**BLOCKING VS NON-BLOCKING:**

1. Blocking Vs Non Blocking: Blocking is when execution of additional javascript code in node.js process is put on hold waiting for completion of non-Javascript operation such as I/O and network calls. This happens because the event loop is blocked by the blocking operation. It is not blocking when execution of remaining javascript code is put on hold because the operation being performed is CPU intensive. In nodejs we are provided with options for such operations that are not blocking which make use of call back functions.
2. Concurrency and throughput: Javascript execution in Nodejs is single threaded. So higher throughput and concurrency is achieved through non blocking feature wherein the rest of the code can finish execution without waiting for the non-javascript operation. Read the example to get a better grip on concept For Example: In a webserver if it take 50ms to complete a request where 45ms are spent on a non-javascript operation then , we can execute them in a non blocking fashion and continue handling of other requests. Thus requests are taken care of concurrently with a singlethread.

Do not mix blocking and non-blocking codes carelessly as it might lead to unwanted outcomes.Here fs.unlinkSync which deletes the file would be called before the callback function Ex: const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) throw err;

console.log(data);

});

fs.unlinkSync('/file.md');

A better option would be to call the unlinkSync in a non-blocking way inside the callback as shown below:

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;

});

});

**EVENT LOOP,TIMERS, and process.nextTick():**

1. Event loop is what allows for nodejs to execute operations in a non-synchronous fashion even though there is only one Javascript thread of execution for the script. Very broadly speaking when a non javascript operation is taking place the event loop delegates the task to be done by the operating system, when the tasks are done the operating system puts their corresponding callbacks are pushed to a FIFO queue which are then executed by the event loop. Since modern operating systems are multithreaded many operations are performed by them and their relevant callbacks are executed by the event loop synchronously one after another as they are placed in a FIFO queue. The event loop is initialized when the nodejs script is run. **{NOTE: the event loop is executed by the main javascript thread which executes the event loop after all the javascript in the nodejs process is finished being executed}**
2. The event loop has seven phases with each phase having their own FIFO queue of callbacks which are executed when event loop enters the phase. The event loop will remain in that phase until the FIFO queue of callbacks becomes empty or the maximum number of callbacks for that phase(which is dependent on the OS also) is reached.
3. Timers is first phase wherein when the time set in them expires their corresponding callbacks are added to the FIFO queue and the event loop executes the callbacks in the queue. Thus the time set in the timers don’t tell the exact duration after which the callback is called instead they signify the amount of time after which event loop will start to execute the callbacks ie if a time is set for 100ms the callback will be executed after 100ms . It has been designed in such a way that the callbacks are executed with minimal delay time.
4. The second phase is pending callbacks which were delegated to the next loop of events when the event loop cannot execute callbacks in a certain phase as the number of executed callbacks for that phase has reached the maximum number.
5. The third phase is idle, prepare which is used by Nodejs internally.
6. The fourth phase is poll wherein when I/O, network operations finish they put their corresponding callbacks in the FIFO queue attached to the poll phase and the event loop executes them.
7. The next phase is check wherein callbacks associated with setImmediate functions are placed. These callbacks are executed when the poll phase is idle.
8. The next is closed callbacks which are called when certain events are triggered due to closing of resources. Callbacks associated with them are executed by the event loop.
9. The event loop goes through the phases described above in the order they are described and then again iterates through the phases. Between iterations of execution of the phases by event loop the event loop checks if there are any callbacks still present or any operations to poll on, if there are none then the event loop terminates.

**EVENT in Nodejs:**

1. Certain objects in nodejs emit events that can be listened on like streams which emit an event when data is ready. Functions can be registered to listen for these named events, and whenever the events are triggered the function registered to listen to the events will be executed.
2. The functions registered to listen to the events are called synchronously in the order they are registered with the return values of the function being ignored.
3. Objects which can emit event are instances of EventEmitter class which is present in the ‘events’ module. The **addListener/on** method of the class can be used to register the event and **emit** method is used to trigger/emit the event.
4. The functions should be registered before the event is called else when the event is triggered the listener functions won’t be executed.

**ERROR (exception handling):**

1. Errors in Javascript fall into the below four categories:
2. Standard javascript errors such as TypeError, RangeError.
3. System errors that are triggered by underlying operating system when we the OS tries to open a file which is not existent, or a socket that has already been closed.
4. User defined errors.
5. Assertion errors that are part of ‘assertion’ module. The assertion errors are triggered when there is an exceptional logic violation in the code.
6. The standard javascript errors and system errors all inherit from Error class.
7. The way these errors are propgated/reported and handled depends on the error style(not type) and API that generates the error.
   1. The standard javascript errors are generated using the **throw** mechanism and as such should be handled by using **try{} and catch() block**.
   2. The errors generated by the async error is passed as the first argument of the callback attached to it (mostly). The try catch mechanism will not work here as by the time the callback function is executed the try and catch block would have finished executing. If these errors are thrown in the callback function then uncaughtexception event is emitted , if no handler for this event is present then the the nodejs process crashes.
   3. The errors generated by objects which are instances of **EventEmitter** class are thrown as ‘error’ event and can only be handled by the listener functions attached to those error event. If no handler is present for the error event then the nodejs process(main javascript thread of execution) emits an uncaughtexception, if no handler for this event is present then the nodejs process crashes.
8. The generic class Error has 2 properties message and stack. stack which captures the stack trace showing how the error was thrown and the message property which might have the text description of the error being thrown.

**STREAMS in nodejs:**

1. A stream is wherein data is read from a source in chunks ie one chunk at a time or when we write data to a destination in chunk. Since in streams small chunks of data are read or written the memory usage is small and streams are effective when dealing with large amount of data.
2. Nodejs provides us with objects with methods that allow us to read and write from streams ie if readeable stream object is provided a source it reads from it in chunks , similarly if a writeable stream object is provided with a destination it writes to in chunks. Ex: fs.createReadStream()- reads data from the associated file source in chunks and httpResponse – writes data in chunks.
3. To make the stream objects have a standard interface there is a **stream interface** which needs to be implemented by all Stream objects.
4. There is also a **module** called **stream** which is used to create custom stream objects.
5. Based on the nature of stream there are four types:
6. Readable streams- stream objects of this nature can only read from a source.
7. Writeable streams- stream objects of this nature can only write to a destination. Use **write()** *method on stream object to write chunk of data.*
8. Duplex streams- stream objects of this nature can read from a source and write to a destination.
9. Transform streams- stream objects of this nature are duplex streams where we can read from a souce, transform them and write to a destination.
10. Buffer:
11. Readable Streams- chunks of data are read from a source and are buffered in a buffer whose size is passed as an argument to the readable stream object constructor. Once the buffer is full no more data is read from the source-readStreamObj.read() method needs to be called to read from the buffer.
12. Writeable Streams-chunks of data are written to a buffer, each time a chunk of data is written using write() , the method return true/false depending upon whether the chunk was written into the buffer. Once the buffer is full the method returns false.
13. Duplex Streams: In duplex stream there will be 2 buffers one each for the readable and writeable stream.
14. All Stream objects are also instances of Event Emitter class.
15. Events:
    1. Readable stream- The readable stream can also be created in such a way that each chunk read will be passed on to the application via events without the need of buffer. So every time data is read ‘**data**’ event is emitted by stream and chunk is passed as an argument to the handler function. Other significant(**not all**) events emitted are:
16. **end** - emitted when stream object has completed reading from the source.
17. **readable**- emitted when there is data present in buffer to be read by application. The handler needs to use **read()** method of the stream to read from the buffer.
18. **error**- emitted if the stream object encounters any errors.
19. **close-** emitted when the stream is destroyed by calling the **destroy()** method on the stream.

The readable stream can be in 2 modes , in one mode buffering occurs and read() method is used in the other we use event and read the chunk as each chunk is read from the source with no buffering. Generally prefer the event based way of reading.

* 1. Writeable stream- Following are some of the important events associated with write stream:
     1. **close**-emited when **destroy()** method is called on the stream.
     2. **finish**-emitted when **end()** method is called signifying that the all data has been written.
     3. **drain**-when buffer is full write() method returns false. This event is emitted when some space is created in the buffer by consumption of some data from the buffer.

1. All streams by default operate upon Buffer objects(octect/binary stream) ie the stream object when reads a chunk of data in the form group of bytes / writes a chunk of data in the form of group of bytes. They can also operate upon string ie by reading /writing a string of data- this can be done by setting the **setEncoding()** with ‘utf-8’ or ‘ascii’ as an argument. It can also be made to operate on objects – this mode is called as an object mode , any object can be used as basis similar to string except for null.
2. All readable streams provide a pipe method which takes a writeable stream as an argument. The pipe() function reads from the source and writes to the destination, using pipes() is recommended as flow rates are controlled to minimize buffering.