Node JS – Events and REPL

Node.js Events

- As node js is an asynchronous event-driven JavaScript runtime.
- Node.js has an event-driven architecture which can perform asynchronous tasks.
- Node.js has 'events' module which emits named events that can cause corresponding functions or callbacks to be called. Functions(Callbacks) listen or subscribe to a particular event to occur and when that event triggers, all the callbacks subscribed to that event are fired one by one in order to which they were called.

The EventEmmitter class

- All objects that emit events are instances of the EventEmitter class.
- The event can be emitted or listen to an event with the help of EventEmitter.

```
const EventEmitter = require("events");
const  event = new EventEmitter();
```

The EventEmitter Object

You can assign event handlers to your own events with the EventEmitter object.

In the example below we have created a function that will be executed when a "scream" event is fired.

To fire an event, use the emit() method.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();


//Create an event handler and assign the event handler to an event:
eventEmitter.on('Saymyname', function() {
console.log('My name is Node js!');
});


//Fire the ' Saymyname ' event:
eventEmitter.emit('Saymyname');
```

Event Loop

1. Node.js is a single-threaded event-driven platform that is capable of running non-blocking, asynchronously programming.
2. It uses async function calls to maintain concurrency.
3. These functionalities of Node.js make it memory efficient.
4. The event loop allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded.
5. It is done by assigning operations to the operating system whenever and wherever possible.
6. Most operating systems are multi-threaded and hence can handle multiple operations executing in the background. When one of these operations is completed, the kernel tells Node.js and the respective callback assigned to that operation is added to the event queue which will eventually be executed.

Features of Event Loop:

1. Event loop is an endless loop, which waits for tasks, executes them and then sleeps until it receives more tasks.
2. The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
3. The event loop allows us to use callbacks.
4. The event loop executes the tasks starting from the oldest first.

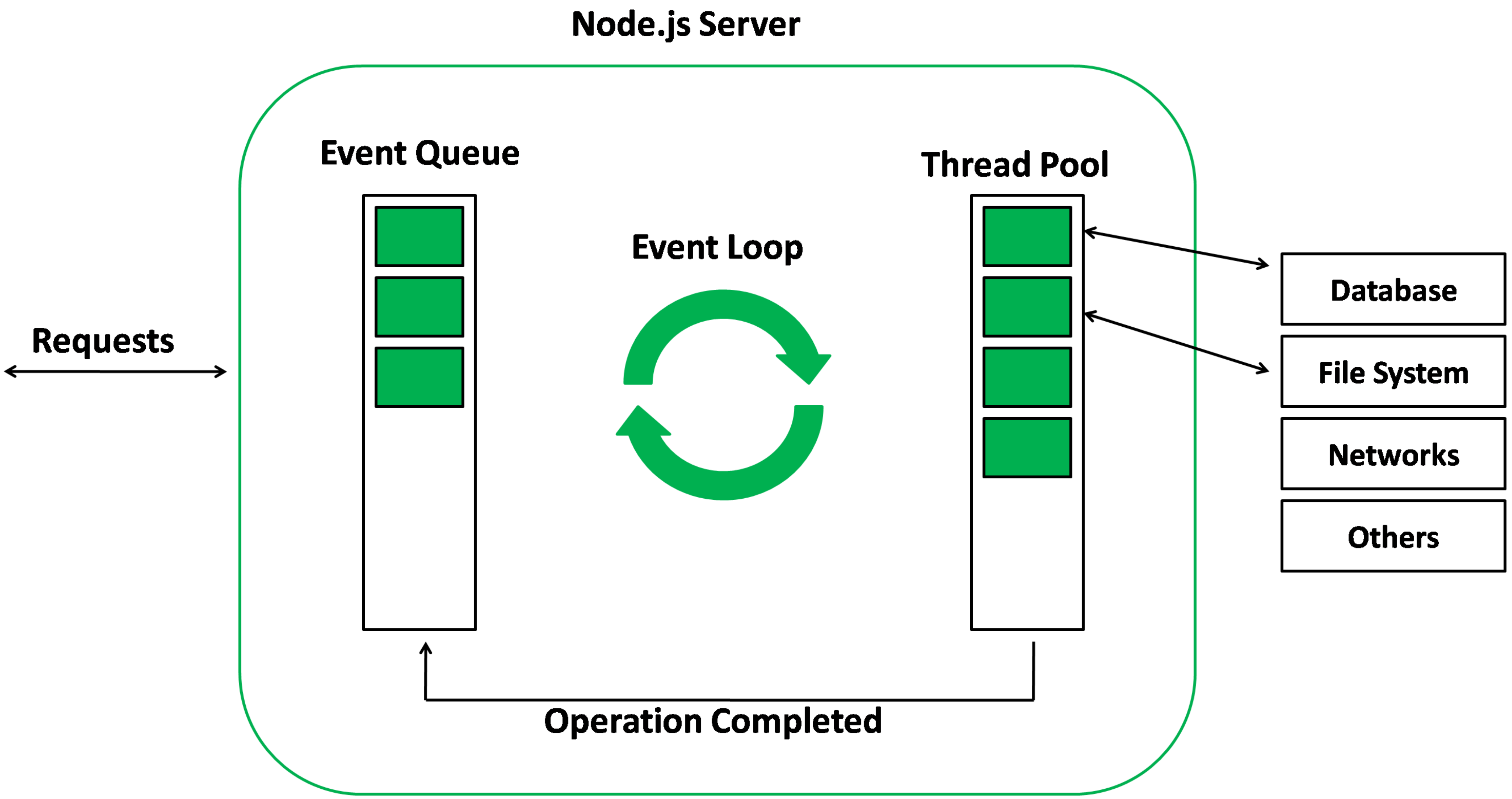


Image source: https://www.geeksforgeeks.org/node-js-event-loop/

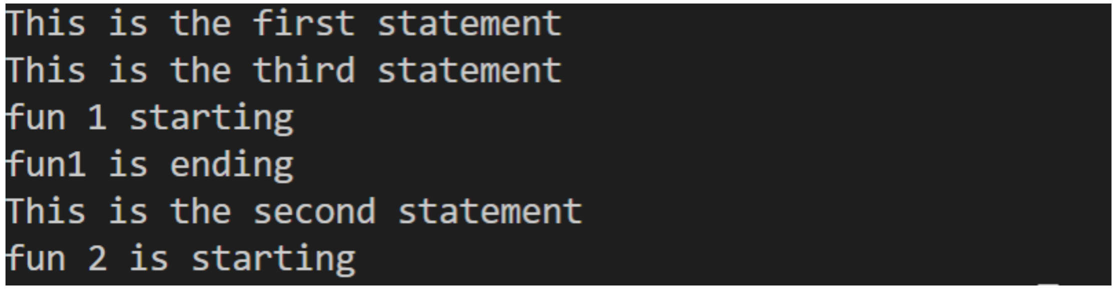
Event Loop

```
console.log("This is the first statement");

setTimeout(function(){
        console.log("This is the second statement");
}, 1000);

console.log("This is the third statement");
```

- ➢The first console log statement is pushed to the call stack and “This is the first statement” is logged on the console and the task is popped from the stack. Next, the setTimeout is pushed to the queue and the task is sent to the Operating system and the timer is set for the task. This task is then popped from the stack. Next, the third console log statement is pushed to the call stack and “This is the third statement” is logged on the console and the task is popped from the stack.
- ➢When the timer set by setTimeout function (in this case 1000 ms) runs out, the callback is sent to the event queue. The event loop on finding the call stack empty takes the task at the top of the event queue and sends it to the call stack. The callback function for setTimeout function runs the instruction and “This is the second statement” is logged on the console and the task is popped from the stack.

Note: In the above case, if the timeout was set to 0ms then also the statements will be displayed in the same order. This is because although the callback with be immediately sent to the event queue, the event loop won’t send it to the call stack unless the call stack is empty i.e. until the provided input script comes to an end.

```
const fun2=()=>{
    setTimeout(()=> {console.log('fun 2 is starting');}, 300
0);
}
 const fun1=()=>{
     console.log('fun 1 starting');
     fun2();
     console.log('fun1 is ending');
 }
 fun1();
```



```
This is the first statement
This is the third statement
fun 1 starting
fun1 is ending
This is the second statement
fun 2 is starting
```

REPL Terminal

Feature

1. REPL future of node is very useful in experimenting with Node.js codes and to debug JavaScript code.
2. REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode. Node.js or Node comes bundled with a REPL environment.

It performs the following tasks −

Read − Reads user's input, parses the input into JavaScript data-structure, and stores in memory.

Eval − Takes and evaluates the data structure.

Print − Prints the result.

Loop − Loops the above command until the user presses ctrl-c twice.

REPL can be started by simply running node on shell/console without any arguments as follows.

```
$ node
```

REPL Terminal

Use Variables

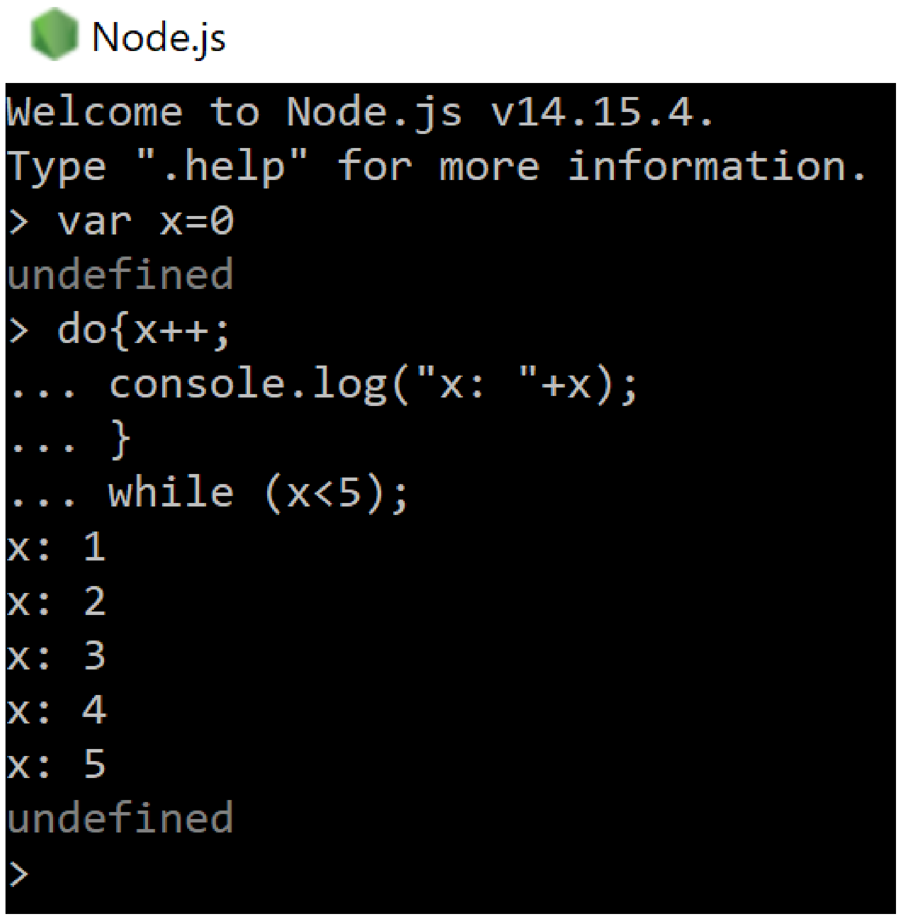
-You can make use variables to store values and print later.

- If **var** keyword is not used, then the value is stored in the variable and printed. Whereas if **var** keyword is used, then the value is stored but not printed. You can print variables using **console.log()**.

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
```

REPL Terminal

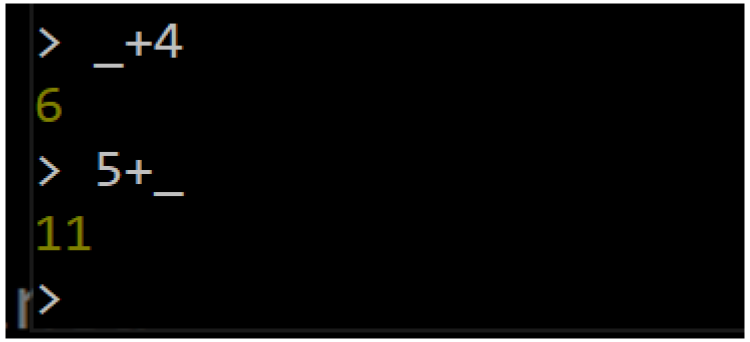
Multi-line Expression



```
Welcome to Node.js v14.15.4.
Type ".help" for more information.
> var x=0
undefined
> do{x++;
... console.log("x: "+x);
... }
... while (x<5);
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

REPL Terminal

Return values



```
> _+4
6
> 5+_
11
>
```

Accessing Modules

If you need to access any of the built-in modules, or any third party modules, they can be accessed with require, just like in the rest of Node.

REPL Commands

- **ctrl + c** − terminate the current command.
- **ctrl + c twice** − terminate the Node REPL.
- **ctrl + d** − terminate the Node REPL.
- **Up/Down Keys** − see command history and modify previous commands.
- **tab Keys** − list of current commands.
- **.help** − list of all commands.
- **.break** − exit from multiline expression.
- **.clear** − exit from multiline expression.
- **.save *filename*** − save the current Node REPL session to a file.
- **.load *filename*** − load file content in current Node REPL session.

Useful Reference Links

- https://www.w3schools.com/nodejs/default.asp
- https://nodejs.dev/learn