**How JavaScript works**

JS is a synchronous single threaded (i.e., JS executes single line at a time) language.

JS was first designed as **Interpreted** language. But now most JS engines are **(Just In Time) JIT Compilation** i.e., JS is both compiled and interpreted. The code is compiled and then interpreted.

While executing a JS code/function, JS engine creates an **Execution Context** which is done in 2 phases:

1. **Variable environment (Memory component):** It is the phase when memory is allocated to the variable and functions. The variables are not assigned any values yet. They are assigned undefined.
2. **Thread of execution (Code component):** It is a thread in which the whole code is executed line by line. Now the variables are assigned any values.  
   The current line of execution is stored in a call stack. Call stack is only for managing the execution context. Whenever an execution context is created (ex: a function is called) it is pushed into the stack and when the execution context is deleted, it is poped out of the stack.  
     
   Functions called will create its own separate Memory and Code component when the global code is running. The functions will have the access to the global variables.  
   Once a function has finished running its instance will be deleted.  
     
   Text, whiteboard

   Description automatically generated with medium confidence

As soon as a JS file runs the engine creates a **Global Execution Context (GEC),** a **Global Object** (called **Window**). “This” points to window in GEC i.e., Global level (main level).

Window holds all the memory allocated.

Whenever an Execution context is created, an Object and a Lexical environment are created with it.

**Hoisting:** Hoisting is a phenomenon is JS by which we can access the variables and functions even before initializing them.

It is possible because the memory is allocated to variable/functions before they are executed or initialized value.

If a value is printed before initializing it, it will print “undefined”, and if the variable is never initialized in the program, then the program will throw variable not defined error.

If a function is printed before defining it, then it will print the function. This happens as in case of function the whole function code is put while allocating memory.

**Lexical environment**: It is the local memory along with the lexical environment (properties / values / functions) of its parent function.

An inner function can be directly called using two parenthesis ()().

Ex: A function created inside main will have access to the values in main. In this case the function is lexically inside main.

**Let**: These variables cannot be used before they are initialized.

* Although it cannot be used, but just like var, it is allocated memory with “undefined” value.
* If we try to use a let variable before initialization, it will throw “Reference Error: Cannot access variable before initialization.”

**Const**: These are just like Let variables, but their values is to be declared in the same line when they are initialized.

* Let and Const are not allocated inside the window (Global object) unless they are initialized, they are kept separately.
* In case of Let, we can initialize the variable first and declare the value later, but this is not possible in const. In const the value declaration should be done in same line.
* In the same scope, Let/const values cannot be changed later. But it can be shadowed in another scope.  
  Ex: If a main function uses a const/let variable, another function which is inside the main function can redeclare a const/let variable with same name and shadow the variable in main. In this case a new variable is created inside the child function.

**Temporal Dead Zone:** It is the time between which the Let/Const variable is Hoisted (has “undefined” value) and it is assigned some value.

**Block (Compound statement):** It is a group of multiple JS code lines written inside {}. It is used when the JS expects a single line statement, but the logic cannot be written in 1 line.

**Closure**: It is a function bind or bundled together with its lexical scope (outer surrounding or parent scope) i.e., A function scope along with its parent function’s scope is closure.

**Advantage of Closures:**

**Graphical user interface, text, application

Description automatically generated with medium confidence**

Data privacy and encapsulation.

**Disadvantage of Closure**: Everytime a closure is formed it consumer a lot of memory, as multiple copies are formed. This may lead to overconsumption of memory which may not be garbage collected until program expires.  
If not handled properly, it may lead to memory leaks.

**setTimeout function**: The JS engine, when encounters this function, it sets a defined timer and moved ahead with its execution of lines after this. When the timer expires then the function inside setTimeout is executed.

* In the below example, the setTimeout function forms a closure with same reference if i since var is not a blocked type. So, JS loops over i 5 times, and then all the 5 setTimeout functions expire after every second and the value of i is printed.

Text

Description automatically generated with low confidence

* In the below example, the inner function forms a closure with new variable i (since let is a blocked type) every time the setTimeout function is called i.e., the copy of i in each iteration is new.  
  Graphical user interface, text, application

  Description automatically generated with medium confidence
* For writing the above program with var, we will need to somehow create a new instance of i every time while creating a closure.

Text, calendar

Description automatically generated

**Types of Functions**

1. Function Statement or Function Declaration: Simple function statement.
2. Function Expression: Function statement as a variable.  
   Since a variable is allocated space first and then while code execution phase it is assigned value, function expression will give error if called before assignment.  
     
   Graphical user interface, text, application

   Description automatically generated
3. Anonymous Functions: Functions without name. JS does not allows functions to be nameless.  
   They are used only as Function expression.

Text

Description automatically generated

1. Named function expression: Function express with name.
2. First class function or call back function: The functions which are used as a argument passed to another function or returned from another function or assigned to another variable is called First class function.
3. Higher Order Function: A function which takes another function as an argument or returns another function.

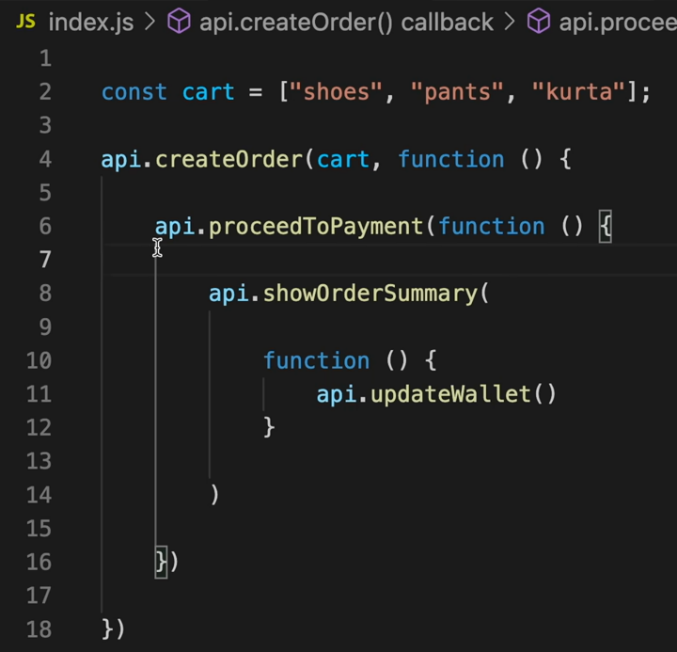
**Asynchronous JavaScript**

**Diagram

Description automatically generated**

**Callback Function:** It is a function which is called by another function or a function which invokes JS engine to run a script after certain event is done.

Ex: setTimeout()

In the above code, JS engine will run the createOrder API, this API will create an order and then call the proceedToPayment function back.  
Then the proceedToPayment will run, it will complete the payment and then call the showOrderSummary function.  
After this in same way, showOrderSummary will run and when it’s done, it will call update Wallet function.

**Callback queue:** It is a queue in which the piece of code is put after the event is done.

**Event Loop:** It keeps checking the callback queue, microtask queue and call stack for any pending call backs or microtasks and pushes the callback/tasks (if any) into call stack.

Ex: If a setTimeout function is there, callback from browser is added into the callback queue and Event loop constantly check if there is any callback pending or not. If there is any callback in callback queue, the Event loop will push the callback queue into the call stack.

**Microtask queue:** It is like a callback queue, but with higher priority than callback queue (i.e., the tasks in this queue are executed before the tasks in callback queue).  
All the callback functions which come from promises and mutation observer (this checks the DOM files for any changes) will go into this queue.

**Starvation of callback queue**: If there are tasks in both callback and microtask queue, and the microtask queue creates more microtasks and that new microtask creates new microtask and so on, then the callback queue will not be able to execute. This is called starvation of callback queue.

**Issues with Callbacks:**

1. **Callback Hell**: If there are many nested callbacks in a code, then the code will start growing horizontally and will become unmanageable, this is called callback hell or pyramid of doom.

Ex:  


1. **Inversion of Control:** While using callbacks, we are giving up the control of our function (or the task of running our function) to another function and we may not have any idea about other function (the other function may have multiple bugs, issues, may not call the function or may call it twice etc). This is called inversion of control.

Ex: While using setTimeout() function we are asking setTimeout function to execute a part of code or function after some time has passed, now it’s the job of setTimeout function to call our code for execution. If the setTimeout function has bugs, then it may not even call the code. This is inversion of control.

**Functional Programming**

Generally, functional programming means using functions to the best effect for creating clean and maintainable software. More specifically, functional programming is a set of approaches to coding, usually described as a programming paradigm.

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”.

**Map:** It is used to transform an array i.e., to apply a mathematical function on each element of array.  
Ex: Double/triple every element in an array.

**Filter:** It is used to filter certain elements from an array based on some logic.  
Ex: To get odd/even values from an array.

**Reduce:** This function is used to iterate through all the elements in array and get a single element.  
Ex: Getting sum of array elements, or maximum number in array.

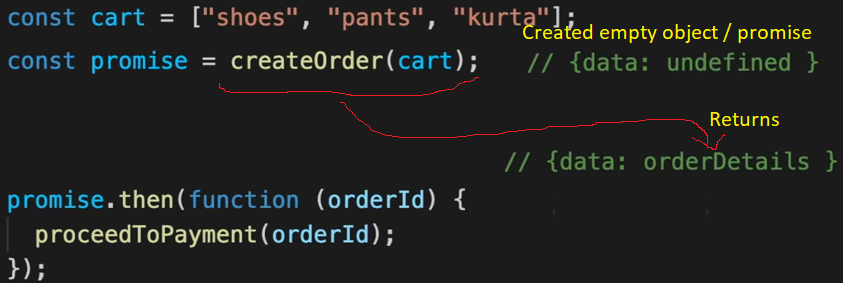
**Promises**

Promises are used to handle async operations in JS. It is a placeholder for a certain data which will be filled later with a value when we receive value from our asynchronous function.

In case of promise, the caller function will create a (promise) empty object (or dictionary) with undefined value first. The object value will be filled later with the value returned by the caller function. And during this time, the JS engine will be executing other line of the code.

The callback function is attached to the object. Now as soon as the caller function returns and fills the promise value, the JS engine will call the callback function.

Ex:



When the code is executed, createOrder creates an empty object (promise) (data: undefined). Since the createOrder function is asynchronous, it will take some time to execute and during this time JS will execute other lines of the code.  
When the createOrder is done and returns some data later, it is filled in the value and the JS engine calls the callback function linked to the promise.

**Catch Block:** It catches all the errors which have happened before this catch block while executing any promise.  
It does not take into consideration the promises below the catch block

**Contents of Promise Object:**

**Graphical user interface, text, application

Description automatically generated**

1. Prototype: This is the data type of the promise variable. i.e., it will be Promise.
2. PromiseState: It tells us if the function has returned any data or not.
   1. Pending state: If a promise is in pending state, this means that the function has not returned any data.
   2. Fulfilled: This means that the function has returned some valid data.
   3. Rejected: It means that the function has failed to return the data.

Initially the PromiseState for a promise will be pending, and after some time it will move to fulfilled/rejected.

1. PromiseResult: It contains the data which the function has returned.

**Promise Chaining:** Promise chaining is the method which helps us to avoid callback hell.

Text

Description automatically generated With Promise Using Callbacks (Without promise)

