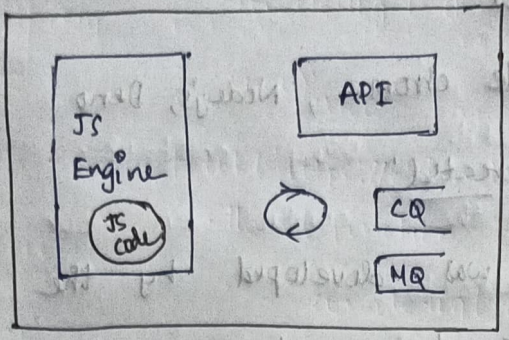
JavaScript Engine

JavaScript Runtime Environment:

A sort of container which has all the things required to run JavaScript code.



→ JavaScript Runtime Env. is not possible without the JS engine.

→ Every browser has JS RE

Browser and Node JS runtime:

→ Node JS is open source JS RE and can run JS code outside the browser.

→ API: provide the features to use inside JS code. These are different inside browser and node JS. Sometimes they are common too.

Ex:

↳ Browser has an API called local storage which gives access to local storage of the browser from the code and in case of node JS, it could be different.

↳ setTimeout API is present in the browser as well as node JS.

↳ console also present in both browser and node JS.

↳ These names may seem to be similar but internally they are implemented differently.

List of JS engines:

→ All browsers have their own JS engine.

Chakra - Microsoft Edge

SpiderMonkey - Firefox.

V8 - Google chrome, Node.js, Deno

First JS engine ever created:

→ first JS engine was developed by the creator of JS. This engine has evolved a lot and now it is known as spiderMonkey.

Myths about JS engine:

↳ JS engine is not a machine.

↳ It's just a normal program written in low level languages.

Ex: Google's V8 is written in C++

Namaste JavaScript - 52

Callbacks!

Callbacks can be used to perform asynchronous tasks in JS.

Ex!

```
const cart = ["shoes", "pant", "shirt"];

api.createOrder(function() {

    api.proceedToPayment();

});
```

↳ we have given callback function to createOrder API and it is the responsibility of createOrder API to create an order and then call the function back.

↳ This way, using callbacks, we can handle async tasks in JS.

If we attach more tasks:

↳ create order

↳ proceed to payment

↳ show Order Summary

↳ updat wallet

here we can see a dependency flow where every inner function is dependent on outer function. In large code bases, APIs are dependent one upon the other. This phenomenon of one callback inside another callback is known as "callback Hell"

→ In such situations the code grows horizontally but not vertically.

→ This type of code structure is unreadable and unmaintainable.

Inversion of control:

→ Another problem while using callbacks.

→ This means, loosing the control on our code when using callbacks.

Ex:

```
api.createOrder(function() {

    api.proceedToPayment();

});
```

Here, we are creating the order and take the callback function, gave it to createOrder API.

Now, a developer is blindly trusting createOrder API that it will create an order and at some point of time will call the function back. This is very risky as createOrder API must be in some other service or there could be a lot of bugs or it could never be called.

We are giving control of our code to some other code and we might not know what happens behind the scenes.