

Async JS

🕒 Created	@May 4, 2023 8:25 PM
📁 Class	
📁 Type	
📎 Materials	
☑ Reviewed	<input type="checkbox"/>

What is a thread ?

All of us know that if we make a program like `test.js` and run it in our computer it becomes a process i.e. program in a running state is a process.

Let's take a simple example:

If a man takes `X` days to complete a job, how many days will it take for `Y` men to complete the same job ?

Ans : X / Y days because we are assuming all of them will work parallel and divide the task among themselves.

Similar concepts is applicable for computers also. Computer have got different mechanisms to achieve parallel computations.

Threads are execution entity, very much like process, but they are extensively very light weight process.

Let's write a very simple program

```
i = 0;
sum = 0;
while(i <= 1000000000) {
  sum += i;
  i++;
}
```

Now if we run the above code on a machine by let's say saving it as a C++ code or JS code, then what will happen is, this program will be loaded in the memory and become a process and will be running on any one of the cores of our processor. Assuming we have an Octa core processor, we will be having 7 other cores sitting idle.

So what the above program is actually doing ? It is simply adding the first 100000000 naturals numbers.

How about we try to do some parallelisation, i.e. how about we try to divide our task of summing up first 100000000 natural numbers between all of our 8 processor cores.

Then there will be 8 process running in different cores which will sum up the things in parallel fashion.

This kind of a model is called as `Processor model`.

Now there is something called as `Thread Mode`.

In thread mode what we will do is we create 1 process with 8 threads, and each thread does 1/8th of the given work. It will be equivalently fast as running it on 8 different processors.

One interesting fact about a thread is that, all the thread maintains their own call stack (and it's obvious right ? They are just light weight process).

You might think that, won't it be slow to run 8 threads as compared to running process on 8 different cores ?

It won't be that slow, as threads are very light weight on memory, the context switching is very fast compared to normal process, and threads can also communicate within themselves.

Real life example:

On earlier mobile phone like android, the main application that you used to see generally used to run on something called as Main thread. Now the app has to download some data, then we should not download the data on the main thread. For this we can make another thread and download data over that thread leaving the main thread unblocked.

Characteristics Of JS:

- Javascript is single threaded (then how does it manage time consuming tasks, like downloading some data ????)
- JS supports Synchronous execution of code (JS will execute everything one after another i.e. if you have some time consuming task, JS will wait for it to complete and then only move forward)

```
function blockingTimeConsumingCode() {  
    for(let i = 0; i < 10000000000; i++) {}  
}  
console.log("Start");  
blockingTimeConsumingCode();  
console.log("End");
```

In this piece of code, JS is running everything one after another, the for loop is a blocking piece of code, so JS has to wait for it to complete before it can move forward. That means this time consuming for loop is not getting executed in parallel fashion, it is blocking our main thread.

Note: The above statement of synchronous execution is only applicable for those piece of code which is native and known to JS for ex: while loop, or for loop etc.

Now you must be thinking, is there anything that JS doesn't know natively? Yes, for example, `setTimeout` function you can execute in your browser's JS console, but this function is nowhere mentioned in the official JS docs. That means, official JS is not shipped with this function. JS is somehow able to execute it, but it is not native to JS.

So the above blocking behaviour or synchronous behaviours will not be shown for non native features.

```
console.log("Start");  
setTimeout(function() {  
    console.log("Completed timer");  
}, 10000);  
console.log("End");
```

In this piece of code, JS doesn't wait for the timer to get completed in 10 sec, and instead moves forward, giving you an output like

```
Start
End
Completed timer
```

Now you should be getting two questions in your mind i.e.

- if `setTimeout` is not native and known to JS, how come it is able to execute it ?
- And how does JS handles execution of these non native features? Doesn't it become unpredictable ?

Runtime Environment

In order to understand how exactly JS is running non native functions, we need to first of all understand from where these are coming up ?

A runtime environment, is where your code/program will be executed and while execution this environment is going to provide multiple different aids in order to add more functionalities to our code. This runtime environment understands how the code should be run and what capabilities might be useful for the program during runtime.

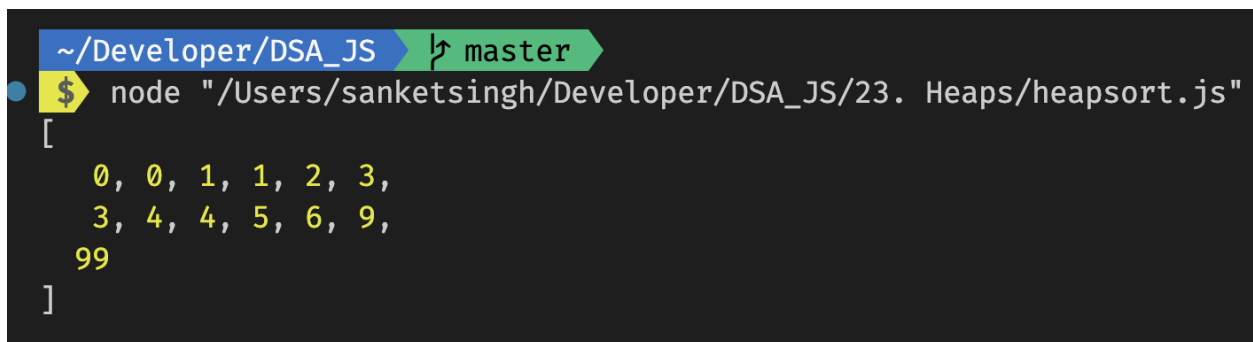
For example: We can run our JS code in a browser, here browser is a Runtime environment. Browser provides additional capabilities to JS in order to run in a efficient way and be more productive. For ex: Browser provides JS functionalities like

- How to run a timer ? So functionalities like `setTimeout` or `setInterval` these are actually provided to JS through the browser runtime.
- Browser also gives JS a functionality to download some data over the network, using the `fetch` or `XMLHttpRequest` functions.
- Browser also provides capabilities using which JS can interact with the DOM or HTML.

Similar to browser we have other runtime environments as well where we can run JS, for example: NodeJS (NodeJS is neither a language nor a framework, instead it is a Runtime environment).

What capabilities NodeJS provide ?

Using NodeJS We can execute JS out of browser and inside our terminal. So whenever you are running a JS code inside VS-Code, then you are actually running it in the NodeJS runtime.

A screenshot of a VS Code terminal window. The top bar shows the file path ~/Developer/DSA_JS and the branch master. The terminal prompt is \$, and the command entered is node "/Users/sanketsingh/Developer/DSA_JS/23. Heaps/heapsort.js". The output of the script is an array of numbers: [0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 6, 9, 99].

```
~/Developer/DSA_JS master  
$ node "/Users/sanketsingh/Developer/DSA_JS/23. Heaps/heapsort.js"  
[  
  0, 0, 1, 1, 2, 3,  
  3, 4, 4, 5, 6, 9,  
  99  
]
```

So here when we actually run the JS file in VS-Code it runs it with the following command:

```
node file.js
```

So here we are invoking the NodeJS env to run our file.

Now as NodeJS is a different runtime, it will be having different behaviour than browser. Because now we are not running JS in browser, so there is no point of giving browser based functionalities like reading html. Instead we give it other functionalities like

- Inside NodeJS , Javascript will be able to read files from our File system.
- It gives access to process level details.

Apart from some new functionalities there can some similar functionalities also.

- NodeJS also provides access to timers.

Note: As I said that functionalities can be similar , they are not necessarily same.

Thats why if you run setTimeout in browser it returns you and ID of the timer which is a Number, where as in NodeJS, it returns an object.

Note: Even two different browsers can act as two different runtimes. Although now days most of the functionalities are consistent in major browsers.

So this technically answers that how JS gets these functions like `setTimeout`

How JS handles the execution of Runtime features?

So, till now you must be already aware that when you run a JS code, it becomes a process. For a process we have access to a memory area in which there are multiple components like `Call Stack`

But there are few other components as well that will actually unravel this mystery for us. These components are

- Macrotask Queue
- MicroTask Queue
- Event Loop