Node JS

Lesson 1:Getting started with Node.js

Introduction to Node.js

- In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications. It is written in C++ and JavaScript.
- Node.js is a platform built on Chrome’s JavaScript runtime(v8 JavaScript Engine) for easily building fast, scalable network applications.
- It’s a highly scalable system that uses asynchronous, non-blocking I/O model (input/output), rather than threads or separate processes
- It is not a framework like jQuery nor a programming language like C# or JAVA . It's a new kind of web server like has a lot in common with other popular web servers, like Microsoft’s Internet Information Services (IIS) or Apache
- IIS / Apache processes only HTTP requests, leaving application logic to be implemented in a language such as PHP or Java or ASP.NET. Node removes a layer of complexity by combining server and application logic in one place.

Why Node.js?

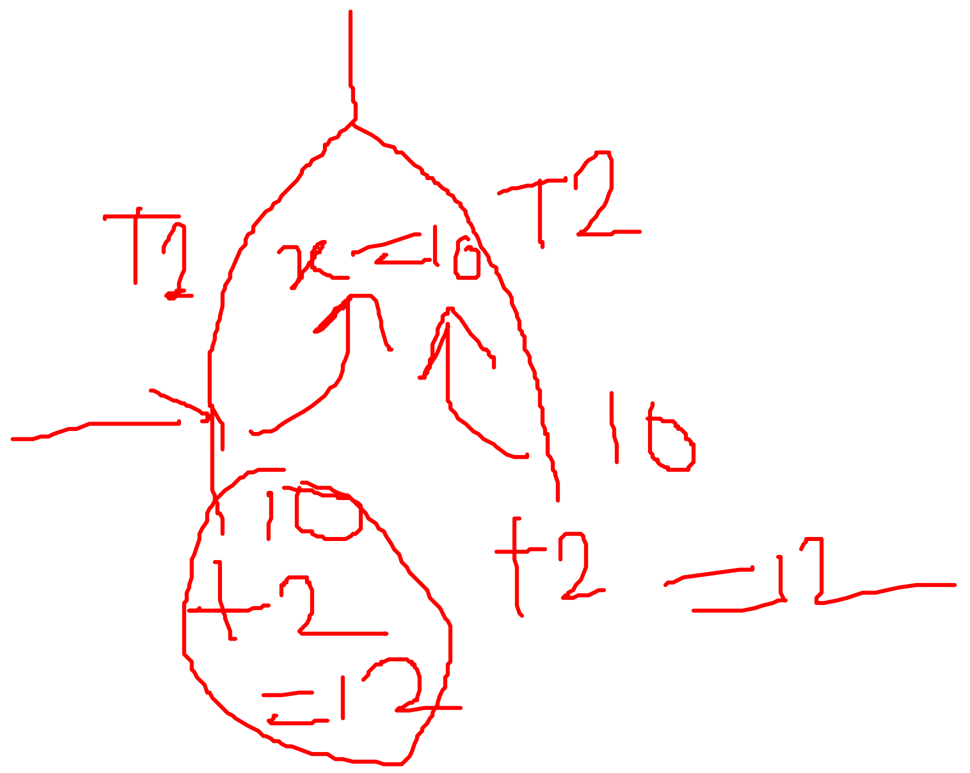
- JavaScript everywhere i.e. Server-side and Client-side applications in JavaScript.
- Node is very easy to set up and configure.
- Vibrant Community
- Small core but large community so far we have 60,000 + packages on npm
- Real-time/ high concurrency apps (I/O bound)
- API tier for single-page apps and rich clients(iOS, Android)
- Service orchestration
- Top corporate sponsors like Microsoft, Joyent, PayPal etc..
- Working with NOSQL(MongoDB) Databases

Traditional Programming Limitations

- In traditional programming I/O (database, network, file or inter-process communication) is performed in the same way as it does local function calls. i.e. Processing cannot continue until the operation is completed.
- When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as “Blocking”
- Due to this blocking behavior we cannot perform another I/O operation, the call stack becomes frozen waiting for the response.
- We can overcome this issue by creating more call stacks or by using event callbacks.

Creating more call stacks

- To handle more concurrent I/O, we need to have more concurrent call stacks.
- Multi-threading is one alternative to this programming model.
  - Makes use of separate CPU Cores as "Threads"
  - Uses a single process within the Operating System
  - Ran out of Ghz, hardware adds more cores
- If the application relies heavily on a shared state between threads accessing and modifying shared state increase the complexity of the code and It can be very difficult to configure, understand and debug.



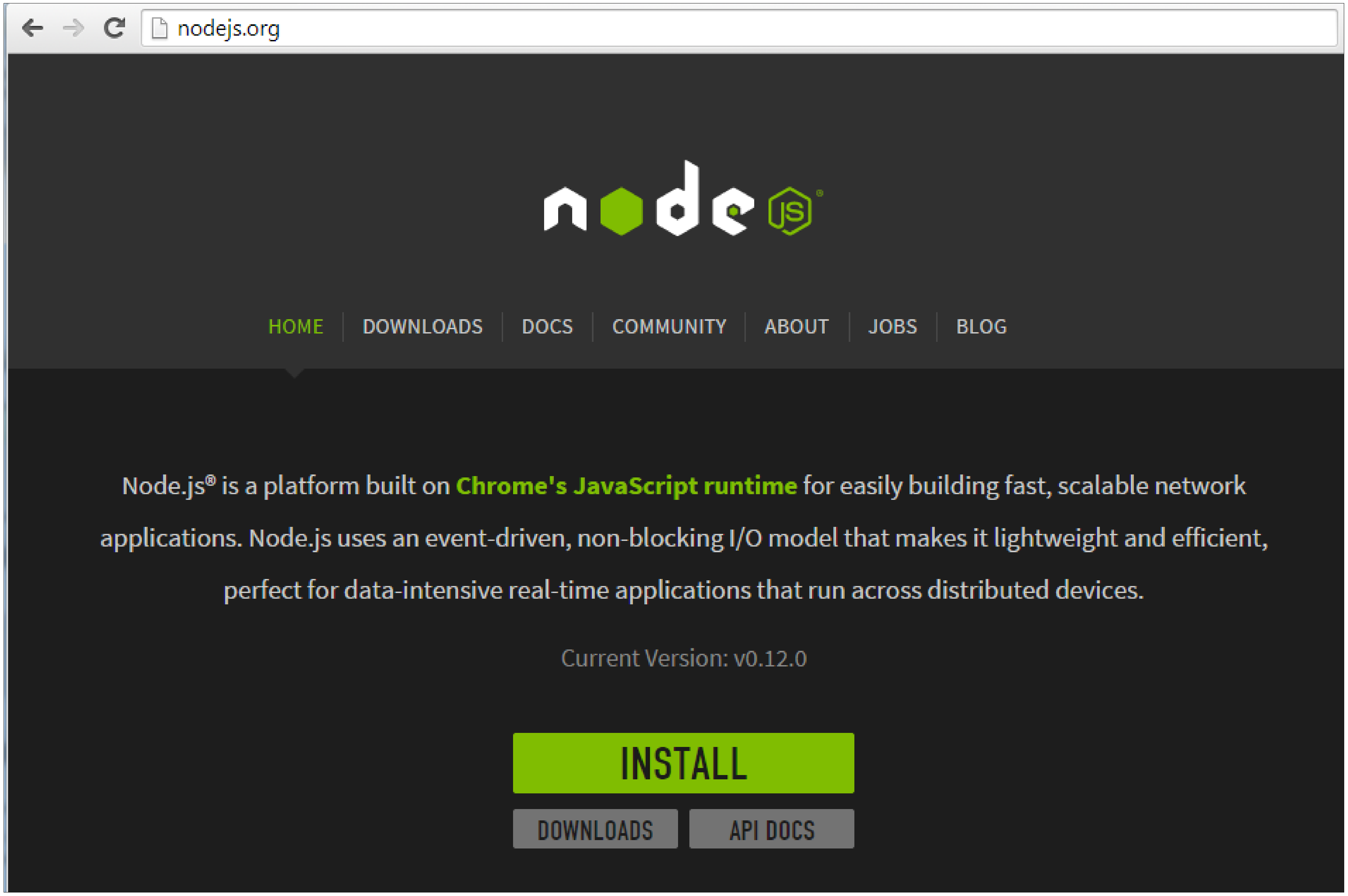
Event-driven Programming

- Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.
- Events are handled by event handlers or event callbacks
- An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.

```
query_finished = function(result) {
    do_something_with(result);
}
query('SELECT Id,Name FROM employees', query_finished);
```

- Now instead of simply returning the result, the query will invoke the query_finished function once it is completed.
- Node.js supports Event-driven programming and all I/O in Node are non-blocking

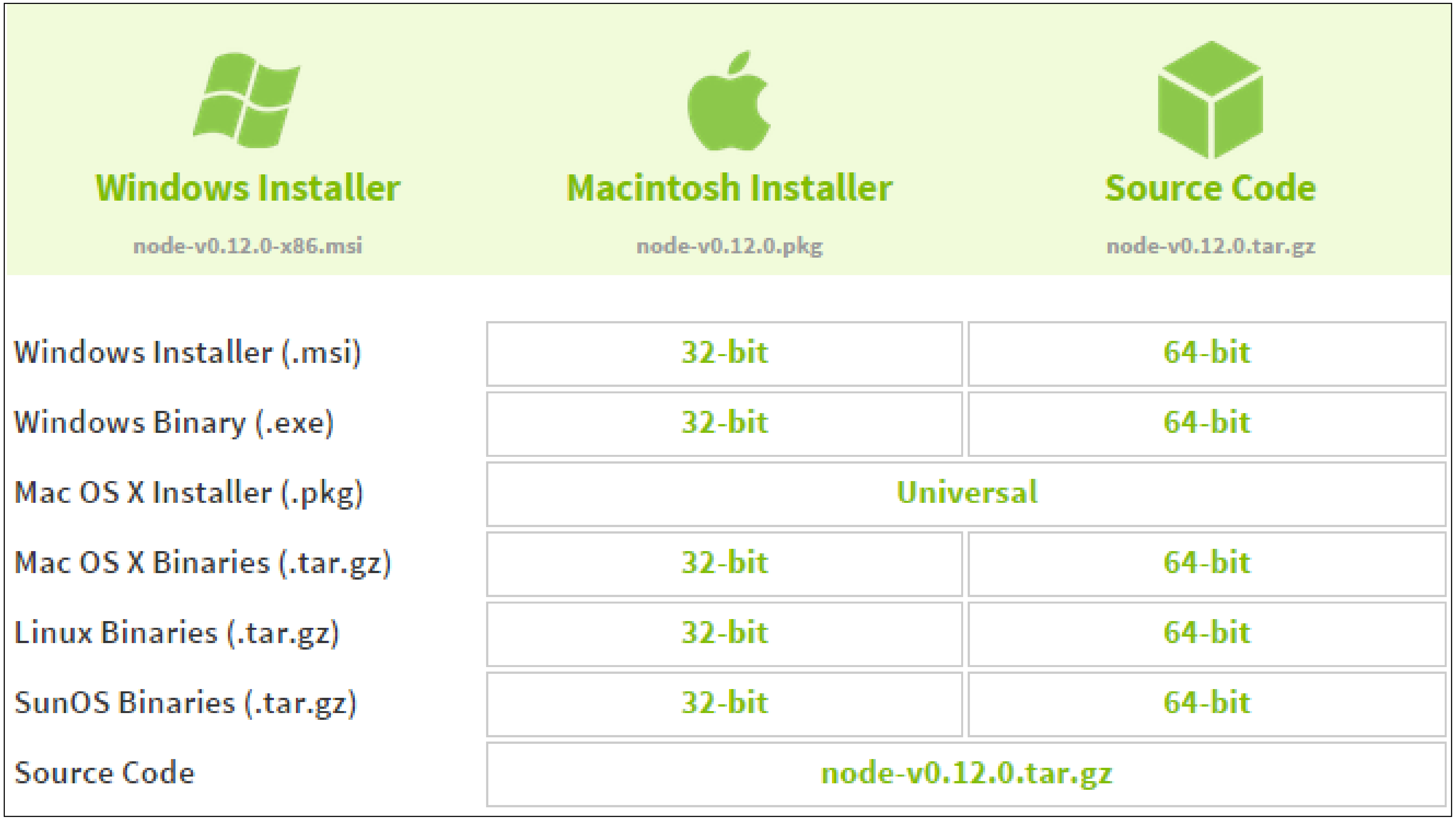
nodejs.org – Node.js official Website



Node.js® is a platform built on **Chrome's JavaScript runtime** for easily building fast, 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.

Current Version: v0.12.0

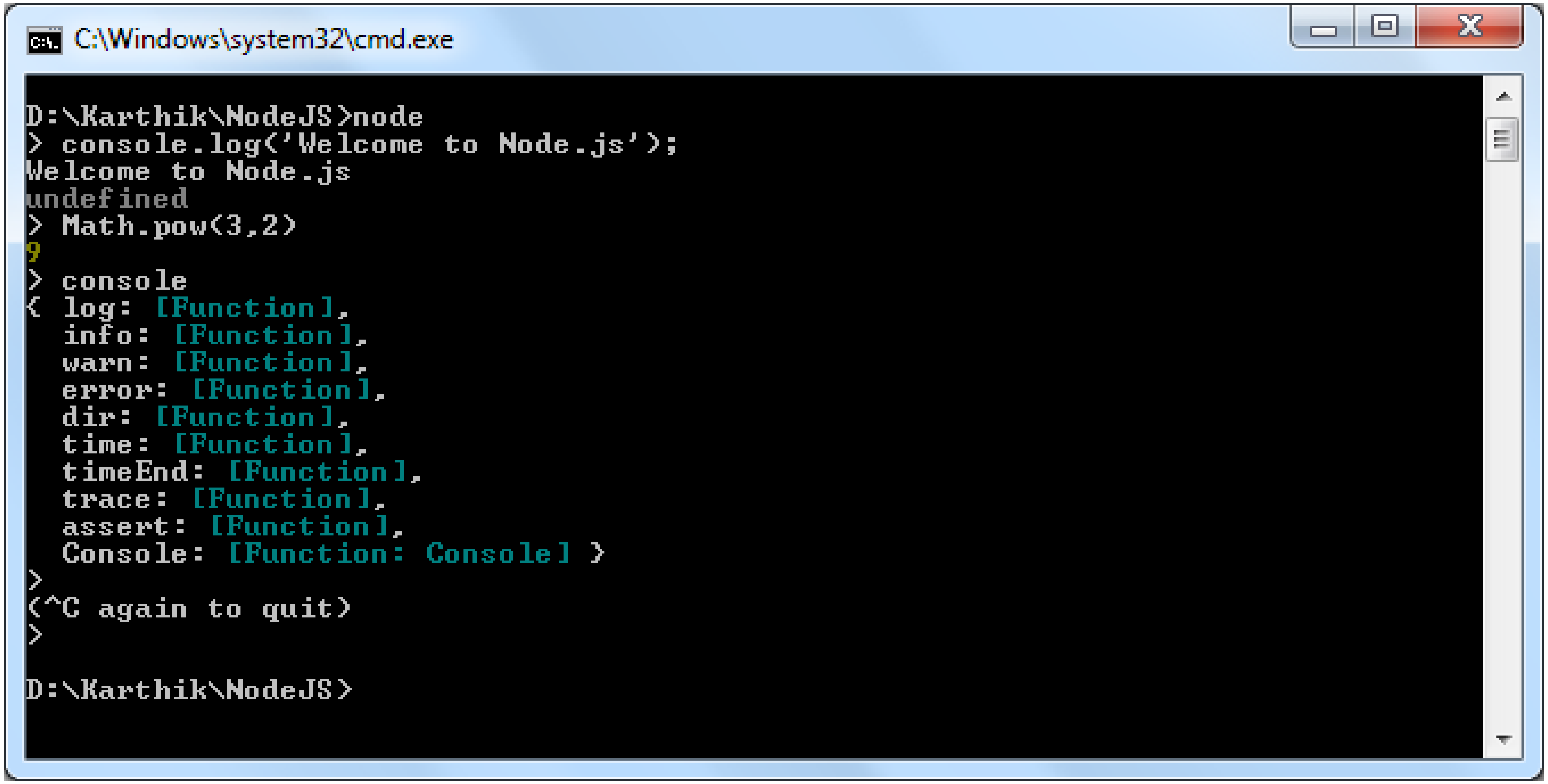
Downloading Node.js



Windows Installer	Macintosh Installer	Source Code
node-v0.12.0-x86.msi	node-v0.12.0.pkg	node-v0.12.0.tar.gz

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.12.0.tar.gz	

Node.js CLI



```
D:\Karthik\NodeJS>node
> console.log('Welcome to Node.js');
Welcome to Node.js
undefined
> Math.pow(3,2)
9
> console
{ log: [Function],
  info: [Function],
  warn: [Function],
  error: [Function],
  dir: [Function],
  time: [Function],
  timeEnd: [Function],
  trace: [Function],
  assert: [Function],
  Console: [Function: Console] }
>
(^C again to quit)
>

D:\Karthik\NodeJS>
```

Node.js Globals

- global
  - Any variables or members attached to global are available anywhere in our application. GLOBAL is an alias for global
    - global.companyName = ‘Capgemini'
    - global['companyName'] // We can directly access the members attached to global.
- process
  - The process object is a global object which is used to Inquire the current process to know the PID, environment variables, platform, memory usage etc.
    - process.platform
    - process.exit( )
- console
  - It provides two primary functions for developers testing web pages and applications
    - console.log(‘Capgemini')

Module Introduction

- A module is the overall container which is used to structure and organize code.
- It supports private data and we can explicitly defined public methods and variables (by just adding/removing the properties in return statement) which lead to increased readability.
- JavaScript doesn’t have special syntax for package / namespace, using module we can create self-contained decoupled pieces of code.
- It avoids collision of the methods/variables with other global APIs.

Demo

- Module

Modules in Node.js

- In Node, modules are referenced either by file path or by name.
- Node's core modules expose some Node core functions(like global, require, module, process, console) to the programmer, and they are preloaded when a Node process starts.
- To use a module of any type, we have to use the require function. The require function returns an object that represents the JavaScript API exposed by the module.
  - *var module = require('module_name');*

Loading a module

- Modules can be referenced depending on which kind of module it is.
- **Loading a core module**
  - Node has several modules compiled into its binary distribution. These are called the core modules. It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
    - *var http = require('http');*
- **Loading a file module (User defined module)**
  - We can load non-core modules by providing the absolute path / relative path. Node will automatically adding the .js extension to the module referred.
    - *var myModule = require('d:/karthik/nodejs/module'); // Absolute path for module.js*
    - *var myModule = require('../module'); // Relative path for module.js (one folder up level)*
    - *var myModule = require('./module'); // Relative path for module.js (Exists in current directory)*

Loading a module

- **Loading a folder module** (User defined module)
  - We can use the path for a folder to load a module.
    - *var myModule = require('./myModuleDir');*
  - Node will presume the given folder as a package and look for a package definition file inside the folder. Package definition file name should be named as ***pagkage.json***
  - Node will try to parse ***package.json*** and look for and use the ***main*** attribute as a relative path for the entry point.
  - We need to use ***npm init*** command to create ***package.json***.
  - Creating ***Package.json*** using ***npm init*** command
    - D:\Karthik\NodeJs\modules> npm init
    - *{ "name": "Karthik_Modules", "version": "1.0.0", "description": "Karthik Modules for Demo",* ***"main": "index.js"****, "scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, "author": "Karthik M <karthik.muthukrishnan@igate.com>", "license": "ISC"}*
    - *var myModule = require('d:/Karthik/NodeJS/modules'); // refer index.js placed in modules folder*

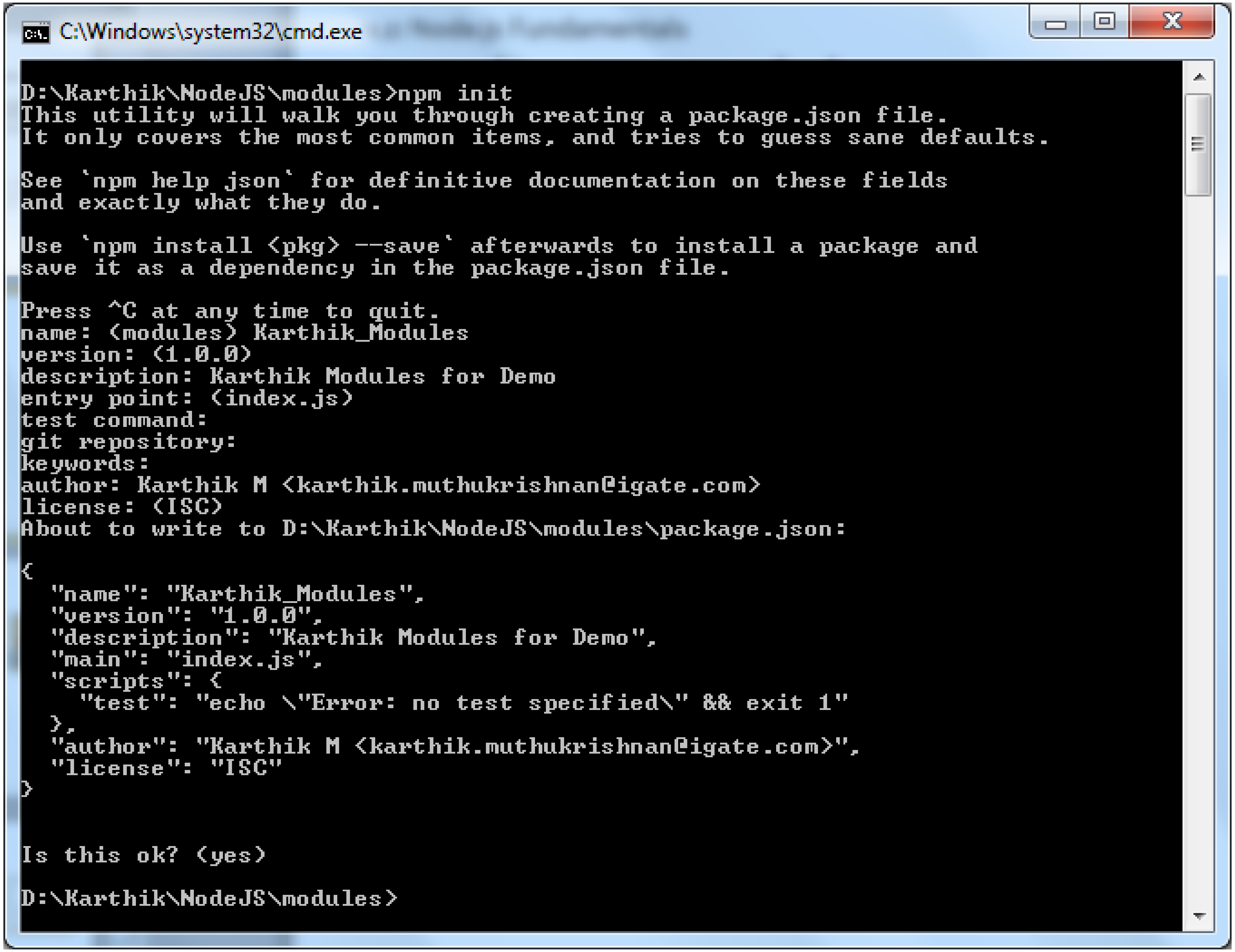
package.json usage

- package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.
- It must be located in project's root directory.
- The JSON data in package.json is expected to adhere to a certain schema. The following fields are used to build the schema for package.json file
  - **name and version** : package.json must be specified at least with a name and version for package. Without these fields, npm cannot process the package.
  - **description and keywords** : description field is used to provide a textual description of package. Keywords field is used to provide an array of keywords to further describe the package. Keywords and a description help people discover the package because they are searched by the npm search command
  - **author** : The primary author of a project is specified in the author field.
  - **main** : Instruct Node to identify its main entry point.

package.json usage

- **dependencies** : Package dependencies are specified in the dependencies field.
- **devdependencies** : Many packages have dependencies that are used only for testing and development. These packages should not be included in the dependencies field. Instead, place them in the separate devdependencies field.
- **scripts :** The scripts field, when present, contains a mapping of npm commands to script commands. The script commands, which can be any executable commands, are run in an external shell process. Two of the most common commands are start and test. The start command launches your application, and test runs one or more of your application's test scripts.

Creating package.json



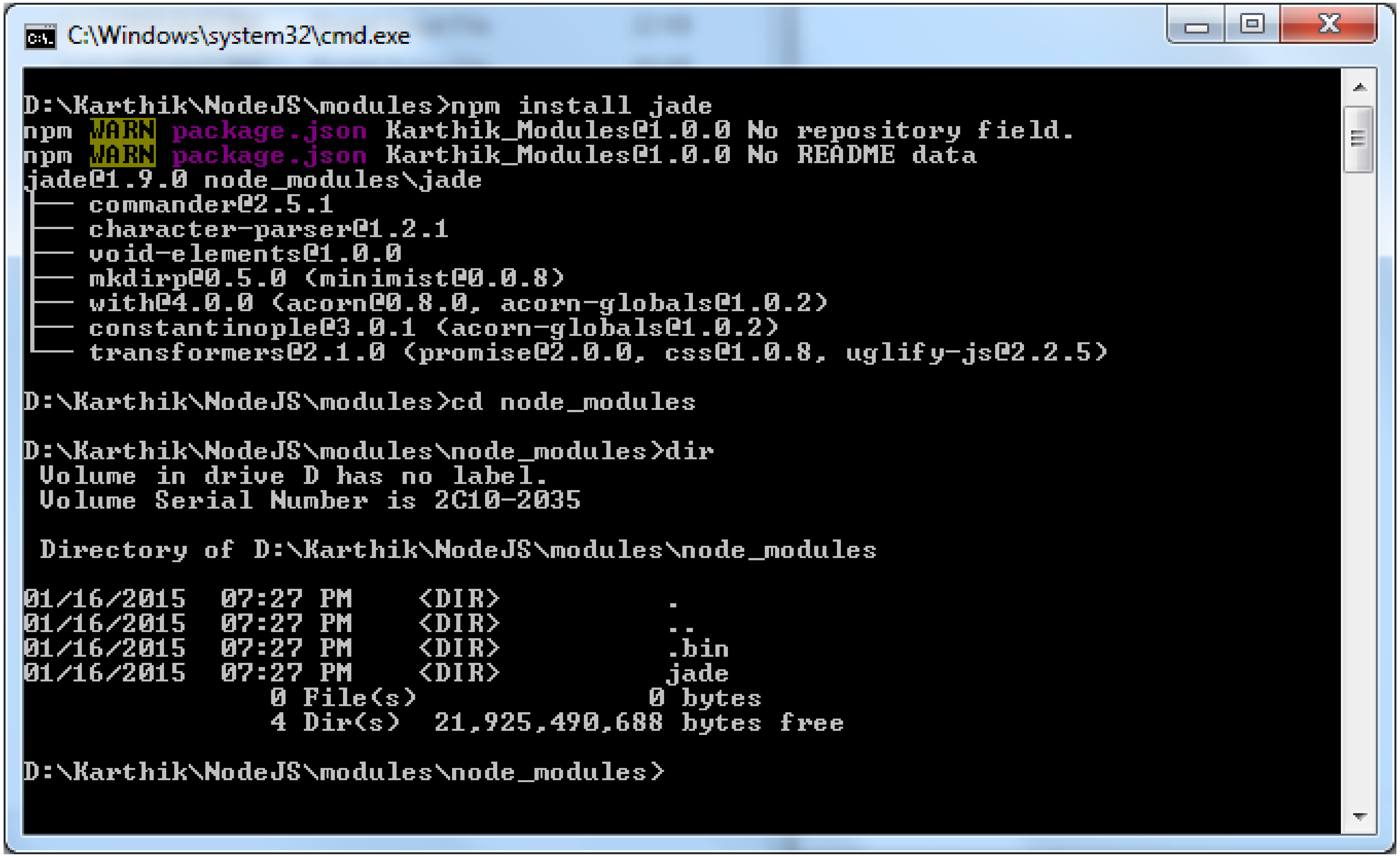
C:\Windows\system32\cmd.exe

```
D:\Karthik\NodeJS\modules>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

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

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

Press ^C at any time to quit.
name: (modules) Karthik_Modules
version: (1.0.0)
description: Karthik Modules for Demo
entry point: (index.js)
test command:
git repository:
keywords:
author: Karthik M <karthik.muthukrishnan@igate.com>
license: (ISC)
About to write to D:\Karthik\NodeJS\modules\package.json:

{
  "name": "Karthik_Modules",
  "version": "1.0.0",
  "description": "Karthik Modules for Demo",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Karthik M <karthik.muthukrishnan@igate.com>",
  "license": "ISC"
}


Is this ok? (yes)

D:\Karthik\NodeJS\modules>
```

Node Package Manager

- **Loading a module(Third party) installed via NPM (Node Package Manager)**
  - Apart from writing our own modules and core modules, we will frequently use the modules written by other people in the Node community and published on the Internet (npmjs.com).
  - We can install those third party modules using the ***Node Package Manager*** which is installed by default with the node installation.
  - To install modules via npm use the **npm install** command.
  - npm installs module packages to the **node_modules** folder.
  - To update an installed package to a newer version use the **npm update** command.
  - If the module name is not relative and is not a core module, Node will try to find it inside the node_modules folder in the current directory.
    - *var jade = require('jade');*
    - *Here jade is not a core module and not available as a user defined module found in relative path, it will look into the node_modules/jade/package.json and refer the file/ folder mentioned in main attribute.*

Loading a third party module (installed via NPM)



```
D:\Karthik\NodeJS\modules>npm install jade
npm WARN package.json Karthik_Modules@1.0.0 No repository field.
npm WARN package.json Karthik_Modules@1.0.0 No README data
jade@1.9.0 node_modules\jade
├── commander@2.5.1
├── character-parser@1.2.1
├── void-elements@1.0.0
├── mkdirp@0.5.0 (minimist@0.0.8)
├── with@4.0.0 (acorn@0.8.0, acorn-globals@1.0.2)
├── constantinople@3.0.1 (acorn-globals@1.0.2)
└── transformers@2.1.0 (promise@2.0.0, css@1.0.8, uglify-js@2.2.5)

D:\Karthik\NodeJS\modules>cd node_modules

D:\Karthik\NodeJS\modules\node_modules>dir
 Volume in drive D has no label.
 Volume Serial Number is 2C10-2035

 Directory of D:\Karthik\NodeJS\modules\node_modules

01/16/2015  07:27 PM    <DIR>          .
01/16/2015  07:27 PM    <DIR>          ..
01/16/2015  07:27 PM    <DIR>          .bin
01/16/2015  07:27 PM    <DIR>          jade
               0 File(s)              0 bytes
               4 Dir(s)  21,925,490,688 bytes free

D:\Karthik\NodeJS\modules\node_modules>
```

Creating and exporting a module

- Creating a module that exposes / exports a function called helloWorld

```
// Save it as myModule.js
exports.helloWorld = function () {
    console.log("Hello World");
}
```

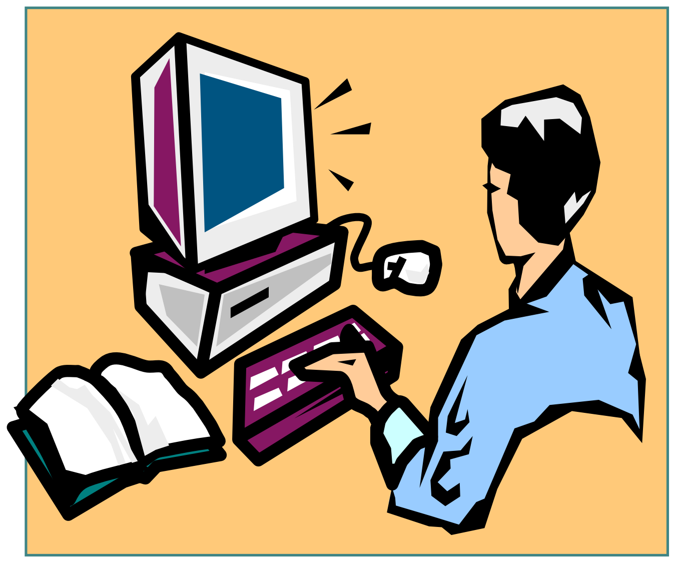
- *exports* object is a special object created by the Node module system which is returned as the value of the require function when you include that module.
- Consuming the function on the exports object created in myModule.js

```
// Save it as moduleTest.js
var module = require('./myModule');
module.helloWorld();
```

- We can replace exports with module.exports
  - exports = module.exports = { }

Demo

- Modules

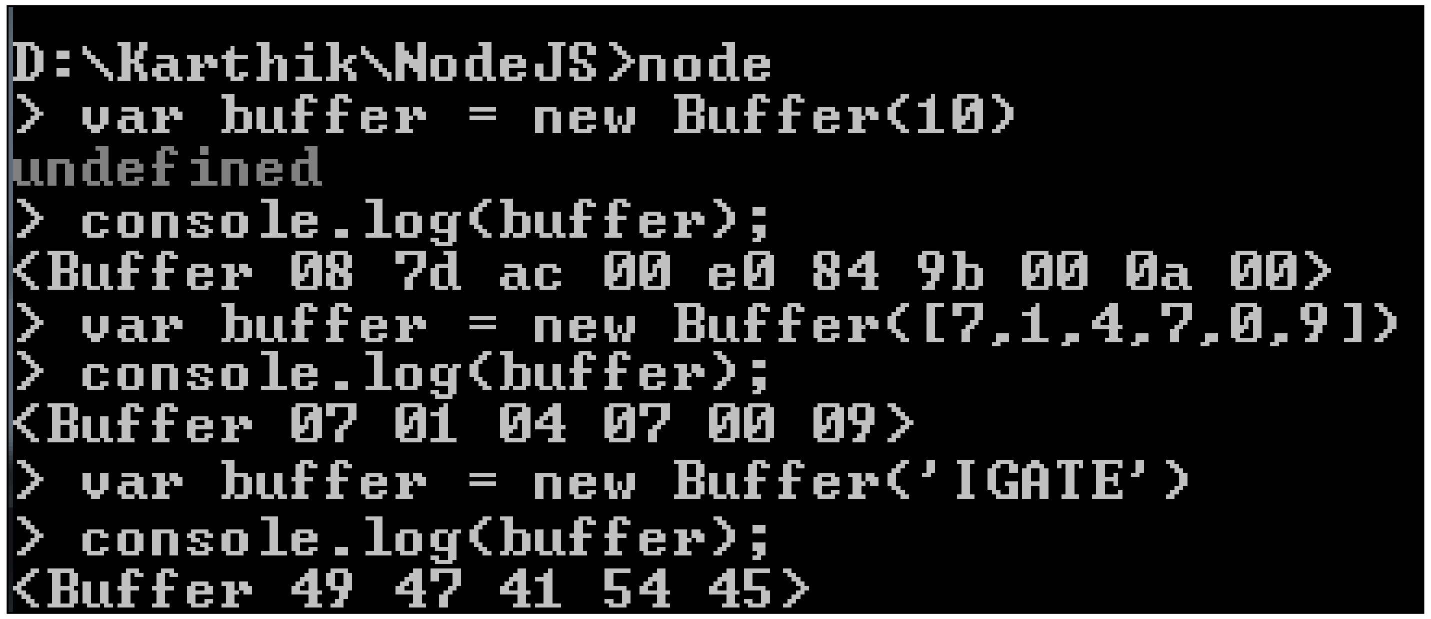


Buffers in Node

- JavaScript doesn’t have a byte type. It just has strings.
- Node is based on JavaScript with just using string type it is very difficult to perform the operations like communicate with HTTP protocol, working with databases, manipulate images and handle file uploads.
- Node includes a binary buffer implementation, which is exposed as a JavaScript API under the Buffer pseudo-class.
- Using buffers we can manipulate, encode, and decode binary data in Node. In node each buffer corresponds to some raw memory allocated outside V8.
- A buffer acts like an array of integers, but cannot be resized

Creating Buffers in Node

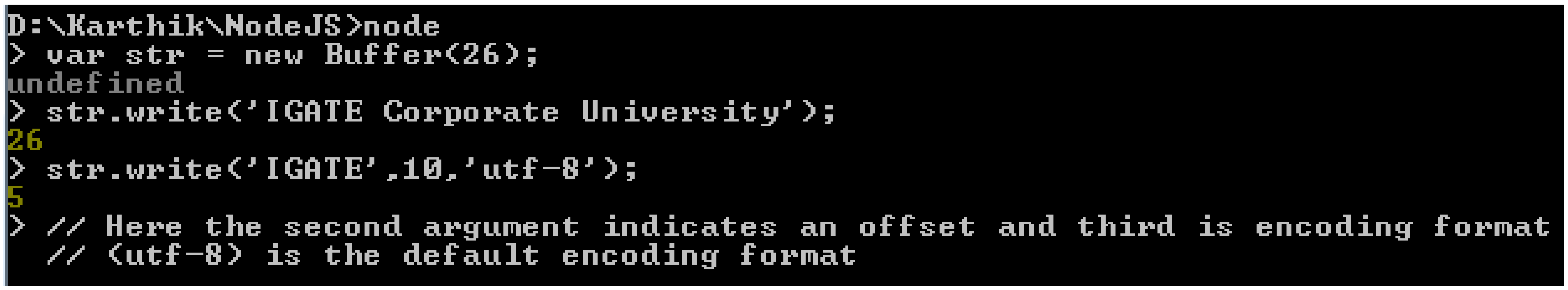
- new Buffer(n) is used to create a new buffer of 'n' octets. One octet can be used to represent decimal values ranging from 0 to 255.
- There are several ways to create new buffers.
  - **new Buffer(n)** : To create a new buffer of 'n' octets
    - var buffer = new Buffer(10);
  - **new Buffer(arr)** : To create a new buffer, using an array of octets.
    - var buffer = new Buffer([7,1,4,7,0,9]);
  - **new Buffer(str,[encoding])** : To create a new buffer, using string and encoding.
    - var buffer = new Buffer("IGATE","utf-8"); // utf-8 is the default encoding in Node.



```
D:\Karthik\NodeJS>node
> var buffer = new Buffer(10)
undefined
> console.log(buffer);
<Buffer 08 7d ac 00 e0 84 9b 00 0a 00>
> var buffer = new Buffer([7,1,4,7,0,9])
> console.log(buffer);
<Buffer 07 01 04 07 00 09>
> var buffer = new Buffer('IGATE')
> console.log(buffer);
<Buffer 49 47 41 54 45>
```

Writing to Buffer

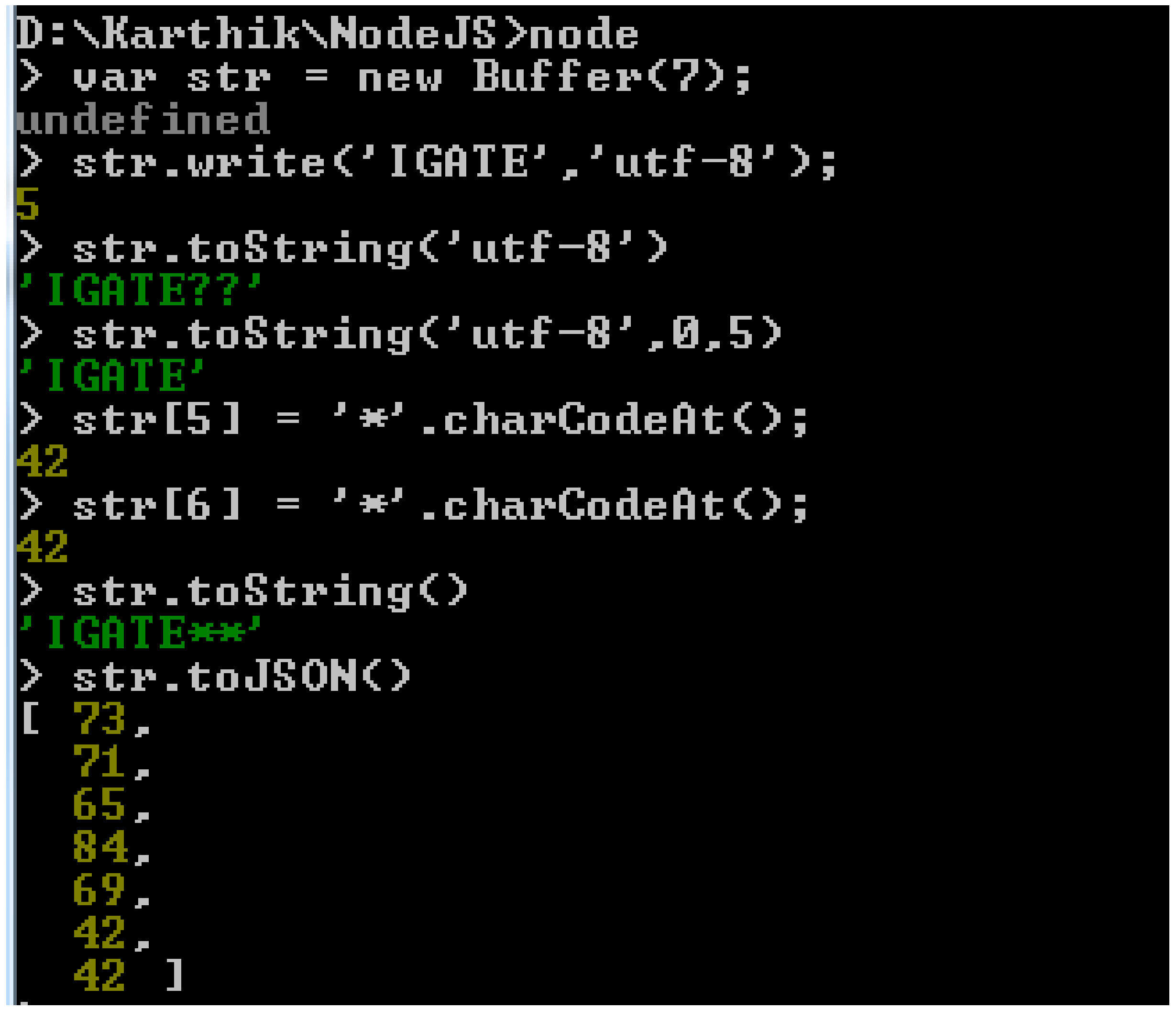
- Writing to Buffer
  - buf.write(str, [offset], [length], [encoding]) method is used to write a string to the buffer.
  - buf.write() returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.
  - An offset or the index of the buffer to start writing at. Default value is 0.



```
D:\Karthik\NodeJS>node
> var str = new Buffer(26);
undefined
> str.write('IGATE Corporate University');
26
> str.write('IGATE',10,'utf-8');
5
> // Here the second argument indicates an offset and third is encoding format
  // (utf-8) is the default encoding format
```

Reading from Buffer

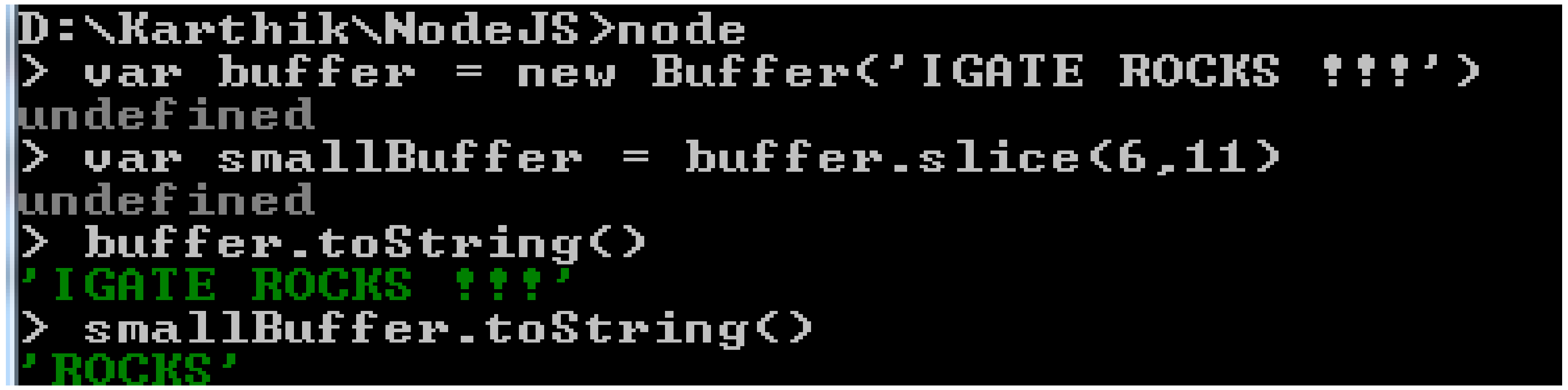
- Reading from buffers
  - buf.toString([encoding], [start], [end]) method decodes and returns a string from buffer data.
  - buf.toString() returns method reads the entire buffer and returns as a string.
  - buf.toJSON() method is used to get the JSON-representation of the Buffer instance, which is identical to the output for JSON Arrays.



```
D:\Karthik\NodeJS>node
> var str = new Buffer(7);
undefined
> str.write('IGATE','utf-8');
5
> str.toString('utf-8')
'IGATE??'
> str.toString('utf-8',0,5)
'IGATE'
> str[5] = '*'.charCodeAt();
42
> str[6] = '*'.charCodeAt();
42
> str.toString()
'IGATE**'
> str.toJSON()
[ 73,
  71,
  65,
  84,
  69,
  42,
  42 ]
```

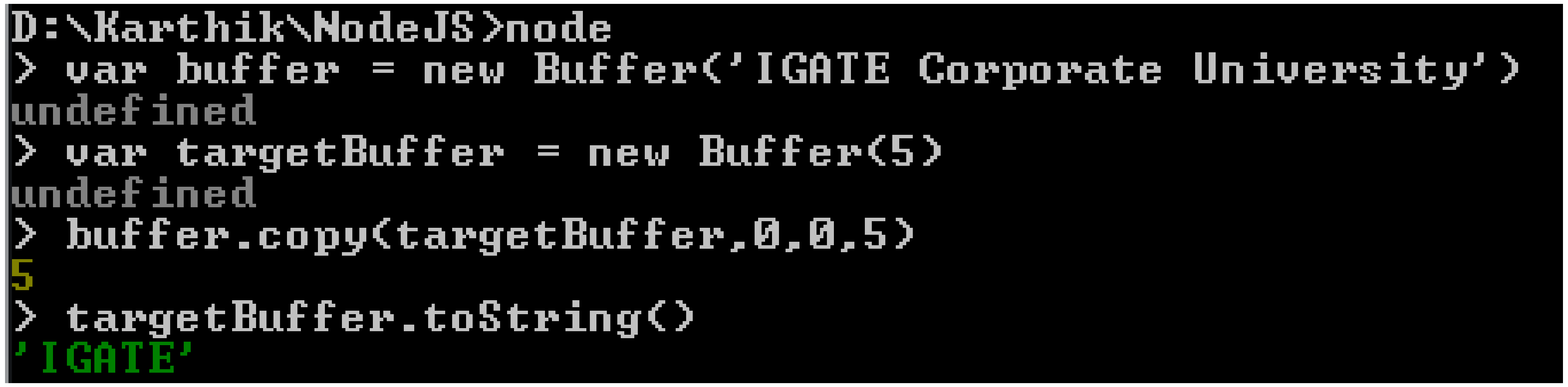
Slicing and copying a buffer

- Slicing a buffer
  - buffer.slice([start],[end]) : We can slice a buffer and extract a portion from it, to create another smaller buffer by specifying the starting and ending positions.



```
D:\Karthik\NodeJS>node
> var buffer = new Buffer('IGATE ROCKS !!!')
undefined
> var smallBuffer = buffer.slice(6,11)
undefined
> buffer.toString()
'IGATE ROCKS !!!'
> smallBuffer.toString()
'ROCKS'
```

- Copying a buffer
  - buffer.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd]) : It is used to copy the contents of one buffer onto another



```
D:\Karthik\NodeJS>node
> var buffer = new Buffer('IGATE Corporate University')
undefined
> var targetBuffer = new Buffer(5)
undefined
> buffer.copy(targetBuffer,0,0,5)
5
> targetBuffer.toString()
'IGATE'
```

Event Handling in Node

- In node there are two event handling techniques. They are called callbacks and EventEmitter.
- Callbacks are for the async equivalent of a function. Any async function in node accepts a callback as it's last parameter

```
var myCallback = function(data) {
  console.log('got data: '+data);
};

var fn = function(callback) {
    callback('Data from Callback');
};

fn(myCallback);
```

EventEmitter

- In node.js an event can be described simply as a string with a corresponding callback and it can be emitted.
- The *on* or *addListener* method allows us to subscribe the callback to the event.
- The emit method "emits" event, which causes the callbacks registered to the event to trigger.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
var myCallback = function(data) {
    console.log('Got data: '+data);
};

eventEmitter.on('karthikEvent', myCallback);
var fn = function() {
     eventEmitter.emit('karthikEvent','Data from Emitter');
};
fn();
```

EventEmitter Methods

- All objects which emit events in node are instances of events.EventEmitter which is available inside Event module.
- We can access the Event module using require("events")
- addListener(event, listener) / on(event, listener)
  - Adds a listener to the end of the listeners array for the specified event. Where listener is a function which needs to be executed when an event is emitted.
- once(event, listener)
  - Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.
- removeListener(event, listener)
  - Remove a listener from the listener array for the specified event
- removeAllListeners([event])
  - Removes all listeners, or those of the specified event

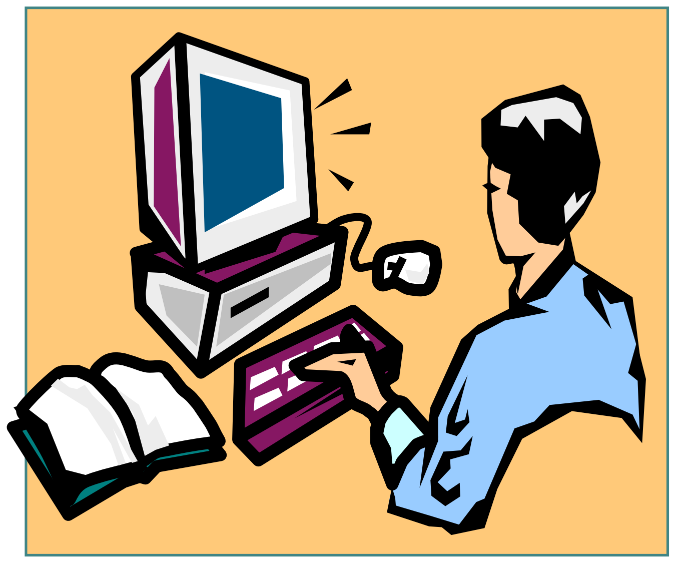
Creating an EventEmitter

- We can create Event Emitter pattern by creating a constructor function / pseudo-class and inheriting from the EventEmitter.

```
var EventEmitter = require('events').EventEmitter,
                          util = require('util');

var Foo = function(){ }

util.inherits(Foo, EventEmitter);

Foo.prototype.someMethod = function() {
     this.emit('customEvent', 'Data from Some Method');
}

var fooObj = new Foo();
fooObj.on('customEvent',function(arg){
       console.log('Custom Event Occurred : '+arg);
});

fooObj.someMethod();
```

Demo

- EventEmitter



File System Module

- By default Node.js installations come with the file system module.
- This module provides a wrapper for the standard file I/O operations.
- We can access the file system module using require("fs")
- All the methods in this module has asynchronous and synchronous forms.
- synchronous methods in this module ends with 'Sync'. For instance *renameSync* is the synchronous method for *rename* asynchronous method.
- The asynchronous form always take a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.
- When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

File I/O methods

- fs.stat(path, callback)
  - Used to retrieve meta-info on a file or directory.
- fs.readFile(filename, [options], callback)
  - Asynchronously reads the entire contents of a file.
- fs.writeFile(filename, data, [options], callback)
  - Asynchronously writes data to a file, replacing the file if it already exists. Data can be a string or a buffer.
- fs.unlink(path, callback)
  - Asynchronously deletes a file.
- fs.watchFile(filename, [options], listener)
  - Watch for changes on filename. The callback listener will be called each time the file is accessed. Second argument is optional by default it is { persistent: true, interval: 5007 }. The listener gets two arguments the current stat object and the previous stat object.

File I/O methods

- fs.exists(path, callback)
  - Test whether or not the given path exists by checking with the file system. The callback argument assigned with either true or false based on the existence.
- fs.rmdir(path, callback)
  - Asynchronously removes the directory.
- fs.mkdir(path, [mode], callback)
  - Asynchronously created the directory.
- fs.open(path, flags, [mode], callback)
  - Asynchronously open the file.
- fs.close(fd, callback)
  - Asynchronously closes the file.
- fs.read(fd, buffer, offset, length, position, callback)
  - Read data from the file specified by fd.

Demo

- FileSystem

Stream

- A stream is an abstract interface implemented by various objects in Node. They represent inbound (ReadStream) or outbound (WriteStream) flow of data.
- Streams are readable, writable, or both (Duplex).
- All streams are instances of EventEmitter.
- Stream base classes can be loaded using *require('stream')*
- *ReadStream* is like an outlet of data, once it is created we can wait for the data, pause it, resume it and indicates when it is actually end.
- *WriteStream* is an abstraction on where we can send data to. It can be a file or a network connection or even an object that outputs data that was transformed(when zipping a file)

Readable Stream

- *ReadStream* is like an outlet of data, which is an abstraction for a source of data that you are reading from
- A Readable stream will not start emitting data until you indicate that you are ready to receive it.
- Readable streams have two "modes": a flowing mode and a non-flowing mode.
  - In flowing mode, data is read from the underlying system and provided to your program as fast as possible.
  - In non-flowing mode, you must explicitly call stream.read() to get chunks of data out.
- Readable streams can emit the following events
  - '**readable**' : This event is fired when a chunk of data can be read from the stream.
  - '**data**' : This event is fired when the data is available. It will switch the stream to flowing mode when it is attached. It is the best way to get the data from stream as soon as possible.

Readable Stream

- **'end'** : This event is fired when there will be no more data to read.
- **'close'** : Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.
- **'error'** : Emitted if there was an error receiving data.

▪ Readable streams has the following methods

- **readable.read([size])** : Pulls data out of the internal buffer and returns it. If there is no data available, then it will return null.
- **readable.setEncoding(encoding)** : Sets the encoding to use.
- **readable.resume()** : This method will cause the readable stream to resume emitting data events.
- **readable.pause()** : This method will cause a stream in flowing-mode to stop emitting data events. Any data that becomes available will remain in the internal buffer.
- **readable.pipe(destination, [options])** : Pulls all the data out of a readable stream and writes it to the supplied destination, automatically managing the flow.

Writable Stream

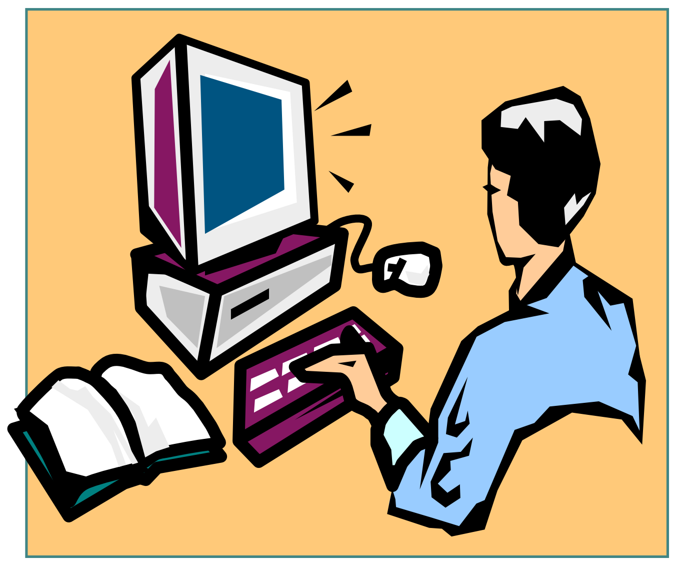
- Writable stream interface is an abstraction for a destination that you are writing data to.
- Writable streams has the following methods
  - **writable.write(chunk, [encoding], [callback])** : This method writes some data to the underlying system and calls the supplied callback once the data has been fully handled. Here chunk is a String / Buffer data to write
  - **writable.end([chunk], [encoding], [callback])** : Call this method when no more data will be written to the stream. Here chunk String / Buffer optional data to write
- Writable streams can emit the following events
  - '**drain**' : If a writable.write(chunk) call returns false, then the drain event will indicate when it is appropriate to begin writing more data to the stream.
  - '**finish**' : When the end() method has been called, and all data has been flushed to the underlying system, this event is emitted

Writable Stream

- **'pipe'** : This is emitted whenever the pipe() method is called on a readable stream, adding this writable to its set of destinations.
- **'unpipe'** : This is emitted whenever the unpipe() method is called on a readable stream, removing this writable from its set of destinations.
- **'error'** : Emitted if there was an error when writing or piping data.

Demo

- Stream



People matter, results count.

About Capgemini

A global leader in consulting, technology services and digital transformation, Capgemini is at the forefront of innovation to address the entire breadth of clients' opportunities in the evolving world of cloud, digital and platforms. Building on its strong 50-year heritage and deep industry-specific expertise, Capgemini enables organizations to realize their business ambitions through an array of services from strategy to operations. Capgemini is driven by the conviction that the business value of technology comes from and through people. It is a multicultural company of 200,000 team members in over 40 countries. The Group reported 2016 global revenues of EUR 12.5 billion.

Visit us at

www.capgemini.com