JavaScript by Akshay Saini

How JS works?
Everything in JS happens inside an Execution Context.

Execution Context: Component where JS is executed. It has:

- **Memory or Variable Env.**
  Here variables are stored as `key: value` pair.
- **Code Env or Thread of execution**
  This is the place where code is executed one line at a time.

Note: JavaScript is a synchronous single-thread language.

1st Phase: Memory creation phase

- Everything happens in a 'Global' execution context.
- In this phase, memory is allocated to all variables and function(s).
- For variable(s), key is the variable name itself and value is `undefined` (even if the variable is initialized). And, for function(s), key is the *function name* and value is *body of the code.*

2nd Phase: Code Execution phase

- Code is traversed line by line and actual value of variables are assigned to them.
- A new 'local' execution context is created, when function 'invocation' is encountered. Again, two phase perform their role.

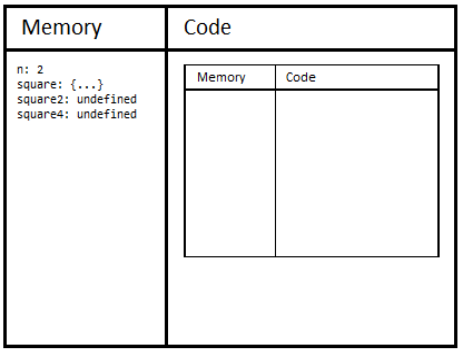


Fig. JS execution context

Q. How it manages all the execution context, i.e., local & global?
A. It is managed through Stack, aka, **Call stack**, with Global execution context at the bottom and all the subsequent function invocation or new execution context is pushed to this call stack. Once the execution context is done with the task, it is popped.
Well, call stack is also called by names like, Execution context, Program, Machine, Control, Runtime stack.

Hoisting
Phenomena in JS through which we can access the variables and functions even before they have initialized, without any error.

```
console.log(x); //perfectly fine but o/p will be 'undefined'
x = 90;
```

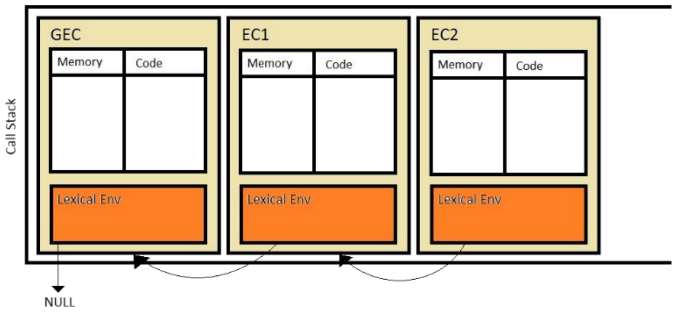
JS Engine
It is responsible for any js code to execute. It creates:

1. Global object (in browsers, it is called window) which is referenced by 'this'.
2. Global execution context

, **even if the file is empty.**
Any global attribute, say `a`, can be accessed, in browser, by `a`, `this.a` or `window.a`

Note: In browser, `window` object, thus created, comes with various *inbuilt* attributes and functions (discussed later).

Lexical Environment
Whenever execution context is created, a lexical environment is also created.
It comprises of local memory and **'reference' (and not 'copy')** of lexical environment of its parent.
For Global Exec. Context, lexical environment points to null.



Whenever a variable is encountered while execution, it is first searched in the local memory space and if not found, it goes to the lexical environment of its parent and perform the same step, until found or reaches global scope.

Scope chain
The above 'chaining' of lexical environments in order to find a particular variable, is called scope chain.

let & const
`let & const` are hoisted but stored in different memory space than other variables like `var`. (And hence they cannot be access via `window` object or `this` specifier)
They cannot be accessed until they're initialized. Hence, the time from hoisting these variable(s) and initialization is **temporal dead zone,** and during this we cannot access `let & const`, in turns throws `Reference error`.
`let` cannot be re-declared in the same scope unlike `var`, it will throw `Syntax Error`.
In case of `const` we need to declare and initialise at the same line/time. If we try changing the value later at some line to const variable, we'll get `Type Error`.

Note: In case of `const` array or object, if we try to change the value, it is perfectly fine/valid. The property of a const object can be change but it cannot be change to reference to the new object.

Block in js
Block combines multiple js statement and can be used at places where single line is expected.
`let & const` cannot be accessed outside the block, in which they reside. Hence, they're called as *block scoped*, and var to be *function scoped*.

Note: The variable(s) in the local scope ***shadows*** the same-named variable in the outer scope. But due to the fact that `let & const` is block scoped, a different memory space is created for `let & const` variable(s) but same variable is over-written in case of `var`. Well, this phenomenon is called **Shadowing.**

Illegal Shadowing
Shadowing a `let` variable with `var` type is not allowed and this is called Illegal Shadowing. (Why?)
Because `var` 'overflows' from the inner scope to outer (since it is function scoped) where `let` variable is present and we are now left with multiple declaration of `let` variable which is invalid.

```
let a = 10;
{
        var a = 100;
}
//illegal shadowing throws Syntax Error
```

Closures
A **Closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).
That means, each function in JS has access to its surrounding environment (which includes variables and functions).

```
//notice use of double parenthesis
function outer(){
        var a = 10;
        function inner(){
                console.log(a);
        }
        return inner;
}
outer()();          // o/p 10
```

The above snippet will execute the `inner()` function in the same line.

Constructor function

Function which is used to create Objects (by using `new` keyword) with properties being accessed by the 'dot' operator. We can see below `counter` as constructor function.

```
function counter(){
        var count = 0;
        this.incrementCounter = function() {
                count++;
        }
        this.decrementCounter = function(){
                count--;
        }
}
var counter1 = new counter(); //Remember to use 'new' keyword
//for constructor function.
```

Disadvantages of Closures

Closure is associated directly with memory consumption. Hence, it leads to high consumption of memory, if lot of closures are created, since the allocated memory are not garbage collected till program expires.

Garbage Collector

It is a program in the browser/JS Engine which is responsible for freeing up the memory which are unutilised.

Function Statement/Declaration

This is defining any function like following:

```
function A () {
        //body
}
```

Function Expression

This is assigning function to a variable.

```
var a = function () {
        //body
}
```

Named Function Expression

Same as function expression but the right-hand side has got a name.

```
var x = function A () {
        //body
}
```

Parameters vs Arguments

The variables defined in the function declaration are called **Parameter**(s) and the variables that are actually passed during a function call is called **Argument**(s).

```
function counter(x, y){  //x & y are parameter
        console.log(x+y);
}
var a = 10, b = 2;
counter(a, b);     //a & b are argument
```

First class function

The ability of functions to be:

1. Assigned to variable
2. Passed as argument to another function
3. Returned from another function

In JS, is called first class function, aka, first class citizens.

CallBack functions

Functions passed as argument into another functions.
JS is synchronous single threaded language but through use of Callback functions we can perform async task.
Eg, Event listeners make use of this.

Why do we need to remove event listeners?

Event listeners are heavy, i.e., it takes memory. And even if the call stack is empty, memory is not freed up from the lexical environment of the event listener function.

Web APIs

All the code execution is done inside call stack, which is present in JS engine, which in turn is in browser.
Browser has some added functionalities like, Local storage, Timer, Address field, etc.
For incorporating additional functionalities and connecting to outer ENV like localStorage, fetching data from remote source, etc, browser comes up with Web APIs like:

1. SetTimeout()
2. DOM APIs
3. Fetch()
4. LocalStorage
5. Console
6. Location

Any js running inside browser gets all this functionality included due to the global object, thus created (`window`).
Since these APIs are present in the global scope, so these can be accessed directly. So, `window.setTimeout()` and `setTimeout()` are essentially same.

Working of setTimeout()

- As soon as `setTimeout()` is encountered, the callback method is first registered (in the Web APIs environment) and the timer is started.
- As soon as timer expires, the callback function is put in the callback queue.
- Event loop continuously checks for the call stack and callback queue, and if we have something in call back queue *AND* the call stack is empty, it pushes callback method into call stack.

Microtask queue

This is exactly similar to callback queue but with higher priority.
All callback function coming through promises and mutation observers, will go into microtask queue. Everything else goes into callback queue.

Note: Since the priority of microtask queue is more than callback queue, the methods in the callback queue cannot get chance to enter call stack if microtask queue is long, which leads to **starvation** of task inside callback queue.

Note: The MutationObserver interface provides the ability to watch for changes being made to the DOM tree.

Javascript Runtime Environment (let call it JsRE)

It contains all components required to run JS code.
Js Engine is the heart of JsRE.

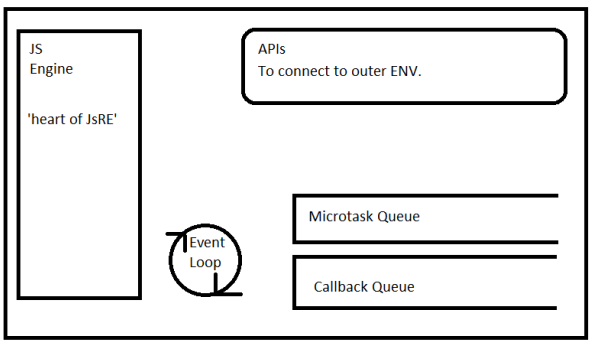


Fig. JavaScript Runtime Environment

For any js engine, it must follow **ECMAScript language standard**.
Js Engine takes code as input and undergoes 3 major phases:
PARSING, COMPILATION and EXECUTION.

1. **Parsing**

In this phase, code is broken down into array of tokens. The syntax parser takes the array of tokens and convert it into AST (Abstract Syntax Tree).

2. Compilation

AST is passed to compilation phase. Implementation of JS engine decides whether JS is interpreted or compiled language. JS Engine can use interpreter along with compiler to make it JIT (Just in time) compiled language.

JIT compilation: Generating machine code during runtime.

AOT compilation: In this compiler takes piece of code (which is supposed to be executed later) and tries to optimize it.

Interpreter and compiler are in sync with execution phase making use of Memory heap and call stack.

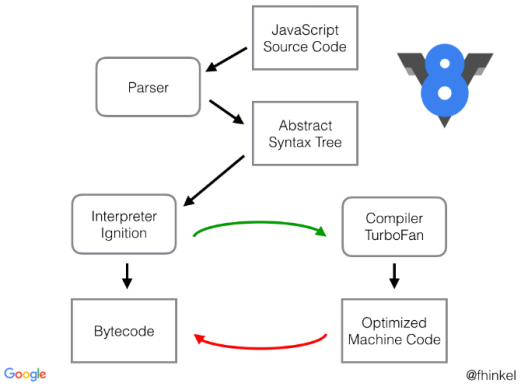


Fig. Js v8 engine architecture

Q. Are only asynchronous web API callbacks, registered in the web API environment?

A. YES, the synchronous callback functions like what we pass inside map, filter, and reduce aren't registered in the Web API environment. It's just those async callback function that are registered.

Higher Order function

Function that takes other function as argument or return function from it, is called higher order function, whereas the passed function is called callback function.

[].map(), [].filter(), [].reduce()

`map()` is used for transformation of each elements of array.

`filter()` is used to filter elements of array based on some condition.

Both of the above method takes function as argument for the logic to be used.

`reduce()` is used to find out some result based on the elements of the array, like sum of all element, max among elements, so on.

It takes function as first arg and initial value for the *accumulator* as second arg.

The callback function in this case has 2 args accumulator and current.

Accumulator: Variable in which final result is expected.

Current: Current element of the array.

```
let a = [1, 2, 3, 4, 5, 6];

function multiplyByTwo(num) {
    return num * 2;
}

function oddNum(num) {
    return num%2;
}

console.log(a.map(multiplyByTwo));    //[2, 4, 6, 8, 10, 12]
console.log(a.filter(oddNum));        //[1, 3, 5]
console.log(a.reduce((acc, curr) => {
    return acc+curr;
}, 0));                               //21
```

(BONUS VIDEOS :P)

Function Currying

Technique in which we convert a function with multiple arguments into several functions of a single arguments in sequence.

`.bind()` method returns a new function.

```
let multiply = function(x, y){
        Console.log(x*y);
}
let multiplyByTwo = multiply.bind(this, 2);
multiplyByTwo(3);  //consoles: 6
```

Prototype

Whenever we create any var/object/function in JS, js engine creates an object with multiple properties/methods (from immediate parent), and attaches to our created object. We can access those implicit features by `.__proto__` or `.prototype` property.

Everything in js is Object, hence each prototype is basically linking immediate parent's property to created var/obj/function. In turn the parent can also have the property attached to them of their immediate parent until `Object` is found, for which the `.prototype` is `null`. This is called **Prototype chain.**

Note: Behaviour-wise, everything inside `__proto__` can be accessed directly via . (dot) operator as if those were the property of object itself.

```
let object = {
    name: "Rohit",
    city: "Pune",
    getIntro: function() {
        console.log(console.log(this.name+" from "+this.city));
    }
}

let obj = {
    name: "ram",
    city: "Mumbai"
}
// JUST FOR DEMONSTRATION, NEVER DO THE FOLLOWING
obj.__proto__ = object;

//obj can directly access the getIntro method
obj.getIntro();    // o/p: ram from Mumbai
```

Event bubbling and capturing

By default, events are bubbled. These propagation of event is expensive and we can stop it by calling, `.stopPropagation()` method.

Because of this event bubbling, we can make use of this, for event delegation where, instead of attaching the events to a lot of child items, we're attaching the event to the parent.

PROS of event delegation:

1. Improves Memory
2. Write less code
3. DOM manipulation

CONS of event delegation:

1. All events are not bubbled up, like, blur, resize, etc.

Call, apply and bind

`.call()` is function borrowing. It takes one argument as the object on which will be acting as 'this' for the called function.

`A.call(B)` is equal to `B.A()`; i.e., A called by B.

If function has arguments, then in this case, first arg of call will be object, and the rest comma separated arg will be corresponding args.

`.apply()` is similar to call but instead of comma separated arg, this method need array of args for the original function.

`.bind()` instead of calling the method, returns copy of the method and bind the method with the obj.

`.call()` and `.apply()` invokes the function in the same line it is written.

```
let empName = {
    firstName: "Akshay",
    lastName: "Saini"
}

let printFullName = function(homeTown, state) {
    console.log(this.firstName + " " + this.lastName);
    console.log( "I'm from " + homeTown + "," + state);
}

printFullName.call(empName, "Pune", "Maharashtra");
printFullName.apply(empName, ["Mumbai", "Maharashtra"]);
```

Debouncing

Calling functions only after certain threshold time, from the last invocation.

If some source is trying to call some function continuously, then, through debouncing we attempt to call the function only after some 'delay'.

Eg. *If we're searching for some product in the search bar(say in e-commerce website), then instead of hitting API for each letter, through this, we can call it, after user has stopped for a while, say 300ms.*

It takes 2 args , function and delay, and the function is only called after the delay, and all previous 'attempt to invocation' is ignored.

```
const getData = () => {
    console.log("fetching data..");
}

const debounce = function(fn, delay) {
    let timer;
    return function() {
        let context = this;
        clearTimeout(timer);
        timer = setTimeout(() => {
            fn.apply(context);
        }, delay);
    }
}

const debouncedMethod = debounce(getData, 300);
```

Throttling

Limiting the fun call rate by 'only' making the 'next' call after some time interval.

Eg. *if api call happen at button click and user is continuously clicking the button , the api call would be made continuously which is costly. So, through throttling, we can prevent this.*

```
const throttle = (fn, limit) => {
    let flag = true;
    return function() {
        if(flag) {
            fn();
            flag = false;
            setTimeout(() => {
                flag = true;
            }, limit);
        }
    }
}
```

Polyfills

It is sort of browser fallback.

Async defer attributes in script

Normally when html parsing is done and script is encountered, parsing is kept on hold. It loads the script and compiles it and the parsing is then resumed.

In case of **async**, the script is loaded parallelly and once loaded, parsing is halted, script is compiled and after that parsing is resumed.

In case of **defer**, the script is loaded parallelly but compilation is only done when whole parsing is done.

```
<script src=" ">
<script async src=" ">
<script defer src=" ">
```

Note: Async doesn't guarantee the order. That is, if the scripts are dependent on some previous scripts, then order is not guaranteed in async. But defer can be used in this case.