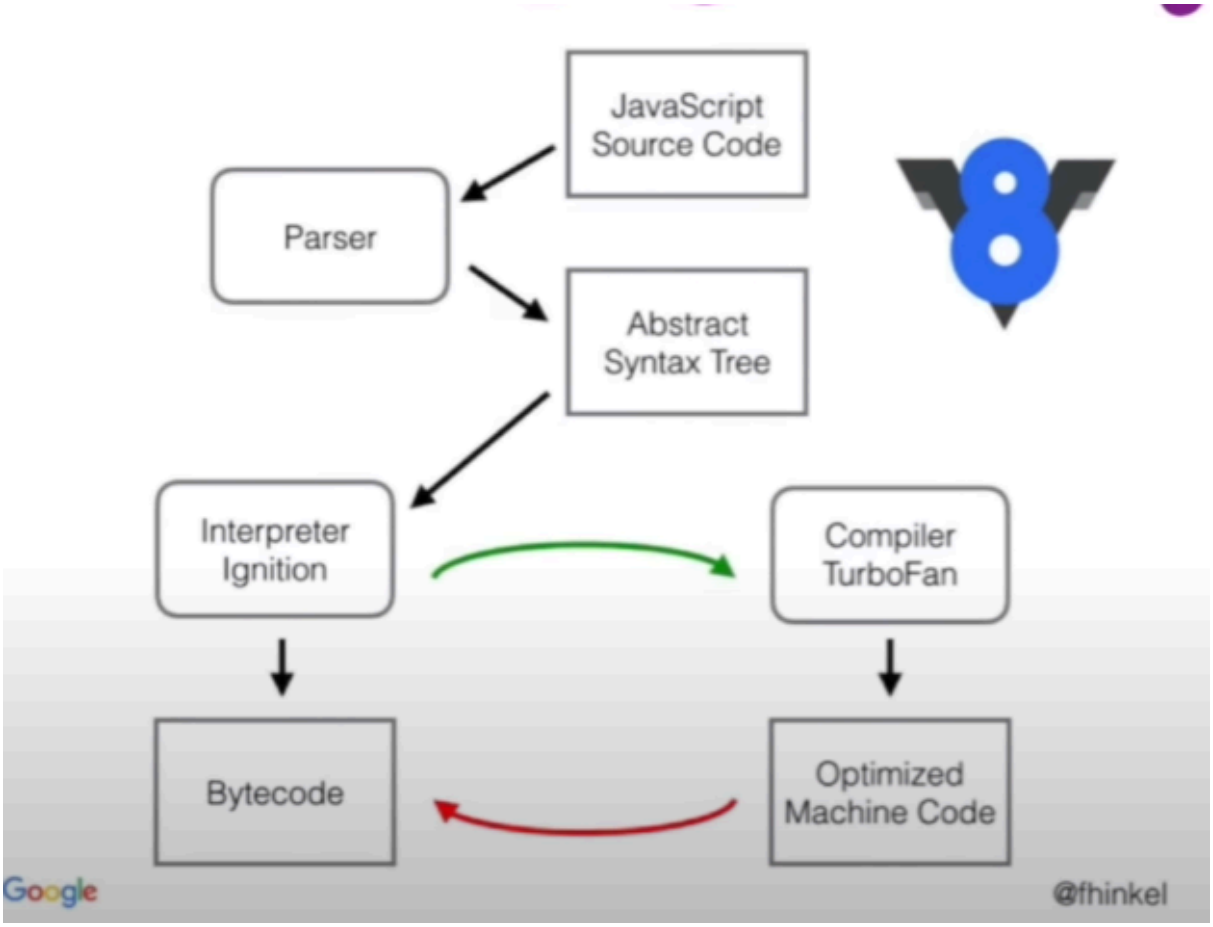


2. Three steps performed by JS Engine.





3. To see the AST (Abstract Syntax Tree) of JS code → astexplorer.net
4. Google’s V8 JS Engine (Used in Chrome and Node.js)



5. **SUMMARY**

- **JavaScript Runtime Environment:** JavaScript needs a runtime environment to execute. This environment provides the necessary tools and resources for JavaScript code to run. Browsers and environments like Node.js serve as JavaScript runtime environments.
- **JavaScript Engine:** Every browser has a JavaScript engine, responsible for executing JavaScript code. Node.js also utilizes a JavaScript runtime environment.
- **ECMAScript:** JavaScript follows a set of standards called ECMAScript. These standards define the language's syntax and features.
- **JavaScript Engine Internals:** A JavaScript engine processes code through several stages: parsing, compilation, and execution.

- **Parsing:** During parsing, the code is transformed into an Abstract Syntax Tree (AST). The AST represents the code's structure and is used in subsequent stages.
- **Compilation and Execution:** Modern JavaScript engines employ a combination of an interpreter and a compiler for optimization. The interpreter converts high-level code to bytecode. The compiler then optimizes the code during runtime, improving performance. This is often referred to as Just-In-Time (JIT) compilation.
- **Memory Heap and Call Stack:** Execution involves memory management. The heap is used for dynamic memory allocation, while the call stack keeps track of function calls.
- **Garbage Collection:** The garbage collector automatically reclaims memory that is no longer in use, preventing memory leaks. (Mark & Sweep algorithm)
- **Optimization Techniques:** JavaScript engines utilize various optimization techniques, such as hidden classes and inline caching, to enhance performance.

TRUST ISSUES with setTimeout()

- **What `setTimeout()` does:** It is used to delay the execution of a function. The specified delay is not a guarantee, but rather a minimum wait time.
- **How `setTimeout()` works:** When `setTimeout()` is called, the callback function is registered in the Web API environment. After the specified delay, the callback is moved to the callback queue. The event loop constantly checks if the call stack is empty. If it is, the callback is moved from the queue to the call stack and executed.
- **Blocking the Event Loop:** If there is a long-running task in the call stack, it can block the event loop and delay the execution of the `setTimeout()` callback, even if the specified time has passed.

- **Non-Blocking Behavior:** JavaScript is single-threaded, but it achieves concurrency through the event loop and asynchronous operations like `setTimeout()`. This allows the program to continue running other tasks while waiting for the `setTimeout()` callback to be executed.
- **Use Cases for `setTimeout()`:** It can be used to defer the execution of less important code until after more critical tasks have completed. It's also useful for tasks that should be executed after a certain delay, such as animations or polling for updates.
- **Concurrency Model:** With the help of this model, we can do asynchronous operations inside a single-threaded language.

Higher-Order Functions ft. Functional Programming

1. A function which takes another function as an argument or returns a function are known as Higher-Order fucntions

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
  return Math.PI * radius * radius;
};

const cicumference = function (radius) {
  return 2 * Math.PI * radius;
};

const diameter = function (radius) {
  return 2 * radius;
};

const calculate = function (radius, logic) {
  const output = [];
```

```
  for (let i = 0; i < radius.length; i++) {
    output.push(logic(radius[i]));
  }
  return output;
};
console.log(calculate(radius, area));
console.log(calculate(radius, cicumference));
console.log(calculate(radius, diameter));
```

```
const area = function (radius) {
  return Math.PI * radius * radius;
};

// For all arrays in code, This function can be used.
Array.prototype.calculate = function (logic) {
  const output = [];
  for (let i = 0; i < this.length; i++) {
    output.push(logic(this[i]));
  }
  return output;
};

console.log(radius.map(area));
console.log(radius.calculate(area));
```

2. Follow DRY (Don't Repeat Yourself) principle while coding.
3. Use function to stop writing repeating line of codes.
4. Function that takes another function as argument (Callback function) is known as Higher order functions.
5. It is this ability that function can be stored, passed and returned, they are called First Class Citizens.
6. If we use Array.property.function-name, The function will be accessible to any array in your code.

map, filter & reduce

1. Example of map

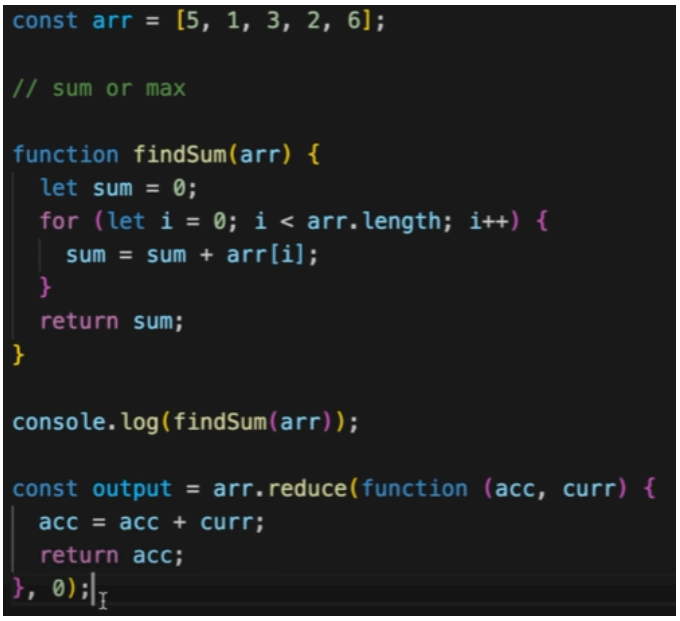
```
const users = [
{ firstName: "akshay", lastName: "saini", age: 26 },
{ firstName: "donald", lastName: "trump", age: 75 },
{ firstName: "elon", lastName: "musk", age: 50 },
{ firstName: "deepika", lastName: "padukone", age: 26}
]
// list of full names

let fullNames = users.map(function (obj) {
   return obj.firstName + " " +  obj.lastName;
});

let fullNamesEasy = users.map((obj) => obj.firstName + " " +  obj.lastNam
e);

console.log(fullNames);
console.log(fullNamesEasy);
```

2. Example of filter

```
// First name of all the people whose age less than 30
// See the same solution using reduce
let ageLessThan30 = users.filter(function(obj) {
   if(obj.age < 30){
      return obj.firstName;
   }
});

let FirstNamesLessThan30 = ageLessThan30.map(obj => obj.firstName);

console.log(ageLessThan30);
console.log(FirstNamesLessThan30);

/IMPORTANT/
```

```
let FirstNamesLessThan30Easy = users.filter(obj => obj.age <30).map(obj =
> obj.firstName);
console.log(FirstNamesLessThan30Easy);
```

3. Examples of reduce
   Understanding reduce




```
let arr = [4, 5, 6, 8, 10, 1, 7];

let output = arr.reduce(function (acc, curr) {
   if(curr > acc) {
      acc = curr;
   }
   return acc;
}, 0);

console.log(output);
```

```
// Number of users with particular age
// reduce to {26: 2, 75: 1, 50: 1}

let ages = users.reduce(function (acc, curr) {
   if(acc[curr.age]) {
```

```
        acc[curr.age]++;
    } else {
        acc[curr.age] = 1;
    }
    return acc;
}, {});

console.log(ages);
```

```
let example = users.reduce(function(acc, curr) {
    if(curr.age < 30){
        acc.push(curr.firstName);
    }
    return acc;
}, []);

console.log(example);
```

4. **SUMMARY**

- **Higher Order Functions**: Map, filter, and reduce are higher-order functions in JavaScript.
- **Map Function**: Used to transform each value of an array and returns a new array. You pass a transformation function to `array.map()`.
- **Filter Function**: Used to filter values from an array based on a condition. It returns a new array containing only the values that meet the specified criteria. You pass a filter logic function to `array.filter()`.
- **Reduce Function**: Used to iterate over all elements of an array and condense them into a single value. It takes a reducer function and an initial value. The reducer function has two parameters: an accumulator and the current value.
- **Chaining**: Map, filter, and reduce can be chained together.
- **Reduce as an alternative**: Reduce can be used to achieve the same results as filter and map.