//FIRST CLASS FUNCTIONS

/\* When functions can be treated like any other variable then those functions are first-class functions. There are many other programming languages, for example, scala, Haskell, etc which follow this including JS. Now because of this function can be passed as a param to another function(callback) or a function can return another function(higher-order function). map() and filter() are higher-order functions that are popularly used. \*/

//NODEJS

/\* Node.js is a virtual machine that uses JavaScript as its scripting language and runs Chrome’s V8 JavaScript engine.It is based on an event - driven architecture where I / O runs asynchronously making it lightweight and efficient.Node.js provides simplicity in development because of its non - blocking I / O and event - based model results in short response time and concurrent processing, unlike other frameworks where developers have to use thread management. It runs on a chrome v8 engine which is written in c++ and is highly performant with constant improvement. \*/

//NODEJS CONTROL FLOW

/\* https://medium.com/@gunendu/control-flow-in-nodejs-8e855996a5c7 \*/

//TIMERS

/\* setTimeout() can be used to schedule code execution after a designated amount of milliseconds. It accepts a function to execute as its first argument and the millisecond delay defined as a number as the second argument. Additional arguments may also be included and these will be passed on to the function. Here is an example of that: \*/

function myFunc(arg) {

console.log(`arg was => ${arg}`);

}

setTimeout(myFunc, 1500, 'funky');

/\* The above function myFunc() will execute as close to 1500 milliseconds as possible due to the call of setTimeout(). The timeout interval that is set cannot be relied upon to execute after that exact number of milliseconds.This is because other executing code that blocks or holds onto the event loop will push the execution of the timeout back. The only guarantee is that the timeout will not execute sooner than the declared timeout interval. setTimeout() returns a Timeout object that can be used to reference the timeout that was set.This returned object can be used to cancel the timeout as well as change the execution behavior.

setImmediate() will execute code at the end of the current event loop cycle. This code will execute after any I/O operations in the current event loop and before any timers scheduled for the next event loop. This code execution could be thought of as happening "right after this", meaning any code following the setImmediate() function call will execute before the setImmediate() function argument. The first argument to setImmediate() will be the function to execute. Any subsequent arguments will be passed to the function when it is executed. Here's an example: \*/

console.log('before immediate');

setImmediate((arg) => {

console.log(`executing immediate: ${arg}`);

}, 'so immediate');

console.log('after immediate');

/\* The above function passed to setImmediate() will execute after all runnable code has executed, and the console output will be:

before immediate

after immediate

executing immediate: so immediate

setImmediate() returns an Immediate object, which can be used to cancel the scheduled immediate.

If there is a block of code that should execute multiple times, setInterval() can be used to execute that code. setInterval() takes a function argument that will run an infinite number of times with a given millisecond delay as the second argument. Just like setTimeout(), additional arguments can be added beyond the delay, and these will be passed on to the function call. Also like setTimeout(), the delay cannot be guaranteed because of operations that may hold on to the event loop, and therefore should be treated as an approximate delay. See the below example: \*/

function intervalFunc() {

console.log('Cant stop me now!');

}

setInterval(intervalFunc, 1500);

/\* In the above example, intervalFunc() will execute about every 1500 milliseconds, or 1.5 seconds, until it is stopped. Just like setTimeout(), setInterval() also returns a Timeout object which can be used to reference and modify the interval that was set.

What can be done if a Timeout or Immediate object needs to be cancelled? setTimeout(), setImmediate(), and setInterval() return a timer object that can be used to reference the set Timeout or Immediate object. By passing said object into the respective clear function, execution of that object will be halted completely. The respective functions are clearTimeout(), clearImmediate(), and clearInterval(). See the example below for an example of each: \*/

const timeoutObj = setTimeout(() => {

console.log('timeout beyond time');

}, 1500);

const immediateObj = setImmediate(() => {

console.log('immediately executing immediate');

});

const intervalObj = setInterval(() => {

console.log('interviewing the interval');

}, 500);

clearTimeout(timeoutObj);

clearImmediate(immediateObj);

clearInterval(intervalObj);

//CHILD PROCESS

/\* The spawn function launches a command in a new process and we can use it to pass that command any arguments. For example, here’s code to spawn a new process that will execute the pwd command. \*/

const { spawn } = require('child\_process');

const child = spawn('pwd');

/\* We simply destructure the spawn function out of the child\_process module and execute it with the OS command as the first argument.

The fork function is a variation of the spawn function for spawning node processes. The biggest difference between spawn and fork is that a communication channel is established to the child process when using fork, so we can use the send function on the forked process along with the global process object itself to exchange messages between the parent and forked processes. We do this through the EventEmitter module interface. \*/

//API FUNCTIONS

/\* There are two types of API functions: Asynchronous, non-blocking functions - mostly I/O operations which can be fork out of the main loop. Synchronous, blocking functions - mostly operations that influence the process running in the main loop. \*/

//ASYNC.QUEUE()

/\* The async module is is designed for working with asynchronous JavaScript in NodeJS. The async.queue returns a queue object which is capable of concurrent processing i.e processing multiple items at a single time. Example: \*/

const queue = async.queue((task, completed) => {

/\* Here task is the current element being

processed and completed is the callback function \*/

console.log("Currently Busy Processing Task " + task);

// Simulating a complex process.

setTimeout(() => {

3

// Number of elements to be processed.

const remaining = queue.length();

completed(null, { task, remaining });

}, 1000);

}, 1);

/\* The concurrency value is set to one,

Which means that one element is being

Processed at a particular time \*/

//EVENT LOOP

/\* https://www.geeksforgeeks.org/node-js-event-loop/

The main loop is single-threaded and all async calls are managed by libuv library. This is because libuv sets up a thread pool to handle such concurrency. How many threads will be there in the thread pool depends upon the number of cores but you can override this. \*/

//nextTick()

/\* Every time the event loop takes a full trip, we call it a tick. When we pass a function to process.nextTick(), we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts: \*/

process.nextTick(() => {

//do something

})

/\* The event loop is busy processing the current function code. When this operation ends, the JS engine runs all the functions passed to nextTick calls during that operation. It's the way we can tell the JS engine to process a function asynchronously (after the current function), but as soon as possible, not queue it. Calling setTimeout(() => {}, 0) will execute the function at the end of next tick, much later than when using nextTick() which prioritizes the call and executes it just before the beginning of the next tick. Use nextTick() when you want to make sure that in the next event loop iteration that code is already executed. \*/

//ASYNC AWAIT

/\* https://nodejs.dev/learn/modern-asynchronous-javascript-with-async-and-await \*/

//EventEmitter

/\* the interaction of the user is handled through events: mouse clicks, keyboard button presses, reacting to mouse movements, and so on. On the backend side, Node.js offers us the option to build a similar system using the events module. This module, in particular, offers the EventEmitter class, which we'll use to handle our events.

const EventEmitter = require('events')

const eventEmitter = new EventEmitter()

This object exposes, among many others, the on and emit methods.

emit is used to trigger an event

on is used to add a callback function that's going to be executed when the event is triggered

For example, let's create a start event, and as a matter of providing a sample, we react to that by just logging to the console: \*/

eventEmitter.on('start', number => {

console.log(`started ${number}`)

});

eventEmitter.emit('start', 23);

/\* the event handler function is triggered, and we get the console log. The EventEmitter object also exposes several other methods to interact with events, like

once(): add a one-time listener

removeListener() / off(): remove an event listener from an event

removeAllListeners(): remove all listeners for an event \*/

//STREAM

/\* https://nodejs.dev/learn/nodejs-streams \*/

//BUFFERS

/\* https://nodejs.dev/learn/nodejs-buffers \*/

//MIDDLEWARE

/\* Middleware comes in between your request and business logic. It is mainly used to capture logs and enable rate limit, routing, authentication, basically whatever that is not a part of business logic. There are third-party middleware also such as body-parser and you can write your own middleware for a specific use case. \*/