Unit 8 Node JS

Introduction

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use. It can be downloaded from this link https://nodejs.org/en/

Definition: Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications.

Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

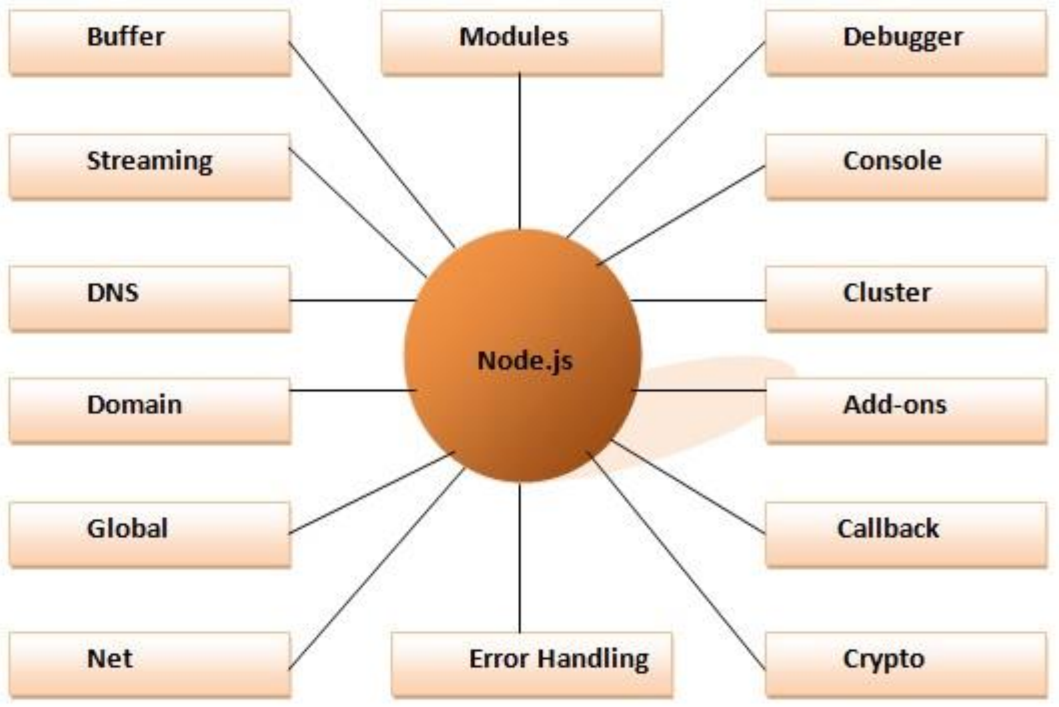
1. Node.js = Runtime Environment + JavaScript Library

Imp Notes:

- Node js is not a language. This is a server environment.
- Node js can connect with database.
- Code and syntax are very similar to JavaScript., but not exactly same. (Java Script does n0t connect with Database.   )
- It is free and open source.
- Node js use Crome's V8 engine to execute code.

Different parts of Node.js

The following diagram specifies some important parts of Node.js:



Features of Node.js

The following is a list of some important features of Node.js that make it the first choice of software architects.

2. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
3. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library is asynchronous i.e. non-blocking. So, a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
4. **Single threaded:** Node.js follows a single threaded model with event looping.
5. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.

6. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
7. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
8. **License:** Node.js is released under the MIT license.

Install Node.js on Windows

To install and setup an environment for Node.js, you need the following two softwares available on your computer:

1. Text Editor.
2. Node.js Binary installable

Text Editor:

The text editor is used to type your program. For example: Notepad is used in Windows, vim or vi can be used on Windows as well as Linux or UNIX. The name and version of the text editor can be different from operating system to operating system.

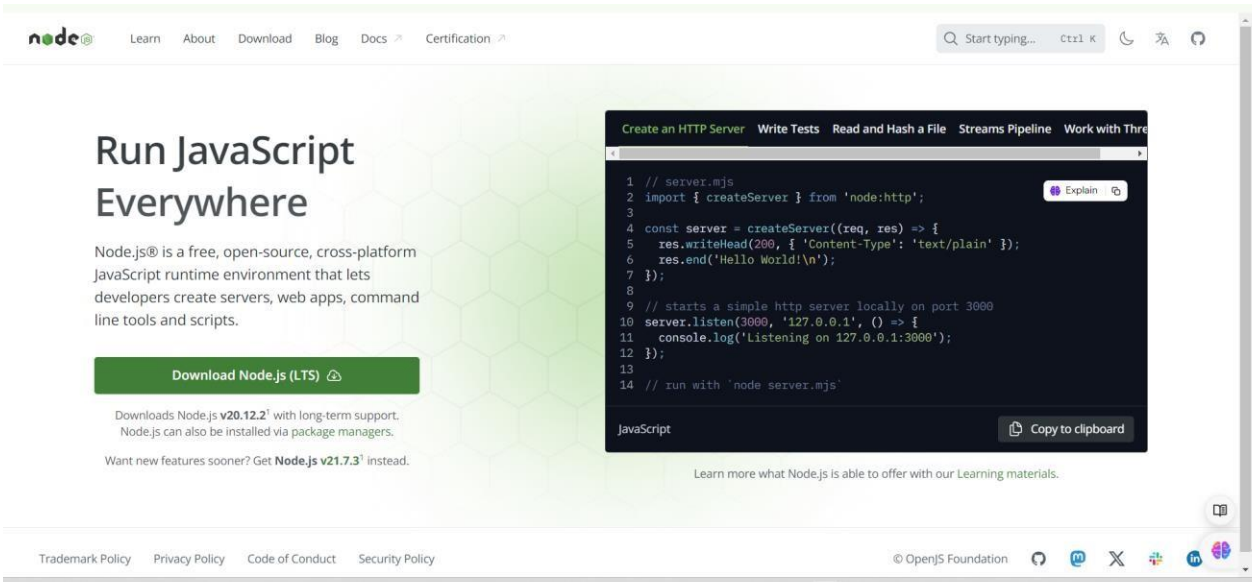
The files created with text editor are called source files and contain program source code. The source files for Node.js programs are typically named with the extension ".js".

The Node.js Runtime:

The source code written in source file is simply JavaScript. It is interpreted and executed by the Node.js interpreter.

How to download Node.js:

You can download the latest version of Node.js installable archive file from https://nodejs.org/en/



Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

Download Node.js (LTS)

Downloads Node.js **v20.12.2**[1] with long-term support. Node.js can also be installed via package managers.

Want new features sooner? Get **Node.js v21.7.3**[1] instead.

Create an HTTP Server | Write Tests | Read and Hash a File | Streams Pipeline | Work with Thre

```
// server.mjs
import { createServer } from 'node:http';

const server = createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!\n');
});

// starts a simple http server locally on port 3000
server.listen(3000, '127.0.0.1', () => {
  console.log('Listening on 127.0.0.1:3000');
});

// run with `node server.mjs`
```

Learn more what Node.js is able to offer with our Learning materials.

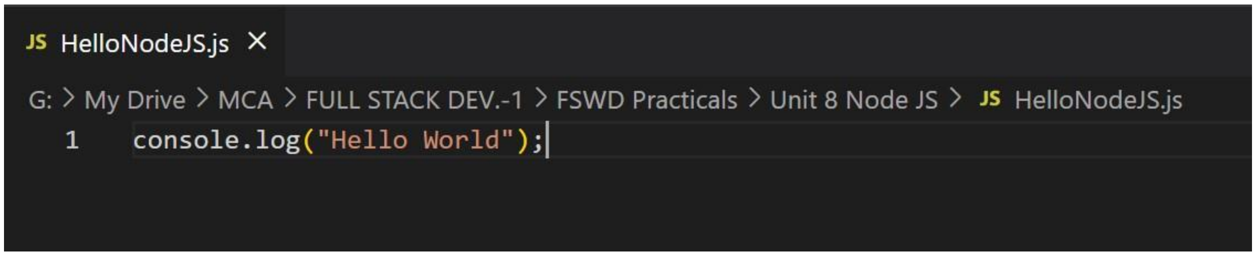
Node.js First Example

There can be console-based and web-based node.js applications.

Node.js console-based Example

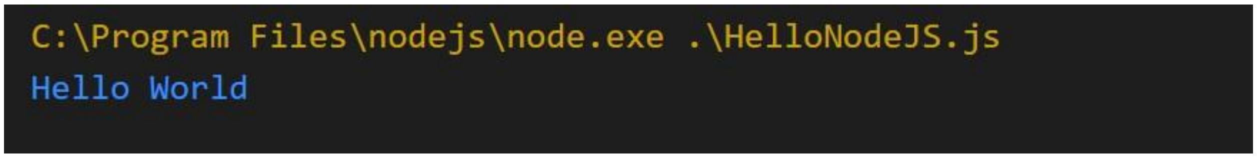
File: console_example1.js console.log('Hello

JavaTpoint');


JS HelloNodeJS.js ✕
G: > My Drive > MCA > FULL STACK DEV.-1 > FSWD Practicals > Unit 8 Node JS > JS HelloNodeJS.js

```
console.log("Hello World");
```

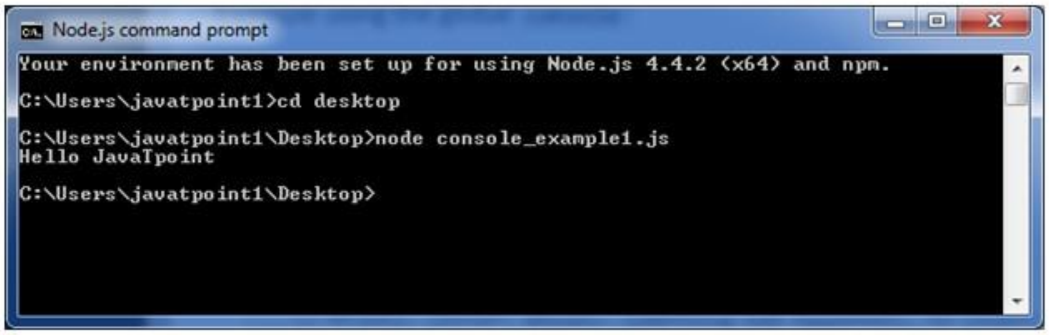
Output:



```
C:\Program Files\nodejs\node.exe .\HelloNodeJS.js
Hello World
```

Open Node.js command prompt and run the following code:

node console_example1.js



```
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.

C:\Users\javatpoint1>cd desktop

C:\Users\javatpoint1\Desktop>node console_example1.js
Hello JavaTpoint

C:\Users\javatpoint1\Desktop>
```

Here, console.log() function displays message on console.

Node.js web-based Example

A node.js web application contains the following three parts:

1. **Import required modules:** The "require" directive is used to load a Node.js module.
2. **Create server:** You have to establish a server which will listen to client's request similar to Apache HTTP Server.
3. **Read request and return response:** Server created in the second step will read HTTP request made by client which can be a browser or console and return the response.

How to create node.js web applications

Follow these steps:

1. **Import required module:** The first step is to use **require** directive to load http module and store returned HTTP instance into http variable. For example: For example, var http = require("http");

2. **Create server:** In the second step, you have to use created http instance and call http.createServer() method to create server instance and then bind it at port 8081 using listen method associated with server instance. Pass it a function with request

and response parameters and write the sample implementation to return "Hello World". For example:

```
http.createServer(function (request, response) {
 // Send the HTTP header
// HTTP Status: 200 : OK        // Content Type: text/plain
response.writeHead(200, {'Content-Type': 'text/plain'});
  // Send the response body as "Hello World"    response.end('Hello
World\n');
}).listen(8081);
// Console will print the message   console.log('Server
running at http://127.0.0.1:8081/');
```

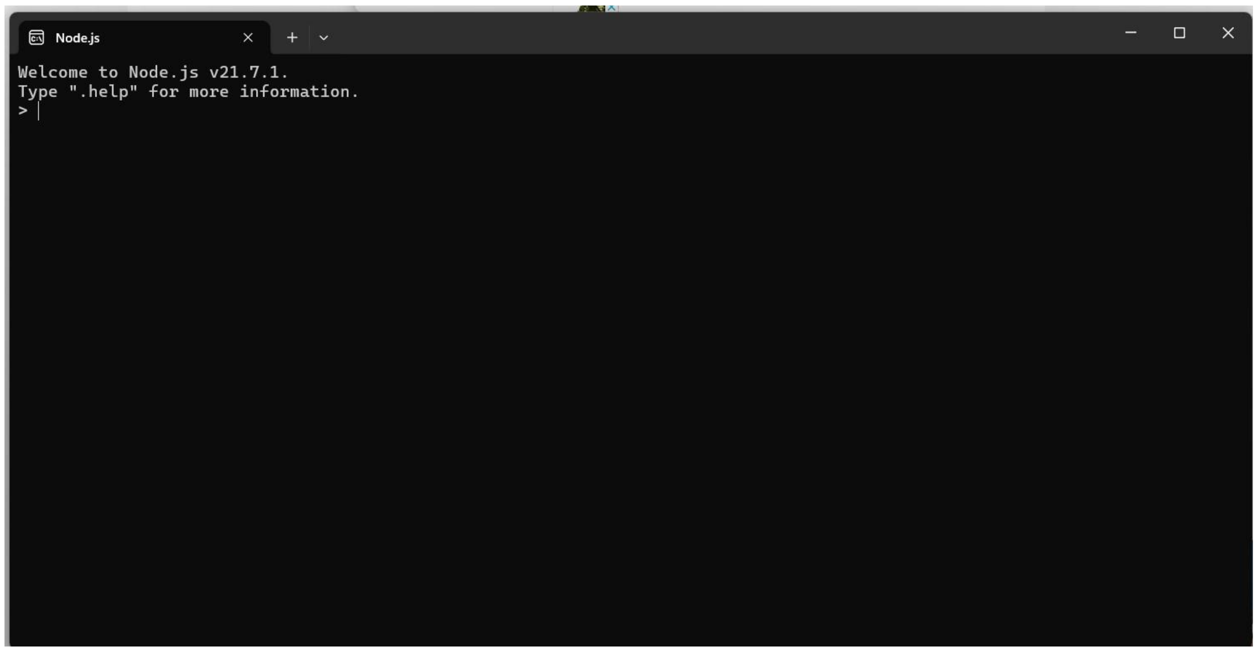
3. **Combine step1 and step2 together** in a file named "main.js".

```
File:   main.js   var   http   =   require("http");
http.createServer(function (request, response) {
 // Send the HTTP header
  // HTTP Status: 200 : OK      // Content Type: text/plain
response.writeHead(200, {'Content-Type': 'text/plain'});
  // Send the response body as "Hello World"    response.end('Hello
World\n');
}).listen(8081);
// Console will print the message   console.log('Server
running at http://127.0.0.1:8081/');
```

How to start your server:

Go to start menu and click on the Node.js command prompt.

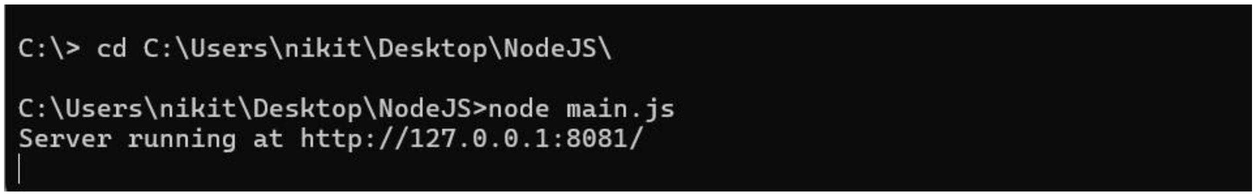
Now command prompt is open:



```
Welcome to Node.js v21.7.1.
Type ".help" for more information.
>
```

Set path: Here we have save "main.js" file on the desktop.

So type **cd desktop** on the command prompt. After that execute the main.js to start the server as follows: node main.js

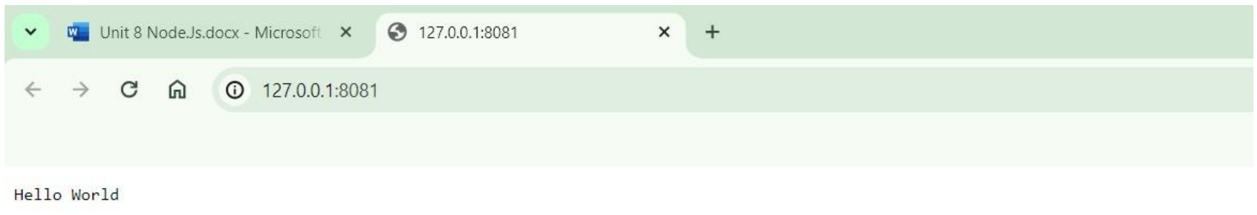


```
C:\> cd C:\Users\nikit\Desktop\NodeJS\

C:\Users\nikit\Desktop\NodeJS>node main.js
Server running at http://127.0.0.1:8081/
```

Now server is started.

Make a request to Node.js server:

Open http://127.0.0.1:8081/ in any browser. You will see the following result.



```
Hello World
```

Node.js Console

The Node.js console module provides a simple debugging console similar to JavaScript console mechanism provided by web browsers.

There are three console methods that are used to write any node.js stream:

1. console.log()
2. console.error()
3. console.warn()

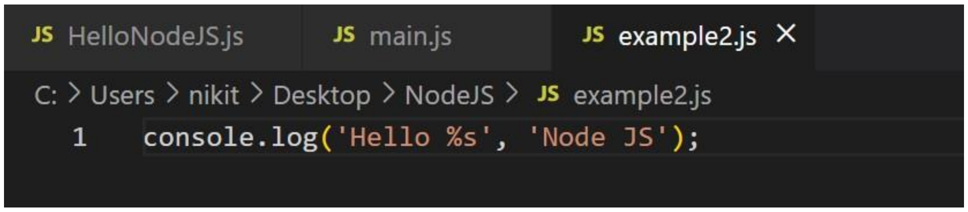
Node.js console.log()

The console.log() function is used to display simple message on console.

File: console_example1.js console.log('Hello

JavaTpoint');

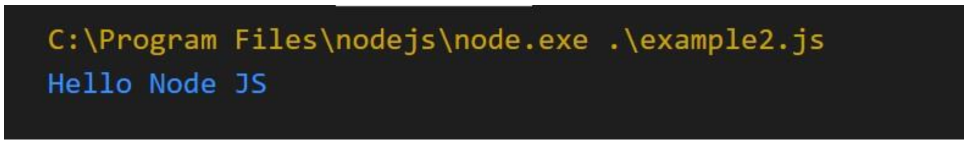
We can also use format specifier in console.log() function.



HelloNodeJS.js   main.js   example2.js

C: > Users > nikit > Desktop > NodeJS > example2.js

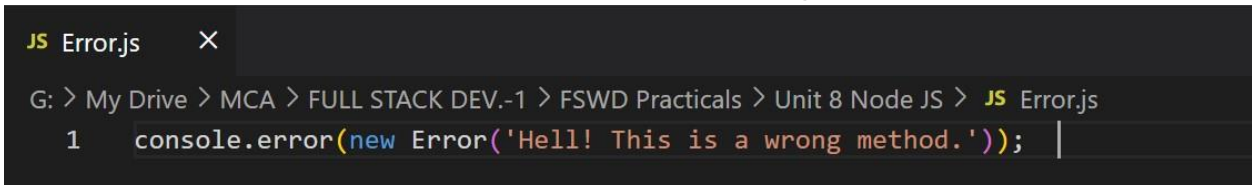
```
console.log('Hello %s', 'Node JS');
```



```
C:\Program Files\nodejs\node.exe .\example2.js
Hello Node JS
```

Node.js console.error()

The console.error() function is used to render error message on console.

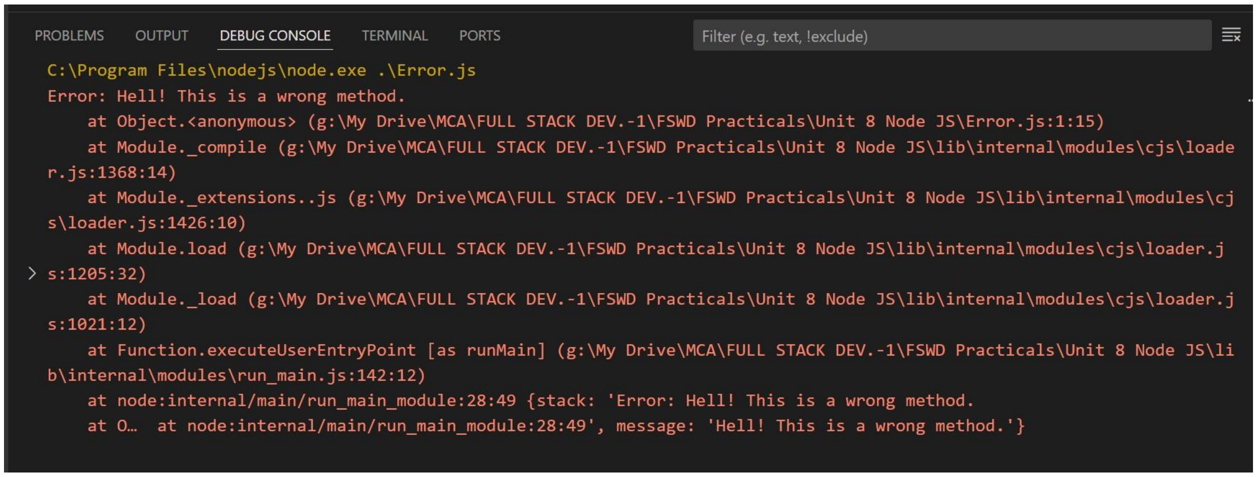


Error.js

G: > My Drive > MCA > FULL STACK DEV.-1 > FSWD Practicals > Unit 8 Node JS > Error.js

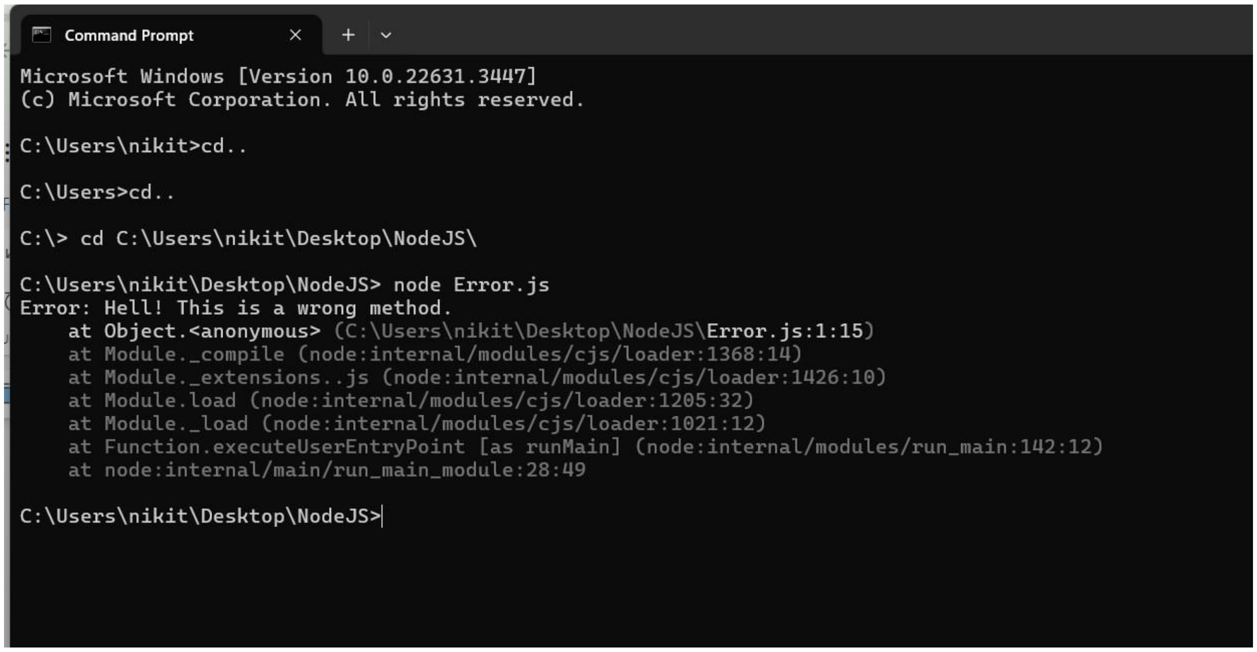
```
console.error(new Error('Hell! This is a wrong method.'));
```

Ouput in VS Code:



```
C:\Program Files\nodejs\node.exe .\Error.js
Error: Hell! This is a wrong method.
    at Object.<anonymous> (g:\My Drive\MCA\FULL STACK DEV.-1\FSWD Practicals\Unit 8 Node JS\Error.js:1:15)
    at Module._compile (g:\My Drive\MCA\FULL STACK DEV.-1\FSWD Practicals\Unit 8 Node JS\lib\internal\modules\cjs\loade
r.js:1368:14)
    at Module._extensions..js (g:\My Drive\MCA\FULL STACK DEV.-1\FSWD Practicals\Unit 8 Node JS\lib\internal\modules\cj
s\loader.js:1426:10)
    at Module.load (g:\My Drive\MCA\FULL STACK DEV.-1\FSWD Practicals\Unit 8 Node JS\lib\internal\modules\cjs\loader.j
s:1205:32)
    at Module._load (g:\My Drive\MCA\FULL STACK DEV.-1\FSWD Practicals\Unit 8 Node JS\lib\internal\modules\cjs\loader.j
s:1021:12)
    at Function.executeUserEntryPoint [as runMain] (g:\My Drive\MCA\FULL STACK DEV.-1\FSWD Practicals\Unit 8 Node JS\li
b\internal\modules\run_main.js:142:12)
    at node:internal/main/run_main_module:28:49 {stack: 'Error: Hell! This is a wrong method.
    at O…  at node:internal/main/run_main_module:28:49', message: 'Hell! This is a wrong method.'}
```

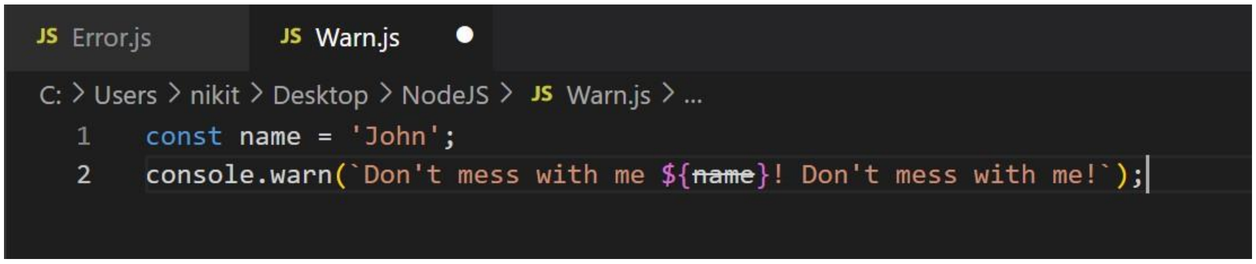
Output in Command prompt:



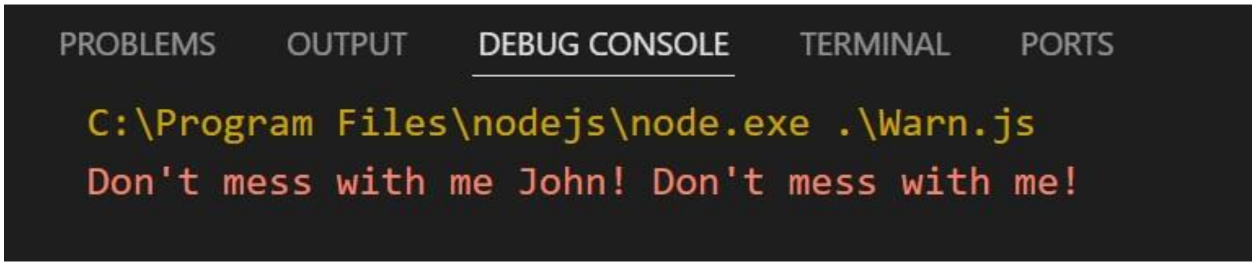
```
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nikit>cd..

C:\Users>cd..

C:\> cd C:\Users\nikit\Desktop\NodeJS\

C:\Users\nikit\Desktop\NodeJS> node Error.js
Error: Hell! This is a wrong method.
    at Object.<anonymous> (C:\Users\nikit\Desktop\NodeJS\Error.js:1:15)
    at Module._compile (node:internal/modules/cjs/loader:1368:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1426:10)
    at Module.load (node:internal/modules/cjs/loader:1205:32)
    at Module._load (node:internal/modules/cjs/loader:1021:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:142:12)
    at node:internal/main/run_main_module:28:49

C:\Users\nikit\Desktop\NodeJS>
```

Node.js console.warn()

The console.warn() function is used to display warning message on console.



```
const name = 'John';
console.warn(`Don't mess with me ${~~name~~}! Don't mess with me!`);
```



```
C:\Program Files\nodejs\node.exe .\Warn.js
Don't mess with me John! Don't mess with me!
```

Node.js REPL

The term REPL stands for **Read Eval Print** and **Loop**. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands and the system responds with an output in an interactive mode.

REPL Environment

The Node.js or node come bundled with REPL environment. Each part of the REPL environment has a specific work.

Read: It reads user's input; parse the input into JavaScript data-structure and stores in memory.

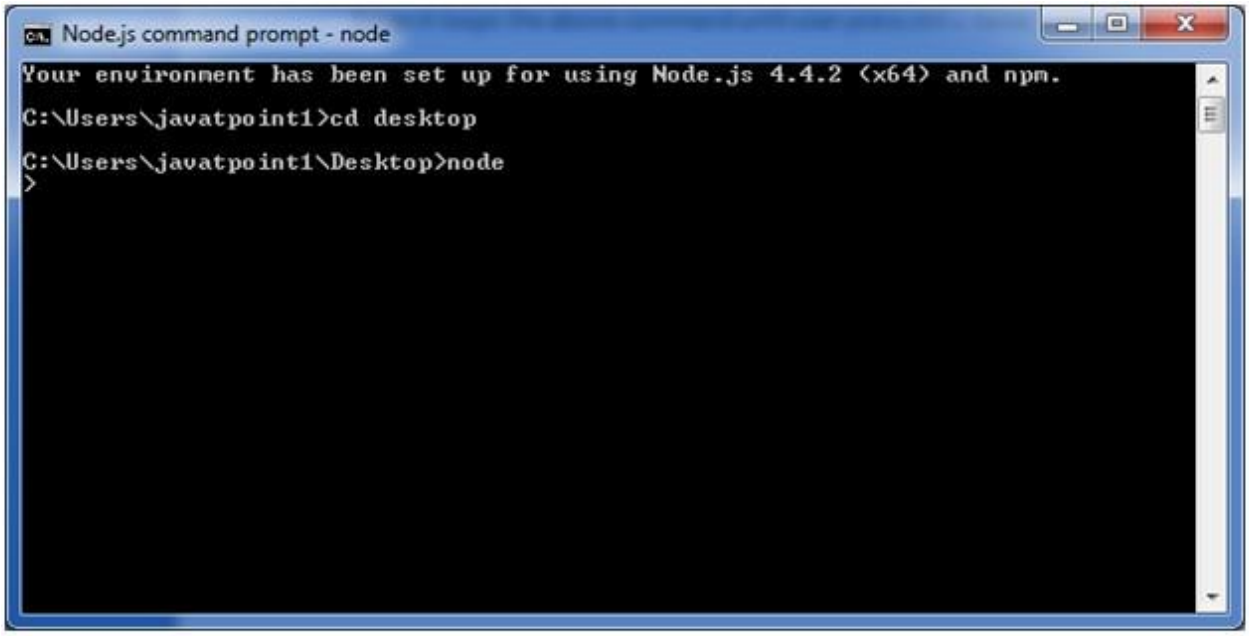
Eval: It takes and evaluates the data structure.

Print: It prints the result.

Loop: It loops the above command until user press ctrl-c twice.

How to start REPL

You can start REPL by simply running "node" on the command prompt. See this:



```
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.

C:\Users\javatpoint1>cd desktop

C:\Users\javatpoint1\Desktop>node
>
```

You can execute various mathematical operations on REPL Node.js command prompt:

Node.js Simple expressions

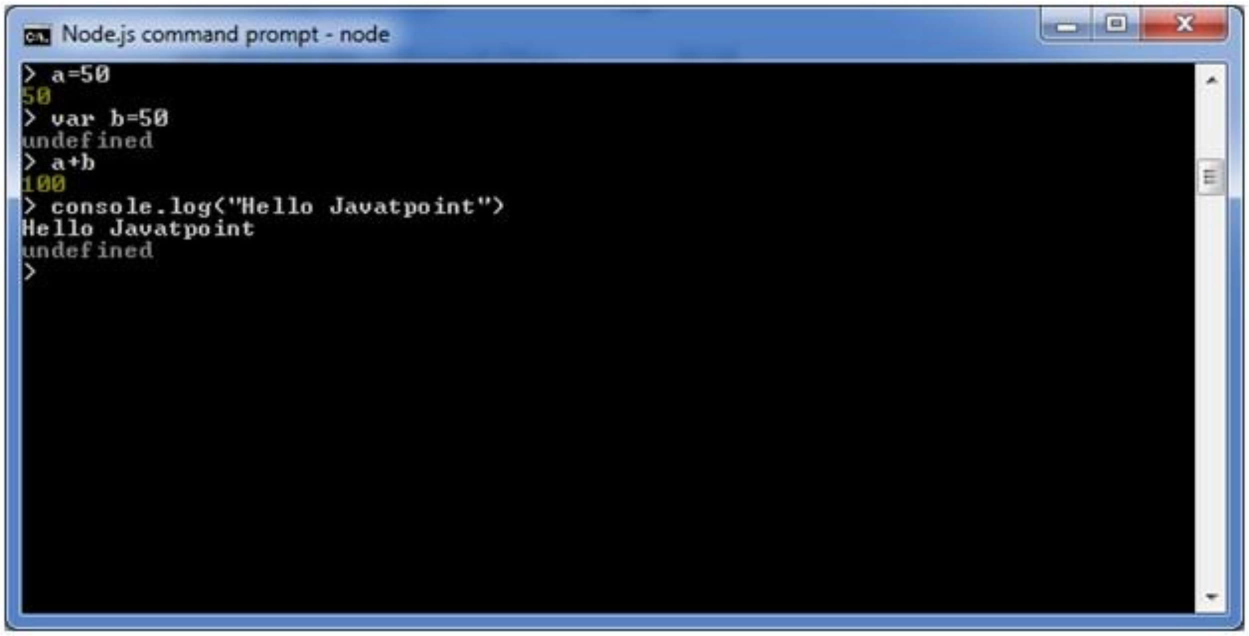
After starting REPL node command prompt put any mathematical expression:

Example: >10+20-5

25

Using variable

Variables are used to store values and print later. If you don't use **var** keyword then value is stored in the variable and printed whereas if **var** keyword is used then value is stored but not printed. You can print variables using console.log().

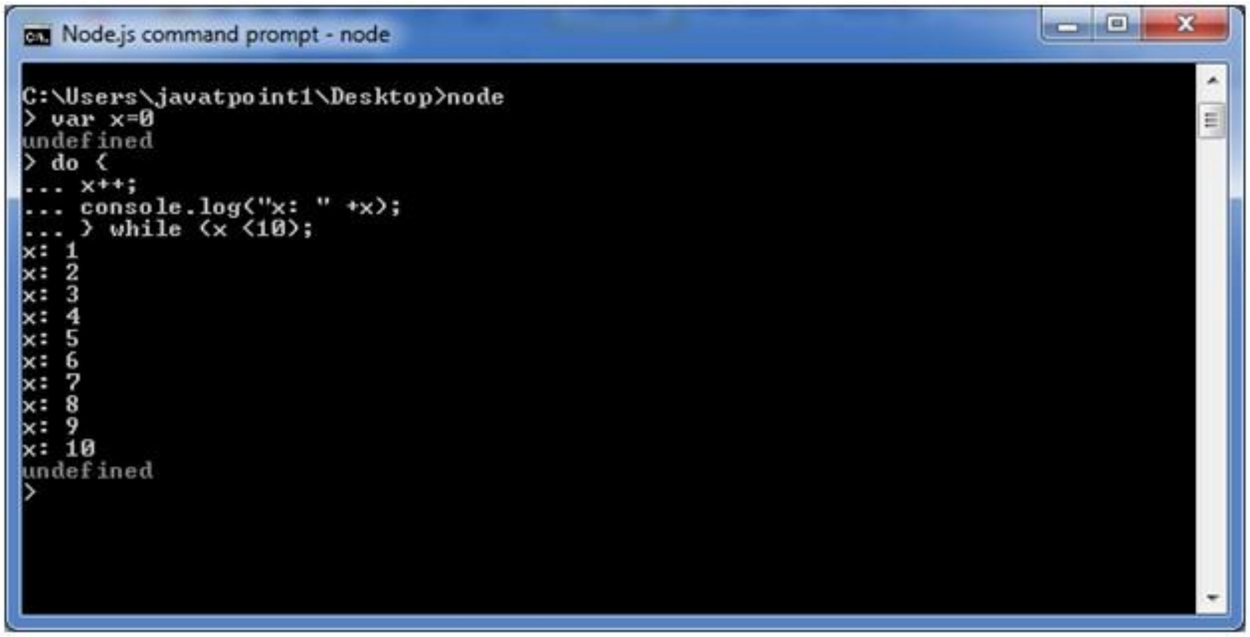


```
> a=50
50
> var b=50
undefined
> a+b
100
> console.log("Hello Javatpoint")
Hello Javatpoint
undefined
>
```

Node.js Multiline expressions

Node REPL supports multiline expressions like JavaScript. See the following do-while loop example:

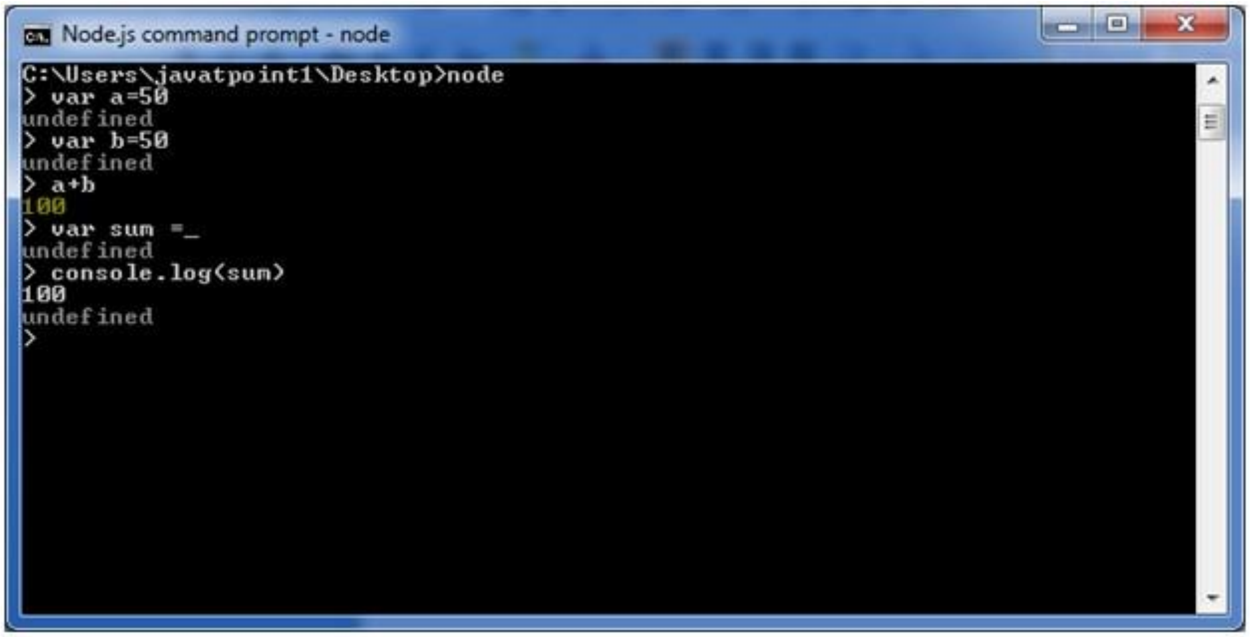
```
var x = 0   undefined
> do {
... x++;
... console.log("x: " + x);
... } while ( x < 10 );
```



```
C:\Users\javatpoint1\Desktop>node
> var x=0
undefined
> do {
... x++;
... console.log("x: " +x);
... } while (x <10);
x: 1
x: 2
x: 3
x: 4
x: 5
x: 6
x: 7
x: 8
x: 9
x: 10
undefined
>
```

Node.js Underscore Variable

You can also use underscore _ to get the last result.



```
C:\Users\javatpoint1\Desktop>node
> var a=50
undefined
> var b=50
undefined
> a+b
100
> var sum =_
undefined
> console.log(sum)
100
undefined
>
```

Node.js REPL Commands

Commands	Description
ctrl + c	It is used to terminate the current command.
ctrl + c twice	It terminates the node repl.
ctrl + d	It terminates the node repl.
up/down keys	It is used to see command history and modify previous commands.
tab keys	It specifies the list of current command.
.help	It specifies the list of all commands.
.break	It is used to exit from multi-line expressions.

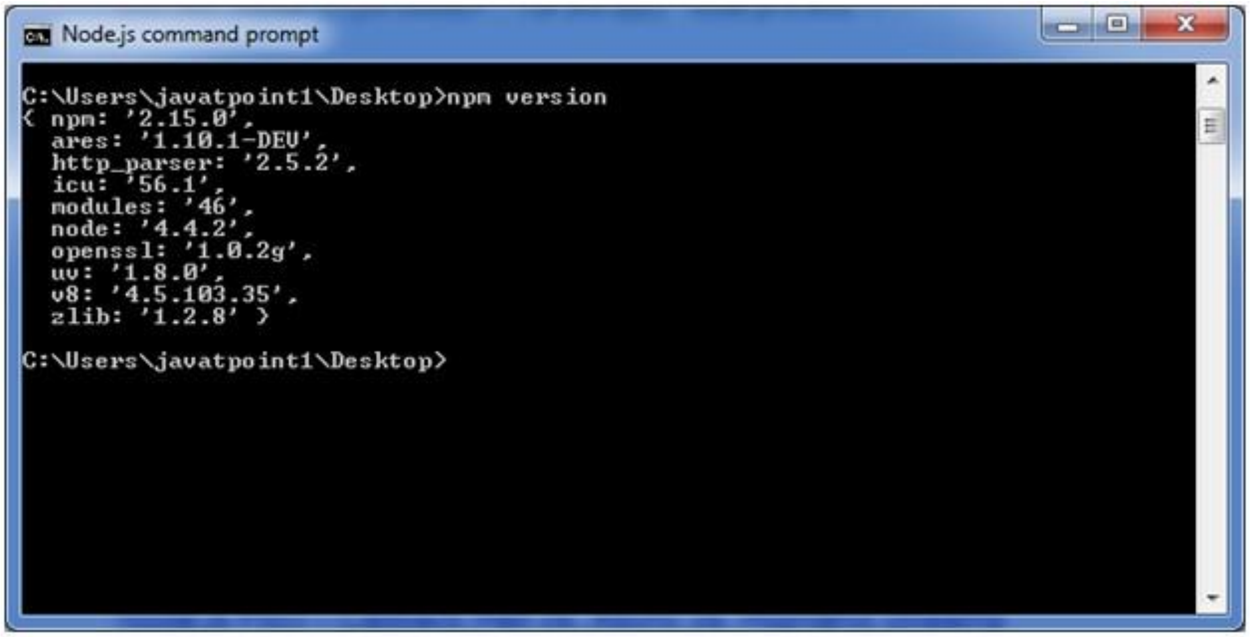
.clear	It is used to exit from multi-line expressions.
.save filename	It saves current node repl session to a file.
.load filename	It is used to load file content in current node repl session.

Node.js Package Manager

Node Package Manager provides two main functionalities:

- It provides online repositories for node.js packages/modules which are searchable on search.nodejs.org
- It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

The npm comes bundled with Node.js installables in versions after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:



```
C:\Users\javatpoint1\Desktop>npm version
{ npm: '2.15.0',
  ares: '1.10.1-DEV',
  http_parser: '2.5.2',
  icu: '56.1',
  modules: '46',
  node: '4.4.2',
  openssl: '1.0.2g',
  uv: '1.8.0',
  v8: '4.5.103.35',
  zlib: '1.2.8' }

C:\Users\javatpoint1\Desktop>
```

Installing Modules using npm

Following is the syntax to install any Node.js module: npm

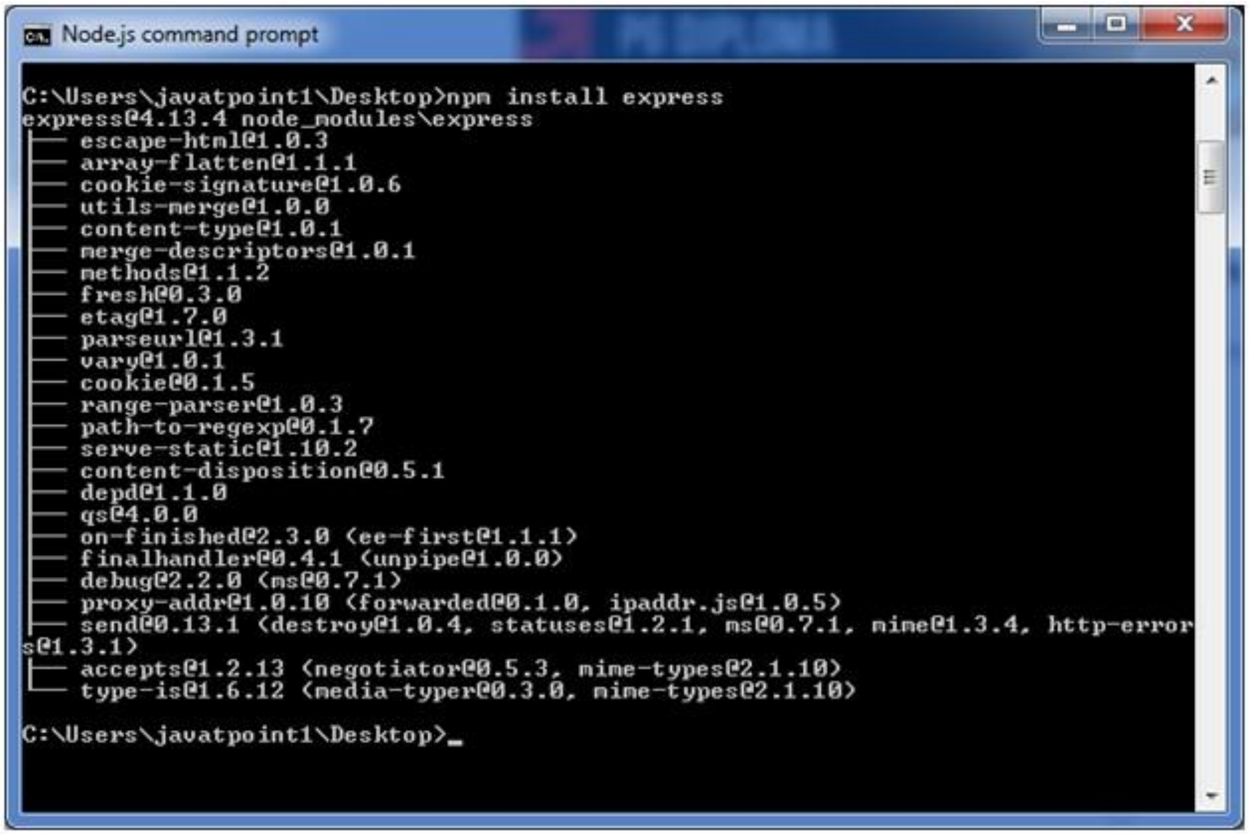
install **<Module** Name**>**

Let's install a famous Node.js web framework called express:

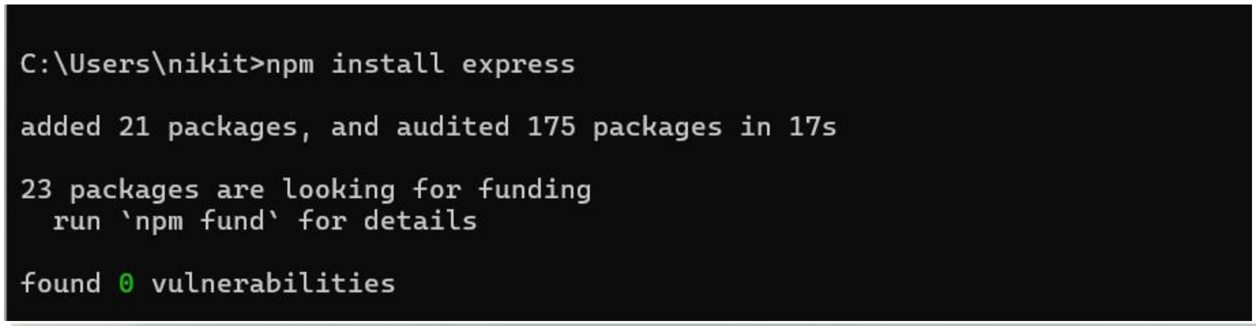
Open the Node.js command prompt and execute the following command:

npm install express

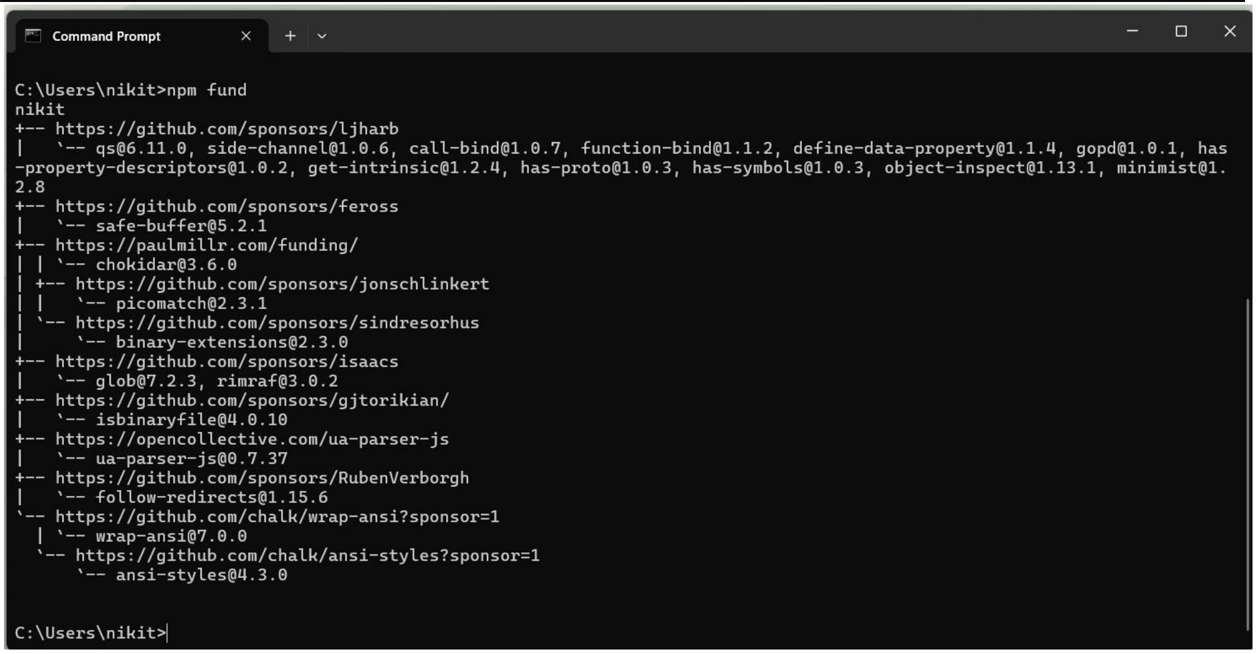
You can see the result after installing the "express" framework.



```
C:\Users\javatpoint1\Desktop>npm install express
express@4.13.4 node_modules\express
├── escape-html@1.0.3
├── array-flatten@1.1.1
├── cookie-signature@1.0.6
├── utils-merge@1.0.0
├── content-type@1.0.1
├── merge-descriptors@1.0.1
├── methods@1.1.2
├── fresh@0.3.0
├── etag@1.7.0
├── parseurl@1.3.1
├── vary@1.0.1
├── cookie@0.1.5
├── range-parser@1.0.3
├── path-to-regexp@0.1.7
├── serve-static@1.10.2
├── content-disposition@0.5.1
├── depd@1.1.0
├── qs@4.0.0
├── on-finished@2.3.0 (ee-first@1.1.1)
├── finalhandler@0.4.1 (unpipe@1.0.0)
├── debug@2.2.0 (ms@0.7.1)
├── proxy-addr@1.0.10 (forwarded@0.1.0, ipaddr.js@1.0.5)
├── send@0.13.1 (destroy@1.0.4, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-error
s@1.3.1)
├── accepts@1.2.13 (negotiator@0.5.3, mime-types@2.1.10)
└── type-is@1.6.12 (media-typer@0.3.0, mime-types@2.1.10)

C:\Users\javatpoint1\Desktop>_
```



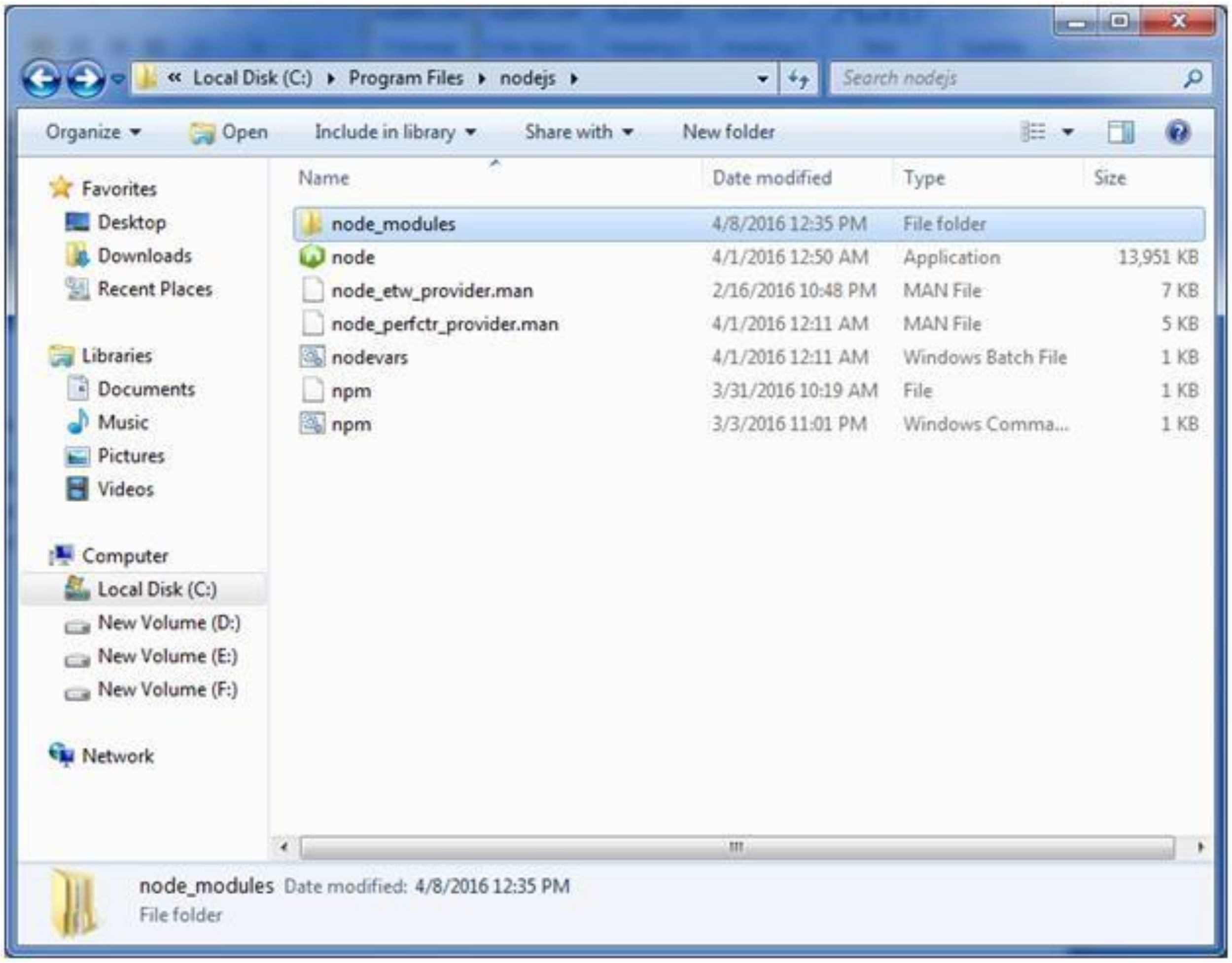
```
C:\Users\nikit>npm install express

added 21 packages, and audited 175 packages in 17s

23 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```



```
C:\Users\nikit>npm fund
nikit
+-- https://github.com/sponsors/ljharb
|   `-- qs@6.11.0, side-channel@1.0.6, call-bind@1.0.7, function-bind@1.1.2, define-data-property@1.1.4, gopd@1.0.1, has
-property-descriptors@1.0.2, get-intrinsic@1.2.4, has-proto@1.0.3, has-symbols@1.0.3, object-inspect@1.13.1, minimist@1.
2.8
+-- https://github.com/sponsors/feross
|   `-- safe-buffer@5.2.1
+-- https://paulmillr.com/funding/
| | `-- chokidar@3.6.0
| +-- https://github.com/sponsors/jonschlinkert
| |   `-- picomatch@2.3.1
| `-- https://github.com/sponsors/sindresorhus
|     `-- binary-extensions@2.3.0
+-- https://github.com/sponsors/isaacs
|   `-- glob@7.2.3, rimraf@3.0.2
+-- https://github.com/sponsors/gjtorikian/
|   `-- isbinaryfile@4.0.10
+-- https://opencollective.com/ua-parser-js
|   `-- ua-parser-js@0.7.37
+-- https://github.com/sponsors/RubenVerborgh
|   `-- follow-redirects@1.15.6
`-- https://github.com/chalk/wrap-ansi?sponsor=1
  | `-- wrap-ansi@7.0.0
  `-- https://github.com/chalk/ansi-styles?sponsor=1
      `-- ansi-styles@4.3.0


C:\Users\nikit>
```

Global vs Local Installation

By default, npm installs dependency in local mode. Here local mode specifies the folder where Node application is present. For example if you installed express module, it created node_modules directory in the current directory where it installed express module.

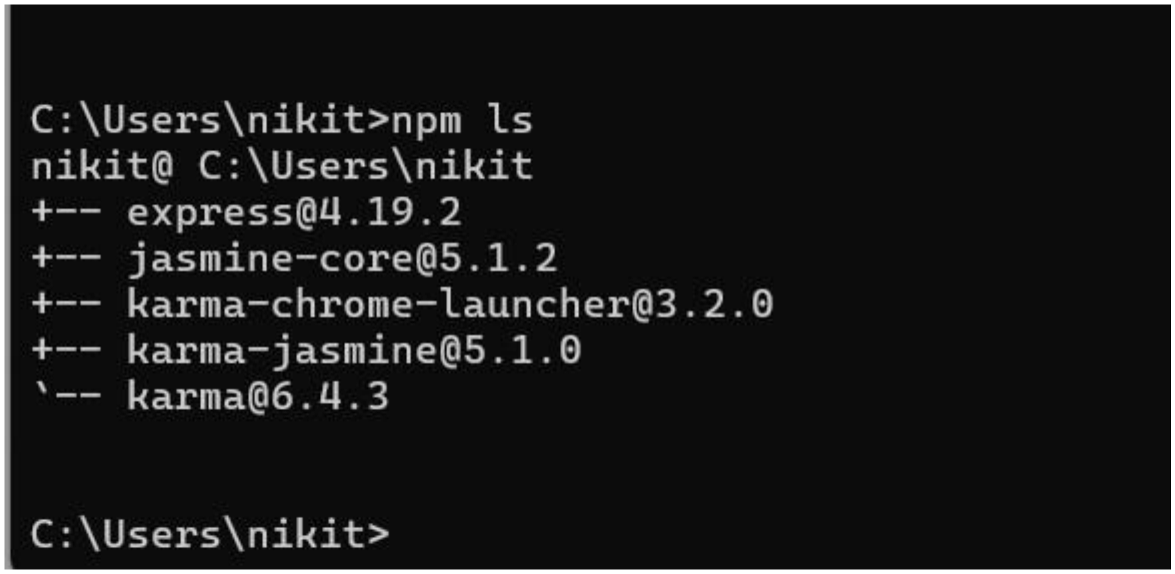


Name	Date modified	Type	Size
node_modules	4/8/2016 12:35 PM	File folder	
node	4/1/2016 12:50 AM	Application	13,951 KB
node_etw_provider.man	2/16/2016 10:48 PM	MAN File	7 KB
node_perfctr_provider.man	4/1/2016 12:11 AM	MAN File	5 KB
nodevars	4/1/2016 12:11 AM	Windows Batch File	1 KB
npm	3/31/2016 10:19 AM	File	1 KB
npm	3/3/2016 11:01 PM	Windows Comma...	1 KB

node_modules Date modified: 4/8/2016 12:35 PM
File folder

You can use npm ls command to list down all the locally installed modules.

Open the Node.js command prompt and execute "npm ls":

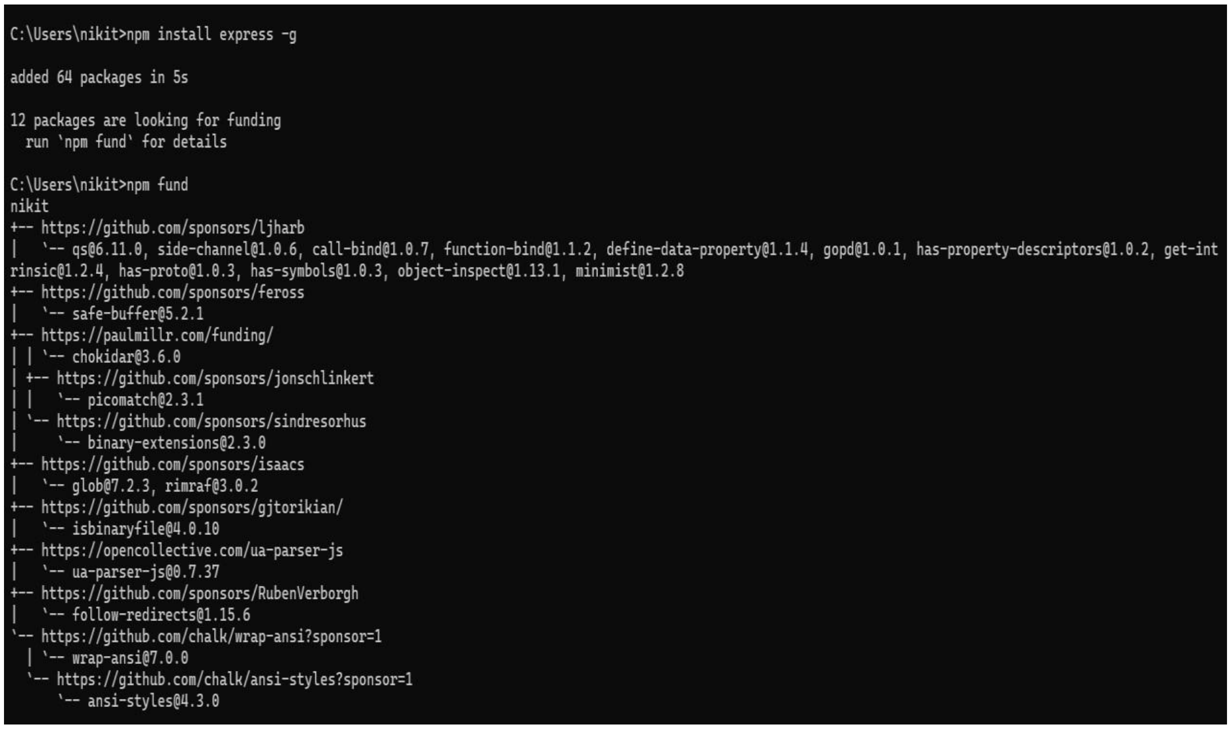


```
C:\Users\nikit>npm ls
nikit@ C:\Users\nikit
+-- express@4.19.2
+-- jasmine-core@5.1.2
+-- karma-chrome-launcher@3.2.0
+-- karma-jasmine@5.1.0
`-- karma@6.4.3


C:\Users\nikit>
```

Globally installed packages/dependencies are stored in system directory. Let's install express module using global installation. Although it will also produce the same result but modules will be installed globally.

Open Node.js command prompt and execute the following code:

npm install express -g



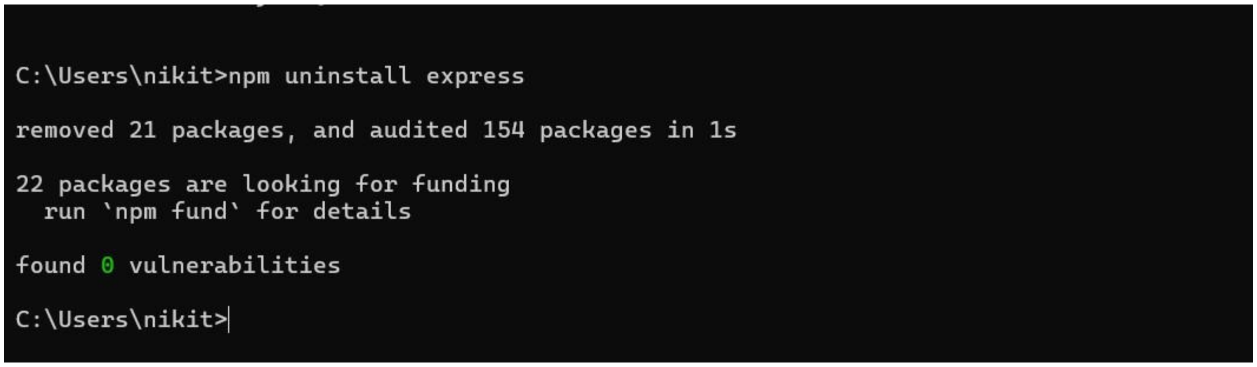
```
C:\Users\nikit>npm install express -g

added 64 packages in 5s

12 packages are looking for funding
  run `npm fund` for details

C:\Users\nikit>npm fund
nikit
+-- https://github.com/sponsors/ljharb
|   `-- qs@6.11.0, side-channel@1.0.6, call-bind@1.0.7, function-bind@1.1.2, define-data-property@1.1.4, gopd@1.0.1, has-property-descriptors@1.0.2, get-int
rinsic@1.2.4, has-proto@1.0.3, has-symbols@1.0.3, object-inspect@1.13.1, minimist@1.2.8
+-- https://github.com/sponsors/feross
|   `-- safe-buffer@5.2.1
+-- https://paulmillr.com/funding/
| | `-- chokidar@3.6.0
| +-- https://github.com/sponsors/jonschlinkert
| |   `-- picomatch@2.3.1
| `-- https://github.com/sponsors/sindresorhus
|     `-- binary-extensions@2.3.0
+-- https://github.com/sponsors/isaacs
|   `-- glob@7.2.3, rimraf@3.0.2
+-- https://github.com/sponsors/gjtorikian/
|   `-- isbinaryfile@4.0.10
+-- https://opencollective.com/ua-parser-js
|   `-- ua-parser-js@0.7.37
+-- https://github.com/sponsors/RubenVerborgh
|   `-- follow-redirects@1.15.6
`-- https://github.com/chalk/wrap-ansi?sponsor=1
  | `-- wrap-ansi@7.0.0
  `-- https://github.com/chalk/ansi-styles?sponsor=1
      `-- ansi-styles@4.3.0
```

Here first line tells about the module version and its location where it is getting installed.

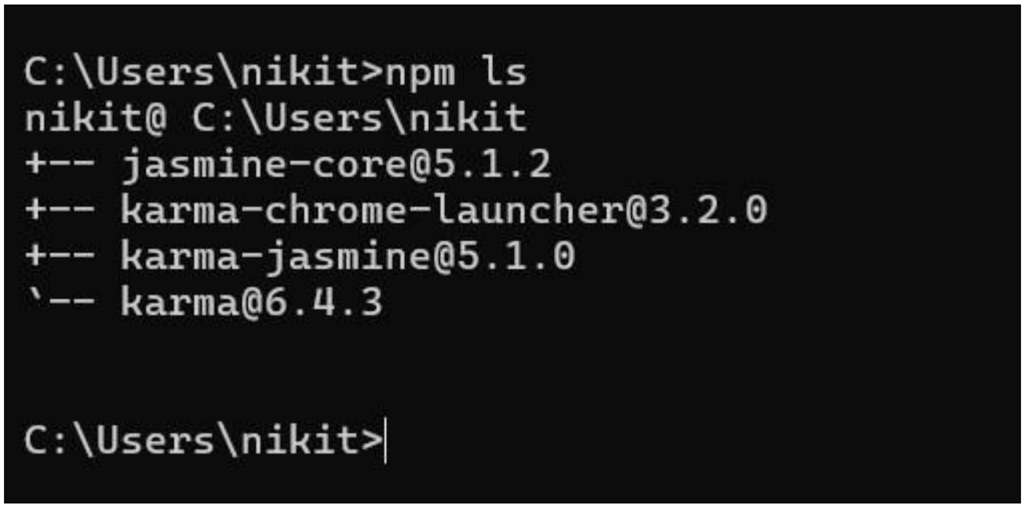
Uninstalling a Module

To uninstall a Node.js module, use the following command: npm

uninstall express



```
C:\Users\nikit>npm uninstall express

removed 21 packages, and audited 154 packages in 1s

22 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Users\nikit>
```

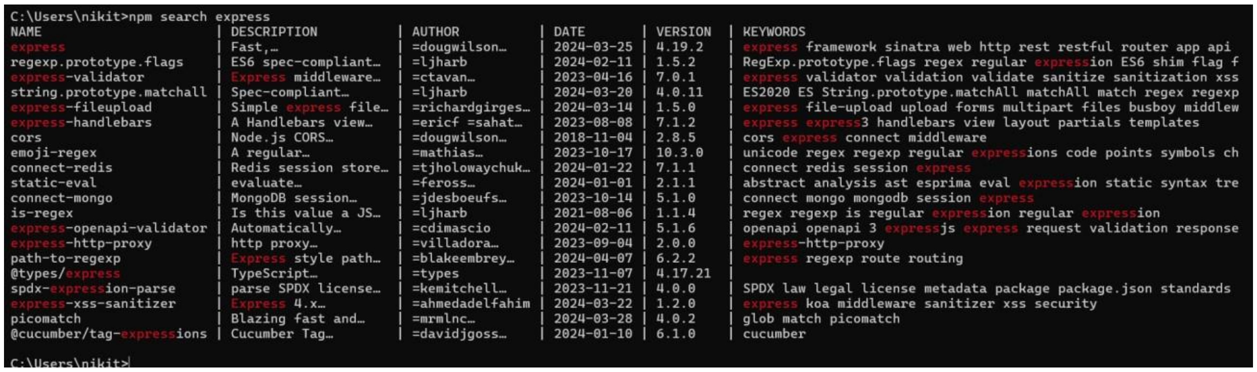


```
C:\Users\nikit>npm ls
nikit@ C:\Users\nikit
+-- jasmine-core@5.1.2
+-- karma-chrome-launcher@3.2.0
+-- karma-jasmine@5.1.0
`-- karma@6.4.3


C:\Users\nikit>
```

Searching a Module

"npm search express" command is used to search express or module.

npm search express



```
C:\Users\nikit>npm search express
NAME                        | DESCRIPTION              | AUTHOR            | DATE       | VERSION  | KEYWORDS
express                     | Fast,…                   | =dougwilson…      | 2024-03-25 | 4.19.2   | express framework sinatra web http rest restful router app api
regexp.prototype.flags      | ES6 spec-compliant…      | =ljharb           | 2024-02-11 | 1.5.2    | RegExp.prototype.flags regex regular expression ES6 shim flag f
express-validator           | Express middleware…      | =ctavan…          | 2023-04-16 | 7.0.1    | express validator validation validate sanitize sanitization xss
string.prototype.matchall   | Spec-compliant…          | =ljharb           | 2024-03-20 | 4.0.11   | ES2020 ES String.prototype.matchAll matchAll match regex regexp
express-fileupload          | Simple express file…     | =richardgirges…   | 2024-03-14 | 1.5.0    | express file-upload upload forms multipart files busboy middlew
express-handlebars          | A Handlebars view…       | =ericf =sahat…    | 2023-08-08 | 7.1.2    | express express3 handlebars view layout partials templates
cors                        | Node.js CORS…            | =dougwilson…      | 2018-11-04 | 2.8.5    | cors express connect middleware
emoji-regex                 | A regular…               | =mathias…         | 2023-10-17 | 10.3.0   | unicode regex regexp regular expressions code points symbols ch
connect-redis               | Redis session store…     | =tjholowaychuk…   | 2024-01-22 | 7.1.1    | connect redis session express
static-eval                 | evaluate…                | =feross…          | 2024-01-01 | 2.1.1    | abstract analysis ast esprima eval expression static syntax tre
connect-mongo               | MongoDB session…         | =jdesboeufs…      | 2023-10-14 | 5.1.0    | connect mongo mongodb session express
is-regex                    | Is this value a JS…      | =ljharb           | 2021-08-06 | 1.1.4    | regex regexp is regular expression regular expression
express-openapi-validator   | Automatically…           | =cdimascio        | 2024-02-11 | 5.1.6    | openapi openapi 3 expressjs express request validation response
express-http-proxy          | http proxy…              | =villadora…       | 2023-09-04 | 2.0.0    | express-http-proxy
path-to-regexp              | Express style path…      | =blakeembrey…     | 2024-04-07 | 6.2.2    | express regexp route routing
@types/express              | TypeScript…              | =types            | 2023-11-07 | 4.17.21  |
spdx-expression-parse       | parse SPDX license…      | =kemitchell…      | 2023-11-21 | 4.0.0    | SPDX law legal license metadata package package.json standards
express-xss-sanitizer       | Express 4.x…             | =ahmedadelfahim   | 2024-03-22 | 1.2.0    | express koa middleware sanitizer xss security
picomatch                   | Blazing fast and…        | =mrmlnc…          | 2024-03-28 | 4.0.2    | glob match picomatch
@cucumber/tag-expressions   | Cucumber Tag…            | =davidjgoss…      | 2024-01-10 | 6.1.0    | cucumber

C:\Users\nikit>
```

Node.js Command Line Options

There is a wide variety of command line options in Node.js. These options provide multiple ways to execute scripts and other helpful run-time options.

Let's see the list of Node.js command line options:

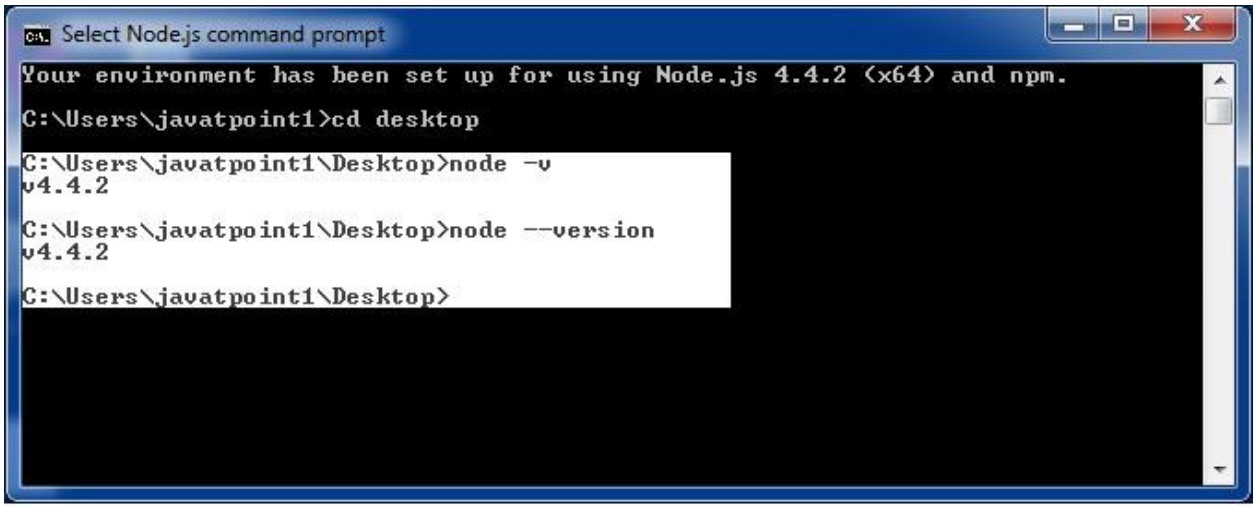
Index	Option	Description
1.	v, --version	It is used to print node's version.
2.	-h, --help	It is used to print node command line options.
3.	-e, --eval "script"	It evaluates the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script.
4.	-p, --print "script"	It is identical to -e but prints the result.
5.	-c, --check	Syntax check the script without executing.
6.	-i, --interactive	It opens the REPL even if stdin does not appear to be a terminal.
7.	-r, --require module	It is used to preload the specified module at startup. It follows require()'s module resolution rules. Module may be either a path to a file, or a node module name.
8.	--no-deprecation	Silence deprecation warnings.
9.	--trace-deprecation	It is used to print stack traces for deprecations.

10.	--throw-deprecation	It throws errors for deprecations.
11.	--no-warnings	It silence all process warnings (including deprecations).
12.	--trace-warnings	It prints stack traces for process warnings (including deprecations).
13.	--trace-sync-io	It prints a stack trace whenever synchronous i/o is detected after the first turn of the event loop.
14.	--zero-fill-buffers	Automatically zero-fills all newly allocated buffer and slowbuffer instances.
15.	--track-heap-objects	It tracks heap object allocations for heap snapshots.
16.	--prof-process	It processes V8 profiler output generated using the v8 option --prof.
17.	--V8-options	It prints V8 command line options.
18.	--tls-cipher-list=list	It specifies an alternative default tls cipher list. (requires node.js to be built with crypto support. (default))
19.	--enable-fips	It enables fips-compliant crypto at startup. (requires node.js to be built with ./configure --openssl-fips)
20.	--force-fips	It forces fips-compliant crypto on startup. (cannot be disabled from script code.) (same requirements as -enable-fips)
21.	--icu-data-dir=file	It specifies ICU data load path. (Overrides node_icu_data)

Node.js Command Line Options Examples

To see the version of the running Node:

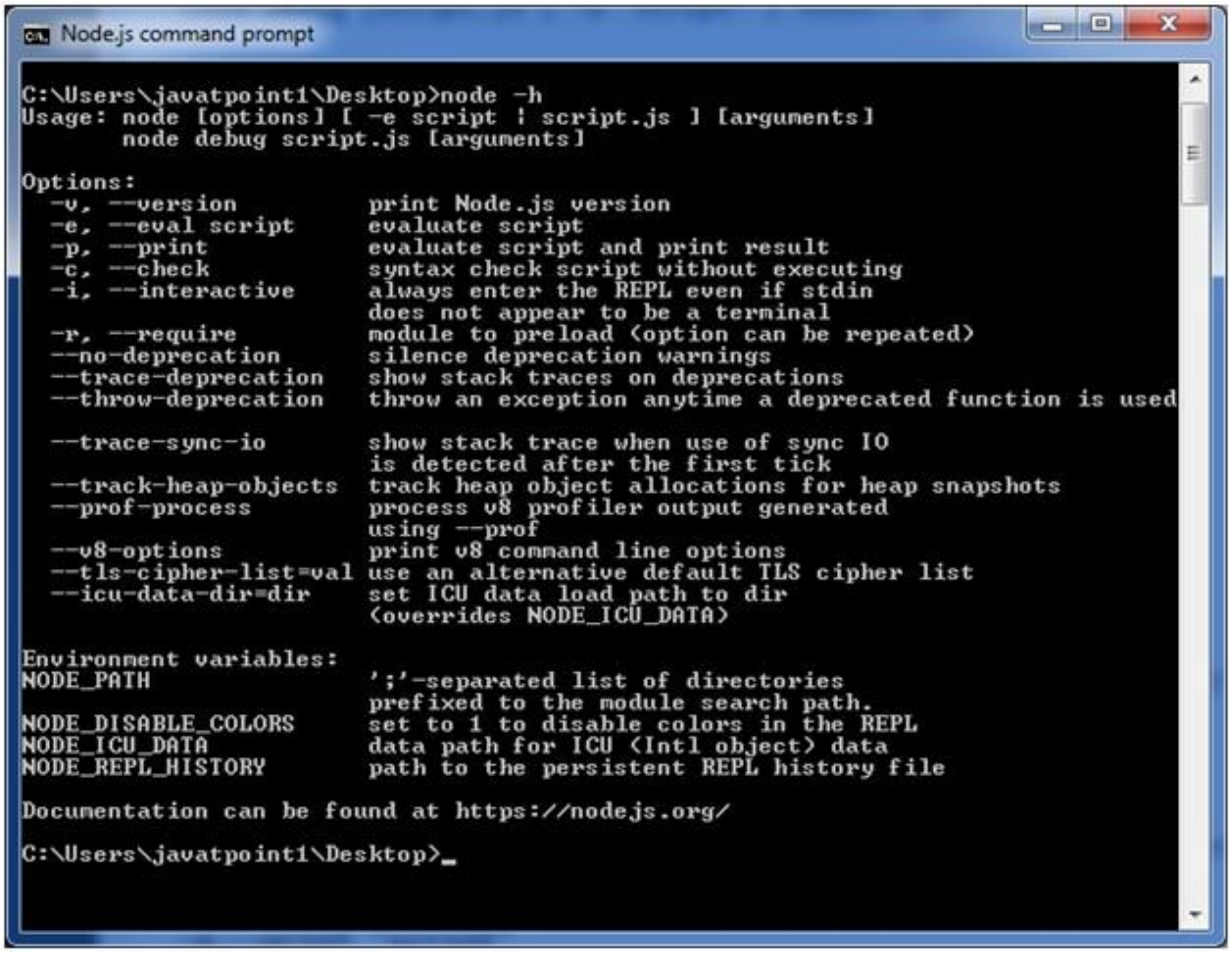
Open Node.js command prompt and run command node -v or node --version



```
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.

C:\Users\javatpoint1>cd desktop

C:\Users\javatpoint1\Desktop>node -v
v4.4.2

C:\Users\javatpoint1\Desktop>node --version
v4.4.2

C:\Users\javatpoint1\Desktop>
```

For Help:

Use command node ?h or node --help



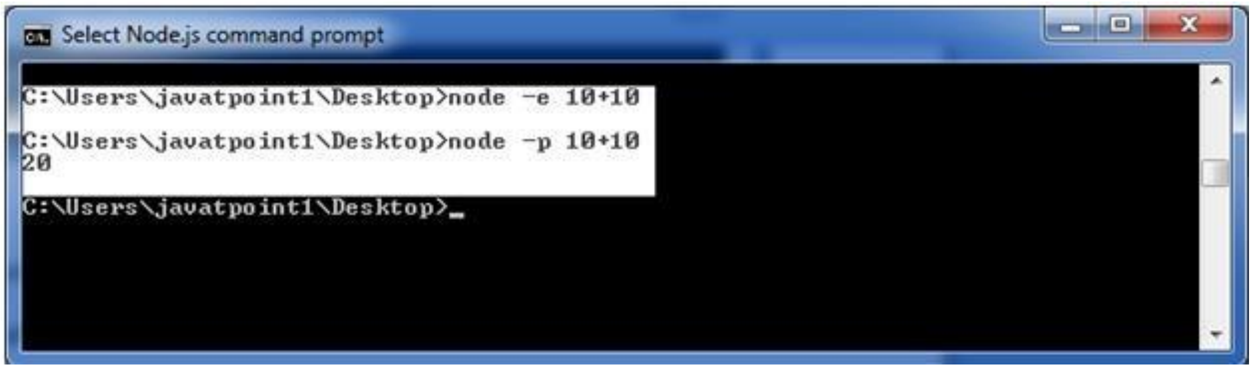
```
C:\Users\javatpoint1\Desktop>node -h
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]

Options:
  -v, --version         print Node.js version
  -e, --eval script     evaluate script
  -p, --print           evaluate script and print result
  -c, --check           syntax check script without executing
  -i, --interactive     always enter the REPL even if stdin
                        does not appear to be a terminal
  -r, --require         module to preload (option can be repeated)
  --no-deprecation      silence deprecation warnings
  --trace-deprecation   show stack traces on deprecations
  --throw-deprecation   throw an exception anytime a deprecated function is used

  --trace-sync-io       show stack trace when use of sync IO
                        is detected after the first tick
  --track-heap-objects  track heap object allocations for heap snapshots
  --prof-process        process v8 profiler output generated
                        using --prof
  --v8-options          print v8 command line options
  --tls-cipher-list=val use an alternative default TLS cipher list
  --icu-data-dir=dir    set ICU data load path to dir
                        (overrides NODE_ICU_DATA)

Environment variables:
NODE_PATH               ';'-separated list of directories
                        prefixed to the module search path.
NODE_DISABLE_COLORS     set to 1 to disable colors in the REPL
NODE_ICU_DATA           data path for ICU (Intl object) data
NODE_REPL_HISTORY       path to the persistent REPL history file

Documentation can be found at https://nodejs.org/

C:\Users\javatpoint1\Desktop>_
```

To evaluate an argument (but not print result):

Use command node -e, --eval "script"

To evaluate an argument and print result also:

Use command node -p "script"



```
C:\Users\javatpoint1\Desktop>node -e 10+10

C:\Users\javatpoint1\Desktop>node -p 10+10
20

C:\Users\javatpoint1\Desktop>_
```

Node.js Global Objects

Node.js global objects are global in nature and available in all modules. You don't need to include these objects in your application; rather they can be used directly. These objects are modules, functions, strings and object etc. Some of these objects aren't actually in the global scope but in the module scope.

A list of Node.js global objects are given below:

- __dirname
- __filename
- Console
- Process
- Buffer
- setImmediate(callback[, arg][, ...])
- setInterval(callback, delay[, arg][, ...])
- setTimeout(callback, delay[, arg][, ...])
- clearImmediate(immediateObject)
- clearInterval(intervalObject)
- clearTimeout(timeoutObject)

Node.js __dirname

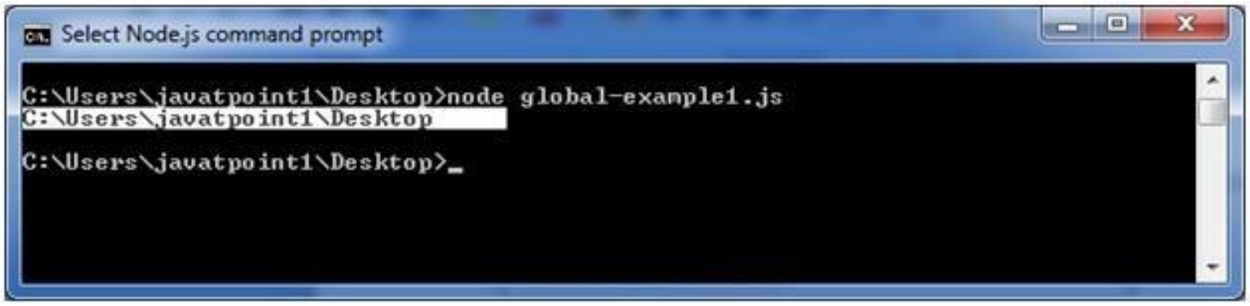
It is a string. It specifies the name of the directory that currently contains the code.

File: global-example1.js

```
console.log(__dirname);
```

Open Node.js command prompt and run the following code:

```
node global-example1.js
```



```
C:\Users\javatpoint1\Desktop>node global-example1.js
C:\Users\javatpoint1\Desktop

C:\Users\javatpoint1\Desktop>_
```

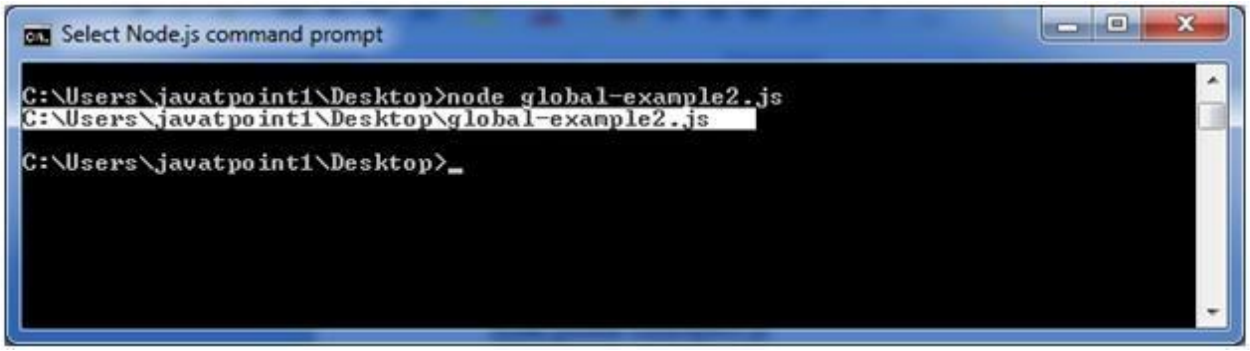
Node.js __filename

It specifies the filename of the code being executed. This is the resolved absolute path of this code file. The value inside a module is the path to that module file.

File: global-example2.js console.log(__filename);

Open Node.js command prompt and run the following code:

node global-example2.js



```
C:\Users\javatpoint1\Desktop>node global-example2.js
C:\Users\javatpoint1\Desktop\global-example2.js

C:\Users\javatpoint1\Desktop>_
```

Node.js Timer

Node.js Timer functions are global functions. You don't need to use require() function in order to use timer functions. Let's see the list of timer functions.

Set timer functions:

- **setImmediate():** It is used to execute setImmediate.
- **setInterval():** It is used to define a time interval.
- **setTimeout():** ()- It is used to execute a one-time callback after delay milliseconds.

Clear timer functions:

- **clearImmediate(immediateObject):** It is used to stop an immediateObject, as created by setImmediate
- **clearInterval(intervalObject):** It is used to stop an intervalObject, as created by setInterval
- **clearTimeout(timeoutObject):** It prevents a timeoutObject, as created by setTimeout

Node.js Timer setInterval() Example

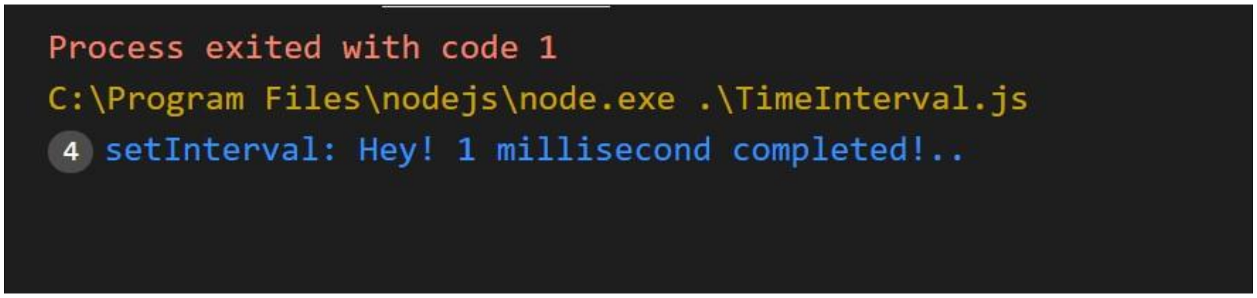
This example will set a time interval of 1000 millisecond and the specified comment will be displayed after every 1000 millisecond until you terminate.

File: timer1.js

```
setInterval(function() {

 console.log("setInterval: Hey! 1 millisecond completed!..");

}, 1000);
```

Open Node.js command prompt and run the following code:

node timer1.js



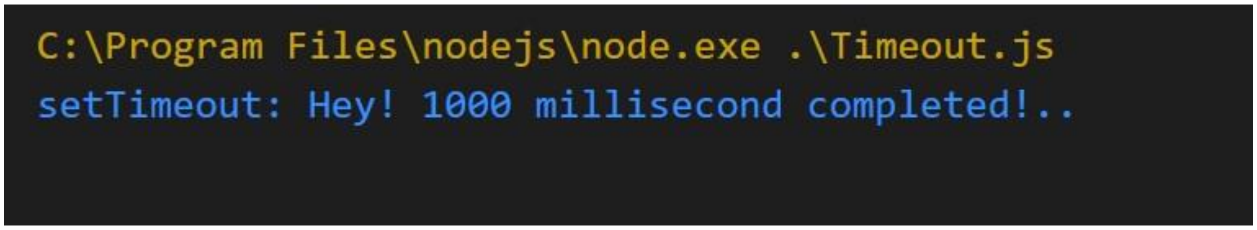
```
Process exited with code 1
C:\Program Files\nodejs\node.exe .\TimeInterval.js
4 setInterval: Hey! 1 millisecond completed!..
```

Node.js Timer setTimeout() Example

File: timer1.js

```
setTimeout(function() {
console.log("setTimeout: Hey! 1000 millisecond completed!..");
}, 1000);
```

Open Node.js command prompt and run the following code:



```
C:\Program Files\nodejs\node.exe .\Timeout.js
setTimeout: Hey! 1000 millisecond completed!..
```

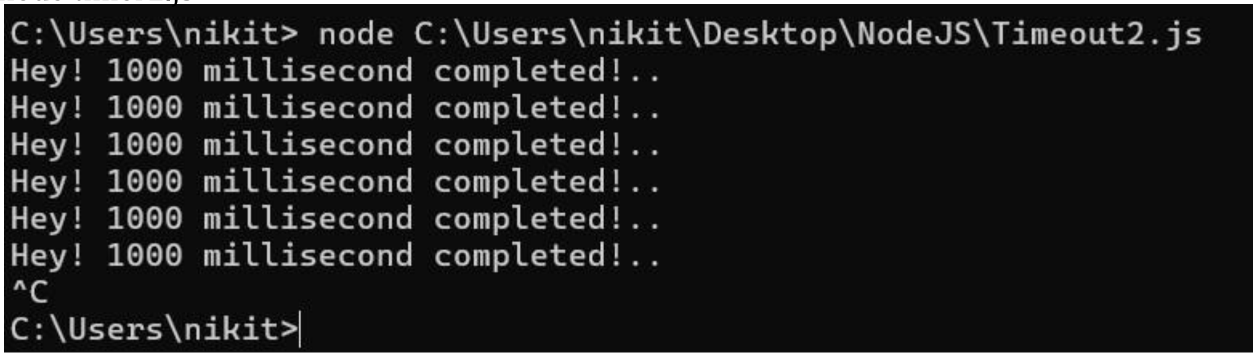
his example shows time out after every 1000 millisecond without setting a time interval. This example uses the recursion property of a function.

File: timer2.js

```
var recursive = function () {
  console.log("Hey! 1000 millisecond completed!..");
  setTimeout(recursive,1000);
}
recursive();
```

Open Node.js command prompt and run the following code:

node timer2.js

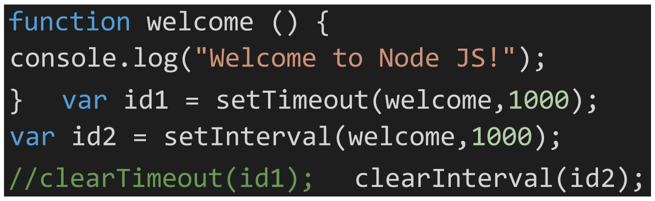


```
C:\Users\nikit> node C:\Users\nikit\Desktop\NodeJS\Timeout2.js
Hey! 1000 millisecond completed!..
Hey! 1000 millisecond completed!..
Hey! 1000 millisecond completed!..
Hey! 1000 millisecond completed!..
Hey! 1000 millisecond completed!..
Hey! 1000 millisecond completed!..
^C
C:\Users\nikit>
```

Node.js setInterval(), setTimeout() and clearInterval()

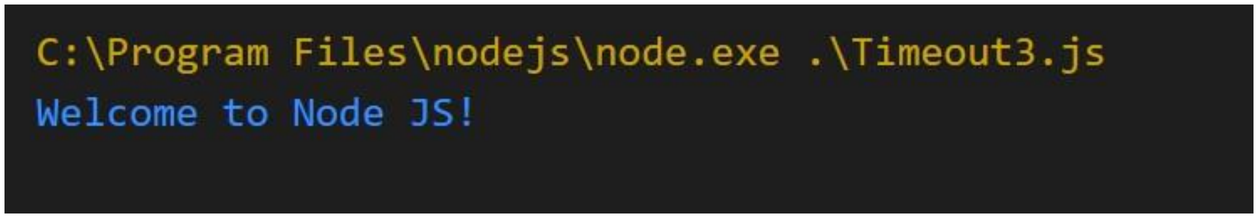
Let's see an example to use clearInterval() function.

File: timer3.js



```
function welcome () {
console.log("Welcome to Node JS!");
}   var id1 = setTimeout(welcome,1000);
var id2 = setInterval(welcome,1000);
//clearTimeout(id1);   clearInterval(id2);
```

Open Node.js command prompt and run the following code:



```
C:\Program Files\nodejs\node.exe .\Timeout3.js
Welcome to Node JS!
```

Node.js Errors

The Node.js applications generally face four types of errors:

- **Standard JavaScript errors** i.e. <EvalError>, <SyntaxError>, <RangeError>, <ReferenceError>, <TypeError>, <URIError> etc.
- **System errors**
- **User-specified errors**
- **Assertion errors**

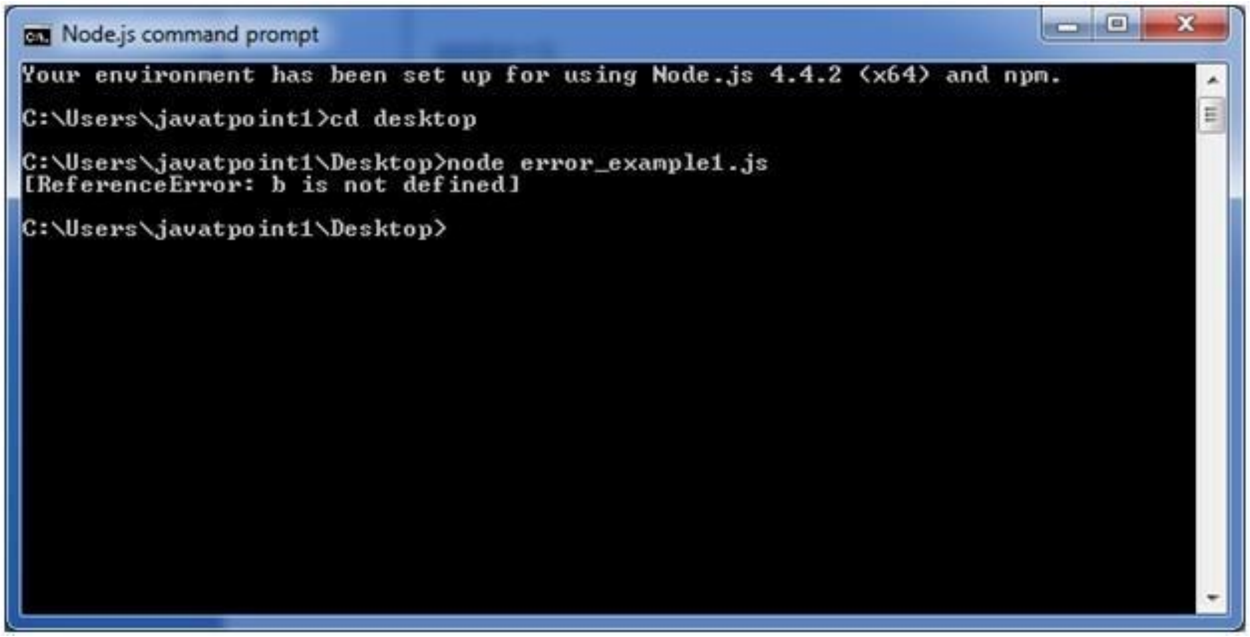
Node.js Errors Example 1

Let's take an example to deploy standard JavaScript error - ReferenceError.

File: error_example1.js

```
// Throws with a ReferenceError because b is undefined

try {     const a =

1;     const c = a +

b;   } catch (err) {

console.log(err);

}
```

Open Node.js command prompt and run the following code:



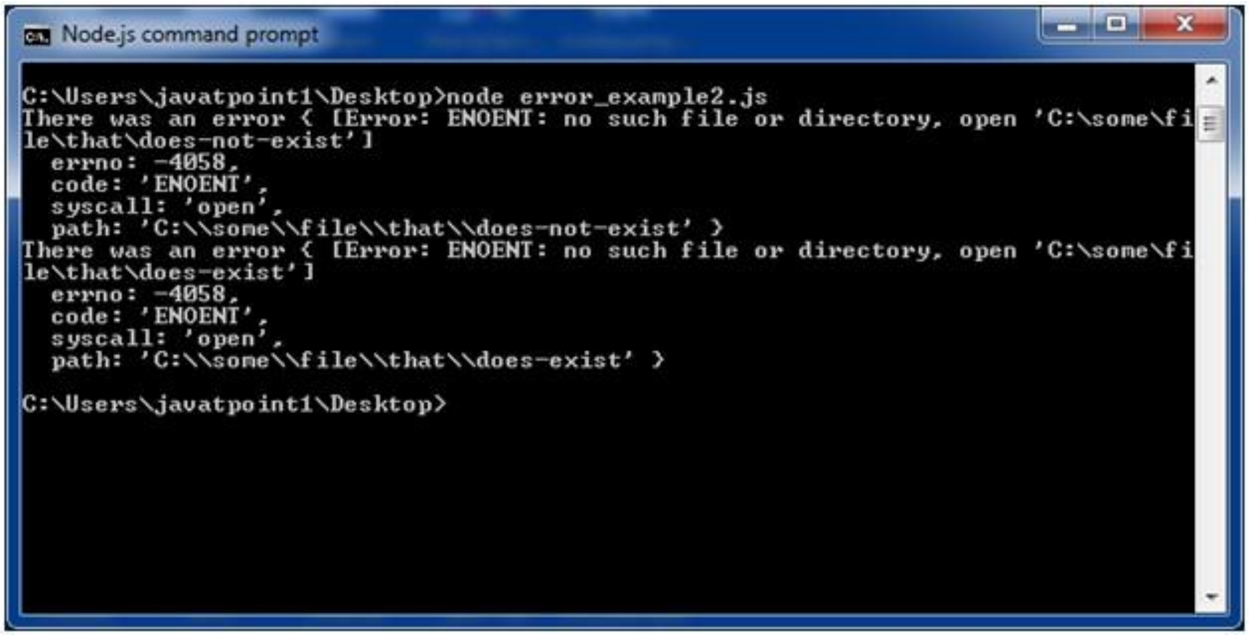
```
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.

C:\Users\javatpoint1>cd desktop

C:\Users\javatpoint1\Desktop>node error_example1.js
[ReferenceError: b is not defined]

C:\Users\javatpoint1\Desktop>
```

Node.js Errors Example 2

File: timer2.js

```
const fs = require('fs');
function nodeStyleCallback(err, data) {   if
(err) {
  console.error('There   was   an   error',    err);
return;
 }
 console.log(data);
}
fs.readFile('/some/file/that/does-not-exist', nodeStyleCallback);
fs.readFile('/some/file/that/does-exist', nodeStyleCallback);
```

Open Node.js command prompt and run the following code: node error_example2.js



```
C:\Users\javatpoint1\Desktop>node error_example2.js
There was an error { [Error: ENOENT: no such file or directory, open 'C:\some\fi
le\that\does-not-exist']
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\\some\\file\\that\\does-not-exist' }
There was an error { [Error: ENOENT: no such file or directory, open 'C:\some\fi
le\that\does-exist']
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\\some\\file\\that\\does-exist' }

C:\Users\javatpoint1\Desktop>
```

Node.js DNS

The Node.js DNS module contains methods to get information of given hostname. Let's see the list of commonly used DNS functions:

- dns.getServers()
  - The **dns.getServers() method** is an inbuilt application programming interface of the dns module which is used to get IP addresses of the current server.

    Syntax: `dns.getServers()` ○ **Parameters:** This method does not accept any parameters.

    Return: This method returns an array of IP addresses in RFC 5952 format as configured in DNS resolution for the current host. A string will be attached as the port number if a custom port is being used.

- dns.setServers(servers)

- dns.lookup(hostname[, options], callback) ○ The **dns.lookup() method** is an inbuilt application programming interface of the dns module which is used to resolve IP addresses of the specified hostname for given parameters into the first found A (IPv4) or AAAA (IPv6) record.
- dns.lookupService(address, port, callback)
- dns.resolve(hostname[, rrtype], callback)
- dns.resolve4(hostname, callback)
- dns.resolve6(hostname, callback)
- dns.resolveCname(hostname, callback)
- dns.resolveMx(hostname, callback)
- dns.resolveNs(hostname, callback)
- dns.resolveSoa(hostname, callback)
- dns.resolveSrv(hostname, callback)
- dns.resolvePtr(hostname, callback)
- dns.resolveTxt(hostname, callback)
- dns.reverse(ip, callback)

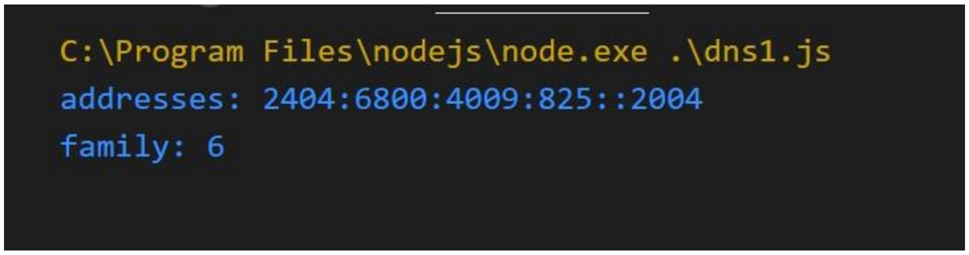
Node.js DNS Example 1

Let's see the example of dns.lookup() function.

File: dns_example1.js

```
const dns = require('dns');
dns.lookup('www.google.com', (err, addresses, family) => {
console.log('addresses:',                    addresses);
console.log('family:',family);
});
```

Open Node.js command prompt and run the following code:



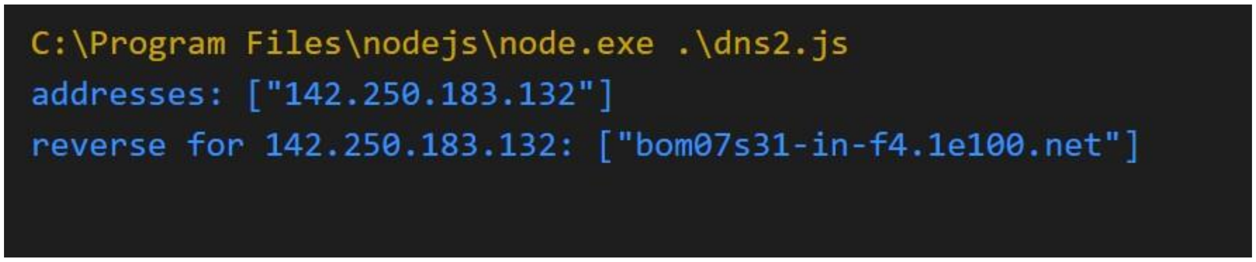
```
C:\Program Files\nodejs\node.exe .\dns1.js
addresses: 2404:6800:4009:825::2004
family: 6
```

Node.js DNS Example 2

Let's see the example of resolve4() and reverse() functions.

File: dns_example2.js

```
const dns = require('dns');
dns.resolve4('www.google.com', (err, addresses) => {
if (err) throw err;     console.log(`addresses:
${JSON.stringify(addresses)}`);     addresses.forEach((a)
=> {       dns.reverse(a, (err, hostnames) => {         if
(err) {
        throw err;
      }
      console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
    });
  });
});
```



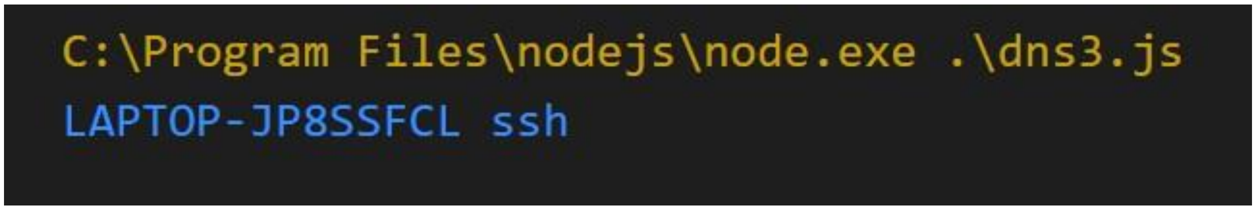
```
C:\Program Files\nodejs\node.exe .\dns2.js
addresses: ["142.250.183.132"]
reverse for 142.250.183.132: ["bom07s31-in-f4.1e100.net"]
```

Node.js DNS Example 3

Let's take an example to print the localhost name using lookupService() function.

File: dns_example3.js

```
const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
  console.log(hostname, service);
    // Prints: localhost
});
```



```
C:\Program Files\nodejs\node.exe .\dns3.js
LAPTOP-JP8SSFCL ssh
```

Node.js Net

Node.js provides the ability to perform socket programming. We can create chat application or communicate client and server applications using socket programming in Node.js. The Node.js net module contains functions for creating both servers and clients.

The syntax for including the Net module in your application:

```
var net = require('net');
```

Net Properties and Methods

Sr.No.	Method & Description
1	**net.createServer([options][, connectionListener])** Creates a new TCP server. The connectionListener argument is automatically set as a listener for the 'connection' event.
2	**net.connect(options[, connectionListener])** A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.
3	**net.createConnection(options[, connectionListener])** A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.

4	**net.connect(port[, host][, connectListener])** Creates a TCP connection to port on host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
5	**net.createConnection(port[, host][, connectListener])** Creates a TCP connection to port on host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
6	**net.connect(path[, connectListener])** Creates Unix socket connection to path. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
7	**net.createConnection(path[, connectListener])** Creates Unix socket connection to path. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
8	**net.isIP(input)** Tests if the input is an IP address. Returns 0 for invalid strings, 4 for IP version 4 addresses, and 6 for IP version 6 addresses.
9	**net.isIPv4(input)** Returns true if the input is a version 4 IP address, otherwise returns false.
10	**net.isIPv6(input)**
	Returns true if the input is a version 6 IP address, otherwise returns false.

Node.js Net Example

In this example, we are using two command prompts:

- Node.js command prompt for server.
- Window's default command prompt for client.

server:

File: net_server1.js

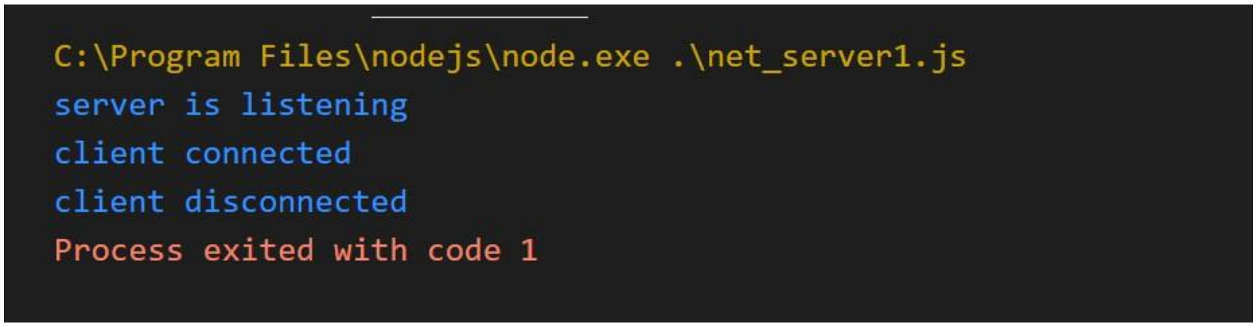
```
var net = require('net');
var server = net.createServer(function(connection) {
   console.log('client connected');

   connection.on('end', function() {
      console.log('client disconnected');
   });

   connection.write('Hello World!\r\n');
   connection.pipe(connection);
});

server.listen(8080, function() {
   console.log('server is listening');
});
```

File: net_client1.js

```
var net = require('net'); var client = net.connect({port: 8080},
function() {    console.log('connected to server!');
});

client.on('data', function(data) {
console.log(data.toString());
   client.end();
});

client.on('end', function() {
   console.log('disconnected from server');
});
```

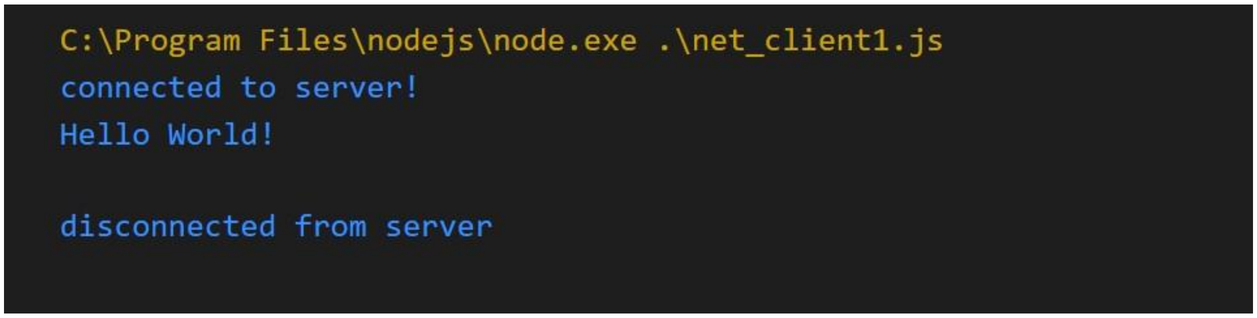
Output:

Server:



```
C:\Program Files\nodejs\node.exe .\net_server1.js
server is listening
client connected
client disconnected
Process exited with code 1
```

Client:



```
C:\Program Files\nodejs\node.exe .\net_client1.js
connected to server!
Hello World!

disconnected from server
```

Node.js Buffers

Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. Buffer class is used because pure JavaScript is not nice to binary data. So, when dealing with TCP streams or the file system, it's necessary to handle octet streams.

Buffer class is a global class. It can be accessed in application without importing buffer module.

Node.js Creating Buffers

There are many ways to construct a Node buffer. Following are the three mostly used methods:

1. **Create an uninitiated buffer:** Following is the syntax of creating an uninitiated buffer of 10 octets:

   ```
   var buf = new Buffer(10);
   ```

2. **Create a buffer from array:** Following is the syntax to create a Buffer from a given array:

var buf = new Buffer([10, 20, 30, 40, 50]);

3. **Create a buffer from string:** Following is the syntax to create a Buffer from a given string and optionally encoding type:

   var buf = new Buffer("Simply Easy Learning", "utf-8");

Node.js Writing to buffers

Following is the method to write into a Node buffer:

Syntax: buf.write(string[, offset][, length][,

encoding])

Parameter explanation:

string: It specifies the string data to be written to buffer. **offset:** It specifies the

index of the buffer to start writing at. Its default value is 0.

length: It specifies the number of bytes to write. Defaults to buffer.length **encoding:**

Encoding to use. 'utf8' is the default encoding.

Return values from writing buffers:

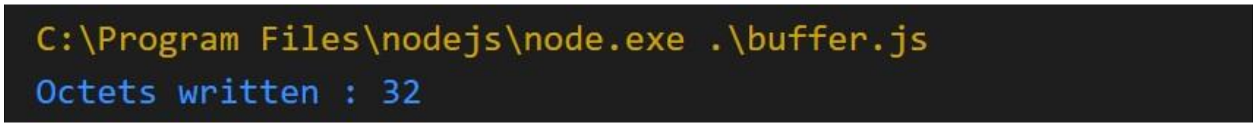
This method is used to return number of octets written. In the case of space shortage for buffer to fit the entire string, it will write a part of the string.

Let's take an example:

Create a JavaScript file named "main.js" having the following code:

File: main.js

```
buf = new Buffer(256);   len = buf.write("Simply
Easy Learning, hello node");
console.log("Octets written : "+  len);   Output:
```



```
C:\Program Files\nodejs\node.exe .\buffer.js
Octets written : 32
```

Node.js Reading from buffers

Following is the method to read data from a Node buffer.

Syntax:

buf.toString([encoding][, start][, end])

Parameter explanation:

encoding: It specifies encoding to use. 'utf8' is the default encoding **start:**

It specifies beginning index to start reading, defaults to 0.

end: It specifies end index to end reading, defaults is complete buffer.

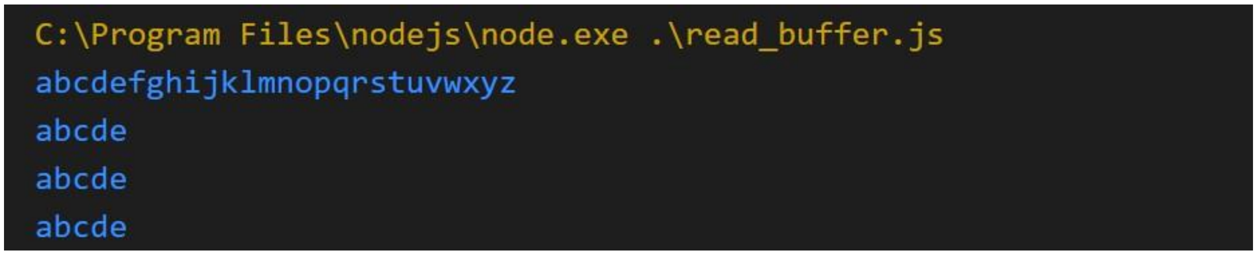
Return values reading from buffers:

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Let's take an example:

```
buf = new Buffer(26);   for (var
i = 0 ; i < 26 ; i++) {
buf[i] = i + 97;
}  console.log( buf.toString('ascii'));       // outputs:
abcdefghijklmnopqrstuvwxyz  console.log( buf.toString('ascii',0,5));   //
outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8', outputs
abcde
```

Output:



```
C:\Program Files\nodejs\node.exe .\read_buffer.js
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

Node.js Streams

Streams are the objects that facilitate you to read data from a source and write data to a destination. There are four types of streams in Node.js:

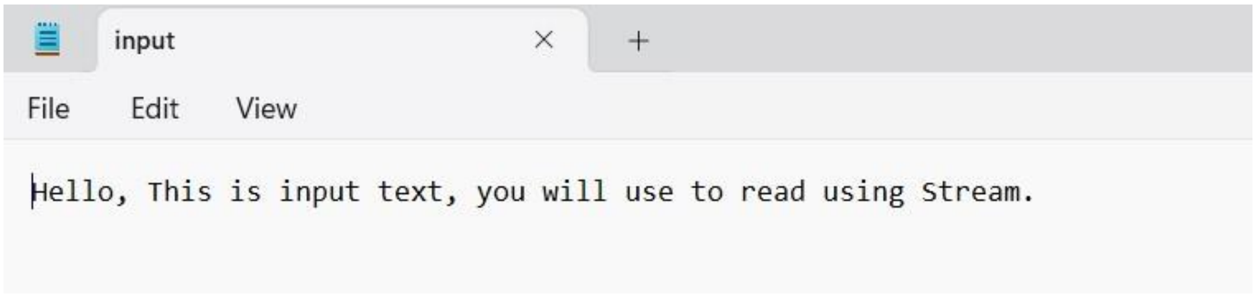
- **Readable:** This stream is used for read operations.
- **Writable:** This stream is used for write operations.
- **Duplex:** This stream can be used for both read and write operations.
- **Transform:** It is type of duplex stream where the output is computed according to input.

Each type of stream is an Event emitter instance and throws several events at different times. Following are some commonly used events:

- **Data:**This event is fired when there is data available to read.
- **End:**This event is fired when there is no more data available to read.
- **Error:** This event is fired when there is any error receiving or writing data.
- **Finish:**This event is fired when all data has been flushed to underlying system.

Node.js Reading from stream

Create a text file named input.txt having the following content:

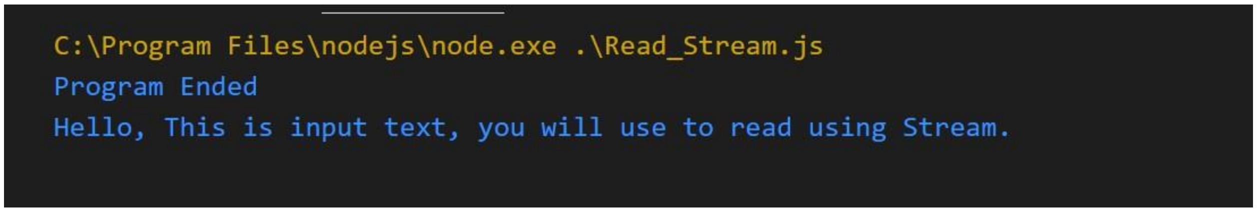


```
Hello, This is input text, you will use to read using Stream.
```

Read_Stream.js

```
var fs = require("fs");
var data = '';
// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');
// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});
readerStream.on('end',function(){
   console.log(data);
});
readerStream.on('error', function(err){
   console.log(err.stack);
});
console.log("Program Ended");
```

Output:



```
C:\Program Files\nodejs\node.exe .\Read_Stream.js
Program Ended
Hello, This is input text, you will use to read using Stream.
```

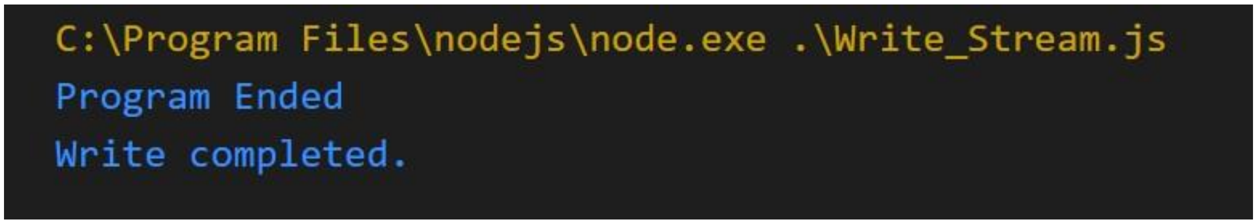
Node.js Writing to stream

Create a JavaScript file named Write_Stream.js having the following code:

```
var fs = require("fs");   var data = 'VS Code: A
Solution of all Technology';
```

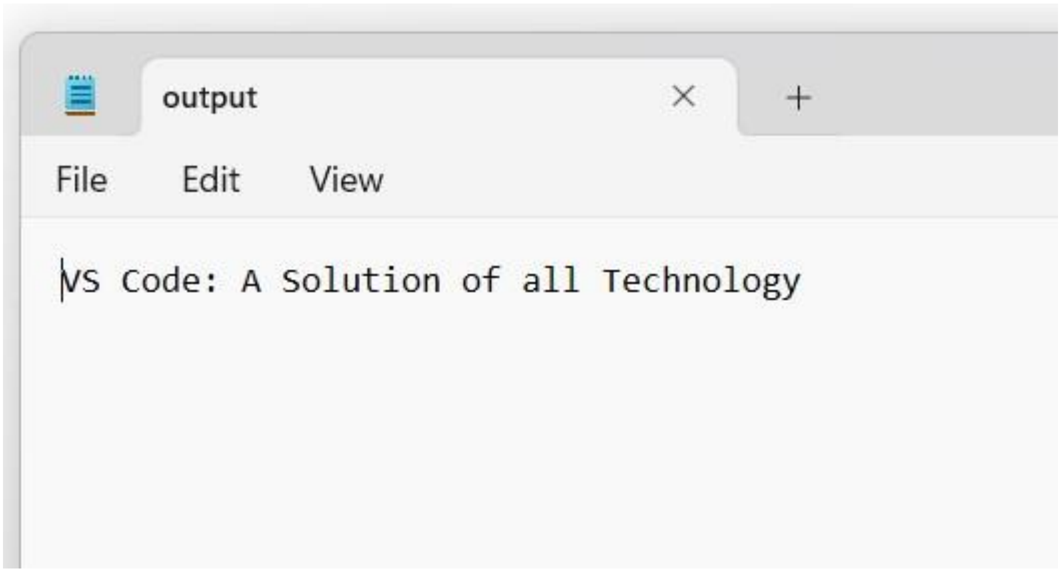
```
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
writerStream.write(data,'UTF8');
// Mark the end of file
writerStream.end();
// Handle stream events --> finish, and error
writerStream.on('finish', function() {
    console.log("Write completed.");
});
writerStream.on('error', function(err){
   console.log(err.stack);
});
console.log("Program Ended");
```

Output:



```
C:\Program Files\nodejs\node.exe .\Write_Stream.js
Program Ended
Write completed.
```

File is created:



```
VS Code: A Solution of all Technology
```

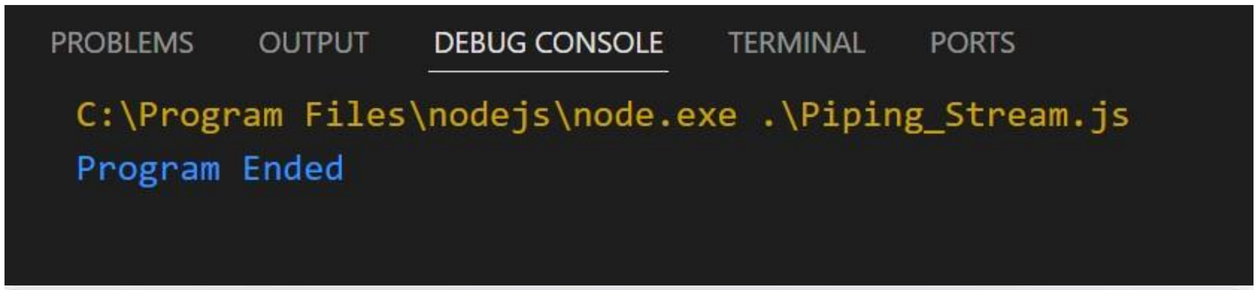
Node.js Piping Streams

Piping is a mechanism where output of one stream is used as input to another stream. There is no limit on piping operation.

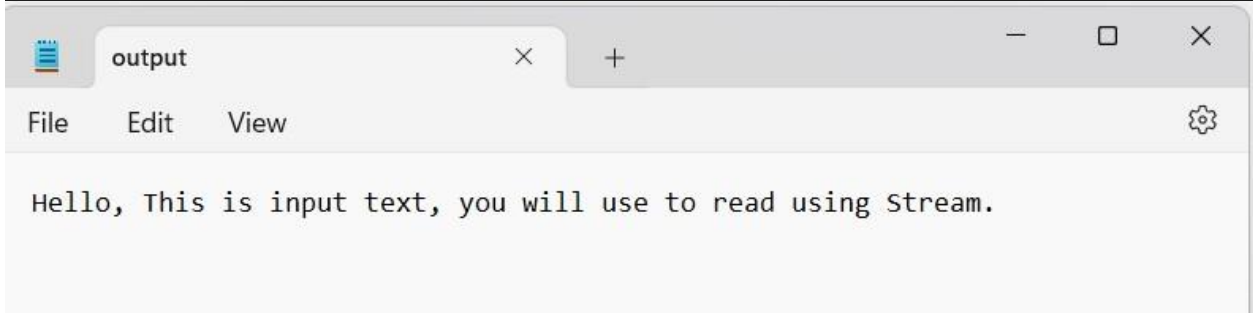
Let's take a piping example for reading from one file and writing it to another file.

```
var fs = require("fs");
// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);
console.log("Program Ended");
```

Piping_Stream.js Output:



```
C:\Program Files\nodejs\node.exe .\Piping_Stream.js
Program Ended
```



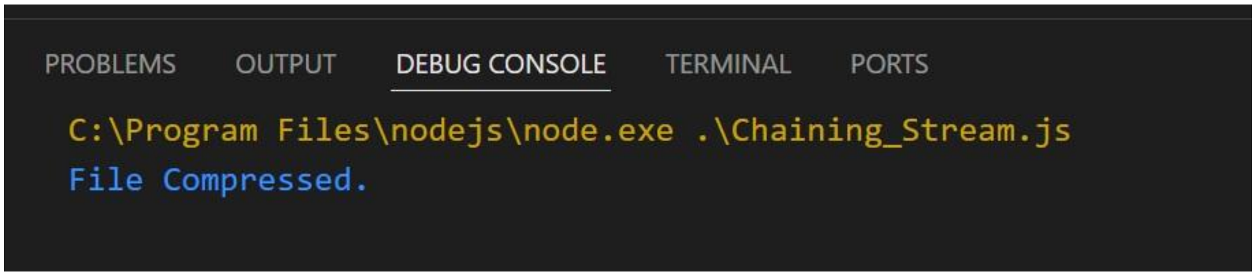
```
Hello, This is input text, you will use to read using Stream.
```

Node.js Chaining Streams

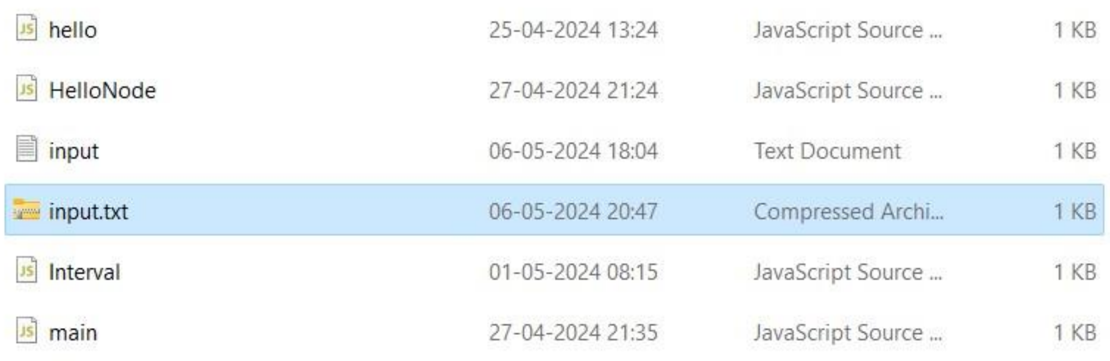
Chaining stream is a mechanism of creating a chain of multiple stream operations by connecting output of one stream to another stream. It is generally used with piping operation.

Let's take an example of piping and chaining to compress a file and then decompress the same file.

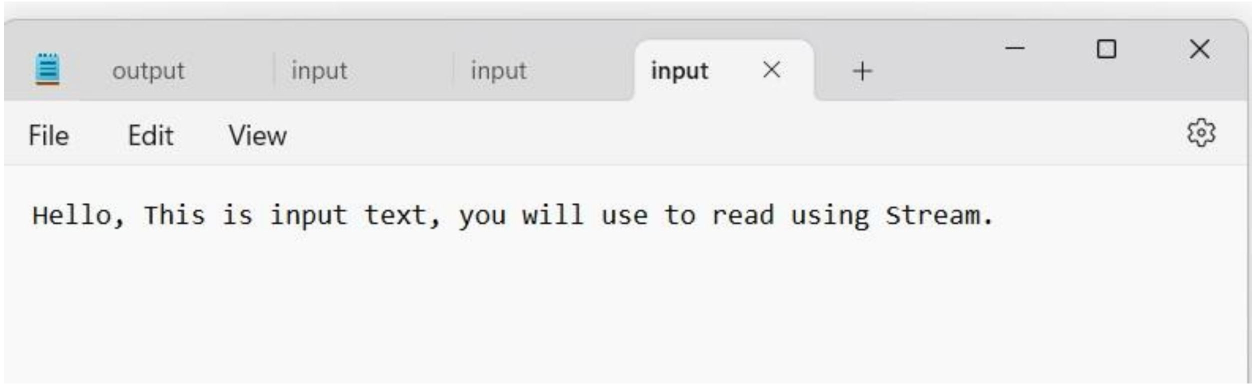
```
var fs = require("fs");
var zlib = require('zlib');
// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));
  console.log("File Compressed.");
```



```
C:\Program Files\nodejs\node.exe .\Chaining_Stream.js
File Compressed.
```



hello	25-04-2024 13:24	JavaScript Source ...	1 KB
HelloNode	27-04-2024 21:24	JavaScript Source ...	1 KB
input	06-05-2024 18:04	Text Document	1 KB
input.txt	06-05-2024 20:47	Compressed Archi...	1 KB
Interval	01-05-2024 08:15	JavaScript Source ...	1 KB
main	27-04-2024 21:35	JavaScript Source ...	1 KB

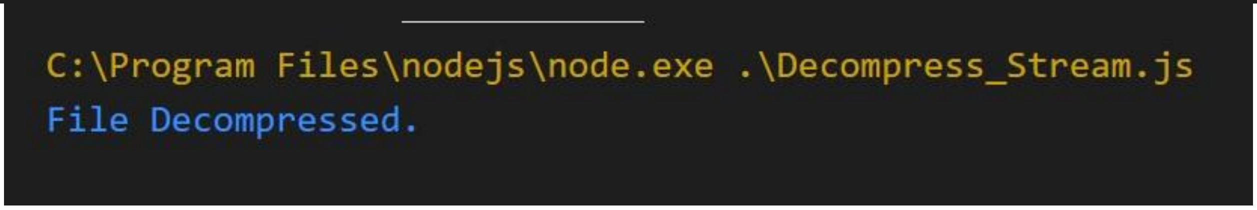


```
Hello, This is input text, you will use to read using Stream.
```

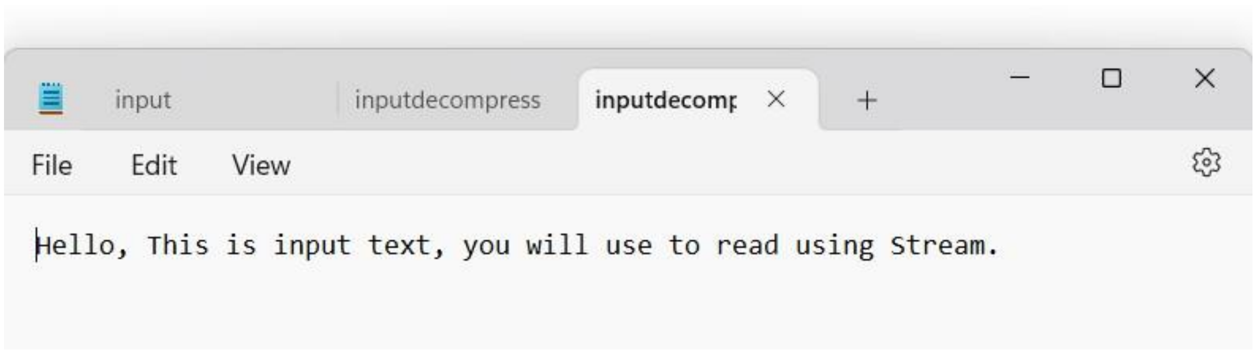
Now you will see that file "input.txt" is compressed and a new file is created named "input.txt.gz" in the current file.

To Decompress the same file: put the following code in the js file "Decompress_Stream.js"

```
var fs = require("fs");
var zlib = require('zlib');
// Decompress the file input.txt.gz to input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));
console.log("File Decompressed.");
```



```
C:\Program Files\nodejs\node.exe .\Decompress_Stream.js
File Decompressed.
```



```
Hello, This is input text, you will use to read using Stream.
```

Node.js File System (FS)

In Node.js, file I/O is provided by simple wrappers around standard POSIX functions. Node File System (fs) module can be imported using following syntax:

Syntax:

var fs = require("fs")

Node.js FS Reading File

Every method in fs module has synchronous and asynchronous forms.

Asynchronous methods take a last parameter as completion function callback. Asynchronous method is preferred over synchronous method because it never blocks the program execution where as the synchronous method blocks.

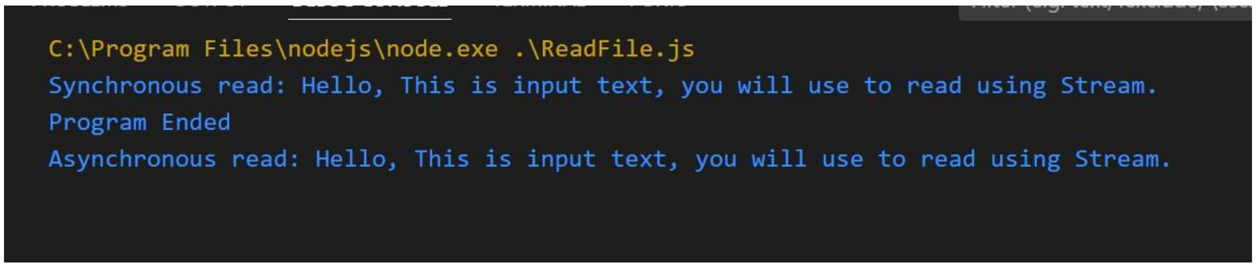
Let's take an example:

Create a text file named "input.txt" having the following content.

File: input.txt

Hello, This is input text, you will use to read using Stream.

```
var fs = require("fs");
// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
       return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});
// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());
console.log("Program Ended");
```



```
C:\Program Files\nodejs\node.exe .\ReadFile.js
Synchronous read: Hello, This is input text, you will use to read using Stream.
Program Ended
Asynchronous read: Hello, This is input text, you will use to read using Stream.
```

Node.js Open a file

Syntax:

Following is the syntax of the method to open a file in asynchronous mode:

9. fs.open(path, flags[, mode], callback)

Parameter explanation:

Following is the description of parameters used in the above syntax: **path:**

This is a string having file name including path.

flags: Flag specifies the behavior of the file to be opened. All possible values have been mentioned below. **mode:** This sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable. **callback:** This is the callback function which gets two arguments (err, fd).

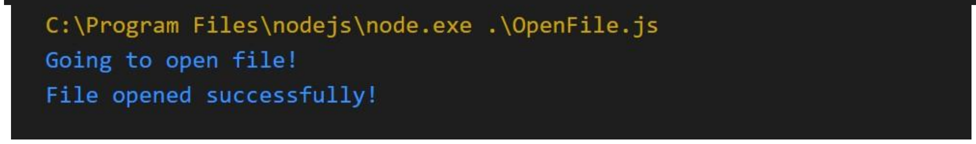
Node.js Flags for Read/Write

Following is a list of flags for read/write operation:

Flag	Description
r	open file for reading. an exception occurs if the file does not exist.
r+	open file for reading and writing. an exception occurs if the file does not exist.
rs	open file for reading in synchronous mode.
rs+	open file for reading and writing, telling the os to open it synchronously. see notes for 'rs' about using this with caution.
w	open file for writing. the file is created (if it does not exist) or truncated (if it exists).
wx	like 'w' but fails if path exists.
w+	open file for reading and writing. the file is created (if it does not exist) or truncated (if it exists).
wx+	like 'w+' but fails if path exists.
a	open file for appending. the file is created if it does not exist.
ax	like 'a' but fails if path exists.
a+	open file for reading and appending. the file is created if it does not exist.
ax+	open file for reading and appending. the file is created if it does not exist.

Create a JavaScript file named "OpenFile.js" having the following code to open a file input.txt for reading and writing.

```
var fs = require("fs");   // Asynchronous -
Opening File   console.log("Going to open
file!");   fs.open('input.txt', 'r+',
function(err, fd) {      if (err) {
return console.error(err);
   }     console.log("File opened
successfully!");
});
```



```
C:\Program Files\nodejs\node.exe .\OpenFile.js
Going to open file!
File opened successfully!
```

Node.js File Information Method

Syntax:

Following is syntax of the method to get file information.

10. fs.stat(path, callback)

Parameter explanation:

Path: This is string having file name including path.

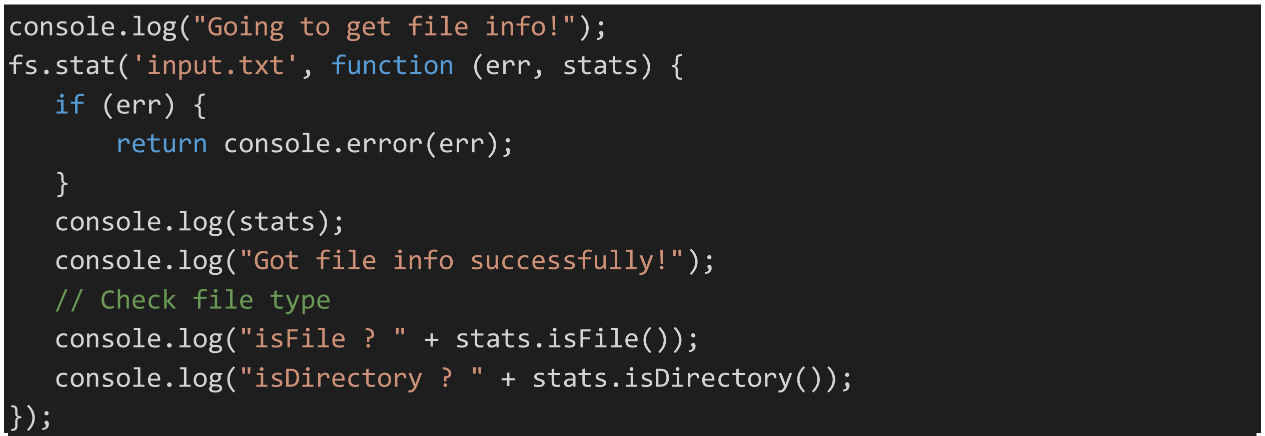
Callback: This is the callback function which gets two arguments (err, stats) where stats is an object of fs.Stats type.

Node.js fs.Stats class Methods

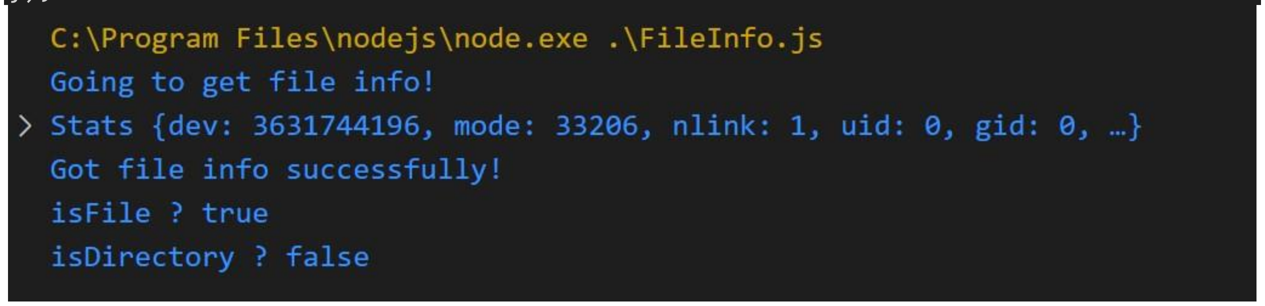
Method	Description
stats.isfile()	returns true if file type of a simple file.
stats.isdirectory()	returns true if file type of a directory.
stats.isblockdevice()	returns true if file type of a block device.
stats.ischaracterdevice()	returns true if file type of a character device.
stats.issymboliclink()	returns true if file type of a symbolic link.
stats.isfifo()	returns true if file type of a fifo.
stats.issocket()	returns true if file type of asocket.

Let's take an example to create a JavaScript file named main.js having the following code:

```
var fs = require("fs");
```



```
console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats) {
   if (err) {
       return console.error(err);
   }
   console.log(stats);
   console.log("Got file info successfully!");
   // Check file type
   console.log("isFile ? " + stats.isFile());
   console.log("isDirectory ? " + stats.isDirectory());
});
```



```
  C:\Program Files\nodejs\node.exe .\FileInfo.js
  Going to get file info!
> Stats {dev: 3631744196, mode: 33206, nlink: 1, uid: 0, gid: 0, …}
  Got file info successfully!
  isFile ? true
  isDirectory ? false
```

Node.js Path

The Node.js path module is used to handle and transform files paths. This module can be imported by using the following syntax:

Syntax:

```
var path = require ("path")
```

Node.js Path Methods

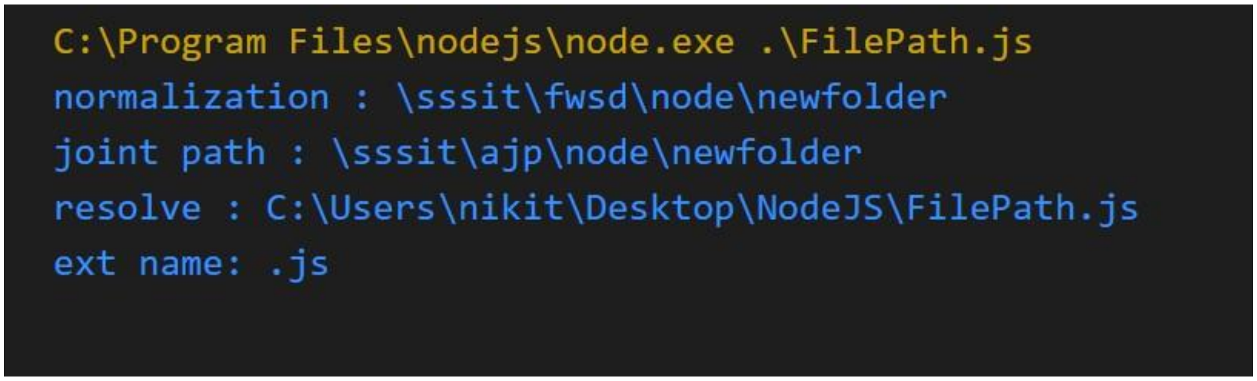
Let's see the list of methods used in path module:

Index	Method	Description
1.	path.normalize(p)	It is used to normalize a string path, taking care of '..' and '.' parts.

2.	path.join([path1][, path2][, ...])	It is used to join all arguments together and normalize the resulting path.
3.	path.resolve([from ...], to)	It is used to resolve an absolute path.
4.	path.isabsolute(path)	It determines whether path is an absolute path. an absolute path will always resolve to the same location, regardless of the working directory.
5.	path.relative(from, to)	It is used to solve the relative path from "from" to "to".
6.	path.dirname(p)	It return the directory name of a path. It is similar to the unix dirname command
7.	path.basename(p[, ext])	It returns the last portion of a path. It is similar to the Unix basename command.
8.	path.extname(p)	It returns the extension of the path, from the last '.' to end of string in the last portion of the path. if there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string.
9.	path.parse(pathstring)	It returns an object from a path string.
10.	path.format(pathobject)	It returns a path string from an object, the opposite of path.parse above.

Node.js Path Example

```
var path = require("path");
// Normalization
console.log('normalization                    :                  '
                                              +
path.normalize('/sssit/fwsd//node/newfolder/tab/..'));
// Join  console.log('joint path : ' + path.join('/sssit', 'ajp',
'node/newfolder',     'tab',     '..'));            //     Resolve
console.log('resolve : ' + path.resolve('FilePath.js'));
// Extension
console.log('ext name: ' + path.extname('FilePath.js'));
```



```
C:\Program Files\nodejs\node.exe .\FilePath.js
normalization : \sssit\fwsd\node\newfolder
joint path : \sssit\ajp\node\newfolder
resolve : C:\Users\nikit\Desktop\NodeJS\FilePath.js
ext name: .js
```

Node.js StringDecoder

The Node.js StringDecoder is used to decode buffer into string. It is similar to buffer.toString() but provides extra support to UTF.

You need to use require('string_decoder') to use StringDecoder module.

```
const StringDecoder = require('string_decoder').StringDecoder;
```

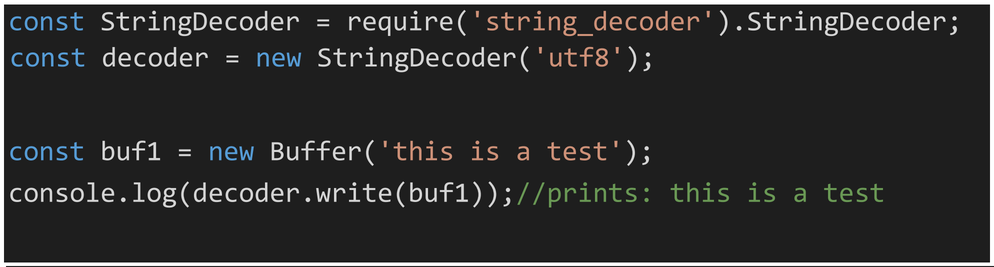
Node.js StringDecoder Methods

StringDecoder class has two methods only.

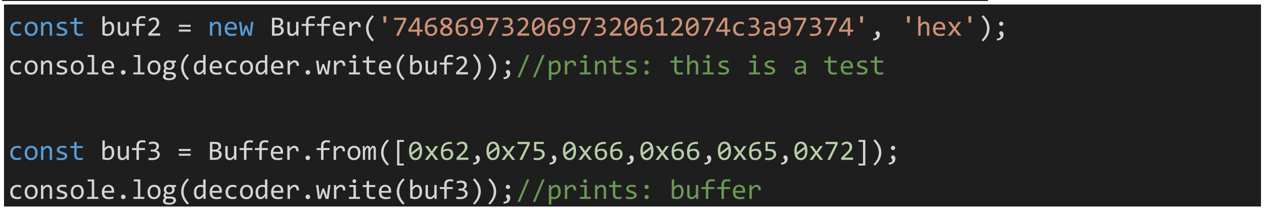
Method	Description
decoder.write(buffer)	It is used to return the decoded string.
decoder.end()	It is used to return trailing bytes, if any left in the buffer.

Node.js StringDecoder Example

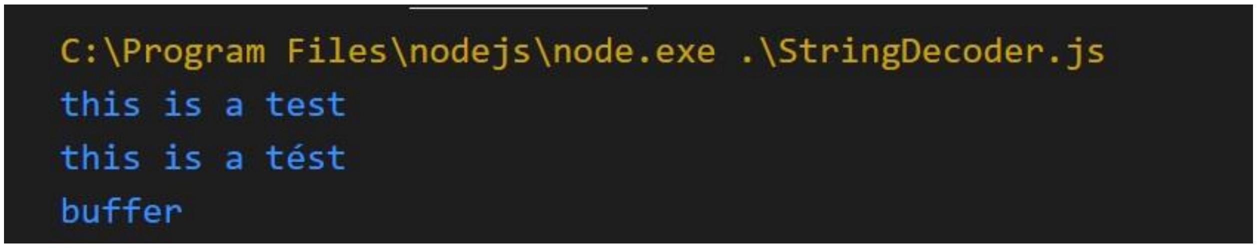
Let's see a simple example of Node.js StringDecoder.



```
const StringDecoder = require('string_decoder').StringDecoder;
const decoder = new StringDecoder('utf8');


const buf1 = new Buffer('this is a test');
console.log(decoder.write(buf1));//prints: this is a test
```



```
const buf2 = new Buffer('7468697320697320612074c3a97374', 'hex');
console.log(decoder.write(buf2));//prints: this is a test

const buf3 = Buffer.from([0x62,0x75,0x66,0x66,0x65,0x72]);
console.log(decoder.write(buf3));//prints: buffer
```



```
C:\Program Files\nodejs\node.exe .\StringDecoder.js
this is a test
this is a tést
buffer
```

Node.js Query String

The Node.js Query String provides methods to deal with query string. It can be used to convert query string into JSON object and vice-versa.

To use query string module, you need to use **require('querystring')**.

Node.js Query String Methods

The Node.js Query String utility has four methods. The two important methods are given below.

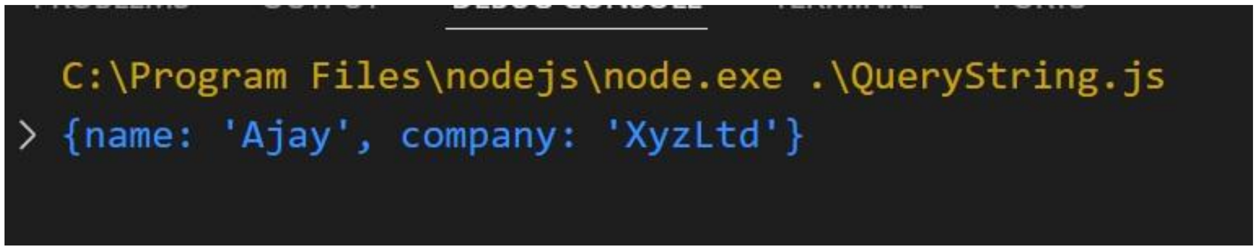
Method	Description
querystring.parse(str[, sep][, eq][, options])	converts query string into JSON object.
querystring.stringify(obj[, sep][, eq][, options])	converts JSON object into query string.

Node.js Query String Example 1: parse()

Let's see a simple example of Node.js Query String parse() method.

[Convert a query string to JSON object]

```
querystring = require('querystring');
const obj1=querystring.parse('name=Ajay&company=XyzLtd');
console.log(obj1);
```



```
C:\Program Files\nodejs\node.exe .\QueryString.js
> {name: 'Ajay', company: 'XyzLtd'}
```

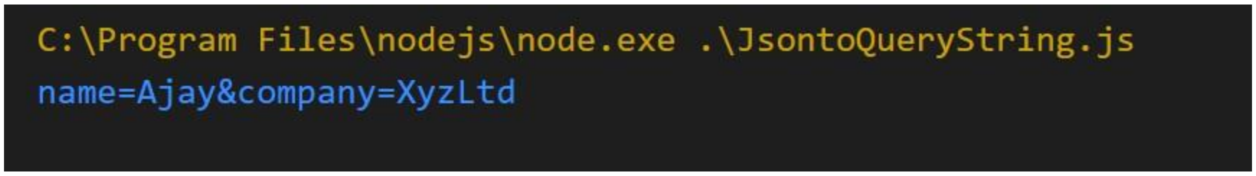
Node.js Query String Example 2: stringify()

Let's see a simple example of Node.js Query String stringify() method.

[Convert JSON object to query string]

File: query-string-example2.js

```
querystring = require('querystring');
const qs1=querystring.stringify({name:'Ajay',company:'XyzLtd'});
console.log(qs1);
```



```
C:\Program Files\nodejs\node.exe .\JsontoQueryString.js
name=Ajay&company=XyzLtd
```

-- **For**

information only: Node.js Callbacks

Callback is an asynchronous equivalent for a function. It is called at the completion of each task. In Node.js, callbacks are generally used. All APIs of Node are written in a way to supports callbacks. For example: when a function start reading file, it returns the control to execution environment immediately so that the next instruction can be executed.

In Node.js, once file I/O is complete, it will call the callback function. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

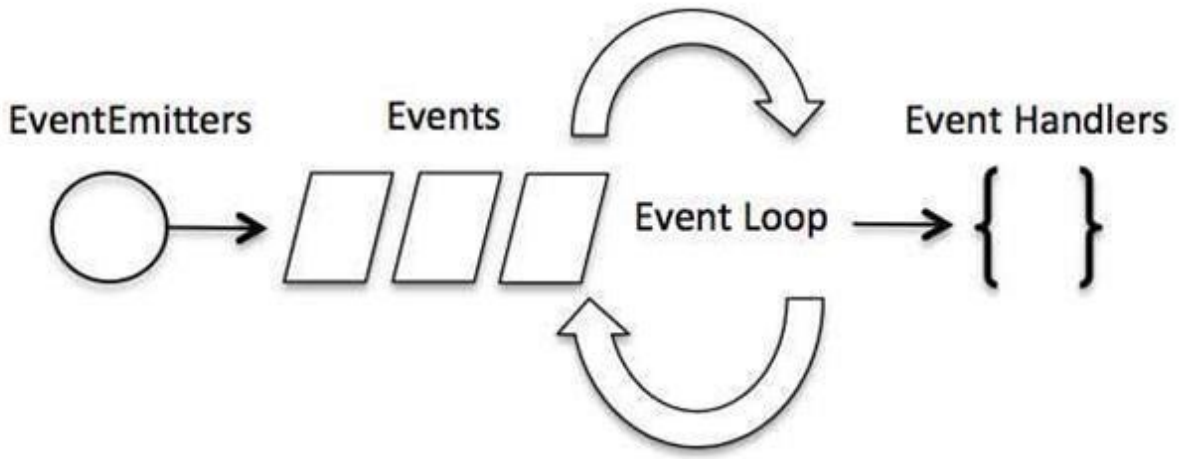
Node.js Events

In Node.js applications, Events and Callbacks concepts are used to provide concurrency. As Node.js applications are single threaded and every API of Node js are asynchronous. So it uses async function to maintain the concurrency. Node uses observer pattern. Node thread keeps an event loop and after the completion of any task, it fires the corresponding event which signals the event listener function to get executed.

Event Driven Programming

Node.js uses event driven programming. It means as soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for event to occur. It is the one of the reason why Node.js is pretty fast compared to other similar technologies.

There is a main loop in the event driven application that listens for events, and then triggers a callback function when one of those events is detected.



Difference between Events and Callbacks:

Although, Events and Callbacks look similar but the differences lies in the fact that callback functions are called when an asynchronous function returns its result where as event handling works on the observer pattern. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which is used to bind events and event listeners.

EventEmitter class to bind event and event listener:

// Import events module   var events =

require('events');   // Create an eventEmitter

object   var eventEmitter = new

events.EventEmitter();

To bind event handler with an event:

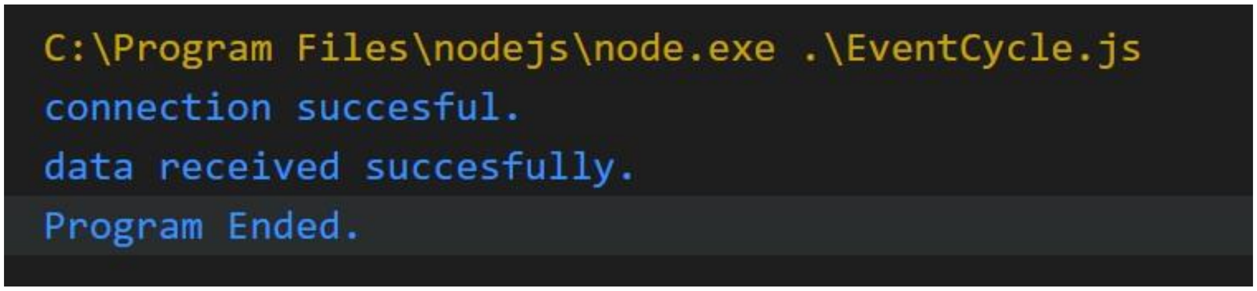
// Bind event and even handler as follows   eventEmitter.on('eventName',

eventHandler);

To fire an event:

// Fire an event   eventEmitter.emit('eventName');

```
// Import events module   var events =
require('events');   // Create an
eventEmitter object   var eventEmitter = new
events.EventEmitter();
// Create an event handler as follows   var
connectHandler = function connected() {
console.log('connection succesful.');
   // Fire the data_received event
eventEmitter.emit('data_received');
}

// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);
 // Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
console.log('data received succesfully.');
});
// Fire the connection event
eventEmitter.emit('connection');
console.log("Program Ended.");
```



```
C:\Program Files\nodejs\node.exe .\EventCycle.js
connection succesful.
data received succesfully.
Program Ended.
```

________________________________End of Unit____