

NODE.js Document

NODE.js:

- Node.js is a very powerful JavaScript-based platform built on Google Chrome's JavaScript V8 Engine.
- It is used to develop I/O intensive web applications like video streaming sites, single-page applications, and other web applications.
- As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications.

What is Node.js?

- Node.js is an open source server environment
- Node.js is free.
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server.

Why Node.js?

Node.js uses asynchronous programming!

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

- Sends the task to the computer's file system.
- Waits while the file system opens and reads the file.
- Returns the content to the client.
- Ready to handle the next request.

Here is how Node.js handles a file request:

- Sends the task to the computer's file system.
- Ready to handle the next request.
- When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

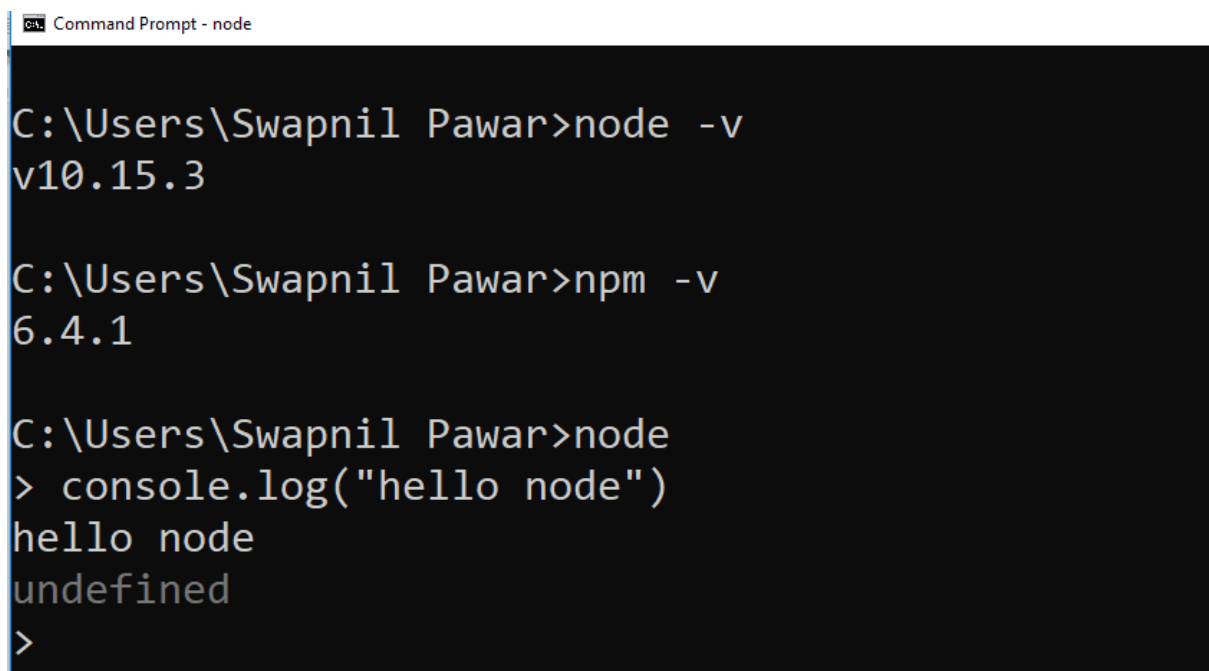
Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

Installation of node js

- Visit <https://nodejs.org/en/> and download the latest version on machine.
- Check the installation of node.js use command prompt and type **node -v**.
- Similarly, check for the installation of npm use **npm -v** and use some command for node.js on command prompt. It shows following results.



```
Command Prompt - node

C:\Users\Swapnil Pawar>node -v
v10.15.3

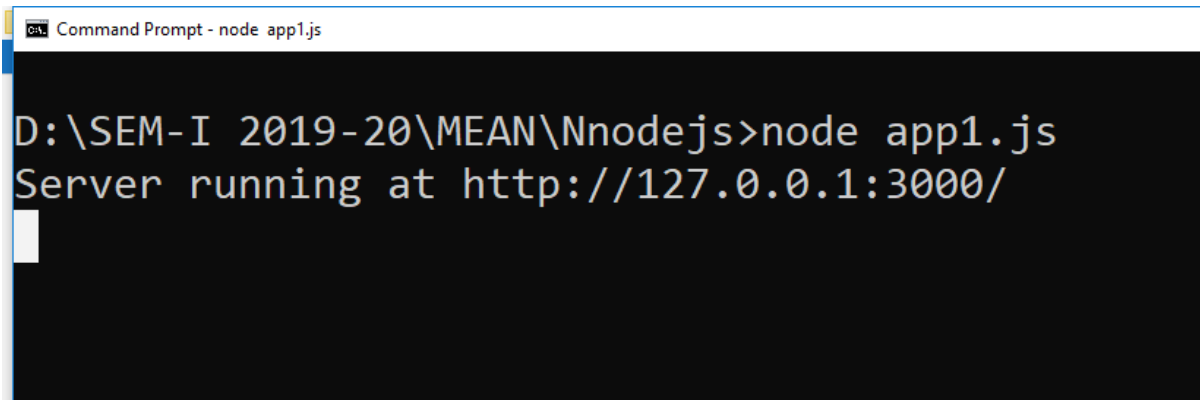
C:\Users\Swapnil Pawar>npm -v
6.4.1

C:\Users\Swapnil Pawar>node
> console.log("hello node")
hello node
undefined
>
```

Now we can try the sample example

Create a file with .js extension and save and run like this.

The server instance is running at <http://127.0.0.1:3000/> or type localhost:3000.



```
Command Prompt - node app1.js
D:\SEM-I 2019-20\MEAN\Nnodejs>node app1.js
Server running at http://127.0.0.1:3000/
```

Features of Node js:

Following is a list of some important features of Node.js that makes it the first choice of software architects.

- 1. Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
- 2. I/O is Asynchronous and Event Driven:** All APIs of Node.js library are 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.
- 3. Single threaded:** Node.js follows a single threaded model with event looping and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
- 4. Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests.
- 5. 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.
- 6. Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.

7. License: Node.js is released under the MIT license.

8. NPM (The Node Package Manager) - it's a tool that handles installing and updating of reusable modules from online collection. It also takes care of version and dependency management of reusable modules from online collection. NPM can be compared to Ruby Gems.

9. Community - There are lots of community tutorials, resources or shared code. Node.js is very popular now and it became one of most of used technologies used nowadays.

10. Data streaming - Because of its asynchronous nature it's very good for handling real-time data streaming. It can be used to stream media, data from multiple sources, file upload or it's great for Websockets server.

11. Modularity: Major advantage of Node JS Platform is that it's modularity. Each and every functionality is divided and implemented as a separate module or package. Some modules were developed by Node JS Community and some were by Third-party Clients.

12. Express JS: Node JS is used to develop Server-side Java Script. It also contains a separate module for Web Application Framework i.e. Express JS.

13. Redis Client Library API: Node JS platform contains a separate module to integrate Redis No SQL database with applications. It provides a Redis wrapper API. We can use this API to write JavaScript easily to interact with Redis database.

14. Jade Template Engine: Node JS platform supports many template engines to write HTML. Default template engine supported by Node JS is "Jade". Jade is a whitespace sensitive template engine for developing HTML applications very easily.

15. MongoDB Wrappers API: Node JS platform contains a separate module to integrate MongoDB No SQL database with applications. It provides a MongoDB wrapper API.

16. Better Socket API: Node JS Platform provides very good Socket Module API to develop Real-time, Multi-User Chat and Multi-Player Gaming Applications very easily. It supports Unix Socket programming like pipe ().

Run First node server

- Use visual studio code to create app.js file.
- Add the following code and run using Command prompt.

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

http.createServer(function(req,res){
res.end("hello");

}).listen(3000);

console.log("Server is running at localhost at port 3000");
```

Creating Node.js Application

Step 1 - Import Required Module

We use the **require ()** directive to load the http module and store the returned HTTP instance into an http variable as follows –

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

Step 2 - Create Server

We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 3000 using the **listen()** method associated with the server instance.

```
http.createServer(function(req,res){
res.end("hello");

}).listen(3000);
```

Node Package Manager(NPM):

Use command prompt or Terminal from Visual studio code and type the following.

1)

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm view underscore

underscore@1.9.1 | MIT | deps: none | versions: 35
JavaScript's functional programming helper library.
http://underscorejs.org

keywords: util, functional, server, client, browser

dist
.tarball: https://registry.npmjs.org/underscore/-/underscore-1.9.1.tgz
.shasum: 06dce34a0e68a7bab29b365b8e74b8925203961
.integrity: sha512-5/4etnCd9c8gwgowi5/om/mY05ajCa0gdzj/ow+0eQV9WxKBDZw5+ycmKmeaTXjInS/W0BzpGLo2xR2aBwZdg==
.unpackedSize: 111.0 kB

maintainers:
- jashkenas <jashkenas@gmail.com>
- jridgewell <justin+npm@ridgewell.name>

dist-tags:
latest: 1.9.1  stable: 1.9.0
```

2)

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm view underscore versions

[ '1.0.3',
  '1.0.4',
  '1.1.0',
  '1.1.1',
  '1.1.2',
  '1.1.3',
  '1.1.4',
  '1.1.5',
  '1.1.6',
  '1.1.7',
  '1.2.0',
  '1.2.1',
  '1.2.2',
  '1.2.3',
  '1.2.4',
  '1.3.0',
  '1.3.1',
  '1.3.2',
```

3) Install Package

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm install underscore
npm WARN saveError ENOENT: no such file or directory, open 'D:\SEM-I 2019-20\MEAN\Nnodejs\First\package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open 'D:\SEM-I 2019-20\MEAN\Nnodejs\First\package.json'
npm WARN First No description
npm WARN First No repository field.
npm WARN First No README data
npm WARN First No license field.

+ underscore@1.9.1
added 1 package from 1 contributor and audited 1 package in 5.193s
found 0 vulnerabilities

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> █
```

4) Uninstall Package

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm uninstall underscore
npm WARN saveError ENOENT: no such file or directory, open 'D:\SEM-I 2019-20\MEAN\Nnodejs\First\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'D:\SEM-I 2019-20\MEAN\Nnodejs\First\package.json'
npm WARN First No description
npm WARN First No repository field.
npm WARN First No README data
npm WARN First No license field.

removed 1 package in 3.045s
found 0 vulnerabilities

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> █
```

5) To view the list

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm list
D:\SEM-I 2019-20\MEAN\Nnodejs\First
|-- underscore@1.9.1

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> █
```

6) Use of NPM init

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (first)
version: (1.0.1)
git repository: █
```

7) See the updates in package.json

```
{ package.json > ...
1  {
2    "name": "first",
3    "version": "1.0.1",
4    "description": "This is my first module",
5    "main": "app.js",
6    "dependencies": {
7      "underscore": "^1.9.1"
8    },
9    "devDependencies": {},
10   "scripts": {
11     "test": "echo \"Error: no test specified\" && exit 1"
12   },
13   "author": "Swapnil",
14   "license": "MIT"
15 }
16
```

8) Now we can add express to npm

See in the above fig. express version is missing. We can view version of express using

npm view express versions

```
'4.15.1',
'4.15.2',
'4.15.3',
'4.15.4',
'4.15.5',
'4.16.0',
'4.16.1',
'4.16.2',
'4.16.3',
'4.16.4',
'4.17.0',
'4.17.1',
'5.0.0-alpha.1',
'5.0.0-alpha.2',
'5.0.0-alpha.3',
'5.0.0-alpha.4',
'5.0.0-alpha.5',
'5.0.0-alpha.6'
```

9) Install specific Version of express

Use command

npm install express@4.17.0. --save


```
{ } package.json > ...
1 {
2   "name": "first",
3   "version": "1.0.1",
4   "description": "This is my first module",
5   "main": "app.js",
6   "dependencies": {
7     "express": "^4.17.0",
8     "underscore": "^1.9.1"
9   },
10  "devDependencies": {},
11  "scripts": {
12    "test": "echo \"Error: no test specified\" && exit 1"
13  }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm install express@4.17.0 --save
npm WARN first@1.0.1 No repository field.

+ express@4.17.0
updated 1 package and audited 127 packages in 6.049s
found 0 vulnerabilities

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> 
```

10)

To check the outdated package and update the same we make use of this commands.

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm updat --save
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm update --save
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm outdated
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> 
```

11) now uninstall express and add it globally to system

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm uninstall express --save
npm WARN first@1.0.1 No repository field.

removed 50 packages and audited 1 package in 2.357s
found 0 vulnerabilities

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm ls
first@1.0.1 D:\SEM-I 2019-20\MEAN\Nnodejs\First
|-- underscore@1.9.1

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> 
```

Install Globally use command

Npm install express -g or npm install express --global

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm install express -g
+ express@4.17.1
updated 1 package in 8.035s
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm ls
first@1.0.1 D:\SEM-I 2019-20\MEAN\Nnodejs\First
|-- underscore@1.9.1

PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> 
```

To view the global packages, install use following command

```
PS D:\SEM-I 2019-20\MEAN\Nnodejs\First> npm ls -g
C:\Users\Swapnil Pawar\AppData\Roaming\npm
+-- @angular/cli@7.3.8
| +-- @angular-devkit/architect@0.13.8
| | +-- @angular-devkit/core@7.3.8 deduped
| | `-- rxjs@6.3.3
| | `-- tslib@1.9.3
| +-- @angular-devkit/core@7.3.8
```

Node.js Docs and Global Objects:

Visit the following link for more details

<https://nodejs.org/dist/latest-v10.x/docs/api/>

Example:

```
var time = 0;
setInterval(function(){
  time +=2;
  console.log(time+' sec have paseed');
},2000);
```

```
var time =0;
var timer = setInterval(function(){
  time += 2;

  console.log(time+'sec have passed ');
  if(time>7)
    clearInterval(timer);
});
```

Creating custom module in node.js

A module is basically a javascript file that exposes internally scoped functions using the **exports** variable

Now let's create our module that we'll call **hello.js**

```
exports.SplitMe = function(stringToSplit, delimiter)
{
    return stringToSplit.split(delimiter);
};
```

As you can see this is a super simple module that splits a string and returns an array using javascript's built-in split function. Using the **exports** variable, we've exposed the **SplitMe** function.

Next we'll create our **app.js** file that will use our **StringSplitter** module.

```
var myCustomModule = require('./hello.js');

console.log(myCustomModule.SplitMe('Texas, Florida, Seattle', ','))
```

We use **require('./hello.js')** to invoke our module. Using require, Node.js will evaluate the file and load the functions defined in the exports object.

Reading and Writing Files using Node.js:

Node.js as a File Server

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

To include the File System module, use the `require ()` method:

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

Read Files

The `fs.readFile()` method is used to read files on your computer.

Read.js file as,

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demo.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    res.end();
  });
}).listen(8080);
```

A simple **demo.html** file as,

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
```

Now initiate the **read.js** file and check output on localhost:8080

Create Files

The File System module has methods for creating new files:

- `fs.appendFile()`
- `fs.open()`
- `fs.writeFile()`

The `fs.appendFile()` method appends specified content to a file. If the file does not exist, the file will be created:

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

//create a file named mynewfile1.txt:
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

The `fs.open()` method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

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

//create an empty file named mynewfile2.txt:
fs.open('mynewfile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

The `fs.writeFile()` method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

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

//create a file named mynewfile3.txt:
fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

Update Files

The File System module has methods for updating files:

- `fs.appendFile()`
- `fs.writeFile()`

The `fs.appendFile()` method appends the specified content at the end of the specified file:

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

//append content at the end of the file:
fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {
  if (err) throw err;
  console.log('Updated!');
});
```

The `fs.writeFile()` method replaces the specified file and content:

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

//Replace the file with a new one:
fs.writeFile('mynewfile3.txt', 'This is my text.', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

Delete Files

To delete a file with the File System module, use the `fs.unlink()` method.

The `fs.unlink()` method deletes the specified file.

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

//Delete the file mynewfile2.txt:
fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
  console.log('File deleted!');
});
```

Rename Files:

To rename a file with the File System module, use the `fs.rename()` method.

The `fs.rename()` method renames the specified file:

Basics of Streams:

Readable stream and Writable Stream:

Streams are collections of data just like the arrays or strings. The difference between arrays and streams is that streams might not be available all at once, and they don't have to fit in the memory. This feature makes streams more powerful when working with the massive amounts of data, or the data that is coming from the external source one chunk at a time. Many of the built-in modules in Node implement the streaming interface. So Stream data are not available at once, but they are available at some point of time in the form of chunk data. That is why it is beneficial for developing a streaming web application like Netflix or Youtube.

Streams can be readable, writable, or both. All streams are instances of EventEmitter.

Streams

Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

The following statement can access the stream module.

```
Const stream = require('stream');
```

Why Streams

Streams primarily provide the following advantages.

- **Memory efficiency:** You don't need to carry the massive amounts of data in memory before you can process it.
- **Time efficiency:** It takes way less time to start processing the data as soon as you have it, rather than waiting till the whole data payload is available to start the process.

Types of Streams

There are four fundamental stream types in Node.js:

- **Writable** – streams to which data can be written (for example, `fs.createWriteStream()`).
- **Readable** – streams from which data can be read (for example, `fs.createReadStream()`).
- **Duplex** – streams that are both **Readable** and **Writable** (for example, `net.Socket`).
- **Transform** – **Duplex** streams that can modify or transform the data as it is written and read

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

const server = http.createServer(function (req, res) {
  fs.readFile('data.txt', (err, data) => {
    res.end(data)
  })
})

var console: Console > {
  console.log('server is running')
}
```

Here, one thing you can note that when the reading of the file is completed then and then it will send a response. So, if the file is very big, then it takes some time to read the whole file and then sending back to the client.

Piping the Streams

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is usually used to get data from one stream and to pass the output of that stream to another stream.

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

const readerStream = fs.createReadStream('data.txt')

const writerStream = fs.createWriteStream('data2.txt')

readerStream.pipe(writerStream)

console.log('Piping ended')
```

Difference between **fs.readFile()** and **fs.createReadStream()**

The **fs.readFile()** loads file and reads it into memory and then writes it to response whereas **fs.createReadStream()** sends chunks of file like writing small chunks of file dividing the entire process into that chunk size of file writing which reduces memory load and memory wastage/garbage reduction.

Node.js Events EventEmitter :

EventEmitter is the implementation of **Node.js's pub-sub** design patterns. Node.js core API has built on an **asynchronous event-driven** architecture. In an **asynchronous event architecture**, certain kinds of objects (called “emitters”) periodically emit named events that cause **Function objects** (“listeners”) to be called.

Each object that emits events are instances of an EventEmitter class.

The event emitter class has two methods.

1. On
2. emit

So, if we make an object of **EventEmitter** class, then we have access to this methods.

```
eventEmitter.on('clicked', function(){
  console.log('happens');
});
```

Here, I have defined an event and till now, not called just determine.

On() method takes an event name and a call back function, which describes the logic and payload of the function. As we know, Node.js has the event-driven architecture, so it first occurs the events and then related to that event, one callback function is returned.

```
eventEmitter.emit('clicked')
```

Here, I have emitted the event, so related to that event, the callback function is invoked, and that function will execute. The first parameter is event name and second is payload (if function with parameter).

```
eventEmitter.on('clicked', function(user){
  console.log(user);
});

eventEmitter.emit('clicked', "swapnil")
```

Example 2

```
//showing inheritance and usage of Util
var events = require('events');
var util = require('util');
//create object of class
var Students = function(name){
  this.name = name;
}
//Two arguments class which inherits and object of EventEmitter

util.inherits(Students, events.EventEmitter);

var max = new Students('max');
max.on('scored', function(marks){
  console.log(max.name + ' Score ' + marks + ' marks');
});

max.emit('scored', 95);
```

In this example, first I have created an **EventEmitter** object and then also create **Students function constructor**.

Then, import the Node.js's core module **util** and inherits the base functionality from EventEmitter module to the newly created Student module.

So, now Students has all the methods and properties of the **EventEmitter** module, and we can use two methods on it.

Now, Students object's behavior is same as EventEmitter, and we can define events on it and emit the events.

The output will be **“max score 95 marks”**