NODE.JS

HANDWRITTEN NOTES

Prepared by: TOPPERWORLD LEARN & GROW

INDEX

NODE.JS

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.
Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly use to run real-time server applications.
The definition given by its official documentation is as follows:
?Node.js is a platform built on chorme'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.

Node.js = Runtime Environment + JavaScript Library

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. Extermely 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 ie. 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 notifaction 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.

4. Highly Scalable:
Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.

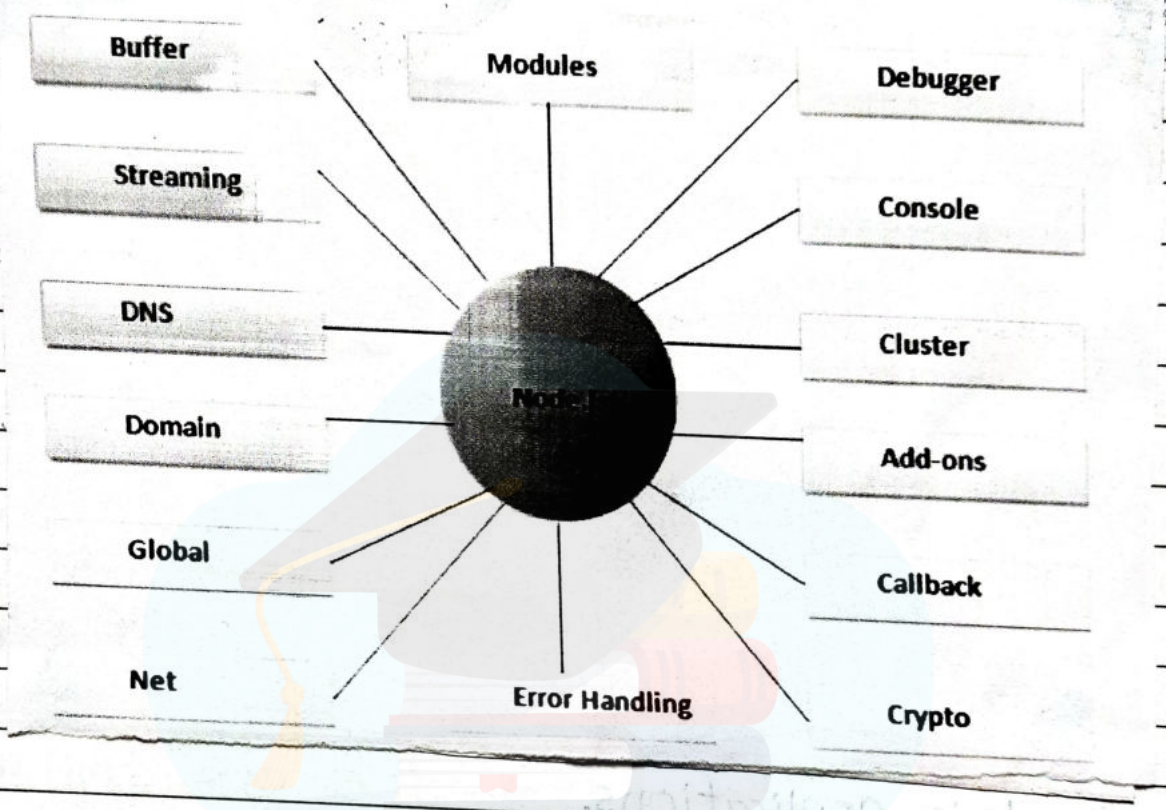
5. No buffering:
Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.

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.

Different parts of Node.js

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



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/

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");
```

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

```
node console_example1.js
```

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.jsmodule

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:

```
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 responce 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 step 1 and step 2 together in a file named "main.js".

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 and node.js stream:

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

Node.js console.log()

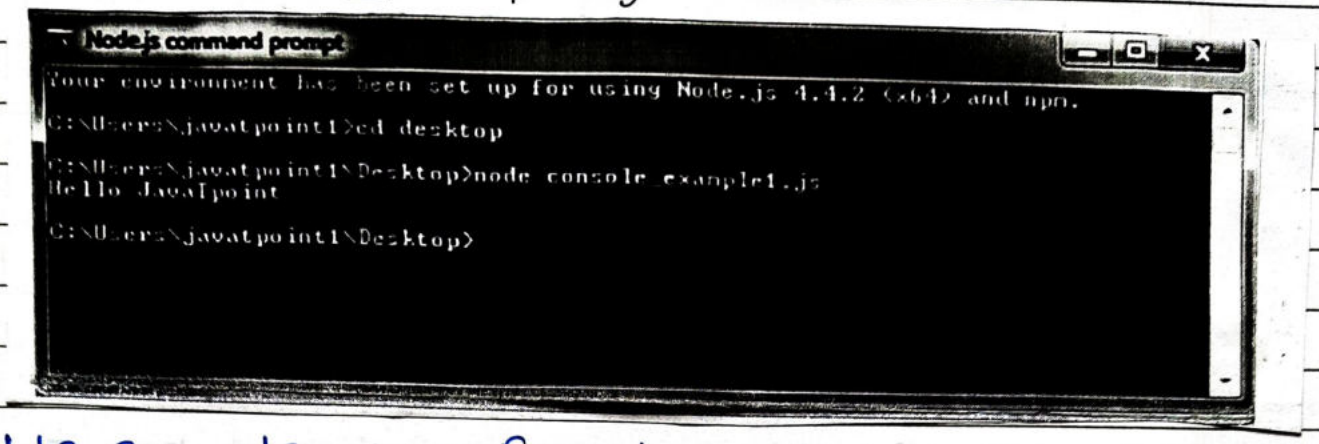
The console.log() function is used to display simple message on console.

```
Console.log('Hello JavaTpoint');
```

Open Node.js Command prompt and run the following

Code:

node console_example1.js



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

File: Console_example2.js

```
Console.log('Hello %s','JavaTpoint');
```

Node.js console.error()

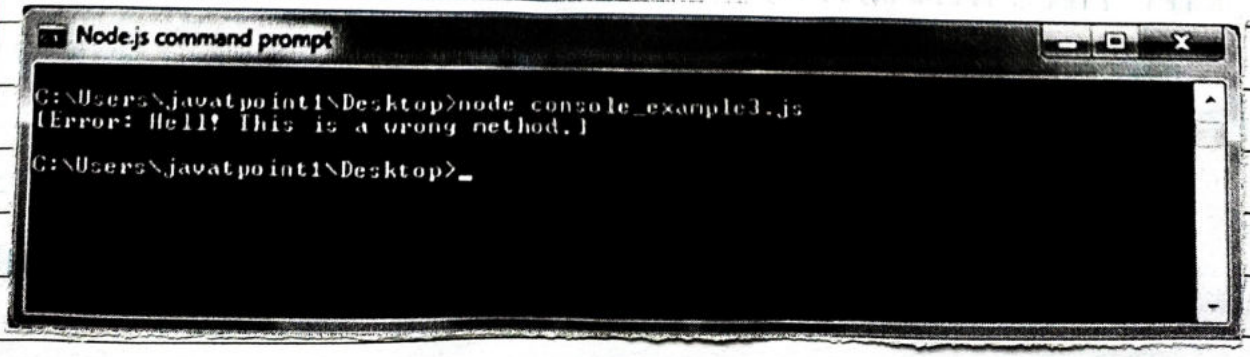
The console.error() function is used to render error message on console.

File: Console_example3.js

```
Console.error(newError('Hell! This is a Wrong method.'));
```

Open Node.js Command prompt and run the following Code:

node console_example3.js



Node.js Console.Warn()

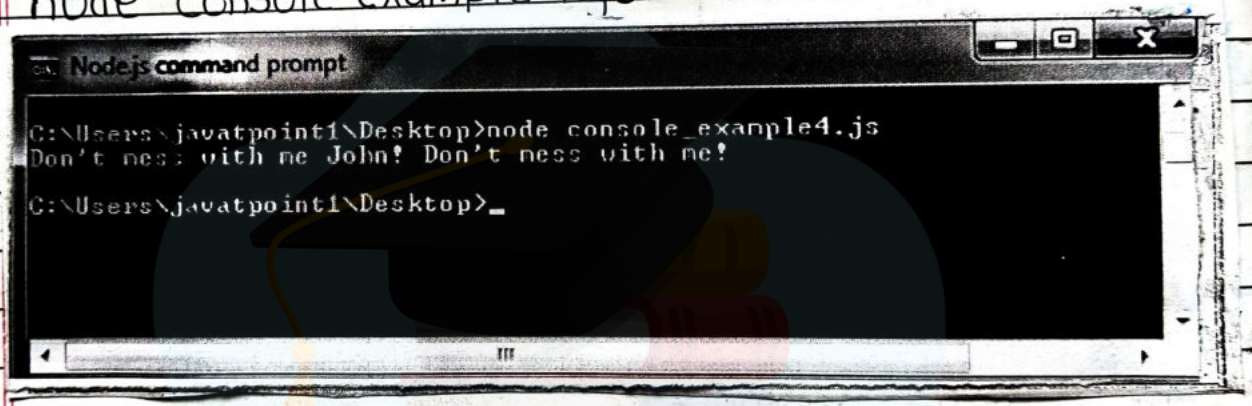
The Console.Warn() function is used to display Warning message on console.

File: Console_example4.js

```
Const name = 'John';
Console.Warn('Don't mess With me ${name}! Don't mess With me!');
```

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

node Console example4.js



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 Shall where you can enter the commands and the system responds With and 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 read 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.

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().
Example: >a=50
50
>var b=50
undefined
> a+b
100

Node.js Multiline expressions

Node REPL supports multiline expressions like JavaScript.

Example

```
Var x = 0
Undefined
> do {
... x++;
... console.log("x:" + x );
... } While (x<10);
```

Node.js Underscore Variable

You can also use underscore_ to get the last result.

Example

```
C:\Users\javatpoint 1\Desktop>node
> Var a = 50
Undefined
> Var b = 50
Undefined
> a+b
100
```

Node.JS REPL Commands

1. Ctrl + C:
   It is used to terminate the current command.

2. Ctrl + C twice:
   It is terminates the node repl.

3. Ctrl + d :

It termintes the node repl.

4. up/down keys:
It is used to see command history and modify previous commands.

5. tab keys:
It specifies the list of current command.

6. help:
It specifies the list of all commands.

7. break:
It is used to exit from multi-line expressions.

8. Clear:
It is used to exit from multi-line expressions.

9. .save filename:
It saves current node repl session to a file.

10. .load filename:
It is used to load file content in current node repl session.

NOde.js Exit REPL

Use ctrl+c command twice to come out of Node.js REPL.

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 version after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:

```
npm version
```

Installing Modules using npm

```
npm install <module Name>
```

Uninstalling a Module

```
npm uninstall express
```

Searching a Module

```
npm search express
```

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.

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 set 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.

12. --trace-Warnings
It prints stack traces for process warnings.

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-chiper-list=list
It specifies an alternative default tls chiper 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 disalbed from script code.)

21. --icu-data-dir=file

It specifies ICU data load path.

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.

Node.js __dirname

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

```
Console.log(__dirname)
```

Node.js __filename

It specifies the filename of the code being exected. This is the resolved asolute 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

Node.js Console

click here to get details of Console class.
http://www.javatpoint.com/nodejs-console

Node.js Buffer

Click here to get details of Buffer class.
http://www.javatpoint.com/nodejs-buffers

Node.js Timer Functions

Click here to get details of Timer functions.
http://www.javatpoint.com/nodejs-timer

Node.js OS

Node.js OS provides some basic operating-system related utility functions.

1. **OS.arch()**
   This method is used to fetch the operating system CPU architecture.

2. **OS.cpus()**
   This method is used to fetch an array of objects containing information about each cpu/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the cpu/core spent in: user, nice, sys, idle and irq).

3. **O.S. endianness()**

This method returns the endianness of the cpu. Its possible values are 'BE' for big endian or 'LE' for little endian.

4. os.freemem()
This method returns the amount of free system memory in bytes.

5. os.homedir()
This method returns the home directory of the current user.

6. os.hostname()
This method is used to returns the hostname of the operating system.

7. os.loadavg()
This method returns an array containing the 1, 5, and 15 minute load averages. The load average is a time fraction taken by system activity, calculated by the operating system and expressed as a fractional number.

8. os.networkinterfaces()
This method returns a list of network interfaces.

9. os.platform()
This method returns the operating system platform of the running computer i.e. 'darwin', 'win32', 'freebsd', 'linux', 'sunos' etc.

10. os.release()
This method returns the operating system release.

11. os.tmpdir()
This method returns the operating system's default directory for temporary files.

12. os.totalmem()
This method returns the total amount of system memory in bytes.

13. os.type()
This method returns the operating system name. For example 'linux' on linux, 'darwin' on os x and 'Windows_nt' on Windows.

14. os.uptime()
This method returns the system uptime in seconds.

15. os.userinfo([options])
This method returns the subset of the password file entry for the current effective user.

Node.js Timer

Node.js Timer functions are global functions. You don't need to use require() function in order to use 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

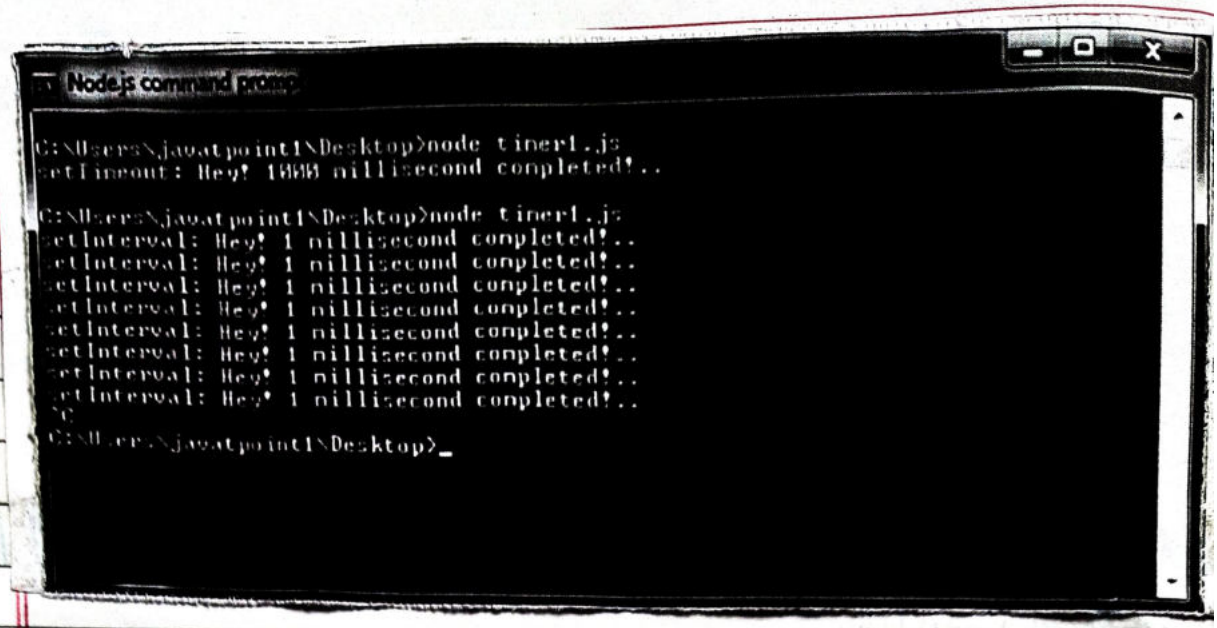
This example will set a time interval of 1000 millisecond and the specified comment will be displayed after every 1000 millisecond until you terminate.

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
```



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
- Use-specified errors
- Assertion errors

Node.js Errors Example

Let's take an example to deplay standard JavaScript error- Reference Error.

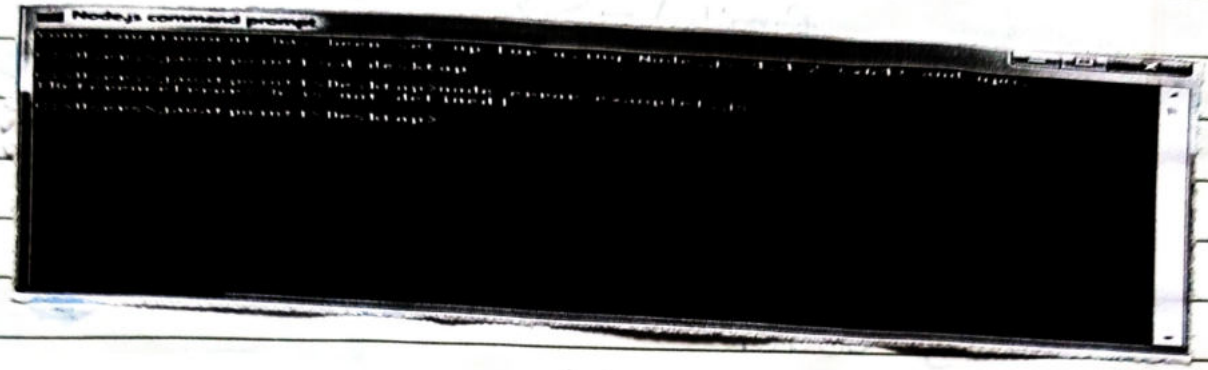
File error_example1.js

```
// Throws With a ReferenceError because bis undefined
 try {
   Const a = 1;
   Const c = a+b;
 } Catch(err) {
   Console.log(err);
 }
```

Open Node.js Command prompt and run the

following code:

node error example 1.js



Node.js DNS

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

- dns.getServers()
- dns.SetServers(servers)
- dns.lookup(hostname[, options], callback)
- dns.lookupService(address, port, callback)
- dns.resolve(hostname[, rrtype], Callback)
- dns.resolve4(hostname, callback)
- dns.resolve6(hostname, callback)
- dns.resolveCname(hostname, callback)
- dns.resolve mx(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

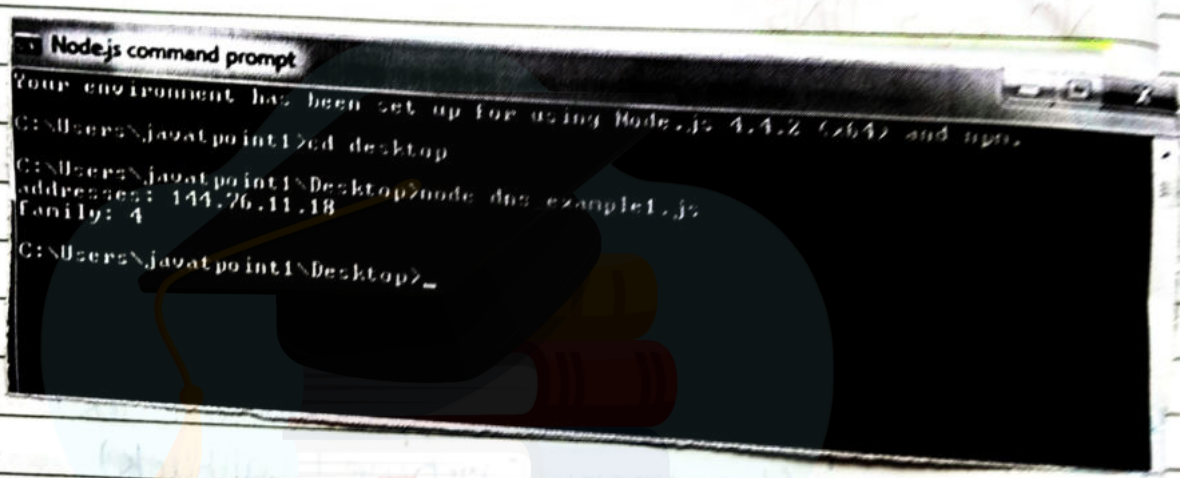
Let's see the example of dns.lookup() function.

File: dns_example1.js

```
Const dns = require('dns');
dns.lookup('www.javatpoint.com',(err, addresses,
                family) => {
  Console.log ('addresses:', addresses);
  Console.log ('family:', family);
});
```

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

```
node dns_example1.js
```



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.

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_server.js

```
const net = require('net');
Var server = net.createServer((socket) => {
   socket.end('goodbye\n');
}).on('error', (err) => {
   // handle errors here
   throw err;
});
// grab a random port.
  server.listen(() => {
   address = server.address();
   console.log('opened server on %j', address);
});
```

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

```
node net_server.js
```

Node.js net example 1

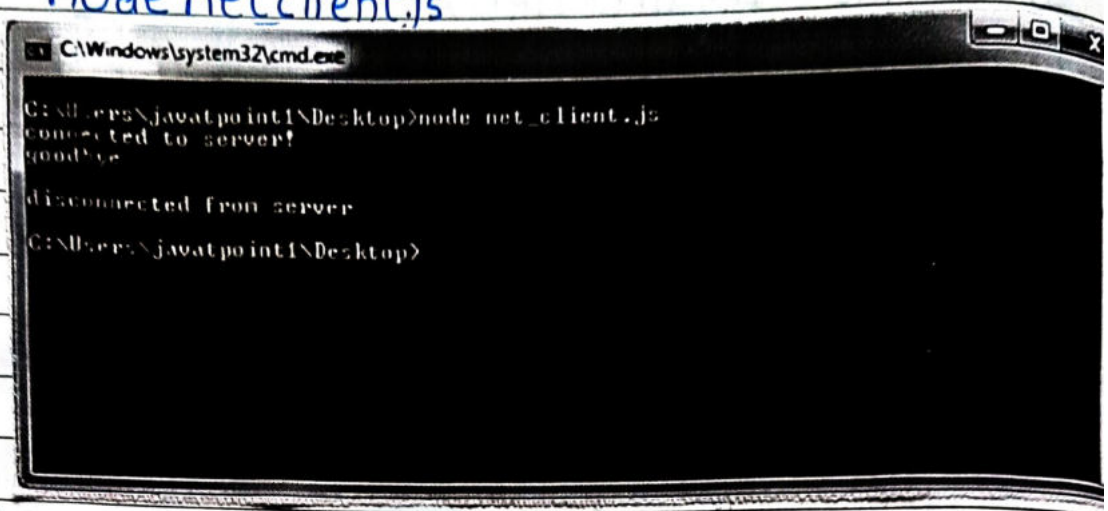
Client:

File: net_client.js

```
Const net = require('net');
Const client = net.connect({port:50302}, () => {//use same
                                  port of server
   cosnole.log('connected to server!');
   client.write('World!\r\n');
});
   client.on('data', (data) => {
   console.log(data.toString());
   client.end();
});
   Client.on('end', () => {
   console.log('disconnected from server');
});
```

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

node net_client.js



Node.js Crypto

The Node.js Crypto module supports cryptography. It provides cryptographic functionality that includes a set of wrappers for open SSL's hash HMAC, cipher, decipher, sign and verify functions.

What it Hash

A hash is a fixed-length string of bits i.e. procedurally and deterministically generated from some arbitrary block of source data.

What is HMAC

HMAC stands for Hash-based Message Authentication Code. It is a process for applying a hash algorithm to both data and a secretkey that results in a single final hash.

Encryption Example using Hash and HMAC

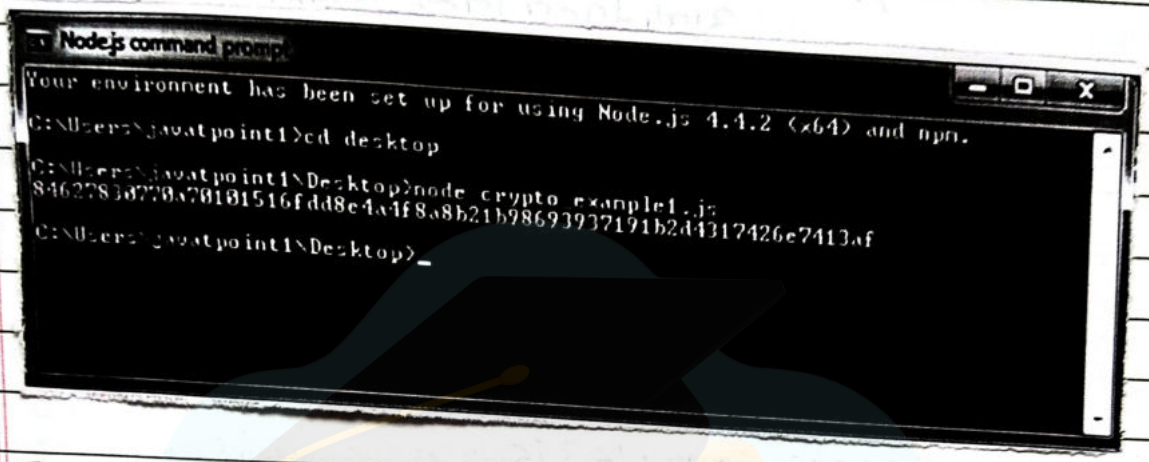
File: crypto_example1.js

```
const crypto = require('crypto');
const secret = 'abcdefg';
```

```
Const hash = crypto.createHmac('sha256', secret)
             .update('Welcome to JavaTpoint')
             .digest('hex');
console.log(hash);
```

Open Nodejs Command prompt and run the following c

```
node crypto_example1.js
```



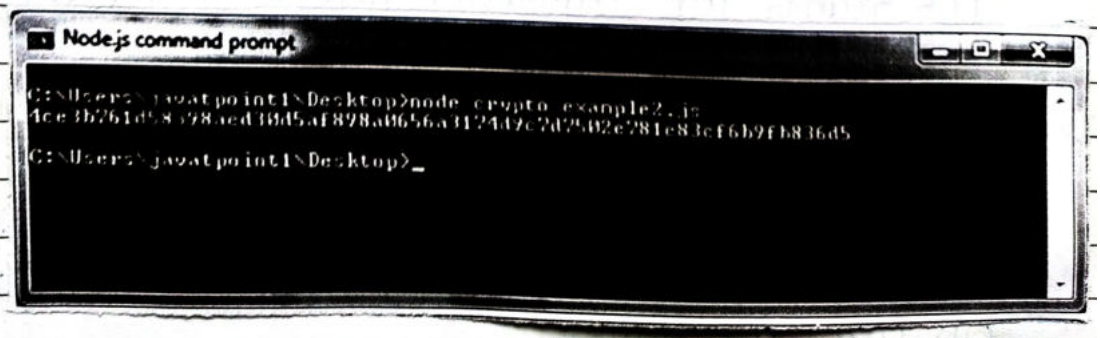
Encryption example using cipher

File: crypto_example2.js

```
Const crypto = require('crypto')
const cipher = crypto.createCipher('aes192', 'a password');
Var encrypted= Cipher.update('Hello JavaTpoint', 'utf8', 'hex'
encrypted + = Cipher.final('hex');
console.log(encrypted);
```

Open Node.js Command prompt and run the following Coo

```
node crypto_example2.js
```



Decryption example using Decipher

File: crypto_example3.js

```
Const crypto = require('crypto');
Const decipher = Crypto.CreateDecipher('aes192',
                                       'a password');
Var encrypted = '4ce3b761d58398aed30d5af898906f
                 3174d9c7d7502e781e83c
Var decrypted = decipher.update(encrypted,'hex',
                                'utf8');
decrypted + = decipher.final('utf8');
console.log(decrypted);
```

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

```
node crypto_example3.js
```

Node.js TLS/SSL

What is TLS/SSL

TLS Stands for Transport Layer Security. It is the successor to Secure Sockets Layer(SSL). TLS along with SSL is used for cryptographic protocols to secure communication over the web.

TLS uses public-key cryptography to encrypt messages. It encrypts communication generally on the TCP layer.

What is public-key cryptography

In public-key cryptography, each client and each server has two keys: public key and private key. Public key is shared with everyone and private key is secured. To encrypt a message, a computer requires its private key and the recipient?s public key. On the other hand, to decrypt the message, the recipient requires its own You have to use require('tls') to access this module.

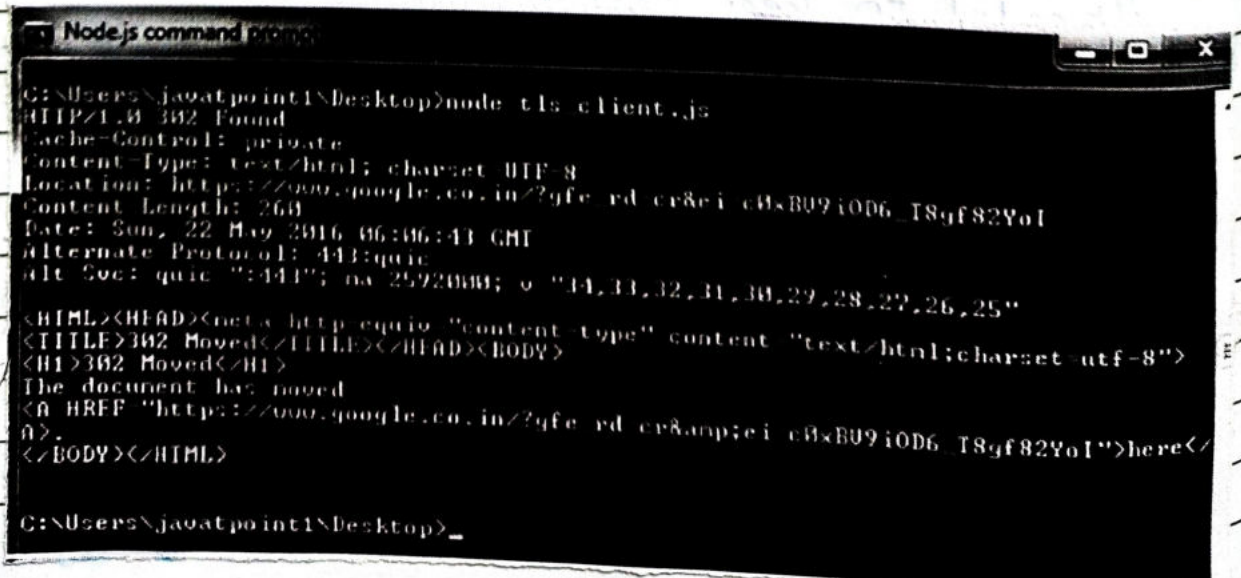
```
Var tls = require('tls');
```

Node.js TLS Client example

File: tls_client.js

```
tls = require('tls');
function connected(stream){
    if(stream){
      //Socket connected
      stream.write("GET/HTTP/1.0\n\rHost:encrypted.
                    google.com:443\n\r\n\r");
    } else{
      console.log("connection failed");
    }
}
//needed to keep socket variable in scope
var dummy = this;
//try to connect to the server
dummy.socket = tls.connect(443,'encrypted.google.com
                           function(){
  // Callback called only after successful socket connection.
  dummy.connected = true;
  if(dummy.socket.authorized){
    //authorization successful
```

```
        dummy.socket.setEncoding('utf-8');
        connected(dummy.socket);
    }else{
        //authorization failed
        console.log(dummy.socket.authorizationError);
        connected(null)
    }
});
dummy.socket.addListener('data', function(data){
    //received data
    console.log(data);
});
dummy.socket.addListener('error', function(error){
    if(!dummy.connected){
        //socket was not connected, notify callback
        connected(null);
    }
    console.log("FAIL");
    console.log(error);
});
dummy.socket.addListener('close', function(){
    // do something
});
```



Node.js Debugger

Node.js provides a simple TCP based protocol and built-in debugging client. For debugging your JavaScript file, you can see use debug argument followed by the js file name you want to debug.

node debug [script.js | -e "script" | <host>:<port>]

Example

node debug main.js

Node.js Process

Node.js provides the facility to get process information such as process id, architecture, platform, version, release uptime, upu usage etc. It can also be used to kill process set uid, set groups, unmask etc.

The process a global object, an instance of Event-Emitter, can be accessed from anyWhere.

Node.js process properties

A list of commonly used Node.js process properties are given below.

Property	Description
arch	returns process architecture: 'arm', 'ia32', or 'x64'
args	returns commands line arguments as an array
env	returns user environment.
pid	returns process id of the process.
platform	returns platform of the process: 'drawin', 'freebsd', 'linux', 'sunos' or 'Win32'
release	returns the metadata for the current node release
Version	returns the node version
versions	returns the node version and its dependencies

Node.js Process Properties Example

File: process_example1.js

```
console.log(`Process Architecture: ${process.arch}`);
console.log(`Process PID: ${process.pid}`);
Console.log(`Process Platform: ${process.platform}`);
console.log(`Process Version: ${process.version}`);
```

Node.js Process Functions

A list of commonly used Node.js process functions are given below.

Function	Description
cwd()	returns path of current working directory
hrtime()	returns the current high-resolution real time in a [seconds, nanoseconds] array
memory-usage()	returns an object having information of memory usage.
process.kill(pid [,signal])	is used to kill the given pid.
uptime	returns the Node.js process uptime in seconds.

Node.js Process Functions Example

File: process_example3.js

```
Console.log(`current directory: ${process.cwd()}`);
Console.log(`uptime: ${process.uptime()}`);
```

Node.js Child Process

The Node.js child process module provides the ability to spawn child processes in a similar manner to popen (3).

There are three major way to create child process:

Node.js child_process.exec() method

The child_process.exec() method runs a command in a console and buffers the output.

child_process.exec(command[, options], callback)

Parameters:

1) command: It specifies the command to run, with space-separated arguments.
2) options: It may contain one or more of the following options:
   - cwd: It specifies the current working directory of the child process.
   - env: It specifies environment key-value pairs.
   - encoding: String (Default: 'utf8')
   - shell: It speifies string shell to execute the command with (Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows. The shell should understand the -c switch on UNIX or /s/c on Windows. On Windows, command line parsing should be compatible with cmd.exe)
   - timeout: Number (Default: 0)
   - maxBuffer: Number (Default: 200 * 1024)
   - killSignal: String (Default 'SIGTERM')
   - uid Number: sets the user identity of the process.
   - gid Number: sets the group identity of the process.

Callback: The callback function specifies three arguments error, stdout and stderr which is called with the following output when process terminates.

Node.js child_process.exec() example

File: child_process_example1.js

```
Const exec = require('child_process');
exec('my.bat', (err, stdout, stderr) => {
  if (err) {
    console.error (err);
    return;
  }
  console.log (stdout);
});
```

Node.js child_process.spawn() method

The child_process.spawn() method launches a new process with a given command. This method returns streams and it is generally used when the process returns large amount of data.

```
child_process.spawn(command[,a[,options]])
```

Node.js child_process.spawn() example

File: support.js

```
console.log("child Process" + process.argv[2] + "executed.");
```

File: master.js

```
const fs = require('fs');
const child_process = require ('child_process');
for (var i =0; i<3; i++ ) {
  Var Worker Process = child_process.spawn('node,support
 WorkerProcess. stdout.on ('data; function(data){
    console.log('stdout:' + data);
});
Worker Process.Stderr.on('data', function(data) {
    Console.log ('stderr' + data)
});
```

```
Worker Process.on('close', function
    Console.log('child process exited With code' + code);
});
}
```

Node.js child_process.fork() method

This child_process.fork method is a special case of the spawn() to create Node processes. This method returns Object With a built-in communication channel in addition to having all the methods in a normal childProccss instance.

```
Child_process.fork(modulePath[, a[, options]])
```

Node.js child_process.fork() example

File : Support.js

```
Const fs = require('fs')
Const Child process = require('child_process');
 for (Var i = 0; i < 3; i++) {
 Var Worker process = child_process.fork("Support.js", [i]
  Worker_process.on('close', function
      Console.log('Child process exited With code' + code);
    });
}
```

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 octect 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 to 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 Shytax 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

```
buf.Write(string[, offset][, length][, encoding])
```

Example
File: main.js

```
buf = new Buffer(256);
```

```
len = buf.write("Simply Easy Learning");
console.log ("Octets written: " + len);
```

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

```
node main.js
```

Node.js Reading from buffers

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

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

Example

File: main.js

```
buf = new Buffer(26);
for (var i = 0 ; i < 26; i++) {
  buf[i] = i + 97;
}
console.log(buf.toString('ascii')); // outputs: abcdefghijkl
                                     mnopqrstuvwxyz
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
        defaulfs to 'utf8'; outputs abcde
```

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

```
node main.js
```

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 has been flushed to underlying stream system.

Node.js Reading from stream

File: main.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");
```

Now, open the Node.js command prompt and run the main.js

```
node main.js
```

Node.js Writing to stream

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

File : main.js

```
Var fs = require ("fs");
Var data = 'A solution of all Technology';
// Create a Writable stream.
Var Writer Stream = fs.createWriteStream ('output.txt');
// Write the data to stream with encoding to be ut
Writer Stream.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(){
    console.log(err.stack);
});
Console.log ("Program Ended");
```

Now open the Node.js Command prompt and run the main.js:

```
node main.js
```

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:

```
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.

Lets take an example:
Create a text file named "input.txt" having the following content.
File: Input.txt
Javatpoint is a one of the best online tutorial website to learn different technologies in a very easy and efficient manner.
Lets take an example to create a JavaScript file named "main.js" having the following code:
File: main.js

```
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");
```

Node.js Open a file

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

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.