NodeJS Events / Async development introduction

Dmytro Kornev

Agenda

EVENTS

- How NodeJS interact with asynchronous code
- Event Loop
- Event Emitter

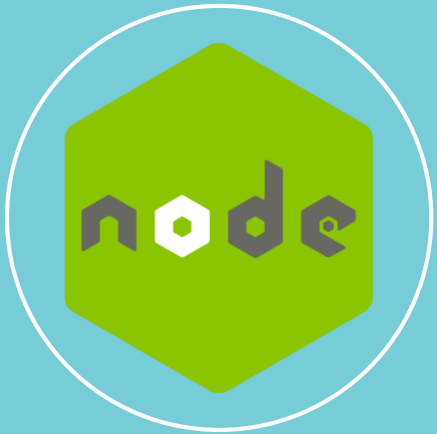
ASYNC DEVELOPMENT

- Async operations
- Asynchronous functions
- Memory Leaks

NODEJS

Why is non-blocking?

- Single-threaded
- Asynchronous
- Event-driven

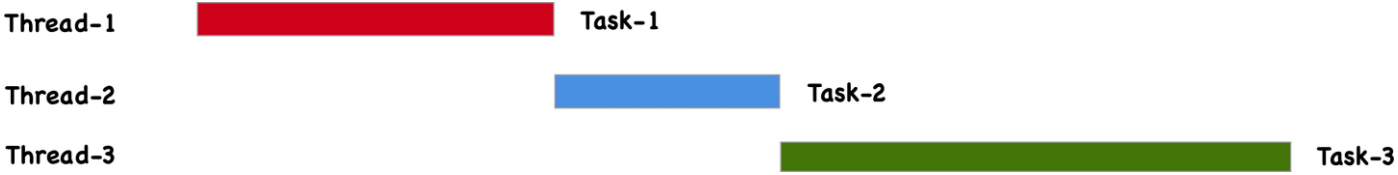


Synchronous

Single Threaded:

Each task gets executed one after another. Each task waits for its previous task to get executed.

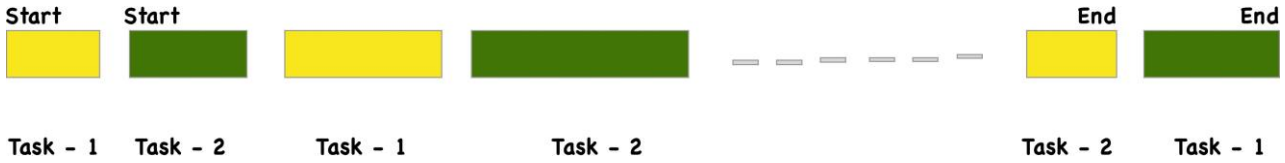
Multi-Threaded:



Tasks get executed in different threads but wait for any other executing tasks on any other thread.

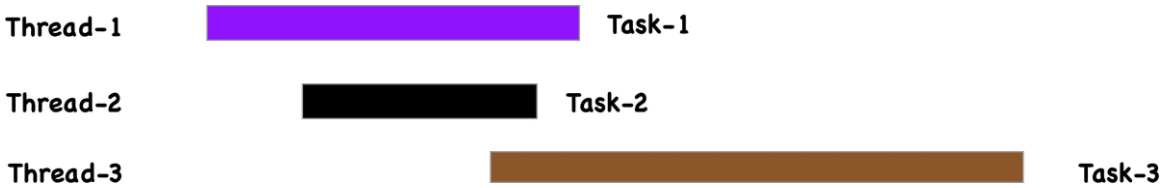
Asynchronous

Single Threaded:



Tasks start executing without waiting for a different task to finish. At a given time a single task gets executed.

Multi-Threaded:



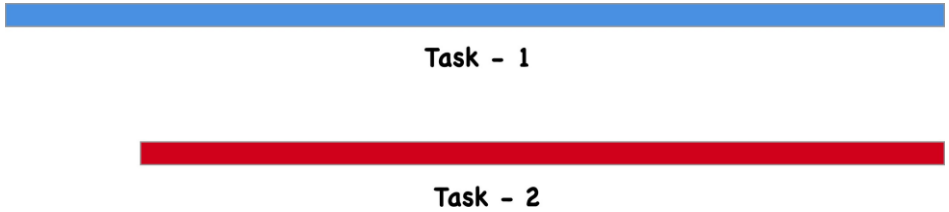
Tasks get executed in different threads without waiting for any tasks and independently finish off their executions.

Concurrency and Parallelism

Concurrency:

Execution of tasks in a single core environment. Tasks are context switched between one another.

Parallelism:



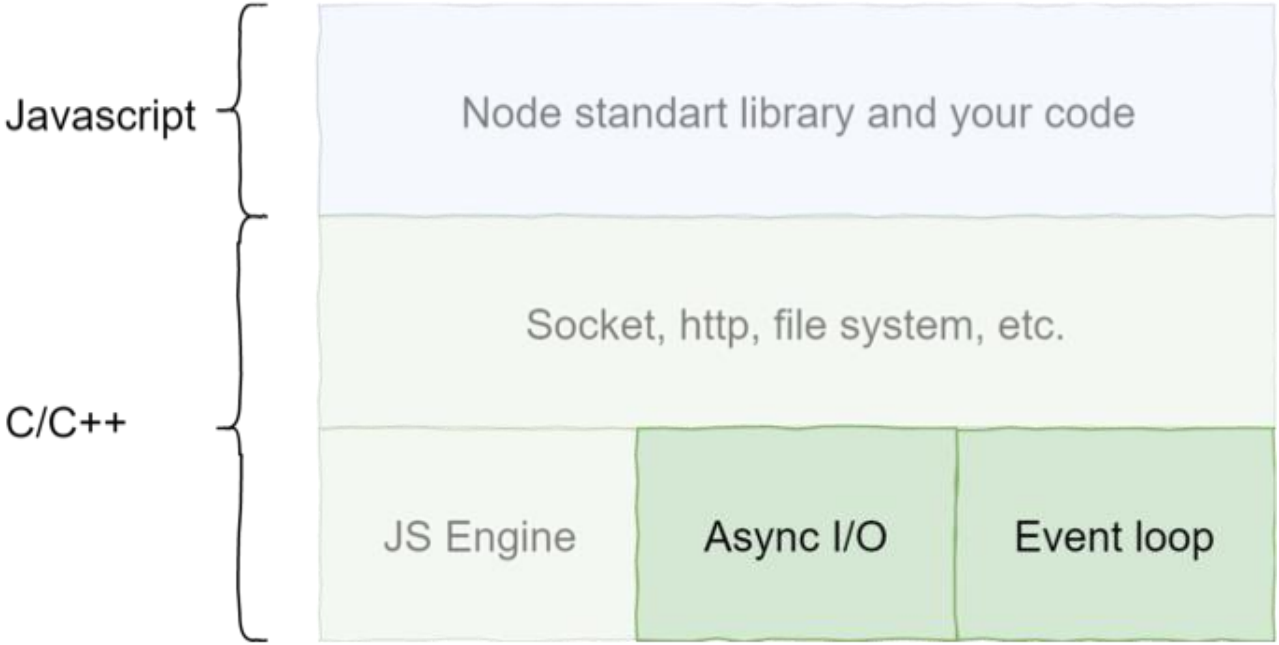
Two tasks are being performed simultaneously over the same time period.

LIBUV

How Node is asynchronous?

- Cross-platform support library which was originally written for NodeJS, but it’s also used by pyUv, Luvit and others
- It’s designed around the event-driven asynchronous I/O model
- A C library that is used to abstract non-blocking I/O operations to a consistent interface across all supported platforms. It provides mechanisms to handle file system, DNS, network, child processes, pipes, signal handling, polling and streaming. It also includes a thread pool for offloading work for some things that can't be done asynchronously at the operating system level.

libuv in depth



libuv in depth

```javascript
const dns = require('dns');

const nSecPerSec = 1e9;
const start = process.hrtime();
dns.setServers(['74.82.42.42'], ['91.239.100.100'], ['77.88.8.8'], ['109.69.8.51']);

for (let i = 0; i < 10; i++) {
    dns.lookup(`fake-servername.${Math.random()}.tld`, (err, address, family) => {
        const [seconds, nanoseconds] = process.hrtime(start);
        console.log(`lookup ${i} finished in ${seconds + nanoseconds / nSecPerSec}s`);
    });
}
```

```
>set UV_THREADPOOL_SIZE=1&node dns-lookupt.js
lookup 0 finished in 0.088076811s
lookup 1 finished in 0.151100661s
lookup 2 finished in 0.242022334s
lookup 3 finished in 0.327145057s
lookup 4 finished in 0.407256381s
lookup 5 finished in 0.493656003s
lookup 6 finished in 0.579198877s
lookup 7 finished in 0.643536862s
lookup 8 finished in 0.715251378s
lookup 9 finished in 0.748912774s
```

```
>node dns-lookupt.js
lookup 3 finished in 0.027999291s
lookup 0 finished in 0.035725534s
lookup 2 finished in 0.036011013s
lookup 5 finished in 0.057695942s
lookup 4 finished in 0.079924522s
lookup 1 finished in 0.080554437s
lookup 6 finished in 0.101843109s
lookup 8 finished in 0.112146421s
lookup 9 finished in 0.113046611s
lookup 7 finished in 0.13003696s
```

```
>set UV_THREADPOOL_SIZE=100&node dns-lookupt.js
lookup 6 finished in 0.073490072s
lookup 2 finished in 0.097054197s
lookup 4 finished in 0.098444046s
lookup 5 finished in 0.099562069s
lookup 3 finished in 0.100674817s
lookup 9 finished in 0.10167058s
lookup 0 finished in 0.102644002s
lookup 1 finished in 0.108199673s
lookup 7 finished in 0.110062114s
lookup 8 finished in 0.173001871s
```

Event Loop

TIMERS (setTimeout, setInterval)

process.nextTick()

PENDING CALLBACKS

process.nextTick()

IDLE, PREPARE

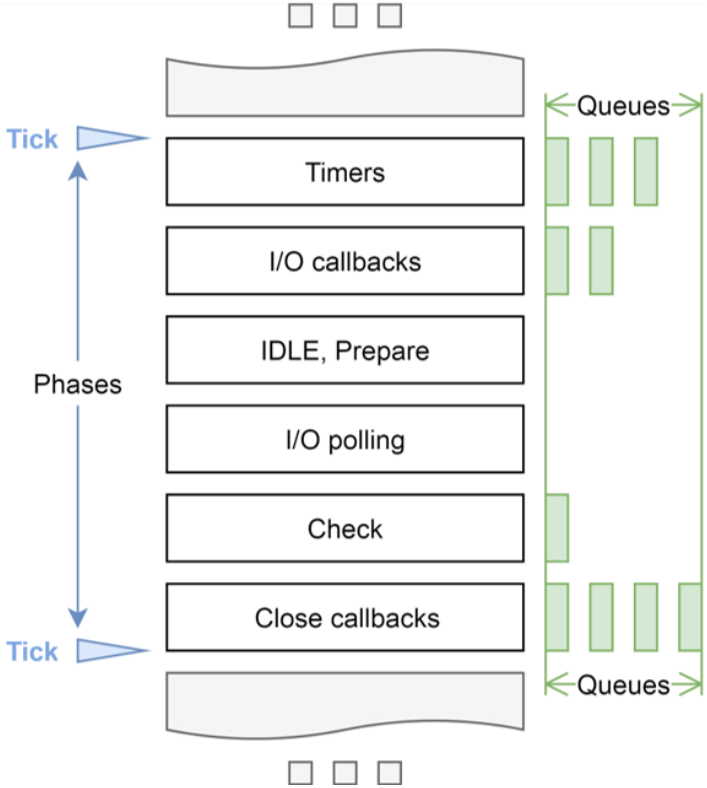
POLL (incoming: connections, data, etc.)

process.nextTick()

CHECK (setImmediate)

process.nextTick()

CLOSE CALLBACKS (socket.on('close', ...))



Timers

setTimeout(function, milliseconds, param1, param2, …)

- Executes a given function after a given time (in milliseconds)
- Return Value: A number, representing the ID value of the timer that is set. Use this value with the clearTimeout() method to cancel the timer

setInterval(function, milliseconds, param1, param2, …)

- Executes a given function at every given milliseconds
- Return Value: A number, representing the ID value of the timer that is set. Use this value with the clearInterval() method to cancel the timer

Node app keeps running in case there are any watch-process like server listening or interval.

You can use *timerObject.unref()* to mark timer as *non-priority,* so app can be closed without waiting until the interval is cleared.

Timers ref/unref Demo

```
1   // 40-timer-ref.js
2
3   for (let i = 0; i < 5; i++) {
4       setTimeout(() => console.log('tick'), 0)
5   }
6
```

```
⇒  node 40-timer-ref.js
tick
tick
tick
tick
tick
```

```
1   // 41-timer-unref.js
2
3   for (let i = 0; i < 5; i++) {
4       const timer = setTimeout(() => console.log('tick'), 0)
5       timer.unref()
6   }
7
```

```
⇒  node 41-timer-unref.js
⇒
```

setImmediate() and process.nextTick()

setImmediate()

- setImmediate(callback)
- Returns an *immediateObject* for possible use with *clearImmediate*. Additional optional arguments may be passed to the callback.

process.nextTick()

- The *process.nextTick()* method adds the callback to the "next tick queue". Once the current turn of the event loop turn runs to completion, all callbacks currently in the next tick queue will be called.
- Allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues
- At times it’s necessary to allow a callback to run after the call stack has unwound but before the event loop continues

setImmediate() vs process.nextTick()

```
1   const operation = process.argv[2];
2   let loops = 11;
3   let delay = 10;
4   let start = process.hrtime();
5
6   function run() {
7       loops--;
8       for (let i = 0; i < 1e7; i++) {
9           Math.pow(Math.random(), Math.random());
10      }
11      if (loops > 0) {
12          switch (operation) {
13              case 'blocked':      run();                 break;
14              case 'nexttick':     process.nextTick(run); break;
15              case 'setimmediate': setImmediate(run);     break;
16          }
17      }
18  }
19
20  setTimeout(() => {
21      const [seconds, nanoSeconds] = process.hrtime(start)
22      const msElapsed = seconds * 1000 + nanoSeconds / 1e6;
23      console.log(`I took ${msElapsed}ms, expected to take ${delay}`);
24  }, delay);
25
26  run();
```

setImmediate() vs process.nextTick()

```
> time node ./51-setImmediate_vs_nextTick.js blocked
I took 3696.961844ms, expected to take 10

real    0m4.125s
user    0m4.111s
sys     0m0.012s
>
> time node ./51-setImmediate_vs_nextTick.js nexttick
I took 3403.666101ms, expected to take 10

real    0m3.832s
user    0m3.816s
sys     0m0.016s
>
> time node ./51-setImmediate_vs_nextTick.js setimmediate
I took 311.39354ms, expected to take 10

real    0m3.802s
user    0m3.765s
sys     0m0.036s
>
```

Callbacks

COMMON FUNCTIONS

- fs.open(path[, flags[, mode]], callback)
- agent.createConnection(options, callback) • server.listen(handle, callback)
- server.listen(path, callback)
- dns.resolve(hostname, rrtype, callback)
- fs.readFile(file, callback)

Add Sync for sync analogue. Ex.: fs.openSync(path[, flags, mode]).

Error-first callbacks

```javascript
const fs = require('fs');

function copyFile(oldFilePath, newFilePath, callback) {
    fs.readFile(oldFilePath, (error, fileContent) => {
        if (error) {
            console.error(error);
            callback(error);
            return;
        }

        fs.writeFile(newFilePath, fileContent, (error) => {
            if (error) {
                console.error(error);
                callback(error);
                return;
            }

            callback(null);
        });
    });
}

copyFile(process.argv[2], process.argv[3], (error) => {
    if (!error) {
        console.log(`File was copied.`);
    } else {
        console.log(`File wasn't copied.`);
    }
});
```

Promises

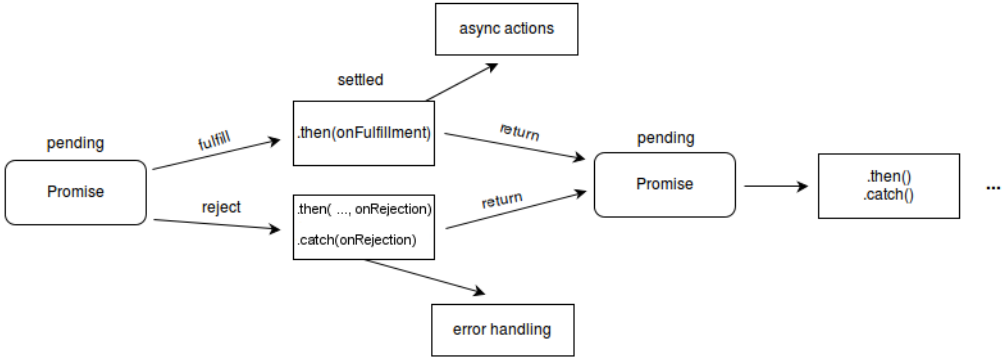
```
1   const fs = require('fs');
2
3   const { promisify } = require('util');
4   const readFileAsync = promisify(fs.readFile);
5   const writeFileAsync = promisify(fs.writeFile);
6
7   function copyFile(oldFilePath, newFilePath) {
8       return readFileAsync(oldFilePath)
9           .then((fileContent) => writeFileAsync(newFilePath, fileContent))
10          .catch((error) => {
11              console.error(error)
12              throw error;
13          });
14  }
15
16  copyFile(process.argv[2], process.argv[3])
17      .then(
18          () => console.log(`File was copied.`),
19          () => console.log(`File wasn't copied.`)
20      );
```

```
const fs = require('fs');

function copyFile(file, encoding) {
  return new Promise(function (resolve, reject) {
    fs.copyFile(file, encoding, function (err, data) {
      if (err) return reject(err); // Rejects the promise with `err` as the reason
      resolve(data); // Fulfills the promise with `data` as the value
    });
  });
}

let promise = copyFile('myfile.txt');
promise.then(console.log, console.error);
```

Promises



- Use promises whenever you are using asynchronous or blocking code.
- resolve maps to then and reject maps to catch for all practical purposes.
- Make sure to write both .catch and .then methods for all the promises.
- If something needs to be done in both cases use .finally.
- We only get one shot at mutating each promise.

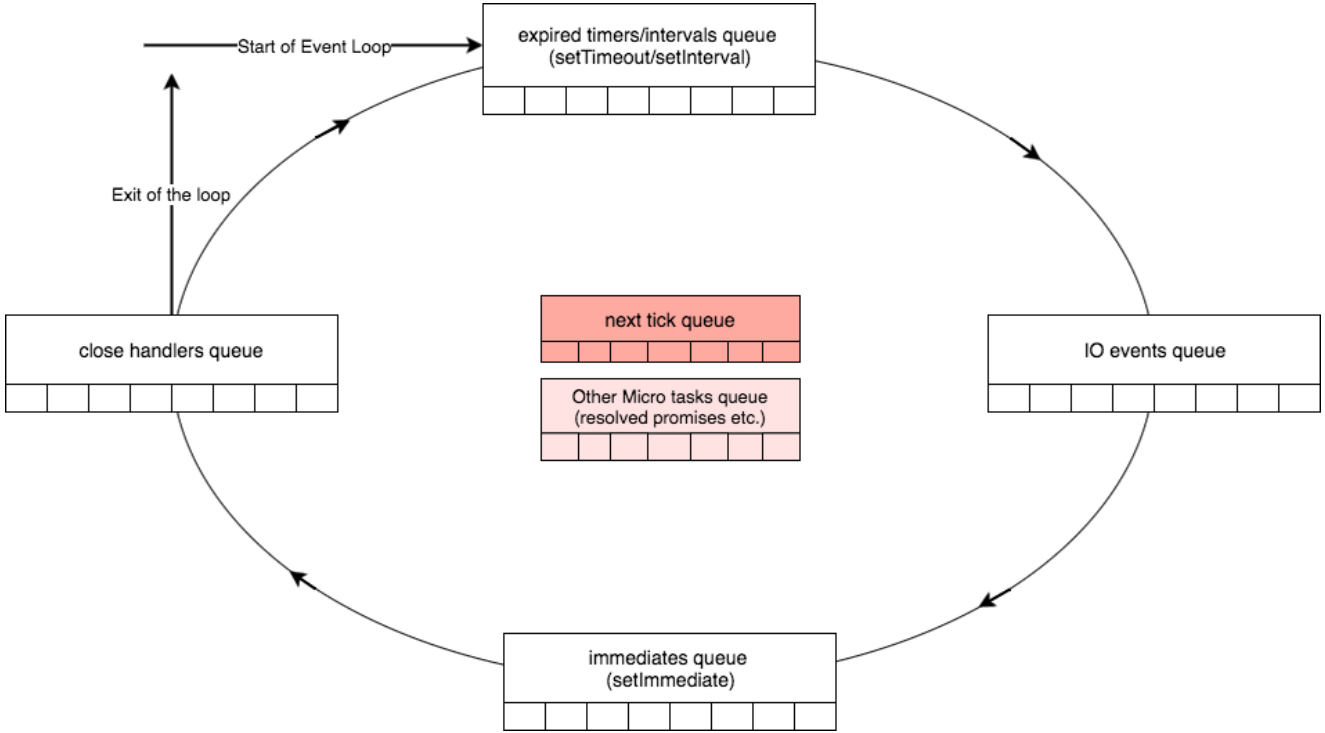
- We can add multiple handlers to a single promise.
- The return type of all the methods in the Promise object, regardless of whether they are static methods or prototype methods, is again a Promise.
- In Promise.all, the order of the promises are maintained in the values variable, irrespective of which promise was first resolved.

Promises

```
setImmediate(() =>console.log('immediate1'));
setImmediate(() => {
   console.log('immediate2’);
   Promise.resolve().then(() => console.log('promise resolve'))
});
setImmediate(() => console.log('immediate3'));
setImmediate(() => console.log('immediate4'));
```

```
➜ ~ nvm use 10
Now using node v10.15.1 (npm v6.8.0)
➜ ~ node setimmediate-and-promises.js
immediate1
immediate2
immediate3
immediate4
promise resolve
➜ ~ nvm use 11
Now using node v11.10.0 (npm v6.7.0)
➜ ~ node setimmediate-and-promises.js
immediate1
immediate2
promise resolve
immediate3
immediate4
➜ ~
```

Microtasks


Start of Event Loop
expired timers/intervals queue
(setTimeout/setInterval)
Exit of the loop
next tick queue
Other Micro tasks queue
(resolved promises etc.)
close handlers queue
IO events queue
immediates queue
(setImmediate)

Microtasks

```
Promise.resolve().then(() => console.log('promise1 resolved'));
Promise.resolve().then(() => console.log('promise2 resolved'));
Promise.resolve().then(() => {
  console.log('promise3 resolved');
  process.nextTick(() => console.log('next tick inside promise resolve handler'));
});
Promise.resolve().then(() => console.log('promise4 resolved'));
Promise.resolve().then(() => console.log('promise5 resolved'));
setImmediate(() => console.log('set immediate1'));
setImmediate(() => console.log('set immediate2'));

process.nextTick(() => console.log('next tick1'));
process.nextTick(() => console.log('next tick2'));
process.nextTick(() => console.log('next tick3'));

setTimeout(() => console.log('set timeout'), 0);
setImmediate(() => console.log('set immediate3'));
setImmediate(() => console.log('set immediate4'));
```

```
next tick1
next tick2
next tick3
promise1 resolved
promise2 resolved
promise3 resolved
promise4 resolved
promise5 resolved
next tick inside promise resolve handler
set timeout
set immediate1
set immediate2
set immediate3
set immediate4
```

Microtasks

```
const Q = require('q');
const BlueBird = require('bluebird');

Promise.resolve().then(() => console.log('native promise resolved'));
BlueBird.resolve().then(() => console.log('bluebird promise resolved'));
setImmediate(() => console.log('set immediate'));
Q.resolve().then(() => console.log('q promise resolved'));
process.nextTick(() => console.log('next tick'));
setTimeout(() => console.log('set timeout'), 0);
```

```
q promise resolved
q promise rejected
next tick
native promise resolved
native promise rejected
set timeout
bluebird promise resolved
bluebird promise rejected
set immediate
```

Async

```
1   const fs = require('fs');
2
3   const { promisify } = require('util');
4   const readFileAsync = promisify(fs.readFile);
5   const writeFileAsync = promisify(fs.writeFile);
6
7   async function copyFile(oldFilePath, newFilePath) {
8       try {
9           const fileContent = await readFileAsync(oldFilePath);
10          await writeFileAsync(newFilePath, fileContent);
11      } catch(error) {
12          console.error(error);
13          throw error;
14      }
15  }
16
17  copyFile(process.argv[2], process.argv[3])
18      .then(
19          () => console.log(`File was copied.`),
20          () => console.log(`File wasn't copied.`)
21      );
```

- *async* functions return a promise.
- *async* functions use an implicit Promise to return results. Even if you don't return a promise explicitly, the async function makes sure that your code is passed through a promise.
- *await* blocks the code execution within the *async* function, of which it (*await statement*) is a part.
- There can be multiple await statements within a single async function.
- When using async await, make sure you use try catch for error handling.
- Be extra careful when using await within loops and iterators. You might fall into the trap of writing sequentially-executing code when it could have been easily done in parallel
- await is always for a single Promise
- Promise creation starts the execution of asynchronous functionality
- await only blocks the code execution within the async function. It only makes sure that the next line is executed when the promise resolves. So, if an asynchronous activity has already started, await will not have any effect on it

Async Library


```
1  async.map(['foo.txt', 'bar.txt'], fs.stat, (error, results) => {
2      console.log(results);
3  });
```

```
1  async.parallel([
2      (callback) => { setTimeout(callback, 1000); },
3      (callback) => { setTimeout(callback, 1000); }
4  ], (error, results) => { console.log('I took 1 second.'); });
```

```
1  async.waterfall([
2      (callback) => { setTimeout(callback, 1000); },
3      (callback) => { setTimeout(callback, 1000); }
4  ], (error, results) => { console.log('I took 2 seconds.'); });
```

Event Emitter

All objects that emit events are members of *EventEmitter* class. These objects expose an *eventEmitter.on*() function that allows one or more functions to be attached to named events emitted by the object.

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {};
const myEmitter = new MyEmitter();
```

- on
- once
- prependListener
- prependOnceListener
- emit
- removeAllListeners
- removeListener

The EventEmitter calls all listeners synchronously in the order in which they were registered. This is important to ensure the proper sequencing of events and to avoid race conditions or logic errors. When appropriate, listener functions can switch to an asynchronous mode of operation using
the setImmediate() or process.nextTick() methods:

Event Emitter

```
const EventEmitter = require('events’);

class MyEmitter extends EventEmitter {};
const myEmitter = new MyEmitter();
const alerter = () => console.log('event happened!’);

myEmitter.on('event', alerter);

myEmitter.emit('event’);
myEmitter.emit('event');
myEmitter.emit('event');
```

```
event happened!
event happened!
event happened!
```

Event Emitter

```
const EventEmitter = require('events’);

class MyEmitter extends EventEmitter {};
const myEmitter = new MyEmitter();
const alerter = () => console.log('event happened!’);

myEmitter.once('event', alerter);

myEmitter.emit('event’);
myEmitter.emit('event');
myEmitter.emit('event');
```

```
event happened!
```

Event Emitter

```
const emitter = require('events');

class MyEmitter extends emitter {
}

const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously');
  });
});
myEmitter.on('event', (a, b) => {
  process.nextTick(() => {
    console.log('this happens on tick');
  });
});
myEmitter.on('event', () => console.log('this happens synchronously'));

myEmitter.emit('event', 'a', 'b');
```

```
this happens synchronously
this happens on tick
this happens asynchronously
```

Event Emitter

```
myEmitter.setMaxListeners(5);
```

myEmitter.getMaxListeners(); // 5

```
const alerter = () => console.log('event happened!');
const helloer = () => console.log('hello there!');
const byebyeer = () => console.log('bye bye');
const alerter2 = () => console.log('event happened2!');
const helloer2 = () => console.log('hello there2!');
const byebyeer2 = () => console.log('bye bye2');

myEmitter.on('event', alerter);
myEmitter.on('event', alerter);
myEmitter.on('event', alerter2);
myEmitter.on('event', alerter2);
myEmitter.on('event', helloer);
myEmitter.on('event', byebyeer);
myEmitter.on('event', helloer2);
myEmitter.on('event', byebyeer2);
myEmitter.emit('event');
```

```
event happened!
event happened!
event happened2!
event happened2!
hello there!
bye bye
hello there2!
bye bye2
(node:25686)
MaxListenersExceededWarning
: Possible EventEmitter
memory leak detected. 6 event
listeners added to [MyEmitter].
Use emitter.setMaxListeners()
to increase limit
```

Event Emitter

```
myEmitter.listenerCount(eventName);
```

.eventNames()

```
const myEmitter = new MyEmitter();
const alerter = () => console.log('event happened!');

console.log(myEmitter.listenerCount('event’)); // 0

myEmitter.on('event', alerter);
myEmitter.on('event', alerter);

console.log(myEmitter.listenerCount('event’)); // 2
```

```
console.log(myEmitter.eventNames()); // []

console.log(myEmitter.eventNames()); // [‘event’]
```

Event Emitter

```
myEmitter.prependListener(eventName);
```

```
const myEmitter = new MyEmitter();
const alerter = () => console.log('event happened!’);
const byebyeer = () => console.log('bye bye');

myEmitter.on('event', alerter);
myEmitter.on('event', alerter);
myEmitter.prependListener('event', byebyeer);

myEmitter.emit('event');
```

```
bye bye
event happened!
event happened!
```

.prependOnceListener()

```
myEmitter.prependOnceListener('event', byebyeer);

myEmitter.emit('event’);
myEmitter.emit('event');
```

```
bye bye
event happened!
event happened!
event happened!
event happened!
```

Event Emitter

```
myEmitter.off(eventName);
```

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
const alerter = () => console.log('event happened!');
myEmitter.on('event', alerter);
myEmitter.emit('event');
myEmitter.emit('event');
myEmitter.off('event', alerter);
myEmitter.emit('event');
```

```
event happened!
event happened!
```

```
.removeAllListeners()
```

```
myEmitter.emit('event’);

myEmitter.removeAllListeners();

myEmitter.emit('event');
```

```
event happened!
event happened!
```

Memory Leaks

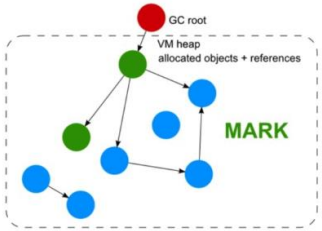
V8 IS HANDLING TWO MAIN MEMORY CATEGORIES:

- **Stack:** In which it stores the primitive data types: Number, String, boolean, Null, Undefined, Symbol and references to non-primitive data types Object.
- **Heap:** stores the non-primitive data types: Object.

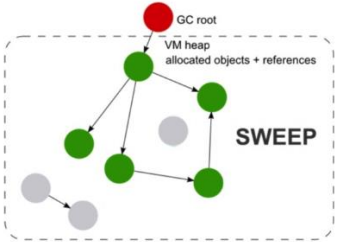
The **V8** has a garbage collector runs mainly **Mark and Sweep** algorithm:
It checks for all objects' reference paths to the root node *'which is the global or window object'*. If any reference has no path to the root node, it will be marked as garbage and will be swept later.

Important Note: When the Garbage Collector runs, it pauses your application entirely until it finishes its work. so you need to minimize its work by taking care of your objects' references.

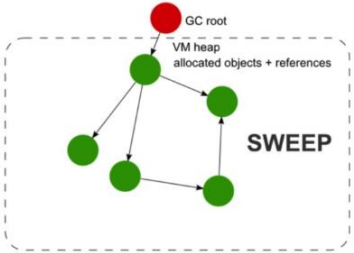
Mark and sweep (MARK)



Mark and sweep (SWEEP)



Mark and sweep (SWEEP)



Memory Leaks

- Global Variables

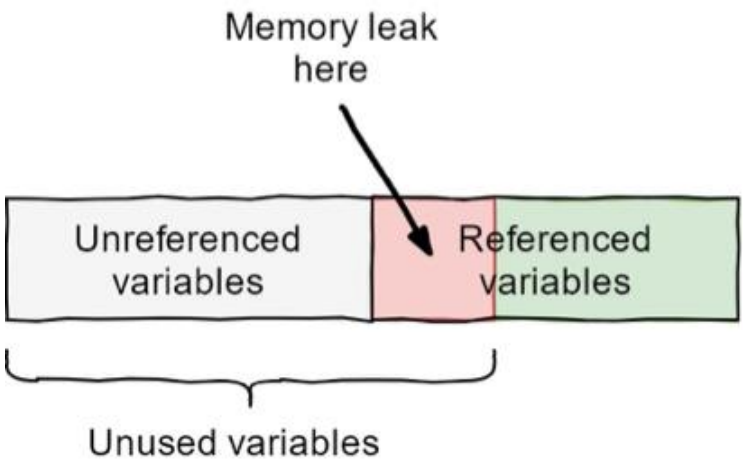
As they have a direct path to the root node, they will stay in memory as long as the application is running so you need to be careful when setting global variables and the amount of data you'll set to them.

- Multiple References

Setting multiple references to the same object may cause a problem also as you may remove one ref and forget the other which will keep your object still exists in the Heap.

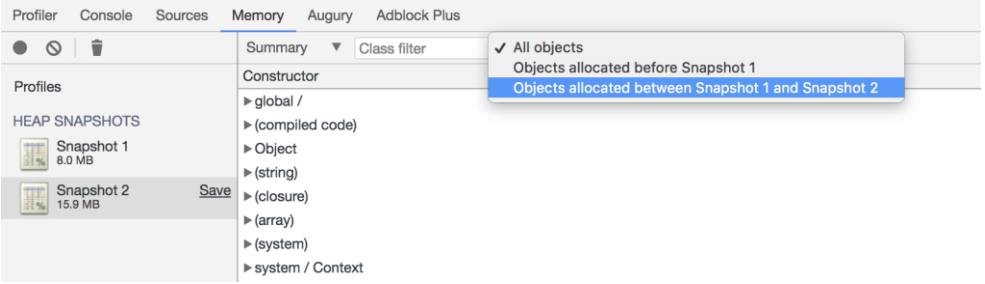
- Closures

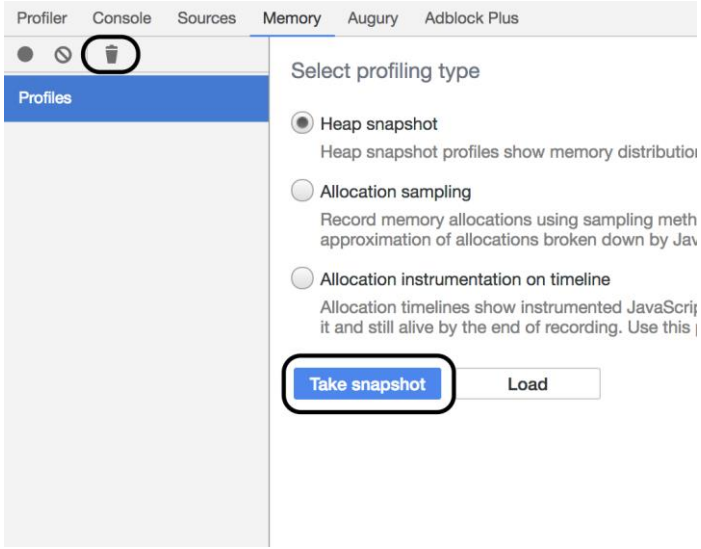
In closures simply you keep references to objects to be used later. this feature has many advantages but if it's used without caution it may cause big issues as these references will keep objects in heap and these objects might be large ones, not just simple objects.



Memory Leaks

Firstly, run your app with options:
expose-gc to be able to run the garbage collector explicitly
inspect=9222 to be able to attach the Chrome Debugger to your app on port *9222*
so the command should be:
node --expose-gc --inspect=9222 app.js





References

Event loop
https://blog.insiderattack.net/event-loop-and-the-big-picture-nodejs-event-loop-part-1-1cb67a182810

Microtasks in node 11+
https://blog.insiderattack.net/new-changes-to-timers-and-microtasks-from-node-v11-0-0-and-above-68d112743eb3

Promises in node
https://blog.insiderattack.net/promises-next-ticks-and-immediates-nodejs-event-loop-part-3-9226cbe7a6aa

Async development
https://medium.com/swift-india/concurrency-parallelism-threads-processes-async-and-sync-related-39fd951bc61d

Library for async
http://caolan.github.io/async/

Memory Leaks
https://medium.com/tech-tajawal/memory-leaks-in-nodejs-quick-overview-988c23b24dba

THANK YOU