Important Terms to Understand Web Development-:

1. //HTML Parsers
2. //CSS Parsers
3. //v8 Engine - Chrome

4. // Multi-Threading - Thread
5. // is JS Multi Thread?
6. //Can we use JS to connect a DB?
7. //Can we use JS to access FS?
8. //Can we use JS to start a web server?
9. // C/C++ + v8 = Node JS
10. //Can we install JS as standalone compiler?
11. //v8 => Events on Click, set Timeout,
12. // Blocking vs Non-Blocking, Sync vs Async

13. //Browser + Event Loop/Thread = Web Browser
14. // v8 + C++ (Libuv)
15. // Call Stack = LIFO/FILO

1. **HTML Parsers :** HTML parsers are software for automated Hypertext Markup Language (HTML) parsing. They have two main purposes:

- HTML traversal: offer an interface for programmers to easily access and modify the "HTML string code". Canonical example: DOM parsers.
- HTML clean: to fix invalid HTML and to improve the layout and indent style of the resulting markup. Canonical example: HTML Tidy.

Example

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag (self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag (self, tag):
        print("Encountered an end tag :", tag)

    def handle_data (self, data):
        print ("Encountered some data  :", data)

parser = MyHTMLParser ()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The Output:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data: Test
Encountered an end tag: title
Encountered an end tag: head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data: Parse me!
Encountered an end tag: h1
Encountered an end tag: body
Encountered an end tag: html
```

2. //CSS Parsers:

The CSS Parser is implemented as a package of Java classes, that inputs Cascading Style Sheets source text and outputs a Document Object Model Level 2 Style tree. Alternatively, applications can use SAC: The Simple API for CSS. Its purpose is to allow developers working with Java to incorporate Cascading Style Sheet information, primarily in conjunction with XML application developments.

3. ///v8 Engine Chrome**:**

V8 is a powerful open source JavaScript engine provided by Google. So what actually is a JavaScript Engine? It is a program that converts JavaScript code into lower level or machine code that microprocessors can understand. All of our systems consist of microprocessors, the thing that is sitting inside your computer right now and allowing you to read this. Microprocessors are tiny machines that work with electrical signals and ultimately do the job. We give microprocessors the instructions. The instructions are in the language that microprocessors can interpret. Different microprocessors speak different languages. Some of the most common are IA-32, x86–64, MIPS, and ARM. These languages directly interact with the hardware so the code written in them is called machine code. Code that we write on our computers is converted or compiled into machine code.

There are different JavaScript engines including Rhino, JavaScriptCore, and SpiderMonkey. These engines follow the ECMAScript Standards. ECMAScript defines the standard for the scripting language. JavaScript is based on ECMAScript standards. These standards define how the language should work and what features it should have. The V8 engine is written in C++ and used in Chrome and Nodejs.

4. Multithreads and Single threads:

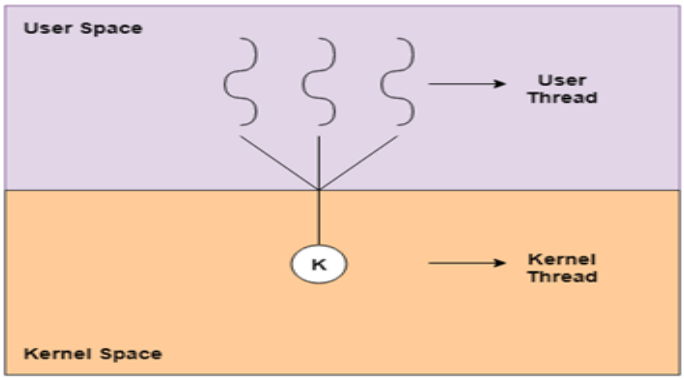
Single Thread Processes:

Single threaded processes contain the execution of instructions in a single sequence. In other words, one command is processes at a time.

The opposite of single threaded processes are multithreaded processes. These processes allow the execution of multiple parts of a program at the same time. These are lightweight processes available within the process.

Multithreaded Processes Implementation:

Multithreaded processes can be implemented as user-level threads or kernel-level threads. Details about these are provided using the following diagram −



User-level Threads

The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. Also, there is no kernel involvement in synchronization for user-level threads.

Kernel-level Threads

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.

Advantages of Multithreaded Processes

Some of the advantages of multithreaded processes are given as follows −

- All the threads of a process share its resources such as memory, data, files etc. A single application can have different threads within the same address space using resource sharing.
- It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time consuming to create processes as they require more memory and resources.
- Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation.
- In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This

is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.

Disadvantages of Multithreaded Processes

Some of the disadvantages of multithreaded processes are given as follows −

- Multithreaded processes are quite complicated. Coding for these can only be handled by expert programmers.
- It is difficult to handle concurrency in multithreaded processes. This may lead to complications and future problems.
- Identification and correction of errors is much more difficult in multithreaded processes as compared to single threaded processes.

5. is JS Multi Thread ?

- JavaScript does not support multi-threading because the JavaScript interpreter in the browser is a single thread (AFAIK). Even Google Chrome will not let a single web page's JavaScript run concurrently because this would cause massive concurrency issues in existing web pages. All Chrome does is separate multiple components (different tabs, plug-ins, etcetera) into separate processes.
- You can however use, as was suggested, setTimeout to allow some sort of scheduling and "fake" concurrency. This causes the browser to regain control of the rendering thread, and start the JavaScript code supplied to setTimeout after the given number of milliseconds. This is very useful if you want to allow the viewport (what you see) to refresh while performing operations on it. Just looping through e.g. coordinates and updating an element accordingly will just let you see the start and end positions, and nothing in between.

6. Can we use JS to connect a DB?

JavaScript can't directly access the database. You'll need to have some server-side component that takes requests (probably via HTTP), parses them and returns the requested data.

7. Can we use JS to JS to access FS?

8. Can we use JS to start a web server?

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server. Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

```
var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) {   // 2 - creating
server

    //handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')
```

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

```
C:\> node server.js

Node.js web server at port 5000 is running..
```

9. **Can we install JS as stand alone compiler ?**

   No.

10. **Blocking vs Non Blocking, Sync vs Async**

This overview covers the difference between **blocking** and **non-blocking** calls in Node.js. This overview will refer to the event loop and libuv but no prior knowledge of those topics is required. Readers are assumed to have a basic understanding of the JavaScript language and Node.js callback pattern.

"I/O" refers primarily to interaction with the system's disk and network supported by libuv.

Blocking

Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as **blocking**. Synchronous methods in the Node.js standard library that use libuv

are the most commonly used **blocking** operations. Native modules may also have **blocking** methods.

All of the I/O methods in the Node.js standard library provides asynchronous versions, which are **non-blocking**, and accept callback functions. Some methods also have **blocking** counterparts, which have names that end with Sync.

Comparing Code

Blocking methods execute **synchronously** and **non-blocking** methods execute **asynchronously**.

Using the File System module as an example, this is a **synchronous** file read:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

And here is an equivalent **asynchronous** example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

The first example appears simpler than the second but has the disadvantage of the second line **blocking** the execution of any additional JavaScript until the entire file is read. Note that in the synchronous version if an error is thrown it will need to be caught or the process will crash. In the asynchronous version, it is up to the author to decide whether an error should throw as shown.

Let's expand our example a little bit:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

And here is a similar, but not equivalent asynchronous example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

In the first example above, console.log will be called before moreWork(). In the second example fs.readFile() is **non-blocking** so JavaScript execution can continue and moreWork() will be called first. The ability to run moreWork() without waiting for the file read to complete is a key design choice that allows for higher throughput.

Concurrency and Throughput

JavaScript execution in Node.js is single threaded, so concurrency refers to the event loop's capacity to execute JavaScript callback functions after completing other work. Any code that is expected to run in a concurrent manner must allow the event loop to continue running as non-JavaScript operations, like I/O, are occurring.

As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing **non-blocking** asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use **non-blocking** methods instead of **blocking** methods.

The event loop is different than models in many other languages where additional threads may be created to handle concurrent work.

Dangers of Mixing Blocking and Non-Blocking Code

There are some patterns that should be avoided when dealing with I/O. Let's look at an example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
fs.unlinkSync('/file.md');
```

In the above example, fs.unlinkSync() is likely to be run before fs.readFile(), which would delete file.md before it is actually read. A better way to write this, which is completely **non-blocking** and guaranteed to execute in the correct order is:

```
const fs = require('fs');
fs.readFile('/file.md', (readFileErr, data) => {
  if (readFileErr) throw readFileErr;
  console.log(data);
  fs.unlink('/file.md', (unlinkErr) => {
    if (unlinkErr) throw unlinkErr;
  });
});
```

The above places a **non-blocking** call to fs.unlink() within the callback of fs.readFile() which guarantees the correct order of operations.

11. Call Stack = LIFO/FILO

The call stack is primarily used for function invocation (call). Since the call stack is single, function(s) execution, is done, one at a time, from top to bottom. It means the call stack is synchronous.

In Asynchronous JavaScript, we have a callback function, an event loop, and a task queue. The callback function is acted upon by the call stack during execution after the call back function has been pushed to the stack by the event loop.

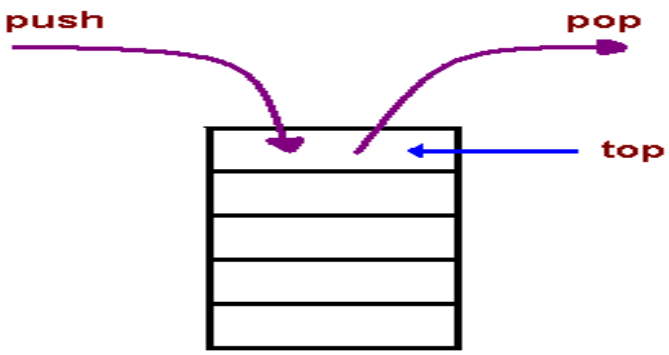
What is the call stack?

At the most basic level, a call stack is a data structure that uses the Last In, First Out (LIFO) principle to temporarily store and manage function invocation (call).

Let's break down our definition:

LIFO: When we say that the call stack, operates by the data structure principle of Last In, First Out, it means that the last function that gets pushed into the stack is the first to be pop out, when the function returns.

Temporarily store: When a function is invoked (called), the function, its parameters, and variables are pushed into the call stack to form a stack frame. This stack frame is a memory location in the stack. The memory is cleared when the function returns as it is pop out of the stack.



I

Manage function invocation (call): The call stack maintains a record of the position of each stack frame. It knows the next function to be executed (and will remove it after execution). This is what makes code execution in JavaScript synchronous.
Think of yourself standing on a queue, in a grocery store cash point. You can only be attended to after the person in front of you have been attended to. That's synchronous.