stack and 1 is printed now.

First class functions:

function statement:

```
function a(){
    console.log("a is called");
}
```

A named function that can be invoked later in the code

function Expression:

we can assign a function to a variable. Here function acts like a value.

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

Difference:

The difference between function statement and function expression is "hoisting".

If we call a(), b() after declaring them, we will get o/p as

```
a called
b called
```

But if we call them even before declaring the function.

```
a()
b()
function a(){
    .
    .
}
var b = function(){
    .
    .
}
```

→ o/p:

```
a called
Uncaught TypeError: b is not a function
```

→ During memory creation phase function a is allocated memory and the function is assigned to a.

→ But in case of function expression, b is treated like any other variable and it is assigned "undefined" until the code hits the function line.

Function Declaration:

It is same as function statement.

Anonymous Function:

A function without a name is an anonymous function.

```
function () {
}
```

→ SyntaxError: Function statements require a function name

These anonymous functions do not have their own identity. If we create an anonymous function as above, it results in a syntax error.

Because an anonymous function looks like a function statement without a name.

→ According to ECMA script specifications, a function statement should always have a name.

Then what's the use of An. Func:

Anonymous functions are used in a place where functions are used as values.

i.e we can assign them to a variable as used in function expression.

Named function Expression:

A named function expression is a function expression where the function is assigned a name. It contrasts with anonymous function expression, which does not have a name.

```
let factorial = function fact(){
}
```

→ variable holding the reference to the function

→ fact: name of the function expression

corner case:

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

If we try to call the function with the name, we will get an error.

Reference Error: xyz is not defined

Here the name of the function is local to the function's scope. It is not created in outer scope.

It's useful for recursion or when you want to refer to the function from within itself.

Parameters vs Arguments:

Parameters are variables listed in the function definition that act as placeholders for values that the function expects when it's called.

Arguments are the actual values or inputs that are passed to the function when it's called

```
function add (a, b){      // a, b → parameters
    return a+b;
}
add (2, 3)                // 2, 3 → Arguments
```

First class functions:

In Javascript, first class functions refer to the idea that functions are treated as "first class citizens". This means that functions can be:

- → Assigned to variables
- → Passed as an Arguments to other functions
- → Returned from other functions
- → Stored in data structures like arrays or objects

Ex:

```
const greet = function(){
    console.log("Hello");
}
greet()
```

→ Assigned to greet variable

Ex:

```
function executeFunction(fn){
    fn();
}
executeFunction(function(){
    console.log('This is a function passed as an argument')
});
```

→ function passed as an argument

Ex:

```
function outerFunction(){
    return function(){
        console.log('This is returned from outer function');
    }
}
const innerFunction = outerFunction();
innerFunction()
```

→ Returning functions

Ex:

```
const arrayOfFunction = [ → Functions stored in array
    function() {console.log('First Function');},
    function () {console.log ('second function');}
];

arrayOfFunction[0]();
arrayOfFunction[1]();
```

~~Arrow functions~~:

Callback Functions: ft. Event Listeners:

A callback function is a function passed as an argument to another function and is executed after the completion of other function.

Callbacks are commonly used to handle asynchronous ~~tasks~~ operations, such as waiting for the data to be fetched from the server or when working with events.

Simple call back function:

```
function greet (name, callback){ → Main function that ~~takes~~ 2 ~~Arguments~~
    console.log ("Hello" + name);
    callback()
}
function sayGoodbye (){ → callback func executed after greet() consolelogs
    console.log (" Goodbye");
}
greet ("harsha", sayGoodbye);
```

// Output:
Hello harsha
Goodbye

→ Here the callback is executed immediately, in the same callstack.

Asynchronous callback:

```
function fetchData(callback){
    setTimeout(() => {
        console.log('Data fetched from the server');
        callback();
    }, 2000);
}

function processData(){
    console.log('Processing Data');
}

fetchData(processData);
```

// After 2 seconds
// Data fetched from the server
// processing Data

→ the callback is executed after some asynchronous operation like, fetching data, setTimeout etc.

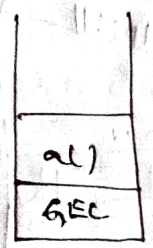
Asynchronous JavaScript & Event Loop:

How JS engine executes the code using Call stack?

JS is a synchronous single threaded language. It has one call stack. This call stack is present inside the JS engine and can do one thing at a time. All the JS code is executed inside the call stack.

Ex:

```
1 function a(){
2     console.log("a");
3 }
4 a();
5 console.log("End");
```



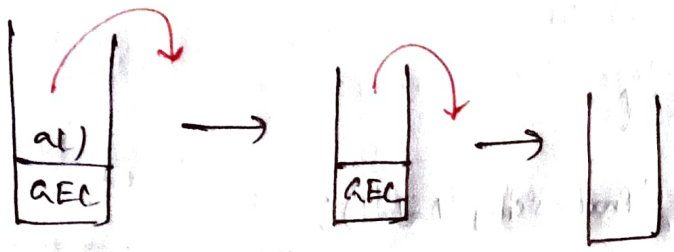
For every JS code, a GEC context is created.
When program reaches line 4, an execution context is

created and pushed in callstack.

Now the control goes into function, at line 2 'a' is printed.

After that there's nothing to execute and control returns to line 4.

Now the EC is deleted and popped out of stack.



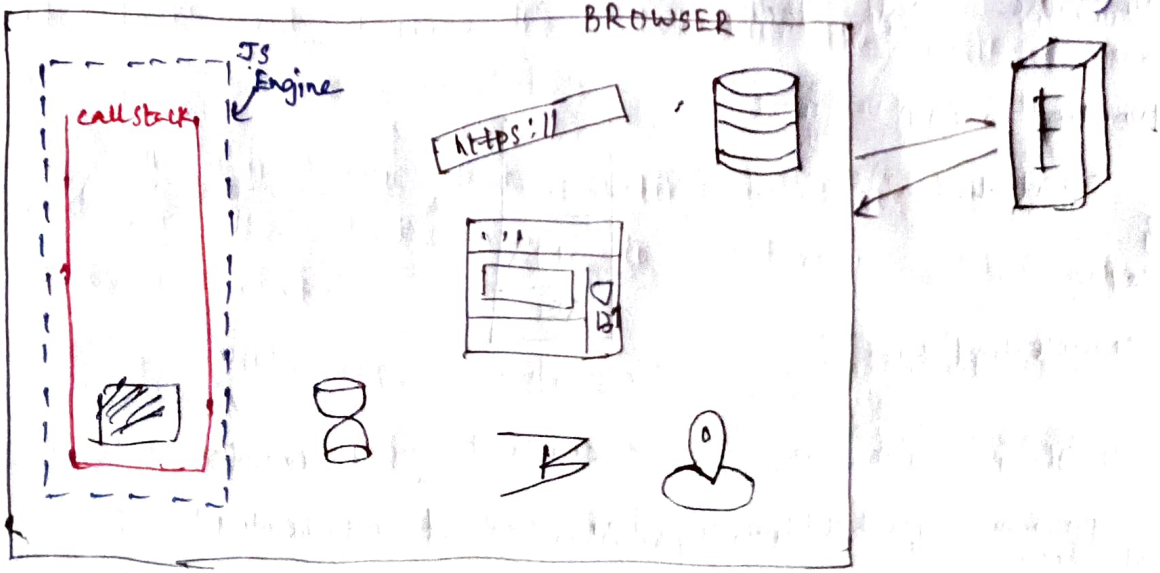
Now at line 5, 'End' is printed and program ends. GEC is also popped out of stack and call stack becomes empty.

So the callstack will immediately executes whatever comes inside it.

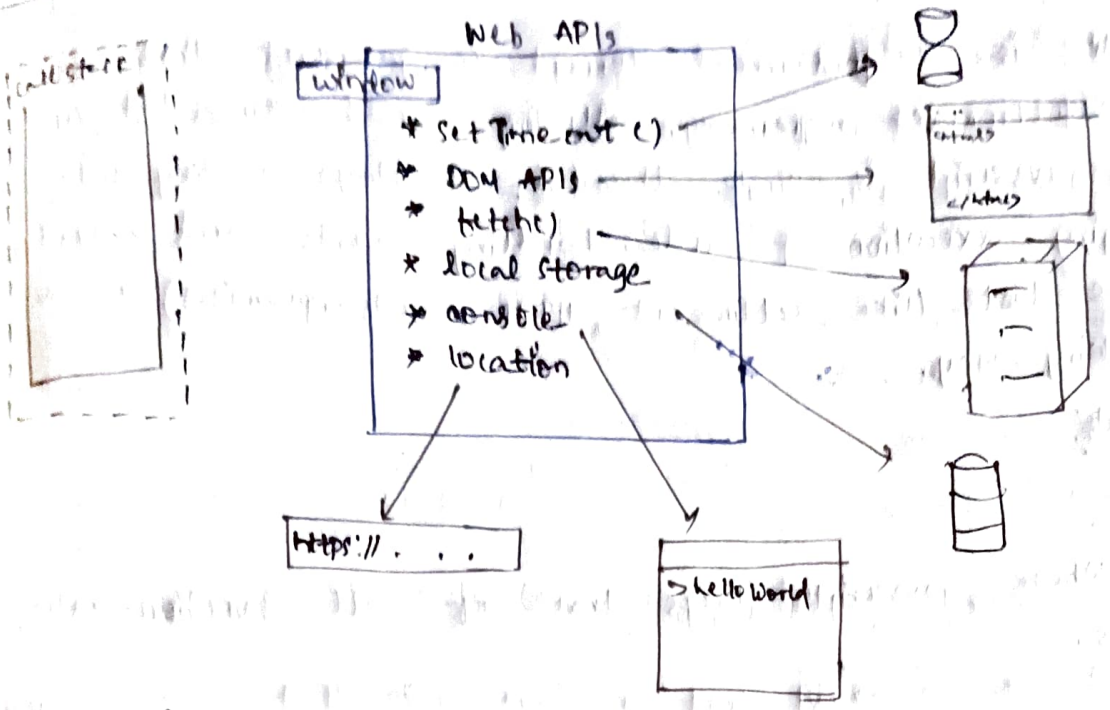
How does JS perform async tasks?

Incase if there is a script that has to be executed after certain time.

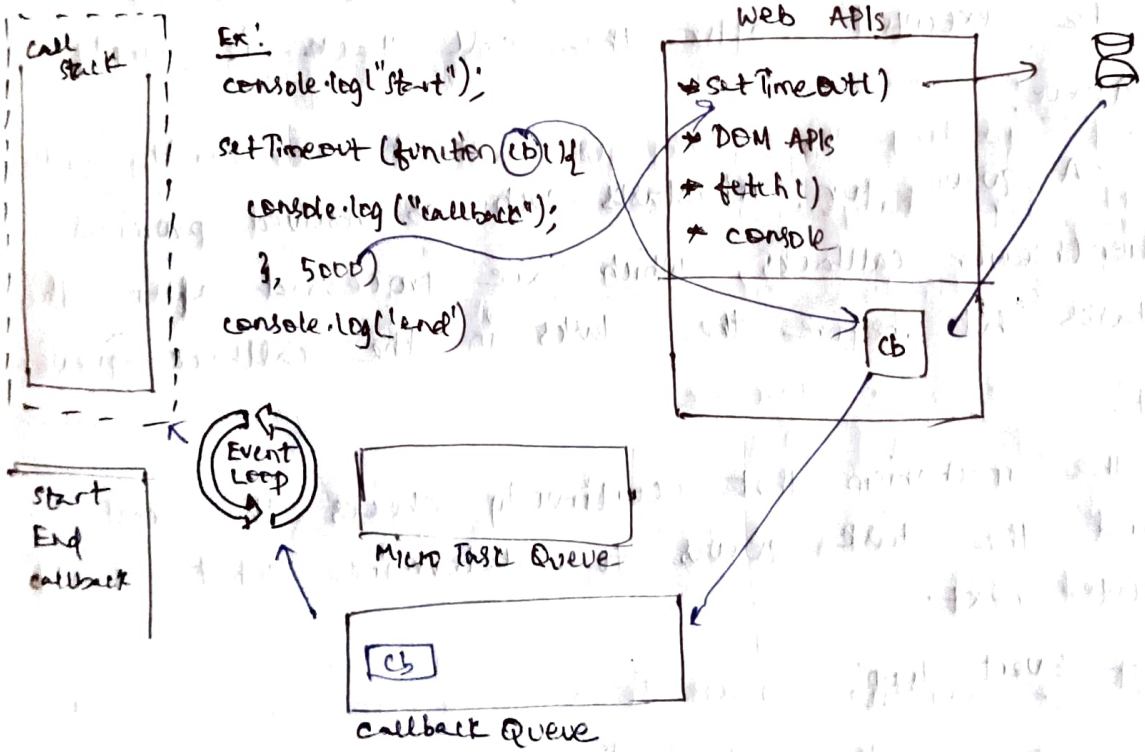
callstack cannot do it as it has no timer. Infact it is the browser who provides the timer. (Web APIs)



Web APIs:



Web APIs in Javascript are a collection of built-in interfaces and functions provided by the browser to enable interaction with web pages and the environment outside of Javascript core functionality. These are not a part of JS but are accessible via JS.



Event Loop:

Event loop is a fundamental concept in javascript that enables asynchronous programming, allowing non-blocking behavior even though javascript is single threaded. It plays a key role in managing execution of code, handling events and executing asynchronous tasks (like setTimeout, HTTP requests, or promises) in a non blocking way.

Key Concept:

Call Stack:

Where javascript keeps track of all function calls, It follows.

Web APIs:

Browser provided functionalities (like setTimeout, fetch, or DOM events) that operate asynchronously.

Callback Queue (Task Queue):

A queue that holds asynchronous callbacks waiting to be executed after the call stack is empty.

Micro Task Queue:

A queue for microtasks such as resolved promises or MutationObsewer callbacks, which are processed after the current tasks but before the tasks in the callback queue.

Event Loop:

The mechanism that continously checks the call stack and the task queues to determine what should be executed next.

Working of Event Loop:

→ Call Stack Execution:

JavaScript begins executing code in the call stack

(which contains the functions to be executed). When a function is called, it's added to the stack and when it returns a value, it is popped off the stack.

→ Handling asynchronous code:

If the code encounters an asynchronous operation (like setTimeout or an HTTP request), the Web APIs handle it. The asynchronous task is delegated to the browser environment, and JavaScript moves onto the next task without waiting for it to finish.

→ Callback Queue:

When the asynchronous operation completes (eg: the timer expires, or an HTTP request is finished), the corresponding callback is placed in the callback queue.

→ Microtask Queue:

Microtasks (resolved promises) are given a higher priority than normal tasks. After the current task finishes executing and before the event loop processes the callback queue, the event loop checks the microtask queue and processes all micro tasks in it.

Ex:

```
console.log('Start');
setTimeout(() => {
    console.log('Timeout callback');
}, 1000);
Promise.resolve().then(() => {
    console.log('promise callback');
});
console.log('End');
```

Start is printed first (synchronous)

setTimeout is called, but its callback is placed in the callback queue to be executed later.

Promise is resolved and its callback is placed in the Microtask Queue.

End is printed next (synch...)

The event loop processes the microtask queue, executing the promise callback.

Finally the event loop processes the callback queue, executing the setTimeout callback.

O/p:

Start
End
promise callback
Timeout callback

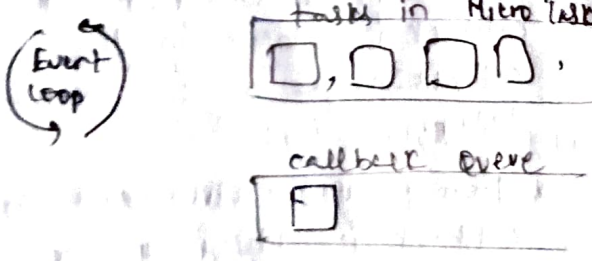
Mutation Observer:

It is a powerful tool in Javascript that allows you to observe changes in the DOM and respond to them. It watches for changes to a specified DOM element and executes a callback function when those changes occur. It can detect changes such as attributes, childNodes or text content of an element.

Uses:

Dynamically updating the webpages without constantly polling the DOM for changes.

Starvation of functions in callback queue:



Starvation of functions refers to a scenario where certain tasks (callbacks), in the callback queue are delayed or never executed because higher-priority tasks like

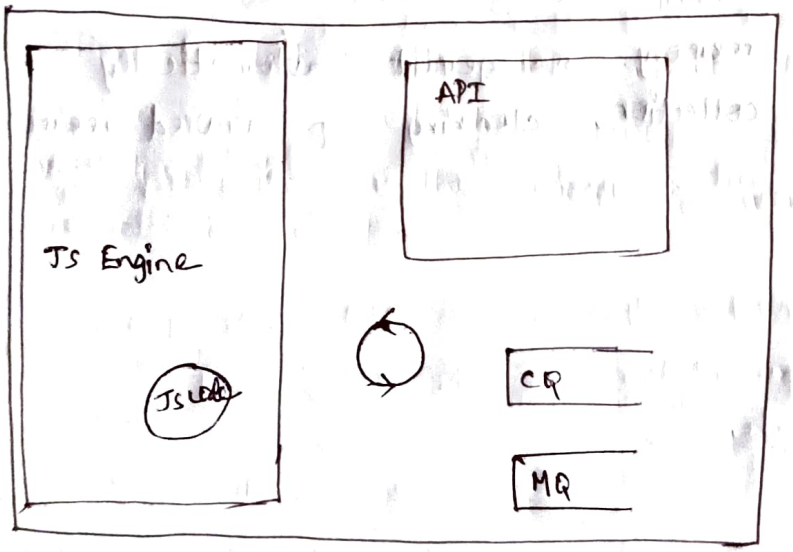
(micro tasks) or long running synchronous code keep the event loop busy. This can lead to certain asynchronous functions waiting indefinitely in the queue for the call stack to become available.

Microtasks blocking callback Execution:

Microtasks have higher priority than callbacks in the event loop. If new microtasks are constantly being added to the queue, they will keep executing preventing any callbacks from the callback queue from running.

So, To avoid starvation of callbacks, you can breakup tasks into smaller chunks, or introduce breaks in microtask chain using setTimeout() or requestAnimationFrame() to allow the event loop to process other tasks.

JavaScript Runtime Environment:



Provides the environment in which javascript code is executed. It includes the necessary components like the engine to interpret and execute the code, along with API and functionalities to interact with the system or browser. This run time environment typically consists of two key parts.

→ Js Engine:

The engine that executes Javascript code. In browsers, this is usually the V8 engine (Chrome, Node.js), SpiderMonkey (Firefox), JavaScriptCore (Safari).

→ Runtime APIs:

These are the additional functionalities provided by the environment, allowing Javascript to interact with the things outside of its core language (DOM in the browser, file system access in Node.js)

Components of Javascript Runtime Environment:

→ Js engine
→ Call stack
→ Heap:
  ↳ Heap is where objects and variables are stored in memory.
  ↳ Javascript handles memory management automatically through garbage collection, cleaning up unused memory.
→ Event loop
→ Callback Queue
→ Microtask Queue

JSRE in a browser:

It includes
→ Js engine
→ Web APIs (DOM, fetch, setTimeout, Event listeners)
→ CB Queue, MT Queue
→ Event loop

Ex:

```
console.log('start');
setTimeout(() => {
    console.log('Timer called');
}, 5000)

Promise.resolve().then(() => {
    console.log('Promise Resolved');
})
console.log('End');
```

JSRE in node.js:

It includes

→ JS Engine

→ Node.js APIs

  ↳ File system : Reading & writing files

  ↳ Networking : Handling n/w requests

  ↳ Child Processes : Running external processes

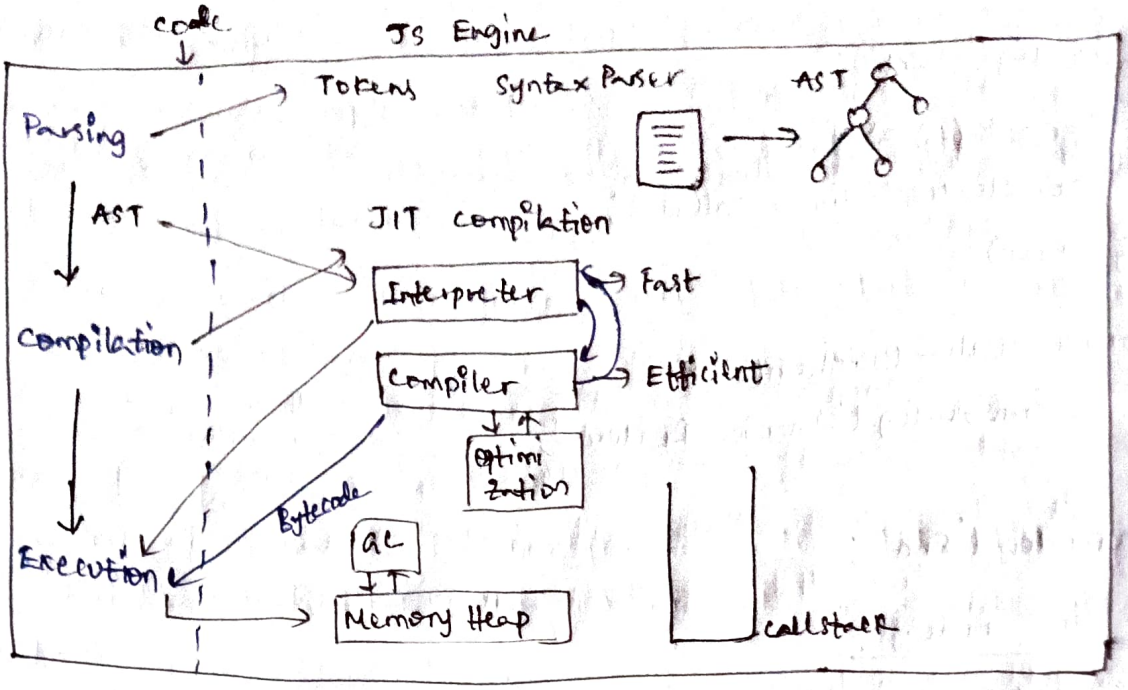
→ Event loop & Queues

Ex:

```
const fs = require('fs')
console.log('start');
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log('File content:', data);
});
console.log('end');
```

JS Engine Architecture:

A JS engine is a program that executes javascript code. It responsible for parsing, interpreting, and executing javascript and its a crucial part of the runtime environment.



Parsing:

→ The parser takes raw javascript code and passes it into an Abstract Syntax Tree (AST).

→ AST is a hierarchial tree structure that represents the program's syntax and structure. Its easier for the engine to optimize and work with than raw code.

Interpreter:

→ Initially, most Javascript engines used an interpreter to execute the AST.

→ The interpreter translates the parsed AST into bytecode, a lower-level representation of the code, which is more efficient to execute.

JIT compilation:

→ For frequently executed code, the engine optimizes the execution by using a JIT compiler.

→ The JIT compiler converts byte code to machine code which the CPU can directly execute. This dramatically improves performance, especially in long running applications.

-> There are two phases in JIT compilation

↳ Baseline compiler: Quickly generates machine code without much optimization. This is faster but less efficient.

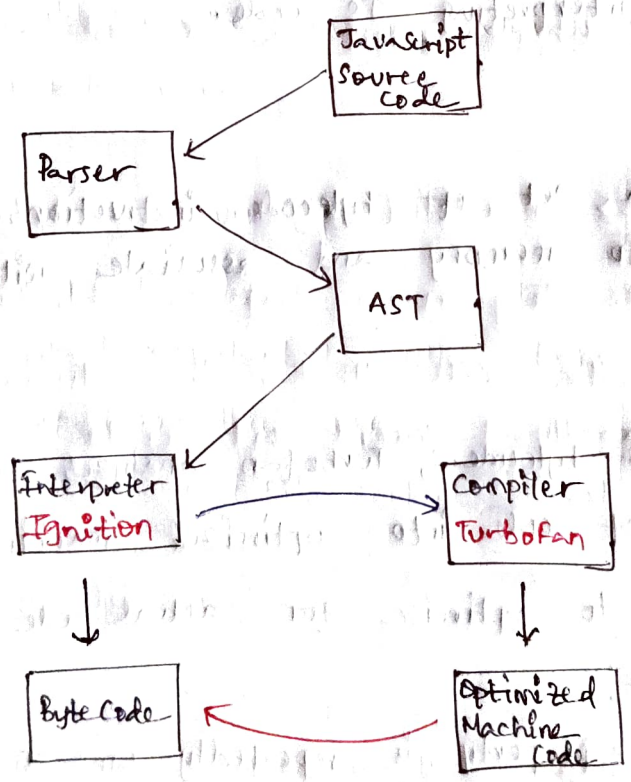
↳ Optimizing compiler: For hot code paths, the optimizing compiler applies various optimizations like inline caching, function inlining, and loop unrolling.

Garbage collection:

JS engine employs a garbage collector to automatically manage memory. This ensures that memory used by variables and objects that are no longer needed is reclaimed.

→ The garbage collector operates in the background to free up memory while minimizing the impact on application performance.

V8 JS engine Architecture:



Parsing and Lexing:

↳ **Lexer / Tokenizer:** Breaks down the source code into meaningful chunks called "tokens."

↳ **Parser:** converts the tokens into AST, which is an internal representation of the code's structure.

```
const a = 5;
```

↓

This would be transformed into AST, which represents this variable declaration and assignment in a tree like format.

Ignition:

→ Ignition is the interpreter in V8 that takes AST and turns into bytecode which is a lower-level platform-independent representation of the code, optimized for faster execution compared to directly interpreting JS code.

Ex: `let x = 10`

↓

translated to a set of bytecode instructions that load value of 10 into memory and associate with variable x.

Turbofan: (V8's compiler)

→ After ignition generates bytecode, Turbofan analyzes and compiles "hot" sections of code into optimized machine code.

→ It uses runtime data to optimize for actual code execution. It applies techniques like:

Inline Caching: If a property is repeatedly on an object, Turbofan generates optimized code that assumes the property will exist in same place.

Function Inlining:

Small functions are inlined, removing the

overhead of a function call.

The machine code executed by Turbofan can be executed directly by the CPU, making it faster than interpreted byte code.

Garbage collection:

→ V8 uses a generational garbage collector, meaning it seperates objects into two groups.

↳ Young generation: Objects that are recently created and expected to be short lived.

↳ Old generation: objects that have been around longer and are likely to stay in memory for a while.

The Scavenge algorithm is used for young generation and Mark-Sweep-Compact is used for old generation This strategy helps minimize the performance impact of garbage collection by focusing most of the clean up efforts on short-lived objects which are cheaper to collect.

Key features of Javascript engine optimization:

→ Hidden classes and Inline caching:

↳ V8 optimizes object property access by creating hidden classes for similar objects. When accessing properties, the engine can optimize repeated access by caching the property locations (inline caching)

→ Deoptimization:

If assumptions made by JIT compiler (like the data type of a variable) are found to be wrong at runtime, the engine can "deoptimize" the code and revert to a less optimized version.

Lazy Compilation:

The engine compiles code only when it's needed. This reduces initial load time for large applications since not all functions need to be compiled upfront.

Issues with setTimeout():

Issues with setTimeout() generally refer to the "unreliable timing" that can occur with setTimeout function. This is due to how javascript's "single threaded" event loop operates, which can lead to delay in execution of tasks when the main thread is busy.

Inaccuracy of delays:

The delay specified in setTimeout (1000 ms) is not guaranteed to be exact. It specifies the minimum time to wait before the callback is executed. If the call stack is busy, the callback might be delayed even further.

Ex:

```
setTimeout(() => console.log("Executed after 1 second"), 1000);
```

If other tasks like long running loops or blocking operations are on the call stack, the timeout callback might not run at exactly 1 second but could be delayed.

Ex:

```
console.log('start');
setTimeout(() => {
  console.log('callback');
}, 5000);
console.log('end');
```

```
let startDate = new Date().getTime();
let endDate = startDate;
while (endDate < startDate + 10){
        endDate = new Date().getTime();
}
console.log('while expires');
```

Here in this example, the callback function will wait for 5 seconds. Meanwhile, while loop will not exit for 10 seconds.

So even if the 5 sec timer expires, the callback func will not be executed as call stack is busy with while loop.

o/p:

start
end
while expires → after 10 sec
callback

setTimeout(0):

```
console.log('start');
setTimeout(() => {
    console.log("callback");
}, 0);
console.log('end')
```

//
start
end
callback

Although the delay is set to 0 milliseconds, the function won't execute immediately - it's placed in queue and will only run once all the current tasks in the call stack are finished.

Higher order functions:

A higher order function is a function that does atleast one of the following:

→ Takes another function as an argument
→ Returns a function as a result

Ex:

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

x → callback function
y → y is higher order function

Introduction to functional Programming:

Map, Filter, Reduce:

Map:

→ Transforms every element in an array based on a given function.
→ Returns a new array with each element modified by the provided function.

Ex:

```
const numbers = [1, 2, 3, 4]
const doubled = numbers.map(num => num * 2)
console.log(doubled) // [2, 4, 6, 8]
```

Filter:

Filters out elements in an array that don't meet certain condition.

→ Returns a new array with elements that pass the condition specified by the provided function.

```
const arr = [5, 1, 2, 3, 6];

function isOdd (x){
    return x%2 ;
}

const output = arr.filter (isOdd) // [5, 1, 3]
```

Reduce:

Reduces an array to a single value by executing a reducer function on each element. Returns a single accumulated value.

Ex:

To find sum of elements in array

```
const arr = [5, 1, 3, 2, 6]

function sum (arr){
    let sum = 0;
    for(let i=0; i<arr.length; i++){
        sum = sum + arr[i];
    }
    return sum;
}
```

```
const arr = [5, 1, 2, 3, 6]

const sum = arr.reduce (function (acc, curr){
        acc = acc + curr;
        return acc;
    }, 0);
```

↳ initial value to accumulator

acc: It accumulates the result of the function as the array is iterated over.

The value of the accumulator is initialized based on the second argument passed to the function.

curr:

It represents the current element of the array being processed during the iteration.