Node JS

Table of Contents

|  |  |  |
| --- | --- | --- |
| Day | Topic | Status |
| Day 1 | Introduction, Modules & NPM |  |
| Node.js Framework |  |
| Advantages of using Node.js |  |
| Day 2 | Creating Node.js Application |  |
| REPL |  |
| Day 3 | Node.js vs Traditional Web Server |  |
| Day 4 | Data Type, Objects, Built – in Objects, Functions |  |
| Day 5 | Asynchronous Iteration |  |
| Advanced Buffers |  |
| Day 6 | Child Processes |  |
| Cryptography |  |
| Day 7 & Day 8 | TCP |  |
| HTTP & How To Create an HTTPS Server Using a Node.js HTTP Framework |  |
| File-System |  |
| Socket.io |  |
| TLS |  |
| Day 9 | Event Emitter |  |
| Streams |  |
| Day 10 | Express.js & Flatiron.js |  |
| Day 11 | Using a relational database with Node.js |  |

Note: Tentative Days may vary based on to the Flow of session and Interaction. Most probably it will on the schedule.

**Day 1**

**Introduction**

Node.js is a very powerful JavaScript-based framework/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. Node.js is open source, completely free, and used by thousands of developers around the world.

**Node.js**

* Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.
* Node.js is part runtime environment and part library for building network applications using server-side JavaScript.
* 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 uses an event-based server execution procedure rather than the multithreaded execution.
* Node.js is an open-source, cross-platform runtime environment for developing server-side web applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on a wide variety of platforms, including OS X, Microsoft Windows, Linux, FreeBSD, NonStop, IBM AIX, IBM System z and IBM i. Its work is hosted and supported by the Node.js Foundation, a collaborative project at the Linux Foundation.
* Node.js provides an event-driven architecture and a non-blocking I/O API designed to optimize an application's throughput and scalability for real-time web applications. It uses Google V8 JavaScript engine to execute code, and a large percentage of the basic modules are written in JavaScript. Node.js contains a built-in library to allow applications to act as a stand-alone web server.

**Node.js Framework**

Node.js is one of the most popular javascript framework that allows you to build scalable network web applications. Node.js contains different kinds of framework, such as MVC framework, full-stack framework, REST API and generators. They are included as server libraries, which allows Node.js to run a web server without the use of external software like Apache and Lighttpd. These frameworks make it user-friendly and allows it to support a large number of features and functions to develop huge web applications in just a few steps.

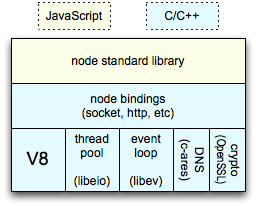
**Web server**

* Web server can refer to either the hardware (the computer) or the software (the computer application) that helps to deliver content that can be accessed through the Internet.
* The primary function of a web server is to deliver web pages on the request to clients. This means delivery of HTML documents and any additional content that may be included by a document, such as images, style sheets and scripts.
* A web server is the basic to delivering requests/pagess to the clients/user on the internet

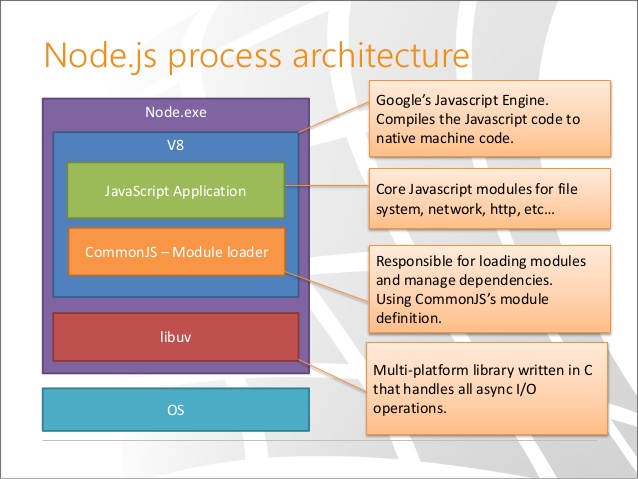
**Web framework**

* A web application framework is a software framework that is designed to support the development of dynamic websites, web applications and web services. The framework aims to alleviate the overhead associated with common activities performed in Web development.
* For example, many frameworks provide libraries for database access, templating frameworks and session management, and they often promote code reuse.
* A web framework uses a webserver to deliver the requests to client, but it is not the web server.

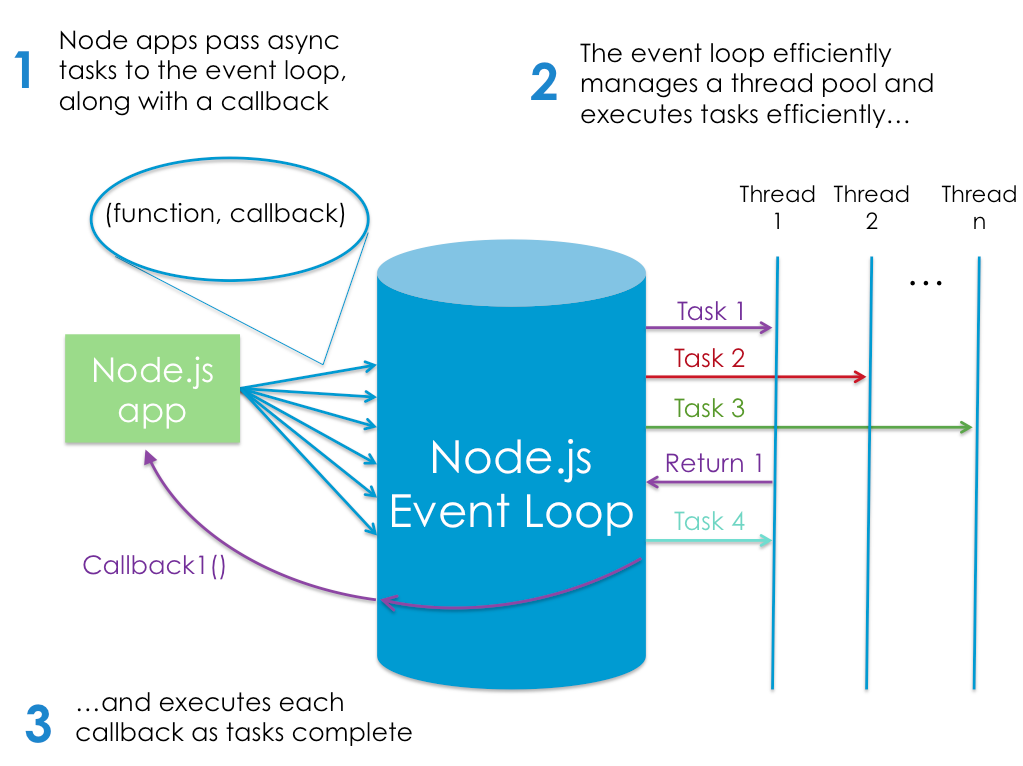
**Figure 1. Block Diagram**



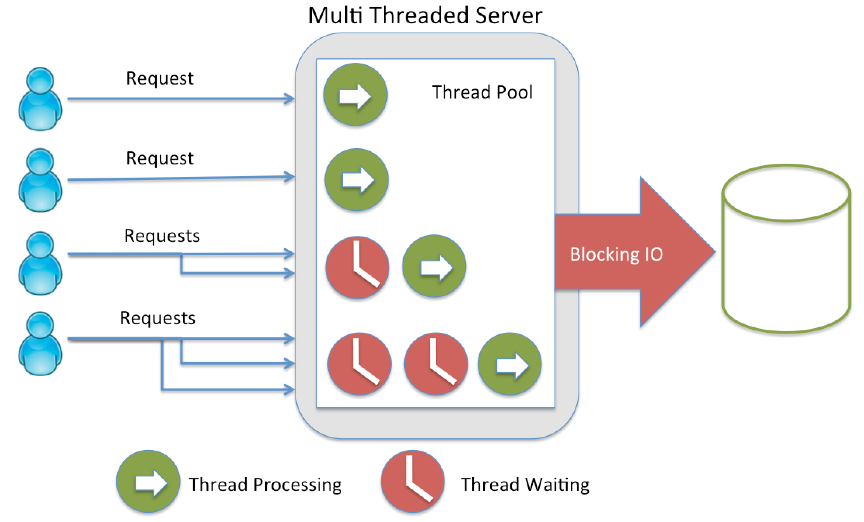
**Figure 2: Node.js Process Architecture**



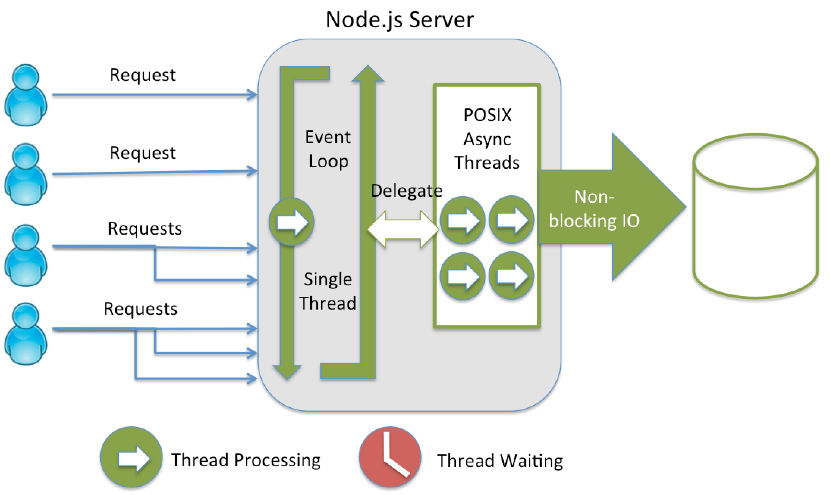
**Figure 3: Node.js Event Loop Process**



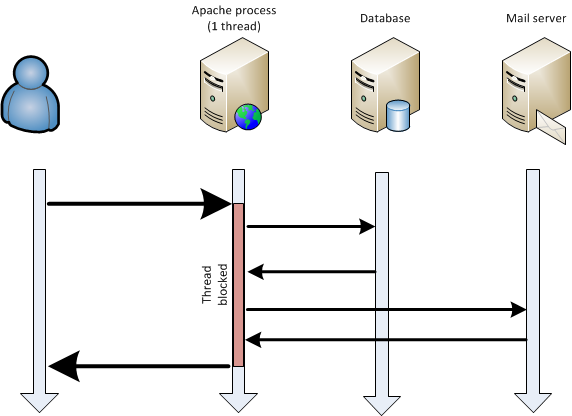
**Figure 4: Traditional Approach**

****

**Figure 5: Current Node.js Approach**

****

**Figure 6 shows a single thread: when a client is connected, the thread is blocked (represented in red) while the final result is not ready. Thus, Apache thread is blocked waiting (doing nothing) during long I/O operations such as writing to disk or database.**



**V8**

* V8 is the JavaScript execution engine built for Google Chrome and open-sourced by Google in 2008. Written in C++, V8 compiles JavaScript source code to native machine code instead of interpreting it in real time.
* Node.js uses libuv to handle asynchronous events. Libuv is an abstraction layer for network and file system functionality on both Windows and POSIX-based systems like Linux, Mac OS X, OSS on NonStop and Unix.
* The core functionality of Node.js resides in a JavaScript library. The Node.js bindings, written in C++, connect these technologies to each other and to the operating system.

**What Node.js can do?**

* Node.js allows the creation of web servers and networking tools using JavaScript and a collection of "modules" that handle various core functionality.
* Modules handle file system I/O, networking (HTTP, TCP, UDP, DNS, or TLS/SSL), binary data (buffers), cryptography functions, data streams and other core functions. Node's modules use an API designed to reduce the complexity of writing server applications.
* Frameworks can be used to accelerate the development of applications, and common frameworks are Express.js, Socket.IO and Connect.
* Node.js applications can run on Microsoft Windows, UNIX, NonStop and Mac OS X servers.
* Node.js is primarily used to build network programs such as web servers, making it similar to PHP. The biggest difference between PHP and Node.js is that PHP is a blocking language, where commands execute only after the previous command has completed, while Node.js is a non-blocking language where commands execute in parallel, and use callbacks to signal completion.

**Node.js for Developers:**

* Node.js implements event-driven programming for web applications in JavaScript. Developers can create highly scalable [clarification needed] servers without using threading by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task. Concurrency is difficult in many server-side programming languages, and often leads to poor performance.
* Thousands of open-source libraries have been built for Node.js, most of which are hosted on the npm website. Its developer community has two main mailing lists and the IRC channel #node.js on freenode. There is an annual Node.js developer conference, NodeConf.
* Node.js applications can alternatively be written with CoffeeScript (an alternative form of JavaScript), Dart or Microsoft TypeScript (strongly typed forms of JavaScript), or any other language that can compile to JavaScript.

**Node.js users:**

Node.js is used by IBM, Microsoft, Yahoo!, Walmart, Groupon, SAP, LinkedIn, Rakuten, PayPal, Voxer and GoDaddy.

**Node.js Milestone:**

* Node.js was invented in 2009 by Ryan Dahl and other developers working at Joyent. Node.js was first released in 2009 supporting only Linux.
* In 2011, a package manager was introduced for the Node.js environment called npm. The package manager makes it easier for the community to publish and share open-source Node.js libraries and is designed to simplify installation, updating and uninstallation of libraries.
* In June 2011, Microsoft and Joyent implemented a native Windows version of Node.js. The first Node.js build supporting Windows was released in July 2011.
* In January 2012, Dahl stepped aside, promoting coworker and npm creator Isaac Schlueter to manage the project. In January 2014, Schlueter announced Timothy J. Fontaine would be the new project lead.
* In December 2014, Fedor Indutny started io.js, a fork of Node.js. Due to internal conflict over Joyent's governance, io.js was created as an open governance alternative with a separate technical committee.
* In February 2015, the intent to form a neutral Node.js Foundation was announced. By June 2015, the Node.js and io.js communities voted to work together under the Node.js Foundation.

**Node’s Goal?**

Its goal is to offer an easy and safe way to build high performance and scalable network applications in JavaScript.

Those goals are achieved thanks it’s architecture:

**a. Single Threaded:**

Node use a single thread to run instead of other server like Apache HTTP who spawn a thread per request, this approach result in avoiding CPU context switching and massive execution stacks in memory. This is also the method used by nginx and other servers developed to counter the C10K problem.

**b. Event Loop:**

Written in C++ using the Marc Lehman’s libev library, the event loop use epoll or kqueue for scalable event notification mechanism.

**c. Non-blocking I/O:**

Node avoid CPU time loss usually made by waiting for an input or an output response (database, file system, web service, …) thanks to the full-featured asynchronous I/O provided by Marc Lehmann’s libeio library.

These characteristics allow Node to handle a large amount of traffic by handling as quickly as possible a request to free the thread for the next one.

Node has a built-in support for most important protocols like TCP, DNS, and HTTP (the one that we will focus on). The design goal of a Node application is that any function performing an I/O must use a callback. That’s why there is no blocking methods provided in Node’s API.

The HTTP implementation offered by Node is very complete and natively support chunked request and response and hanging request for comet applications. The Node’s footprint for each http stream is only 36 bytes (source).

**Real world usage of Node.js**

Though pretty young, Node.js is already being used by a lot of Organizations, Projects and Applications. The following list is an indication of, for what kind of applications Node is being used:

* HTTP Proxy
* IRC Bot / Chat / Messaging
* Web service (data)
* real time analysis (data, email)
* Client of another web service (Like Twitter client)
* real time collaboration
* Log and log monitoring
* Web based software to serve a specific client requirement (like reservations book for restaurateurs, Grade book)
* Application server

**Advantages of using Node.js**

Node.js is part runtime environment and part library for building network applications using server-side JavaScript. It uses Chrome's JavaScript runtime engine to execute JS code directly without the need for the browser sandbox.

Node.js is supposed to be three things: easy to use, fast, and scalable. It's easy in that a little code goes a long way and uses it's own concurrency model that's a lot simpler than traditional O/S threads. Much like client-side JS libraries, node.js does provide abstraction so that a lot of the boilerplate code is handled under the covers.

The scalability results from node.js's better memory efficiency under high-loads and non-blocking execution model. Functions in node.js almost never perform I/O directly, so processes never block. That leaves the developer free from worrying about dead-locking processes and able to focus on the code at hand. That's just another reason why beginner to intermediate programmers are able to use node.js to develop fast system.

* Event driven and Non-Blocking, so handles concurrent connections efficiently than it's counterparts.
* It's great to have common language on both client and server.
* Event better, if used with Databases like MongoDb, then it's 100% JavaScript, From DataBase to Server to Client.
* Clean, Functional API.

**Disadvantages of using Node.js**

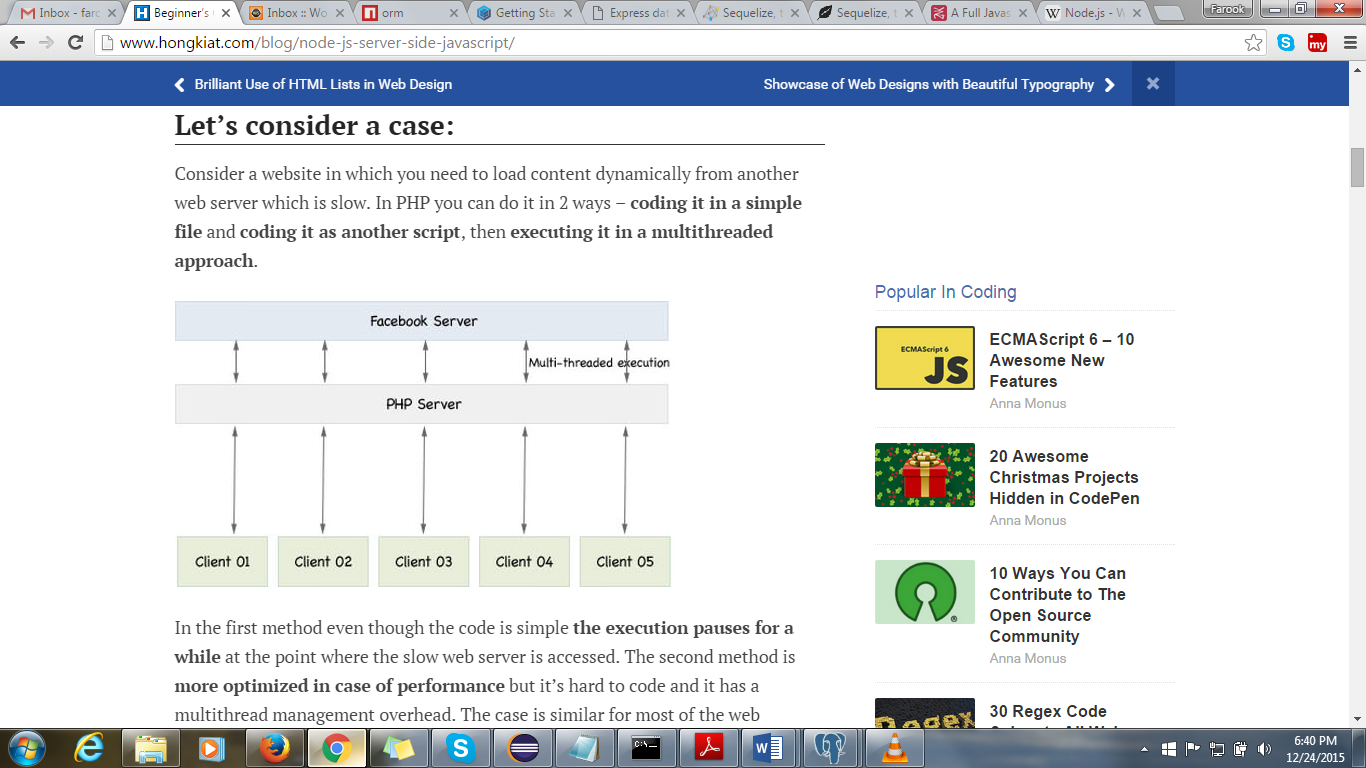
Since Node is very young, lots of database drivers (required to connect to databases) and modules (for adding some specific functionality), are unstable. It is still unstable on Windows platform. But Node has a very active and responsive community and hope that these problems will be marginalized in near future.

**Let’s consider a case:**

Consider a website in which you need to load content dynamically from another web server which is slow. In PHP you can do it in 2 ways – coding it in a simple file and coding it as another script, then executing it in a multithreaded approach.

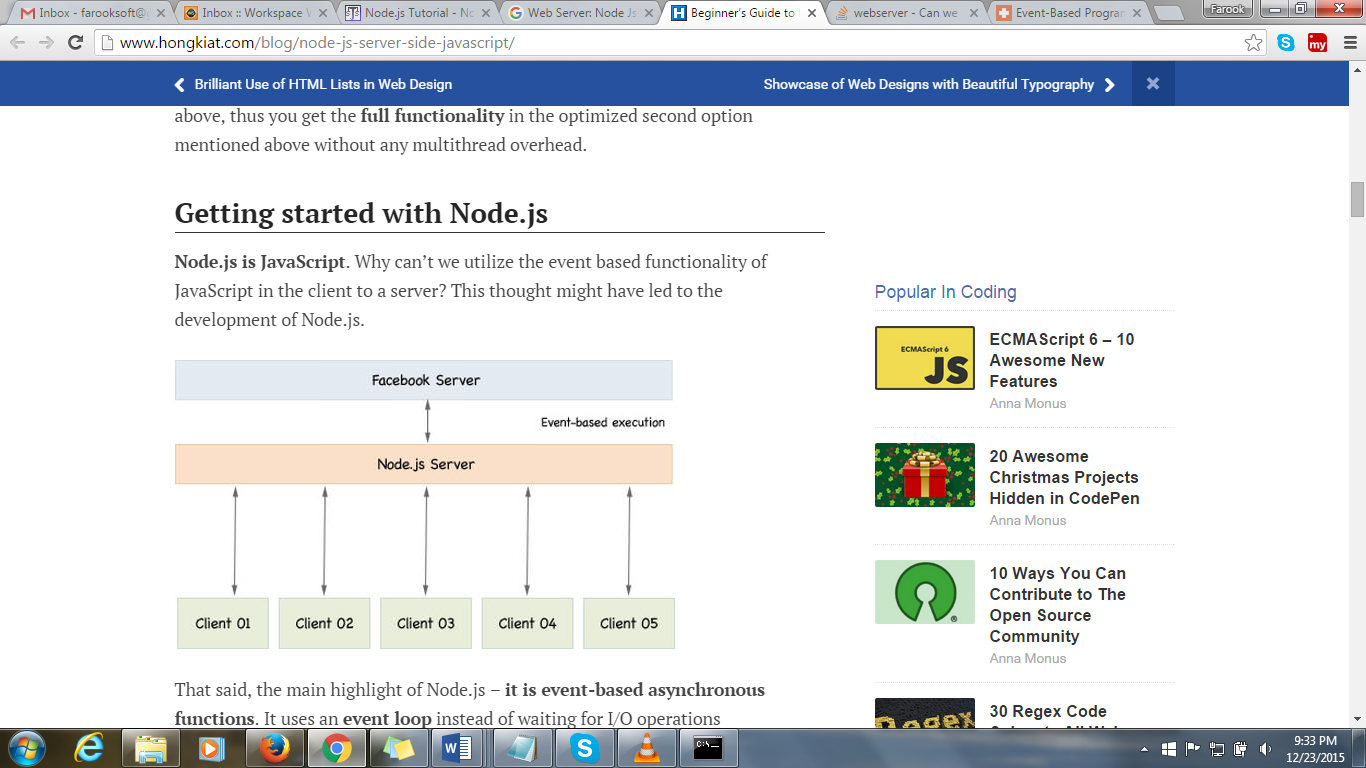
**Multi-threaded execution**

In the first method even though the code is simple the execution pauses for a while at the point where the slow web server is accessed. The second method is more optimized in case of performance but it’s hard to code and it has a multithread management overhead. The case is similar for most of the web programming languages other than the server-side JavaScript, i.e., Node.js.



**What’s the difference in Node.js?**

In order to understand Node.js you must keep in mind the JavaScript’s event based programming in the browser. We utilize the same technology here. Instead of using a separate thread, a function is attached to the finish event of the “slow web server access” mentioned above, thus you get the full functionality in the optimized second option mentioned above without any multithread overhead.



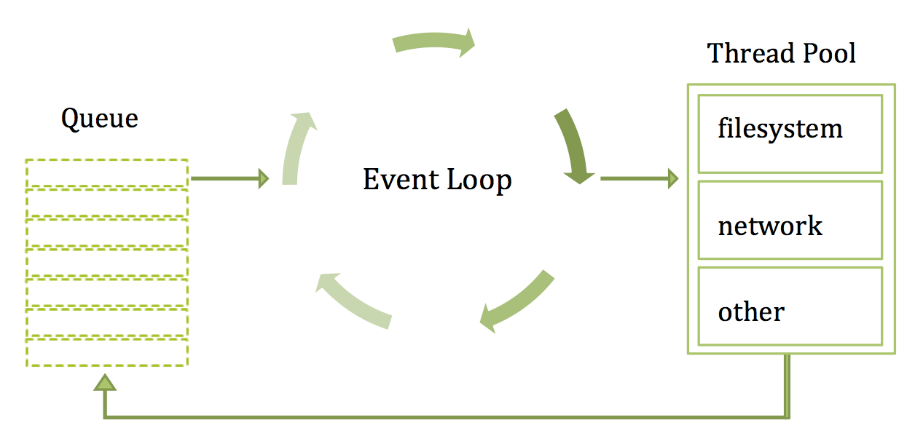
**How Does Node Differ From Other Frameworks?**

The major difference between Node and other server-side technologies is Node’s use of a single thread and asynchronous architecture. Many other server-side technologies are multi-threaded and synchronous, meaning that threads can be blocked while waiting for replies from the database. Each request creates a new thread from a limited pool based on system RAM usage. Node’s asynchronous design allows it to handle a large number of concurrent connections with high throughput on a single-thread, which makes it highly scalable.

Node is not meant as a replacement for other technology stacks, but it can provide scalability and increased performance to applications which fit its purpose. Some examples of application types which can benefit from using Node are REST APIs, Chat applications and Real-Time Tracking applications (Brokerage trading dashboards, real-time user statistics, etc.)

Node’s homepage describes it as “lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.” The applications above fit that description well and can take advantage of Node’s features.

**How does Node.js work asynchronously without multithreading?**



* The main event loop is single-threaded by nature. But most of the I/O (network, disk, etc) is run on separate threads, because the I/O APIs in Node.js are asynchronous/non-blocking by design, in order to accommodate the event loop.
* One of the biggest pluses about Node is the very fact that more often than not all the code the developer writes is running on one thread in an event loop – that’s the rub – you don’t have to deal with a multi-threaded environment because it’s abstracted away.
* As long as you don’t do computationally heavy work on the main Javascript thread, then you are golden, because the slow I/O stuff is taken care of for you by the asynchronous non-blocking I/O libraries. If you do need to do computationally heavy work (CPU bound work), don’t do it on the main thread, instead fork a child\_process and run it on a separate thread than the main Javascript event loop thread.
* So yes, you can’t get away from multithreading, even in Node.js. However, with the Node.js event loop, you can truly avoid many of the pitfalls of more traditional multithreaded servers and programs.
* Let me give you an analogy. Client side Javascript has no traditional I/O. Node.js was created in the first place because Javascript had no existing I/O libraries so they could start clean with non-blocking I/O. For the client, I/O takes the form of AJAX requests. AJAX is by default asynchronous. If AJAX were synchronous then it would lock up the front-end. By extension, the same thing is true with Node.js. If I/O in Node was synchronous/blocking then it would lock up the event-loop. Instead asynchronous/non-blocking i/o APIs in Node allow Node to utilize background threads that do the I/O work behind the scenes, thus allowing the event loop to continue ticking around and around freely, about once every 20 milliseconds.

**NPM: The Node Package Manager**

The NPM tool comes by default with every Node.js installation.

Some of the most popular NPM modules today are:

* **express -** Express.js, a Sinatra-inspired web development framework for Node.js, and the de-facto standard for the majority of Node.js applications out there today.
* **connect** - Connect is an extensible HTTP server framework for Node.js, providing a collection of high performance “plugins” known as middleware; serves as a base foundation for Express.
* **socket.io and sockjs -** Server-side component of the two most common websockets components out there today.
* **Jade -** One of the popular templating engines, inspired by HAML, a default in Express.js.
* **mongo and mongojs -** MongoDB wrappers to provide the API for MongoDB object databases in Node.js.
* **redis -** Redis client library.
* **coffee-script** - CoffeeScript compiler that allows developers to write their Node.js programs using Coffee.
* **underscore (lodash, lazy) -** The most popular utility library in JavaScript, packaged to be used with Node.js, as well as its two counterparts, which promise better performance by taking a slightly different implementation approach.
* **forever -** Probably the most common utility for ensuring that a given node script runs continuously. Keeps your Node.js process up in production in the face of any unexpected failures.

**Day 2**

**Creating Node.js Application**

A Node.js application consists of following three important parts −

1. **Import required modules** − we use require directive to load a Node.js module.
2. **Create server** − A server which will listen to client's request similar to Apache HTTP Server.
3. **Read request and return response** −
4. server created in earlier step will read HTTP request made by client which can be a browser or console and return the response.

**Step 1 - Import required module**

We use require directive to load http module and store returned HTTP instance into http variable as follows −

|  |
| --- |
| var http = require("http"); |

**Step 2: Create Server**

At next step we created http instance and call http.createServer () method to create server instance and then we bind it at port 8080 using listen method associated with server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

|  |
| --- |
| 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://127.0.0.1:8080/'); |

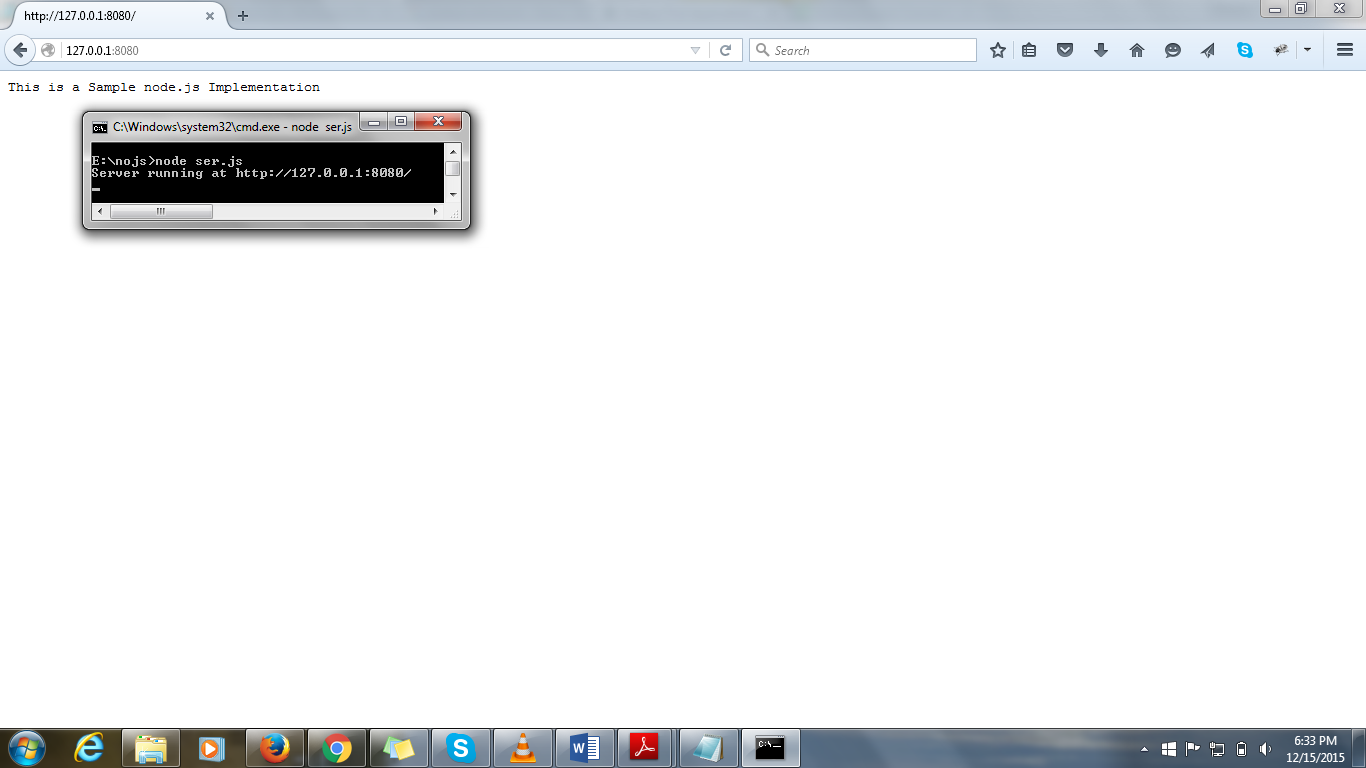
Above code is enough to create an HTTP server which listens i.e. wait for a request over 8080 port on local machine.

**Step 3: Testing Request & Response**

Let's put step 1 and 2 together in a file called ser.js and start our HTTP server as shown below

|  |
| --- |
| 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  response.end('This is a Sample node.js Implementation \n');}).listen(8080);  // Console will print the message  console.log('Server running at http://127.0.0.1:8080/'); |

Now execute the ser.js to start the server as follows −



**REPL**

REPL stands for Read Eval Print Loop. Node.js or Node comes bundled with a REPL environment. It performs the following desired tasks.

|  |  |
| --- | --- |
| Read | Reads user's input, parse the input into JavaScript data-structure and stores in memory. |
| Eval | Takes and evaluates the data structure |
| Print | Prints the result |
| Loop | Loops the above command until user press ctrl-c twice. |

REPL feature of Node is very much useful in experimenting with Node.js codes and to debug JavaScript codes.

|  |  |
| --- | --- |
| **REPL Commands** | |
| **ctrl + c** | Terminate the current command. |
| **ctrl + c twice** | terminate the Node REPL. |
| **ctrl + d** | terminate the Node REPL. |
| **Up/Down Keys** | See command history and modify previous commands. |
| **tab Keys** | List of current commands. |
| **.help** | List of all commands. |
| **.break** | Exit from multiline expression. |
| **.clear** | exit from multiline expression |
| **.save filename** | Save current Node REPL session to a file. |
| **.load filename** | Load file content in current Node REPL session. |

**console**

* Node.js has global variable console.
* console.log function can output string to the console window or debug window from browser.
* console.warn(msg) prints on stderr.
* console.time(label) marks a time stamp and console.timeEnd(label) prints out the elapsed time since the time function was called.
* console.assert(cond, message) throws an AssertionFailure exception if cond evaluates to false.

**global**

* The variable global is our handle to the global namespace in Node.js.
* All the true globals we have seen, such as console, setTimeout, and process, are members of the global variable.
* We can even add members to the global variable to make it available everywhere.

|  |
| --- |
| **Example:**  console.log(console === global.console); // true  console.log(setTimeout === global.setTimeout); // true  console.log(process === global.process); // true  **Output:**  true  true  true |

* Usage of global variables across javascript file.

|  |
| --- |
| **Example:**  **//GlobalJS.js**  global.id = 1987;  **//Sample.js**  require('./GlobalJS');  console.log(id);  **Output:**  1987 |

**a. \_\_filename and \_\_dirname**

* These variables are available in each file and give you the full path to the file and directory for the current module.
* They include everything right up to the root of the current drive this file resides on.
* The following code shows values.

|  |
| --- |
| **Example:**  console.log(\_\_dirname);  console.log(\_\_filename);  **Output:**  D:\Node\_Pjt  D:\Node\_Pjt\sample.js |

**process**

* process is one of the most important globals provided by Node.js.
* It has useful member functions and properties.
* It is a source of a few critical events.

**a. Command Line Arguments**

* We use the process object to access the command line arguments.
* The arguments are available as the process.argv member property, which is an array.
* The first element is the node executable, and the second element is the name of the JavaScript file passed into Node.js to start the process, and the remaining elements are the command line arguments.

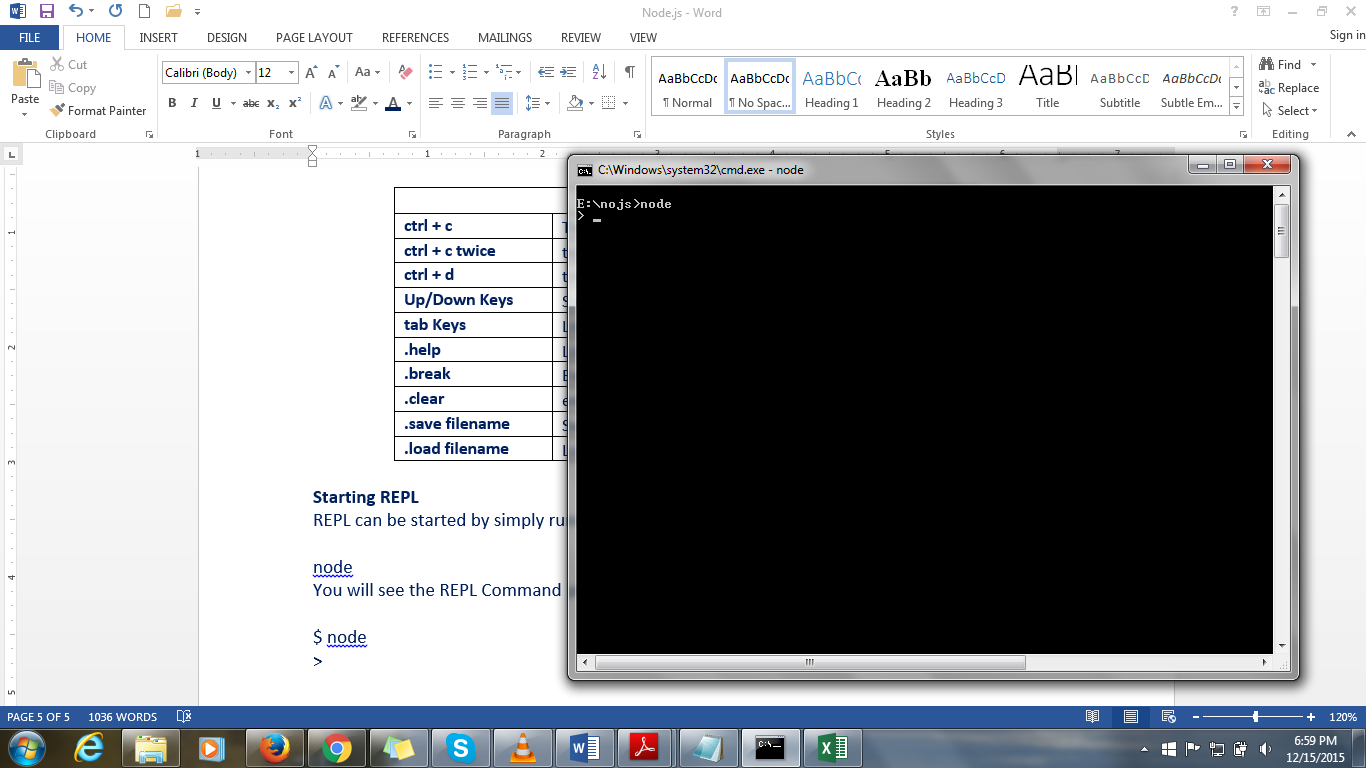
**b. process.nextTick**

* process.nextTick is a simple function that takes a callback function.
* It is used to put the callback into the next cycle of the Node.js event loop.
* It is designed to be highly efficient, and it is used by a number of Node.js core libraries.

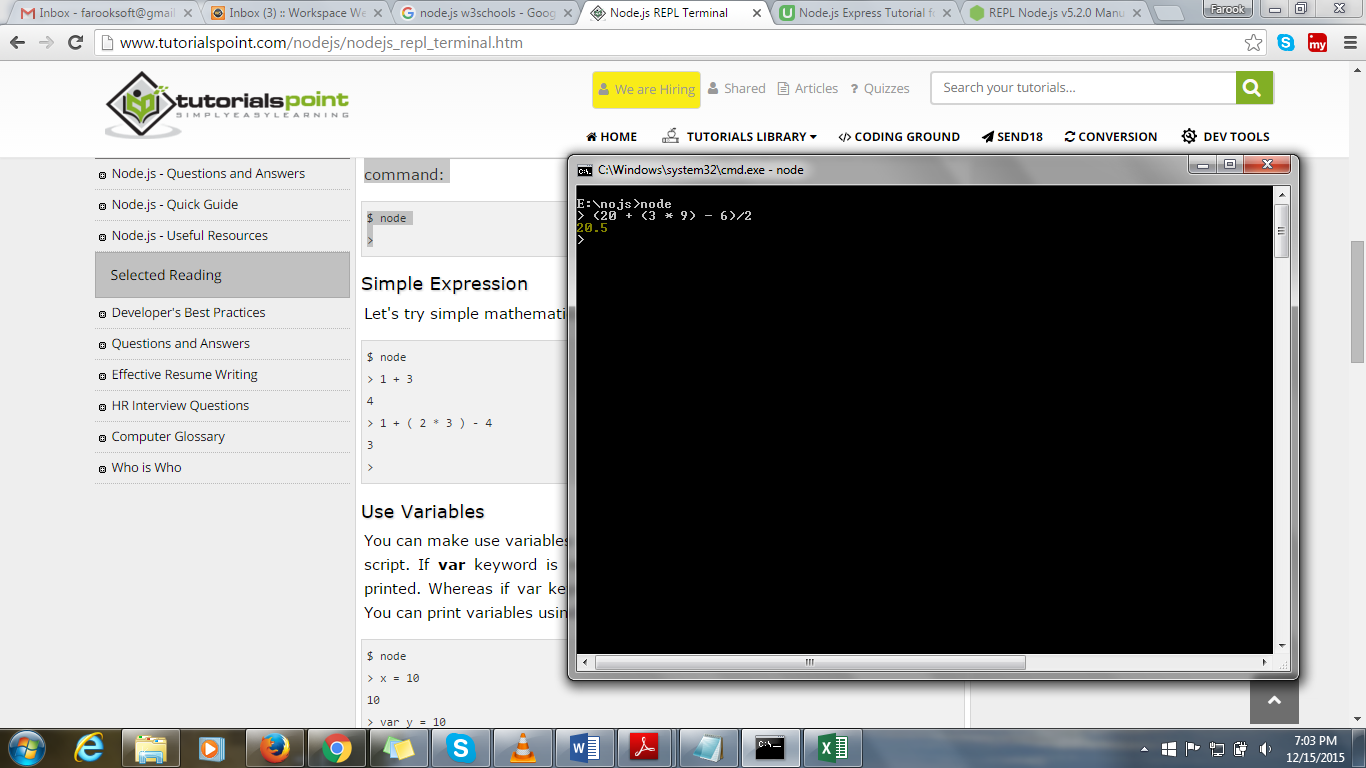
|  |
| --- |
| **Example:**  //argv  console.info(process.argv);  //process.nextTick  process.nextTick(**function**( ){  console.log('next tick');  });  console.info("Right here");  **Output:**  [ 'node', 'D:\\Node\_Pjt\\sample.js' ]  Right here  next tick |

**Starting REPL**

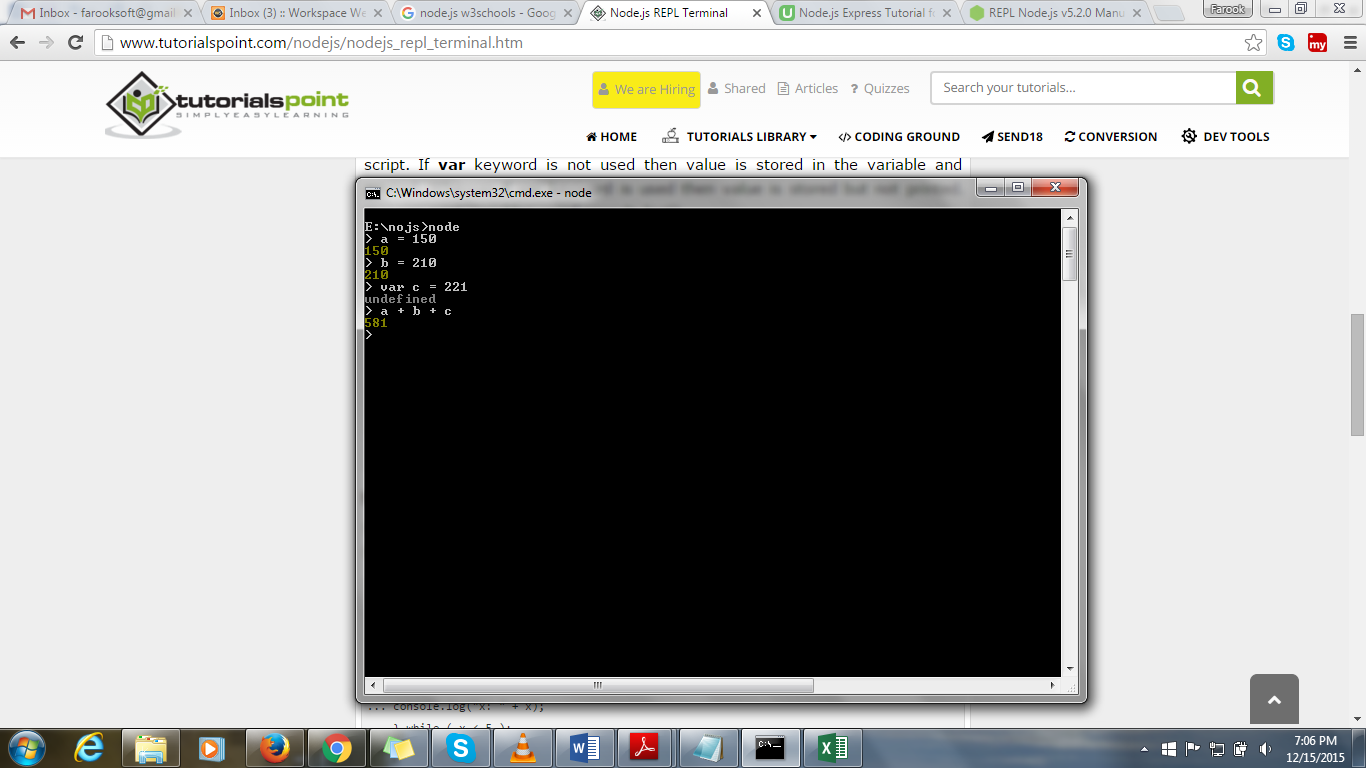
REPL can be started by simply running node on shell/console without any argument and you will see the REPL Command prompt > where you can type any Node.js command:



**Example 1:**

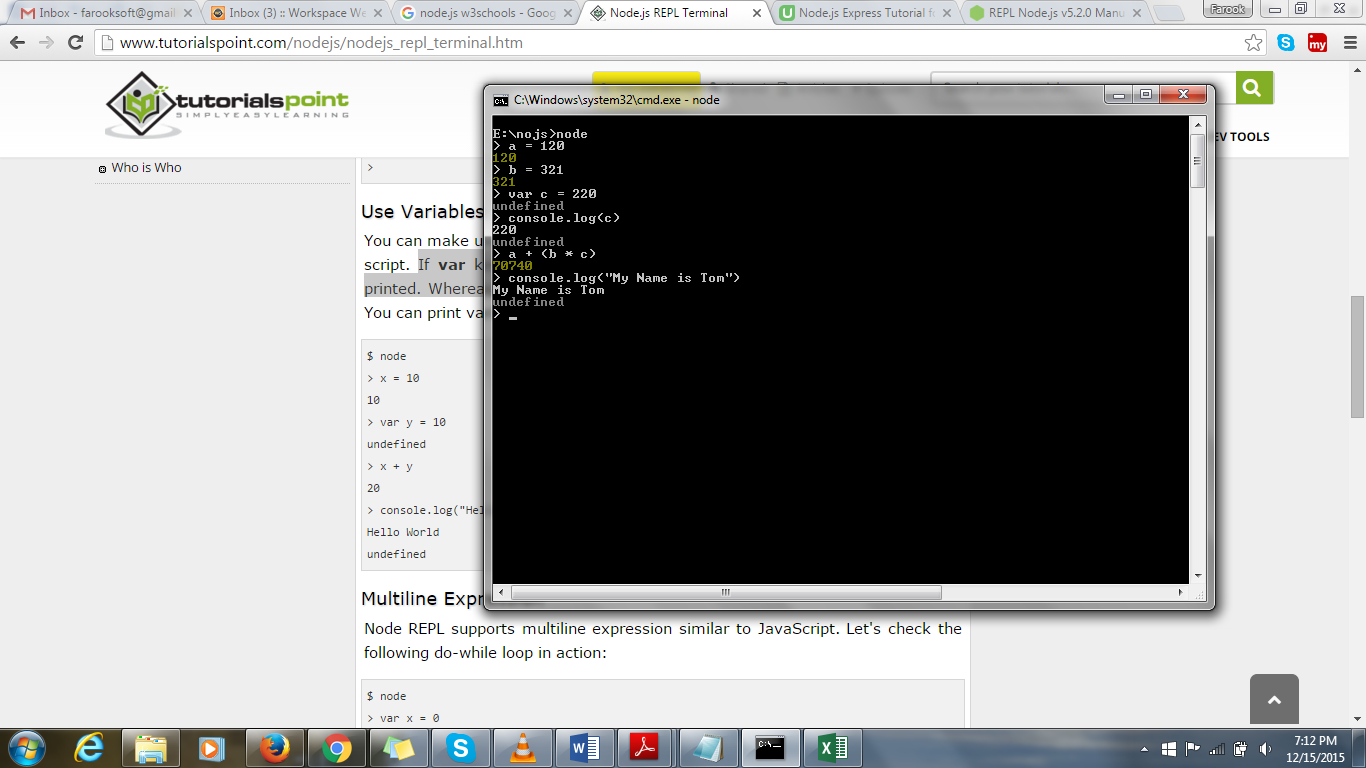


**Example 2:**



**Example 3:**

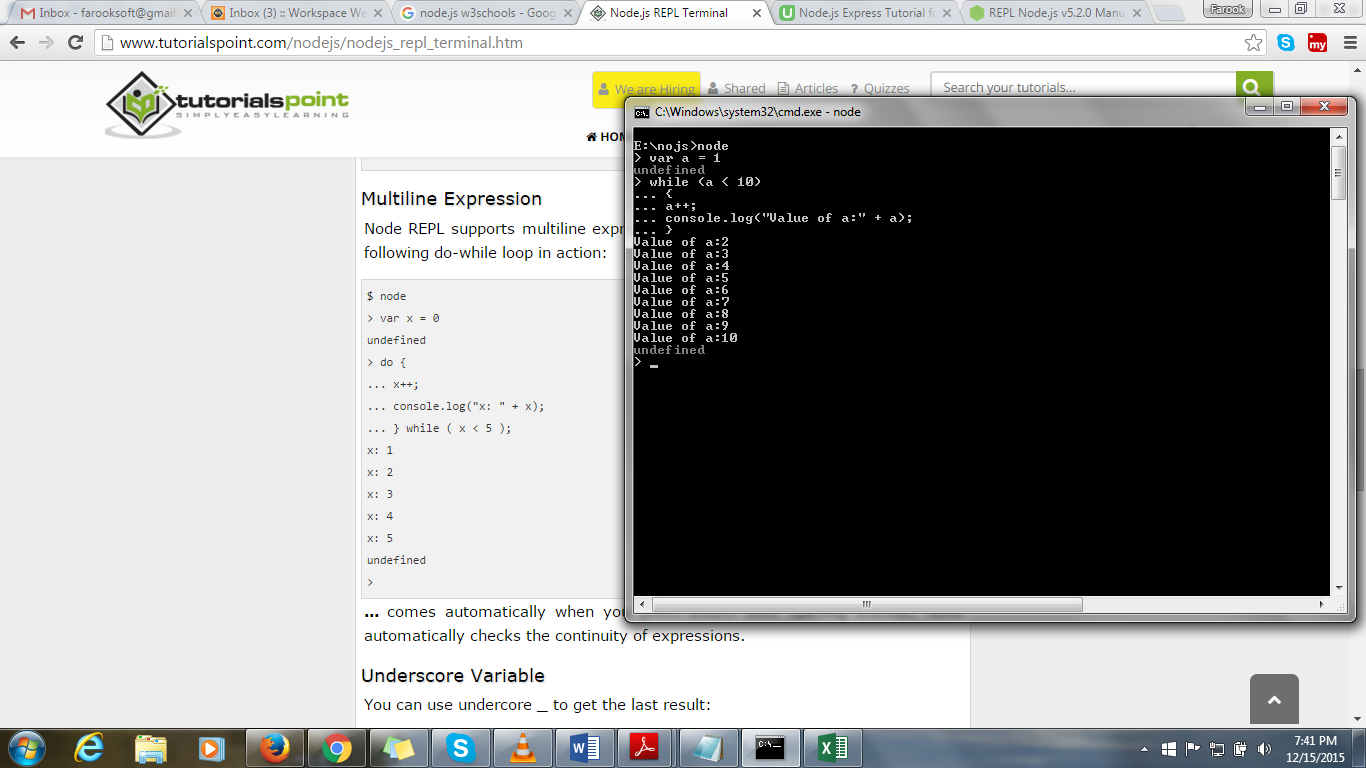
If var keyword is not used then value is stored in the variable and printed, whereas if var keyword is used then value is stored but not printed. You can print variables using console.log() as in the pic below.



**Example 4:**

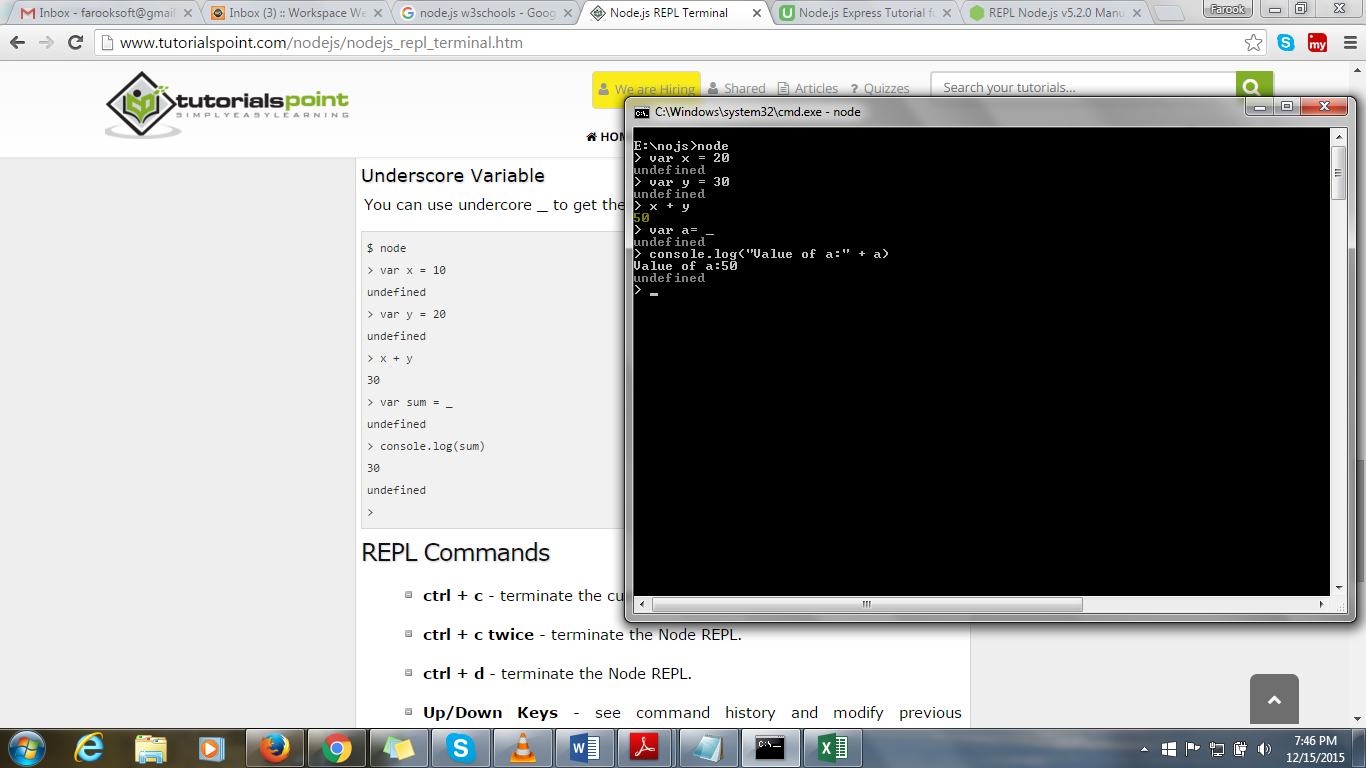
Node REPL supports multiline expression similar to JavaScript.

... comes automatically when you press enters after opening bracket. Node automatically checks the continuity of expressions.



**Example 5:**

Underscore \_ is used to get previous or last result.



**Node.js vs Traditional Web Server**

* Node.js is focused on creating highly performant applications.
* Most web applications depend on reading data from disk or from another network source.
* Traditional Web Servers uses a Process per Request.
* Traditional servers used to spin up a new process to handle every single web request.
* Spinning a new process for each request is an expensive operation, both in terms of CPU and memory.
* Traditional Web Servers Using a Thread Pool

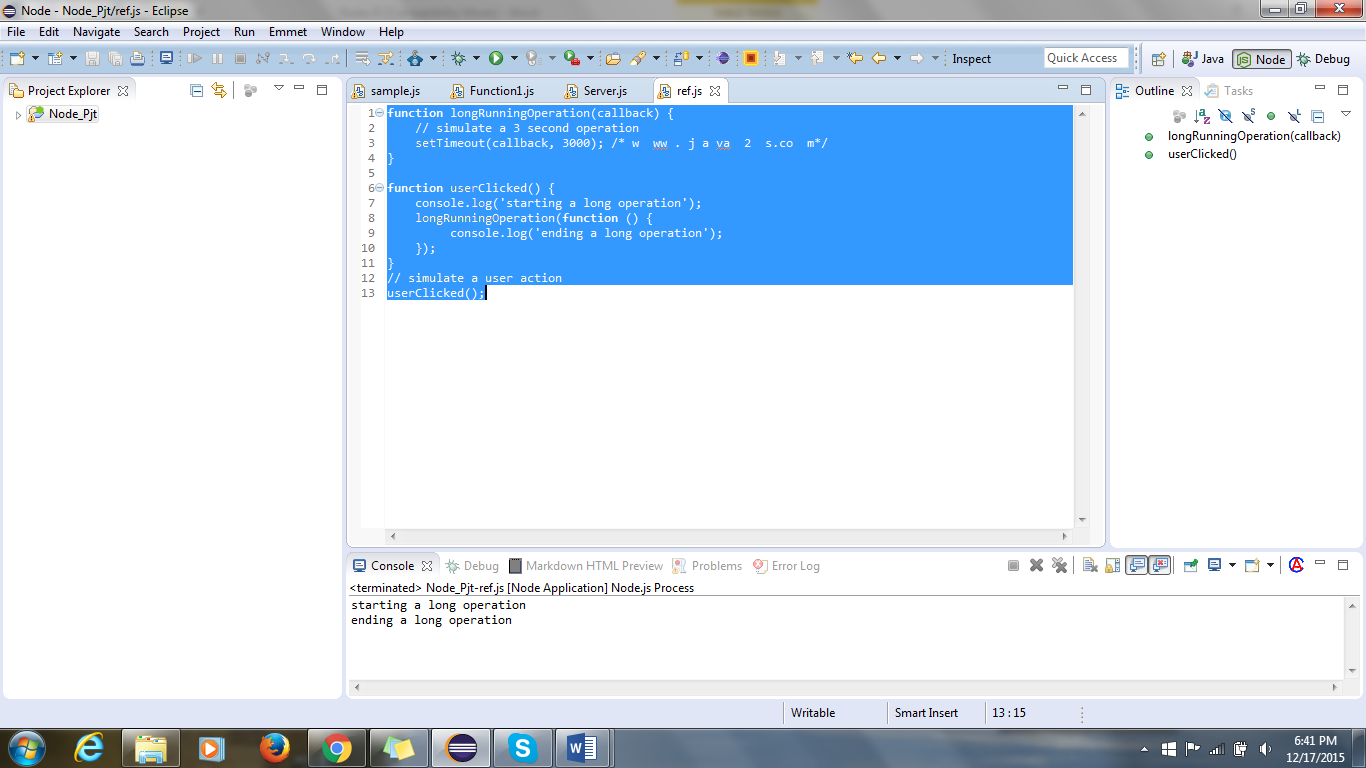
**Example**

Node.js uses a single thread to handle requests.

|  |
| --- |
| function longRunningOperation(callback) {  // simulate a 3 second operation  setTimeout(callback, 3000);  }  function userClicked() {  console.log('starting a long operation');  longRunningOperation(function () {  console.log('ending a long operation');  });  }  // simulate a user action  userClicked(); |

**Output:**

In the console view the second line gets executed just after 3 seconds.



**Data Type**

**1. Node.js Data Type**

* Node.js is highly performant, and it uses JavaScript because JavaScript supports first-class functions and closures.
* Node.js has a few core types: number, boolean, string, and object.
* The value undefined means that a value has not been set yet or simply does not exist.

|  |
| --- |
| For Example:  var x;  console.log(x);  The above code when executed results in: |

* **Null:** null is an explicit assertion that there "is no value":

|  |
| --- |
| For Example:  var y;  console.log(y);  y = null ;  console.log(y);  The above code when executed results in:  undefined  null |

* **typeof:** To see the type of anything in JavaScript, use the typeof operator:

|  |
| --- |
| **For Example:**  console.log(**typeof** 10);  console.log(**typeof** "SMI");  console.log(**typeof** **function** () { **var** x = 20; });  The above code when executed results in:  number  string  function |

* **Constants:** The standard practice is to use uppercase letters and variable declarations:

|  |
| --- |
| For Example:  **var** SECONDS\_PER\_DAY = 86400;  **var** PI=3.14159;  console.log("Total Number of Seconds per Day:" +SECONDS\_PER\_DAY);  console.log("The Value of PI:" +PI);  The above code when executed results in:  Total Number of Seconds per Day:86400  The Value of PI:3.14159 |

* **Type Comparisons and Conversions**
  + JavaScript has both the equality operator == and the precise equality operator ===.
  + A number of different values evaluate to false.

|  |
| --- |
| For Example:  console.log(234 == '234');  console.log(234 === '234');  console.log(234234.235235 == 'SMI');  console.log("smi" == "SMI");  console.log("smi".toUpperCase() == "SMI");  The above code when executed results in:  true  false  false  false  true |

* + A number of different values evaluate to false

|  |
| --- |
| For Example:  console.log('' == **false** == **null** == **undefined** == 0);  console.log(**null** === **undefined**);  The above code when executed results in:  true  false |

* + The primitive wrapper.

|  |
| --- |
| For Example:  **var** x = 234;  **var** x1 = **new** Number(234);  console.log(**typeof** x);  console.log(**typeof** x);  console.log(x1 == x);  console.log(x1 === x);  The above code when executed results in:  number  number  true  false |

**2. Node.js Numbers**

* All numbers in JavaScript are 64-bit IEEE 754 double-precision floating-point numbers.
* All numbers in JavaScript have the same floating point number type.
* Arithmetic operations (+,-,\*, /, %) work on numbers as you would expect.

|  |
| --- |
| For Example:  **var** myData = 1;  **var** myValue = 2;  console.log(myData + 1);  console.log(myData / myValue);  console.log(myData \* myValue);  console.log(myData - myValue);  console.log(myData % 2);  The above code when executed results in:  2  0.5  2  -1  1 |

* The number type in JavaScript behaves much like integer data types in other languages
* The tricky part of using the number type, however, is that for many numeric values, it is an approximation of the actual number.
* When performing floating-point mathematical operations, we cannot manipulate arbitrary real numbers and expect an exact value.

|  |
| --- |
| Example for above points:  console.log(1024 \* 1024);// integer datatype  console.log(0.1 + 0.2); //produces exact value  console.log(10.1 + 09.0);  console.log(1 - 0.3 + 0.1 == 0.8);//performing floating point computation provides exact values  The above code when executed results in:  1048576  0.30000000000000004  19.1  false |

* In JavaScript dividing a number by zero returns the value Infinity or -Infinity instead of generating a runtime exception.
* Infinity and -Infinity are valid values that we can compare against in JavaScript.

|  |
| --- |
| For Example:  console.log(67/0);  console.log(-67/0);  console.log(67/3);  **var** x = 10, y = 0;  console.log("Divide by Zero :"+ (x / y == Infinity));  The above code when executed results in:  Infinity  -Infinity  22.333333333333332  Divide by Zero :true |

* You can use the functions parseInt and parseFloat to convert strings to numbers.

|  |
| --- |
| For Example:  console.log(parseInt("32"));  console.log(parseFloat("8.24"));  console.log(parseInt("234.12345"));  console.log(parseFloat("10"));  The above code when executed results in:  32  8.24  234  10 |

* If we provide these functions with not-parsable value, they return the special value NaN.

|  |
| --- |
| For Example:  console.log(parseInt("SMI"));  console.log(parseFloat("SMI"));  console.log("To check if NAN:" +isNaN(parseInt("smi")));  The above code when executed results in:  NaN  NaN  To check if NAN:true |

* To test whether a given number is a valid finite number, use the isFinite function.

|  |
| --- |
| For Example:  console.log(isFinite(10/5));  console.log(isFinite(10/0));  console.log(isFinite(parseFloat("smi")));  The above code when executed results in:  true  false  false |

**3. Node.js Booleans**

* The boolean type Values in JavaScript can either be true or false.
* Two literals are defined for boolean values: true and false.
* We can convert values to boolean with the Boolean function, and the language converts everything to boolean when needed, according to the following rules:
* false, 0, empty strings "", NaN, null, and undefined all evaluate to false.
* All other values evaluate to true.

|  |
| --- |
| **For Example:**  console.log(0 == **false**);  console.log("" == **false**);  **if**(**null**){  }**else**{  console.log("false");  }  **if**(**undefined**){  }**else**{  console.log("false");  }  **if**(NaN){  }**else**{  console.log("false");  }  **The above code when executed results in:**  true  true  false  false  false |

* We can assign these to variables and apply boolean operations to them.

|  |
| --- |
| **For Example:**  **var** myData = **true**;  console.log(myData); // true  // Boolean operations (&&, ||, !) work as expected:  console.log(**true** && **true**); // true  console.log(**true** && **false**); // false  console.log(**true** || **false**); // true  console.log(**false** || **false**); // false  console.log(!**true**); // false  console.log(!**false**); // true  **The above code when executed results in:**  true  true  false  true  false  false  true |

**4. Node.js Strings**

* Strings in JavaScript are sequences of Unicode characters.
* We use a string of length 1 to represent character.
* Strings can be wrapped in single or double quotation marks.
* They are functionally equivalent.
* To include a single quotation mark inside a single-quoted string, we can use \', and similarly for double quotation marks inside double-quoted strings, we can use \":

|  |
| --- |
| **For Example:**  console.log('Sri Mookambika Infosolutions Pvt Ltd..')  console.log("\"Hey, This is situated !\", in Madurai.")  **The above code when executed results in:**  Sri Mookambika Infosolutions Pvt Ltd..  "Hey, This is situated !", in Madurai. |

* To get the length of a string in JavaScript, just use the length property.
* To add two strings together, you can use the + operator.
* Javascript + can convert them as best it can.

|  |
| --- |
| **For Example:**  **var** text = "SMI"; //finding length of string  console.log(text.length);  **var** string = "SMI" + " is " + "cool";//Adding strings  console.log(string);  **var** pincode = 625014; //using + symbol  **var** string = "My Office Pincode: " + pincode + ".";  console.log(string);  **The above code when executed results in:**  3  SMI is cool  My Office Pincode: 625014. |

**5. Node.js String Functions**

**indexOf:** To find a string with another string, use the indexOf function.

|  |
| --- |
| **For Example:**  **var** i = "Sri Mookambika Infosolutions".indexOf("ik");  console.log("Index value: " + i);  **The above code when executed results in:**  Index value: 11 |

**substr and splice:** To extract a substring from a string, use the substr or splice function.

* substr takes the starting index and length of string to extract.
* splice takes the starting index and ending index:

|  |
| --- |
| **For Example:**  **var** s = "Sri Mookambika Infosolutions".substr(5, 15);  **var** s1 = "Sri Mookambika Infosolutions".slice(5, 15);  console.log("Sub String:\t"+s);  console.log("Slice:\t\t" +s1);  **The above code when executed results in:**  Sub String: ookambika Infos  Slice: ookambika |

**Split**

* To split string into substrings, use the split function and get an array as the result
* The trim function from V8 Javascript function removes whitespace from the beginning and end of a string.

|  |
| --- |
| **For Example:**  **var** s = "S>r>i M>o>o>k>a>m>b>i>k>a Infosolutions".split(">");  **var** s1 = " Sri Mookambika Infosolutions".trim();  console.log("Split String:\t"+s);  console.log("Trim Function:\t"+s1);  **The above code when executed results in:**  Split String: S,r,i M,o,o,k,a,m,b,i,k,a Infosolutions  Trim Function: Sri Mookambika Infosolutions |

**Language**

**1. Operator**

* The ternary operator

|  |
| --- |
| **Example:**  **var** isChocolate = **true**;  **var** snack = isChocolate ? "snicker" : "Hotdog";  console.log("Result\t:"+ snack);  **Output:**  Result :snicker |

* **Bitwise operations**

Bitwise operations are supported in JavaScript: & (and), | (or), ~(inverse), and ^ (xor) operators.

2. **Loop Statement**

* JavaScript also supports while , do...while , and for loops.
* for...in loop are also supported in Node.js.
* The following code shows that we can get the names of all the keys on an object.

|  |
| --- |
| **Example:**  **var** employee = {  first\_name: "Scooby",  last\_name: "Doo",  age: 34,  EmailId: "scooby@gmail.com"  };  console.log(employee.age);  **for** (key **in** employee) {  console.log("Employee Data\t:" +key);  }  **Output:**  34  Employee Data :first\_name  Employee Data :last\_name  Employee Data :age  Employee Data :EmailId |

**Objects**

**1. Class**

Javascript classes are all declared as functions:

|  |
| --- |
| **Example:**  **function** Shape () {  **this**.X = 0;  **this**.Y = 0;  **this**.move = **function** (x, y) {  **this**.X = x;  **this**.Y = y;  }  **this**.distance\_from\_origin = **function** () {  **return** Math.sqrt(**this**.X\***this**.X + **this**.Y\***this**.Y);  }  }  **var** s = **new** Shape();  console.log(s.move(19, 10));  console.log("Square Root\t:"+s.distance\_from\_origin());  **Output:**  undefined  Square Root :21.470910553583888 |

* You can add as many properties and methods to your classes, at any time:

var s = new Shape(15, 35);

s.FillColour = "red";

* The function that declares the class is its constructor!

**2. Class Creation**

Functions that return objects are a great way to create similar objects.

|  |
| --- |
| **Example:**  **function** Message() {  **var** message = 'Its a Sample Program';    **function** setMessage(newMessage) {  **if** (!newMessage)  **throw** **new** Error('cannot set empty message');  message = newMessage;  }    **function** getMessage() {  **return** message;  }    **function** printMessage() {  console.log(message);  }  **return** {  setMessage: setMessage,  getMessage: getMessage,  printMessage: printMessage  };  }  // Pattern in use  **var** hi1 = Message();  hi1.printMessage(); // Its a Sample Program  **var** hi2 = Message();  hi2.setMessage('Hello World');  hi2.printMessage(); // Hello World  hi1.printMessage(); // Its a Sample Program  **Output:**  Its a Sample Program  Hello World  Its a Sample Program |

**a. this Keyword:**

* Javascript this object behaves differently depending upon how we call it.
* this object refers to the calling context.
* The calling context is the prefix used to call a function.

|  |
| --- |
| **Example:**  **var** employee = {  name : "Tom",  age : 23,  Emp\_details: **function**(){  console.log("Emp Name:",**this**.name);  }  }  console.log("Emp Age:",employee.age);  employee.Emp\_details();  **Output:**  Emp Age: 23  Emp Name: Tom |

* The default calling context is the Node.js global variable.

function myData() {

console.log('is this called from globals? : ', this === global); // true

}

myData();

* We can attach a function to any object and change the calling context.

|  |
| --- |
| **Example:**  **var** data = {  value: 123  };  **function** fun() {  **if** (**this** === global)  console.log('called from global');  **if** (**this** === data)  console.log('called from data');  }  **function** funData() {  **this**.funData = 1987;  console.log('Is this global?: ', **this** == global);  }  // global context  fun(); // called from global  // from data  data.fun = fun;  data.fun(); // called from data  // without the new operator  funData(); // Is this global?: true  console.log(global.funData); // 1987  // with the new operator  **var** newFoo = **new** funData(); // Is this global?: false  console.log(newFoo.funData); // 1987  **Output:**  called from global  called from data  Is this global?: true  1987  Is this global?: false  1987 |

**b. Understanding Prototype**

* Every object in JavaScript has an internal link to another object called the prototype.
* When you read a property on an object, myData.myValue reads the property myValue from myData, JavaScript checks that this property exists on myData.
* If not, JavaScript checks if the property exists on myData. \_\_proto\_\_ and so on till \_\_proto\_\_ itself is not present.
* If a value is found at any level, it is returned.
* Otherwise, JavaScript returns undefined.
* Prototypes are shared between all the objects created from the same function.

|  |
| --- |
| **Example:**  **function** employee() { };  employee.prototype.Emp\_id = 143;  employee.prototype.name="Tom";  **var** bas = **new** employee();  **var** info = **new** employee();  console.log(bas.Emp\_id+"\t"+ bas.name); // 143 Tom  console.log(info.Emp\_id +"\t"+ info.name); // 143 Tom  console.log(info.name +"\t"+ bas.name);//Tom Tom  employee.prototype.Emp\_id = 456;  console.log(bas.Emp\_id +"\t"+ info.Emp\_id); // 456  **Output:**  143 Tom  143 Tom  Tom Tom  456 456 |

* Suppose we have 1,000 instances created of a certain object. All the properties and functions on prototype is shared.
* Therefore prototype saves memory.
* A prototype property is shadowed by a property on an object.
* this object is a perfect candidate for read/write properties(data) and you should use it for all properties(data).
* But functions are generally not altered after creation. So functions are great candidates to put on .prototype.
* functionality (functions/methods) is shared between all instances, and properties belong on individual objects.

|  |
| --- |
| **Example:**  **function** employee() { };  employee.prototype.emp\_id = 123;  employee.prototype.emp\_Name="Tom";  **var** data1 = **new** employee();  **var** data2 = **new** employee();  data1.emp\_id = 456;  console.log(data1.emp\_id + "\t"+ data2.emp\_Name);  data1.emp\_Name = "Jerry";  console.log(data2.emp\_id +"\t" +data1.emp\_Name); // 123  **Output:**  456 Tom  123 Jerry |

|  |
| --- |
| **Example: A pattern to write a class in JavaScript.**  **function** Test() {  // Properties go here  **this**.testProp = "%'Some Initial Data'%";  }  // Member functions go here:  Test.prototype.memberFun = **function** () {  **this**.testProp = "'modified value'";  console.log("called from prototype");  }  // Creation  **var** instance = **new** Test();  // Usage  console.log(instance.testProp); // some initial value  instance.memberFun();  console.log(instance.testProp); // modified value  **Output:**  %'Some Initial Data'%  called from prototype  'modified value' |

* Within the member function, we can get access to the current instance using this even though the same function body is shared between all instances.
* Most classes inside core Node.js are written using this pattern.

**3. Globals**

Node.js has a few key global variables that are always available to you.

**a. global**

* JavaScript in web browsers have the window object as a "global" variable.
* Any variables or members attached to global are available anywhere in the application.
* Node.js has the global object.
* Anything attached to it is available anywhere in your node application:

|  |
| --- |
| **Example:**  **function** printInitial(var\_name) {  console.log(global[var\_name]);  }  global.Tom = "J";  global.Scooby = "A";  printInitial("Tom");  printInitial("Henry");  printInitial("Scooby");  printInitial("Flinstone");  **Output:**  J  undefined  A  undefined |

**4. Errors and Exceptions**

* Error handling is an important part of any application.
* To throw an exception, use the throw JavaScript keyword.
* In JavaScript, you traditionally signal errors using an Error object and a message.
* You throw this error, to signal the error condition:

|  |
| --- |
| **Example:**  **function** a() {  **throw** **new** Error("Something bad happened!");  }  a();  **Result:**  D:\Node\_Pjt\sample.js:3  throw new Error("Something bad happened!");  ^  Error: Something bad happened!  at a (D:\Node\_Pjt\sample.js:3:11)  at Object.<anonymous> (D:\Node\_Pjt\sample.js:5:1)  at Module.\_compile (module.js:456:26)  at Object.Module.\_extensions..js (module.js:474:10)  at Module.load (module.js:356:32) |

**a. Catch Exception**

You can catch it with a try / catch block as seen in other languages:

|  |
| --- |
| **Example:**  **function** exceptionHandler () {  **throw** **new** Error("Something bad happened!");  }  **try** {  exceptionHandler();  } **catch** (e) {  console.log("I caught an error: " + e.message);  }  console.log("program is still running");  **Output:**  I caught an error: Something bad happened!  program is still running |

**b. finally**

* To catch an exception, you can use the catch keyword.
* For code to run regardless of whether an exception was caught or not, you can use the finally keyword.
* The following code shows a simple example that demonstrates this.
* The catch section executes only if an error is thrown.
* The finally section executes despite any errors thrown within the try section.

|  |
| --- |
| **Example:**  **try** {  console.log('About to throw an error');  **throw** **new** Error('Error thrown');  } **catch** (e) {  console.log('I will only execute if an error is thrown');  console.log('Error caught: ', e.message);  } **finally** {  console.log('I will execute irrespective of an error thrown');  }  **Output:**  About to throw an error  I will only execute if an error is thrown  Error caught: Error thrown  I will execute irrespective of an error thrown |

* This method of exception handling is great for synchronous JavaScript.
* For asynchronous JavaScript we should handle the error inside the callback.

|  |
| --- |
| **Example:**  setTimeout(**function** () {  **try** {  console.log('About to throw an error');  **throw** **new** Error('Error thrown');  } **catch** (e) {  console.log('Error caught!');  console.log(e.message);  }  }, 1000);  **Output:**  About to throw an error  Error caught!  Error thrown |

5. **prototype chain**

* The util core module (require('utils')) provides a function to create the prototype chain.
* The function is called inherits and takes a child class followed by parent class.

|  |
| --- |
| **Example:**  **var** inherits = require('util').inherits;  **function** Car(n){  **this**.name = n;  }  Car.prototype.drive= **function** (destination) {  console.log(**this**.name, 'can drive to', destination);  }  **function** FlyingCar(name) {  // Call parent constructor  Car.call(**this**, name);  // Additional construction code  }  inherits(FlyingCar, Car);  // Additional member functions  FlyingCar.prototype.fly = **function** (destination) {  console.log(**this**.name, 'can fly to', destination);  }  **var** bird = **new** FlyingCar('Tom');  bird.drive('New York');  bird.fly('London');  **Output:**  Tom can drive to New York  Tom can fly to London |

a. **Overriding Functions in Child Classes**

* To override parent functions but still utilize some of the original functionality, simply do the following:
* Create a function on the child prototype with the same name.
* Call the parent function similar to the way we called the parent constructor, basically using the Parent.prototype.memberfunction.call (this, /\*any original args\*/) syntax.

|  |
| --- |
| **Example:**  // util function  **var** inherits = require('util').inherits;  // Base  **function** Base() {  **this**.message = "This is a message";  };  Base.prototype.relationship = **function** () {  **return** **this**.message + " From Parent"  };  // Child  **function** Child() { Base.call(**this**); };  inherits(Child, Base);  // Overide parent behaviour in child  Child.prototype.relationship = **function** () {  // Call base implementation + customize  **return** Base.prototype.relationship.call(**this**) + " to child ";  }  // Test:  **var** child = **new** Child();  console.log(child.relationship()); // message base relationship child relationship  **Output:**  This is a message From Parent to child |

b. **Checking Inheritance Chain**

|  |
| --- |
| **Example:**  **var** inherits = require('util').inherits;  **function** A() { }  **function** B() { }; inherits(B, A);  **function** C() { }  **var** b = **new** B();  console.log(b **instanceof** B); // true because b.\_\_proto\_\_ == B.prototype  console.log(b **instanceof** A); // true because b.\_\_proto\_\_.\_\_proto\_\_ == A.prototype  console.log(b **instanceof** C); // false  **Output:**  true  true  false |

**Build – In Objects**

**1. Objects**

* To create an object, we can use either of the following,

var obj1 = new Object();

var obj2 = {};

* The latter, known as object literal syntax, is preferred.
* We can specify the contents of objects using object literal syntax.
* We can specify member names and values at initialization time.
* We can add a new property to your user object by using any of the following methods:

|  |
| --- |
| **For Example:**  user.name= "Harry";  user["name"] = "Harry";  var attribute = 'name';  user[attribute] = "Harry"; |

* If we try to access a property that does not exist, we do not receive an error, but instead just get back undefined.
* To remove a property from an object, we can use the delete keyword:

|  |
| --- |
| **For Example:**  **var** user = {  first\_name: "Stuart",  last\_name: "Little",  age: 32,  Email\_id: "stuart@gmail.com"  };  console.log("Object Example\t:" + user.age);  console.log("Delete\t:"+ **delete** user.Email\_id);  console.log("Delete Confirmation :" +user.Email\_id);  **The above code when executed results in:**  Object Example :32  Delete :true  Delete Confirmation :undefined |

**2. Object Literals**

* The most common way of creating an object in JavaScript is using the object notation, {}.
* Objects can be extended arbitrarily at runtime.

|  |
| --- |
| **For Example:**  **var** myData = {};  console.log(myData); // {}  myData.myValue = 123; // extend myData  console.log(myData); // { myValue: 123 }  **The above code when executed results in:**  {}  { myValue: 123 } |

* We can define which properties go on an object upfront by using the object literal notation.

|  |
| --- |
| **For Example:**  **var** myData = {  myValue: 123  };  console.log(myData); // { myValue: 123 }  **The above code when executed results in:**  { myValue: 123 } |

* We can nest object literals inside object literals.

|  |
| --- |
| **For Example:**  **var** myData = {  myValue: 123,  add: {  add1: "SMI",  add2: "Madurai",  add3: [6,2,5,0,1,4]  }  };  console.log(myData);  **The above code when executed results in:**  { myValue: 123,  add: { add1: 'SMI', add2: 'Madurai', add3: [ 6, 2, 5, 0, 1, 4 ] } } |

* We can also have these arrays themselves contain object literals.

|  |
| --- |
| **For Example:**  **var** myData = {  myValue: 123,  nest: [{  myItem: 1  },  {  myItem: 2  },  {  myItem: 3  }]  };  console.log(myData.myValue); // 123  console.log(myData.nest[0].myItem); // 1  console.log(myData.nest[2].myItem); // 2  **The above code when executed results in:**  123  1  3 |

**3. JSON**

* JSON, or JavaScript Object Notation is the data exchange format.
* JSON is similar to object literal notation with two key differences.
* In object literal notation, wrapping the property names in single or double quotation marks is optional, in JSON it is mandatory.
* All strings should be double-quoted in JSON as follows.
* To generate JSON, we can use the V8 Javascript functions JSON.parse and JSON.stringify.
* The former takes a JSON string and converts it to an object, while the latter takes an object and returns a JSON string representation of it.

|  |
| --- |
| // valid object literal notation, INVALID JSON:  **var** obj = {  // JSON strings are supposed to use ", not '  "first\_name": 'HTML',  // Must wrap property names for JSON  last\_name: "CSS"  }  // valid JSON and object literal notation:  **var** obj = {  "first\_name": "HTML",  "last\_name": "CSS"  } |

**4. Arrays**

* To create arrays, you can either use traditional notation or array literal syntax :

var arr1 = new Array();

var arr2 = [];

* As with objects, the literal syntax version is preferred.
* We can test if an object is an array using the Array.isArray function:

var arr2 = [];

Array.isArray(arr2);

Array.isArray({});

* We can create arrays quite easily in JavaScript using [].
* Arrays have many useful functions.

|  |
| --- |
| **For Example:**  **var** data = [];  data.push(1); // add at the end  console.log(data); // prints [1]  data.push(5);  data.push(7);  data.push(9);  console.log(data);  data.unshift(2); // add to the top  console.log(data); // prints [2,1]  data.unshift(43);  data.unshift(22);  console.log(data);  // Arrays are zero index based:  console.log(data[0]);  console.log(data[6]);  **The above code when executed results in:**  [ 1 ]  [ 1, 5, 7, 9 ]  [ 2, 1, 5, 7, 9 ]  [ 22, 43, 2, 1, 5, 7, 9 ]  22  9 |

* The array type in JavaScript length property returns the element count.
* By default, arrays in JavaScript are numerically indexed.

|  |
| --- |
| **For Example:**  **var** array1 = [];  array1.length;  console.log("Data Of Array 1"+array1);  **var** array2 = [ 'Tom', 'Dick', 'Harry', 'Olive' ];  console.log("Length of array:" +array2.length);  **for** (**var** i = 0; i < array2.length; i++) {  console.log("Array 2's Data:"+ array2[i]);  }  **The above code when executed results in:**  Data Of Array 1  Length of array:4  Array 2's Data:Tom  Array 2's Data:Dick  Array 2's Data:Harry  Array 2's Data:Olive |

* To add an item to the end of an array, you can do one of two things.

|  |
| --- |
| **For Example:**  **var** array = [ 'Tom', 'Dick', 'Harry', 'Olive' ];  array.push("Scooby");  console.log(array);  array[array.length] = "fattu";  console.log(array);  **The above code when executed results in:**  [ 'Tom', 'Dick', 'Harry', 'Olive', 'Scooby' ]  [ 'Tom', 'Dick', 'Harry', 'Olive', 'Scooby', 'fattu' ] |

* We can specify the index of the element where you want to insert a new element.
* If this element is past the last element, the elements in between are created and initialized with the value undefined.

|  |
| --- |
| **For Example:**  **var** array = [ 'Tom', 'Dick', 'Harry', 'Olive' ];  array[15] = "Scooby";  array[4] = "Dexter";  console.log(array);  **The above code when executed results in:**  [ 'Tom', 'Dick', 'Harry', 'Olive', 'Dexter', , , , , , , , , , , 'Scooby' ] |

**5. Array Functions**

1. **push and pop**

The push and pop functions let you add and remove items to the end of an array, respectively:

|  |
| --- |
| **Example:**  **var** nums = [ 1, 12, 42, 53, 57, 98 ];  nums.push(103);  console.log("Implementation of Push():"+nums);  nums.pop();  console.log("Implementation of pop():"+nums);  Output:  Implementation of Push():1,12,42,53,57,98,103  Implementation of pop():1,12,42,53,57,98 |

1. **unshift and shift**

To insert or delete items from the front of an array, use unshift or shift, respectively:

|  |
| --- |
| **Example:**  **var** nums = [ 1, 12, 42, 53, 57, 98 ];  nums.unshift(1);  console.log("Implementation of unshift():"+nums);  nums.shift();  console.log("Implementation of shift():"+nums);  **Example:**  Implementation of unshift():1,1,12,42,53,57,98  Implementation of shift():1,12,42,53,57,98 |

1. **join**

The array function join returns a string from the array:

|  |
| --- |
| **Example:**  **var** nums = [ 1, 12, 42, 53, 57, 98 ];  **var** s = nums.join(", ");  console.log("Implementation of join():"+s);  **Output:**  Implementation of join():1, 12, 42, 53, 57, 98 |

1. **Sort**

You can sort arrays using the sort function, which can be used with the built-in sorting function:

|  |
| --- |
| **Example:**  **var** nums = [ 1, 12, 42, 53, 57, 98 ];  **var** names = [ 'Tom', 'Henry', 'Jacob', 'Scooby', 'Alice', 'Adams', 'Flinstone'];  nums.sort();  names.sort();  console.log("Implementation of sort() for numbers\t:"+nums);  console.log("Implementation of soft() for names\t:"+ names);  **Output:**  Implementation of sort() for numbers :1,12,42,53,57,98  Implementation of soft() for names :Adams,Alice,Flinstone,Henry,Jacob,Scooby,Tom |

|  |
| --- |
| **Example 2:**  **var** names = [ 'Tom', 'Henry', 'Jacob', 'Scooby', 'Alice', 'Adams', 'Flinstone'];  console.log("Array values\t\t:"+names);  names.sort(**function** (a, b) {  **var** a1 = a.toLowerCase(), b1 = b.toLowerCase();  **if** (a1 < b1) **return** 1;  **if** (a1 > b1) **return** -1;  **return** 0;  });  console.log("Implementation of soft() for names in reverse order\t:"+ names);  **Output:**  Array values :Tom,Henry,Jacob,Scooby,Alice,Adams,Flinstone  Implementation of soft() for names in reverse order :Tom,Scooby,Jacob,Henry,Flinstone,Alice,Adams |

1. **Loop**

To iterate over items in arrays, we can use the for loop or forEach function

|  |
| --- |
| **Example:**  [ 'Tom', 'Henry', 'Jacob', 'Scooby', 'Alice', 'Adams', 'Flinstone'].forEach( **function** (value)  {  console.log(value);  });  **Output:**  Tom  Henry  Jacob  Scooby  Alice  Adams  Flinstone |

**Function**

**1. Node.js Functions**

JavaScript is a functional programming language, functions are fully typed objects that can be manipulated, extended, and passed around as data.

* A normal function structure in JavaScript is defined as follows.

function functionName() {

// function body

// optional return;

}

* All functions return a value in JavaScript.
* In the absence of an explicit return statement, a function returns undefined.

|  |
| --- |
| **Example:**  **function** fn1() {  **return** 1983;  }  console.log("Data of fn1()\t:"+fn1()); // 123  **function** fn2() {  }  console.log("Data of fn2()\t:"+fn2()); // undefined  **function** fn3(data){  console.log("This is a sample Data "+ data);  }  fn3("of Function.");  **Output:**  Data of fn1() :1983  Data of fn2() :undefined  This is a sample Data of Function |

* To declare parameters for a function in JavaScript, list them in the parentheses.
* There is no checking of these parameters at runtime.
* If too few parameters are passed into a function call, the resulting variables are assigned the value undefined.
* If too many are passed in, the extras are simply unused.
* All functions have a predefined array in the body called arguments.
* It has all the values that were passed in to the function, and we can do extra checking on the parameter list.

|  |
| --- |
| **Example:**  **function** fn1(data){  console.log("This is a sample Data "+ data);  }  fn1("of Function.");  fn1();  fn1("Tom", "Jerry", "AAA", 4);  **Output:**  This is a sample Data of Function.  This is a sample Data undefined  This is a sample Data Tom |

* Functions in JavaScript do not even need to have names.

|  |
| --- |
| **Example:**  **var** x = **function** (a, b) {  **return** a + b;  }  console.log("Function without name:"+x(10, 20));  **Output:**  Function without name:30 |

**1. Function Scope**

* Every time a function is called, a new variable scope is created.
* Variables declared in the parent scope are available to that function.
* Variables declared within the new scope are not available when the function exits.

|  |
| --- |
| **Example:**  **var** pet = 'cat';  **function** myMethod() {  **var** pet = 'dog';  console.log("Inside\t:"+ pet);  }  myMethod();  console.log("Outside\t:"+ pet);  **Output:**  Inside :dog  Outside :cat |

* Combining this scoping with anonymous functions is better way to use private variables that will disappear when the anonymous function exits.

|  |
| --- |
| **Example 2:**  **var** height = 15;  **var** radius = 3;  **var** volume;  // declare and immediately call anonymous function to create scope  (**function** () {  **var** pir2 = Math.PI \* radius \* radius; // temp var  volume = (pir2 \* height) / 3;  })();  console.log("Volume of a cone\t:"+volume);  **Output:**  Volume of a cone :141.3716694115407 |

**2. Higher Order Function**

**a. Immediately Executing Function**

* We can execute a function immediately after you define it.
* Simply wrap the function in parentheses () and invoke it.

(function myData() {

console.log('myData was executed!');

})();

* An immediately executing function creates a new variable scope.
* An if, else, or while does not create a new variable scope in JavaScript.

|  |
| --- |
| **Example:**  **var** fun1 = 123;  **if** (**true**) {  **var** fun1 = 456;  }  console.log(fun1); // 456;  **Output:**  **456** |

* The only recommended way of creating a new variable scope in JavaScript is using a function.
* The following code shows how to create a new variable scope with an immediately executing function.

|  |
| --- |
| **Example:**  **var** fun1 = 123;  **if** (**true**) {  (**function** () { // create a new scope  **var** fun1 = 456;  })();  }  console.log(fun1); // 123;  **Output:**  **123** |

**b. Anonymous Function**

* A programming language is said to have first-class functions if a function can be treated the same way as any other variable in the language.
* JavaScript has first-class functions.
* A function without a name is called an anonymous function.
* In JavaScript, you can assign a function to a variable.
* If you are going to use a function as a variable, you don't need to name the function.
* The following example shows two ways of defining a function inline.
* Both of these methods are equivalent.

|  |
| --- |
| **Example:**  **var** function1 = **function** namedFunction() {  console.log('I belong to Named function');  }  function1(); // function1  **var** function2 = **function** () { // no function name i.e. anonymous function  console.log('I Belong to Anonymous function');  }  function2(); // function2  **Output:**  I belong to Named function  I Belong to Anonymous function |

**c. Higher-Order Functions**

* Since JavaScript allows us to assign functions to variables, we can pass functions to other functions.
* Functions that take functions as arguments are called higher-order functions.
* A very common example of a higher-order function is setTimeout.
* The following code shows how to use setTimeout function.

|  |
| --- |
| **Example:**  setTimeout(**function** () {  console.log('4000 milliseconds have passed since this demo started');  }, 4000);  **Output:**  4000 milliseconds have passed since this demo started |

* If you run this application in Node.js, you will see the console.log message after four seconds and then the application will exit.
* We used an anonymous function as the first argument to setTimeout.
* This makes setTimeout a higher-order function.
* We can create a function and passing that in.

|  |
| --- |
| **Example:**  **function** fun() {  console.log('4000 milliseconds have passed since this demo started');  }  setTimeout(fun, 4000);  **Output:**  4000 milliseconds have passed since this demo started |

**d. Closures**

* If there is a function defined inside another function, the inner function has access to the variables declared in the outer function.
* The variables in the outer function have been closed by the inner function.
* The concept in itself is simple enough and fairly intuitive.
* The inner function can access the variables from the outer scope even after the outer function has returned.
* Because the variables are still bound in the inner function and not dependent on the outer function.

|  |
| --- |
| **Example:**  **function** outerFunction(arg) {  **var** variableInOuterFunction = arg;  **function** myValue() {  console.log(variableInOuterFunction);  }  myValue();  }  outerFunction('hello closure!'); // logs hello closure!  **Output:**  hello closure! |

|  |
| --- |
| **Example 2:**  **function** outerFunction(arg) {  **var** variableInOuterFunction = arg;  **return** **function** () {  console.log(variableInOuterFunction);  }  }  **var** innerFunction = outerFunction('hello closure!');  innerFunction();  **Output:**  hello closure! |

**Node.js Advanced**

**1. Asynchronous Programming**

The following code shows how Node.js handles the nonblocking, asynchronous model.

setTimeout() function takes a function to call and a timeout after which it should be called:

|  |
| --- |
| **Example:**  setTimeout(**function** () {  console.log("done");  }, 3000);  console.log("waiting");  **Output:**  waiting  done |

* The program sets the timeout for 3000ms (3s), and then continues with execution, which prints out the "waiting" text.
* In Node.js, to call a function that needs to wait for some external resource, instead of calling fopen(path, mode) and waiting, we should call fopen(path, mode, function callback(file\_handle) { ... }).

|  |
| --- |
| **Example:**  **var** fs = require('fs');  fs.open('sample.js', 'r',  **function** (err, handle) {  **var** buf = **new** Buffer(100000);  fs.read(handle, buf, 0, 100000, **null**,  **function** (err, length) {  console.log(buf.toString('utf8', 0, length));  fs.close(handle, **function** () { /\* don't care \*/ });  }  );  }  );  **Output:**  var fs = require('fs');  fs.open('sample.js', 'r',  function (err, handle) {  var buf = new Buffer(100000);  fs.read(handle, buf, 0, 100000, null,  function (err, length) {  console.log(buf.toString('utf8', 0, length));  fs.close(handle, function () { /\* don't care \*/ });  }  );  }  ); |

* The require function is a way to include additional functionality in your Node.js programs.
* callback asynchronous functions has at least one parameter, the success or failure status of the last operation. It commonly has a second parameter which has additional results or information from the last operation.

do\_something(param1, param2, ..., paramN, function (err, results) { ... });

**2. Web**

**a. HTTP Response Codes**

* The HTTP specification contains a large number of response codes a server can return to clients.
* We'll use a few of the more common responses in most of our applications.

|  |  |  |
| --- | --- | --- |
| **Code** | **Meaning** | **Description** |
| 200 | OK | Everything went fine. |
| 301 | Moved Permanently | The requested URL has been moved, and the client should rerequest it at the URL specified in the response. |
| 400 | Bad Request | The format of the client's request is invalid and needs to be fixed. |
| 401 | Unauthorized | The client has asked for something it does not have permission to view. |
| 403 | Forbidden | The server is refusing to process this request. This is not the same as 401, where the client can try again with authentication. |
| 404 | Not Found | The client has asked for something that does not exist. |
| 500 | Internal Server Error | Something happened resulting in the server being unable to process the request. |
| 503 | Service Unavailable | This indicates some sort of runtime failure. |

**b. Your First JSON Server**

Here is the trivial server, which is saved to sample.js:

|  |
| --- |
| **Example:**  **var** http = require('http');  **function** handle\_incoming\_request (req, res) {  console.log("INCOMING REQUEST: " + req.method + " " + req.url);  res.writeHead(200, { "Content-Type" : "application/json" });  res.end(JSON.stringify( { error: **null** }) + "\n");  }  **var** s = http.createServer(handle\_incoming\_request);  s.listen(8080);  **Console View:** INCOMING REQUEST: GET /  INCOMING REQUEST: GET /favicon.ico  INCOMING REQUEST: GET /favicon.ico  INCOMING REQUEST: GET /  INCOMING REQUEST: GET /favicon.ico  INCOMING REQUEST: GET /favicon.ico  INCOMING REQUEST: GET /  INCOMING REQUEST: GET /favicon.ico  INCOMING REQUEST: GET /favicon.ico  **Browser View:** |

**3. Buffer**

* In Node.js we can manipulate Binary Data with Buffers.
* When working with streams and files, we work mostly with the Buffer class.
* Buffer hold binary data that can be converted into other formats, used in operations to file writes, or broken apart and reassembled.
* Buffer's length property does not return the size of the content, but that of the buffer itself!
* To work with TCP streams and the file system, the developers added native and fast support to handle binary data. The developers did this in Node.js using the Buffer class, which is available globally.

|  |
| --- |
| **Example:**  **var** b = **new** Buffer(1000);  **var** str = "apple";  b.write(str); // default is utf8, which is what we want  console.log(b.length); // will print 1000 still!  console.log( str.length ); // prints 5  console.log( Buffer.byteLength(str) ); // prints 5  **Output:**  1000  5  5 |

**a. Encoding**

* Node.js supports all the popular encoding formats like ASCII, UTF-8, and UTF-16.
* To convert strings to buffers, call the Buffer class constructor passing in a string and an encoding.
* Call the Buffer instance's toString method and pass in an encoding scheme to convert buffer to string.
* To convert a buffer to a string, use the toString method.

|  |
| --- |
| **Example:**  **var** str = "Sri Mookambika Infosolutions Pvt Ltd.";  // From string to buffer  **var** buffer = **new** Buffer(str, 'utf-8');  // From buffer to string  **var** roundTrip = buffer.toString('utf-8');  console.log("The result of RoutdTrip: "+roundTrip);  **Output:**  The result of RoutdTrip: Sri Mookambika Infosolutions Pvt Ltd. |

* To append one buffer to the end of another, you can use the concat method.
* We can fill in all the values in the buffer by using the fill method, such as buf.fill("\0").
* buf.fill("\0") zero out the buffer.

|  |
| --- |
| **Example:**  **var** b1 = **new** Buffer("Sri Mookambika Infosolutions Pvt Ltd, ");  **var** b2 = **new** Buffer("Madurai");  **var** b3 = Buffer.concat([ b1, b2 ]);  console.log(b3.toString('utf8'));  **Output:**  Sri Mookambika Infosolutions Pvt Ltd, Madurai |

**4. setTimeout setInterval**

**a. setTimeout**

* setTimeout sets up a function to be called after a specified delay in milliseconds.
* The following code shows a quick example of setTimeout, which calls a function after 1,000 milliseconds (one second).

**b. setInterval**

* Similar to the setTimeout function is the setInterval function.
* setTimeout only executes the callback function once after the specified duration.
* setInterval calls the callback repeatedly after every passing of the specified duration.

|  |
| --- |
| **Example:**  setTimeout(**function**() {  console.log("Hello");  }, 2000)  setInterval(**function**() {  console.log("World");  }, 2000)  **Output:**  Hello  World  World  World  World  World  World  World (continues printing World after 2 sec) |

* Both setTimeout and setInterval return an object that can be used to clear the timeout/interval using the clearTimeout/clearInterval functions.
* The following code demonstrates how to use clearInterval to call a function after every second for five seconds, and then clear the interval after which the application will exit.

|  |
| --- |
| **Example:**  **var** count = 0;  **var** intervalObject = setInterval(**function** () {  count++;  console.log(count, 'seconds passed');  **if** (count == 5) {  console.log('exiting');  clearInterval(intervalObject);  }  }, 1000);  **Output:**  1 'seconds passed'  2 'seconds passed'  3 'seconds passed'  4 'seconds passed'  5 'seconds passed'  Exiting |

**Node.js Module**

**1. Module System**

* Node.js uses File Based Module System.
* Each file is its own module.
* Each file has access to the current module definition using the module variable.
* The export of the current module is determined by the module.exports variable.
* To import a module, use the globally available require function.

|  |
| --- |
| **Example:**  **//MyData.js**  module.exports = **function** () {  console.log('Hello Iam from MyData.js File');  };  **//sample.js**  //import myData using the globally require function and store the returned value in a local variable.  **var** myData = require('./MyData');  myData();  **Output:**  Hello Iam from MyData.js File |

**a. module.exports**

* Each file in Node.js is a module.
* The items to export from a module should be attached to the module.exports variable.
* module.exports is defined to be a new empty object in every file.
* module.exports = {} is implicitly present.
* By default, every module exports an empty object, {}.

console.log(module.exports); // {}

**b. Exports Alias**

* We can export more than one variable from a module.
* One way of achieving this is to create a new object literal and assign that to module.exports.
* Node.js helps us by creating an alias for module.exports called exports so instead of typing module.exports.something every time, you can simply use exports.something.
* exports is just like any other JavaScript variable.
* Node.js simply does exports = module.exports.
* If we add something for example, myData to exports, that is exports.myData = 123, we are effectively doing module.exports.myData = 123 since JavaScript variables are references.
* The following code shows that all of these methods are equivalent from consumption (import) point of view.

|  |
| --- |
| **Example:**  **//myData1.js**  **var** a = **function** () {  console.log('a called from myData1');  };  **var** b = **function** () {  console.log('b called from myData1');  };  module.exports = {  a: a,  b: b  };  **//myData2.js**  module.exports.a = **function** () {  console.log('a called from myData2');  };  module.exports.b = **function** () {  console.log('b called from myData2');  };  **//myData3.js**  exports.a = **function** () {  console.log('a called from myData3');  };  exports.b = **function** () {  console.log('b called from myData3');  };  **//appForData.js**  console.log("\tExample for Exports Alias");  **var** myData1 = require('./myData1');  myData1.a();  myData1.b();  **var** myData2 = require('./myData2');  myData2.a();  myData2.b();  **var** myData3 = require('./myData3');  myData3.a();  myData3.b();  **Output:**  Example for Exports Alias  a called from myData1  b called from myData1  a called from myData2  b called from myData2  a called from myData3  b called from myData3 |

**Modules Best Practices**

**a. Do Not Use the .js Extension**

It is better to do require('./myData') instead of require('./myData.js') even though both work fine for Node.js.

**b. Relative Paths**

When using file-based modules, you need to use relative paths (in other words, do require('./myData') instead of require('myData')).

**c. Utilize exports**

Try and use the exports alias when you want to export more than one thing.

The following code shows how to Create a Local Variable and Also Export

var myData = exports.myData = /\* whatever you want to export as `myData` from this module \*/ ;

**d. Export an Entire Folder**

If you have too many modules that go together that you keep importing into other files, try to avoid repeating the import.

|  |
| --- |
| **var** myData = require('../something/myData');  **var** myValue = require('../something/myValue');  **var** another = require('../something/another');  **var** third = require('../something/third'); |

Instead, create a single index.js in the something folder. In index.js, import all the modules once and then export them from this module.

|  |
| --- |
| **// index.js**  exports.myData = require('./myData');  exports.myValue = require('./myValue');  exports.another = require('./another');  exports.third = require('./third'); |

Now you can simply import this index.js whenever you need all these things:

|  |
| --- |
| **var** something = require('./index'); |

**2. Module Require**

**a. Node.js require Function**

* The Node.js require function is the main way of importing a module into the current file.
* There are three kinds of modules in Node.js:

1. core modules,
2. file modules,
3. external node\_modules.

* When we make a require call with a relative path-for example, something like require('./filename') or require('../foldername/filename'), Node.js runs the destination JavaScript file in a new scope and returns the final value assigned to module.exports in that file.
* Using the require function only gives you the module.exports variable, and you need to assign the result to a variable locally in order to use it in scope.

|  |
| --- |
| **var** yourChoiceOfLocalName = require('./myFile'); |

**b. Conditionally Load a Module**

* require behaves just like any other function in JavaScript.
* We can call require() based on some condition and load the module only if you need it.

|  |
| --- |
| **Example:**  **if**(iNeedThisModule){  **var** myData = require('./myData');  } |

**c. Blocking**

* The require function blocks further code execution until the module has been loaded.
* The code following the require() call is not executed until the module has been loaded and executed.

|  |
| --- |
| **Example:**  // Blocks execution till module is loaded  **var** myData = require('./myData');  // Continue execution after it is loaded  console.log('loaded myData');  myData(); |

**d. Cached**

After the first time a require call is made to a particular file, the module.exports is cached.

**3. Path Module**

* Use require('path') to load Path module.
* The path module has functions that works with the file/path string.
* For example, path.join uses the forward slash / on UNIX-based systems like Mac OS X vs. backward slash \\ on Windows systems.

**a. path.normalize (str)**

This function fixes up slashes to be OS specific, takes care of . and .. in the path, and also removes duplicate slashes.

**b. path.join([str1], [str2], ...)**

This function joins any number of paths together, taking into account the operating system.

|  |
| --- |
| **Example:**  **var** path = require('path');  //Fixes up .. and .  //logs on Unix: /myData  //logs on Windows: \myData  console.log(path.normalize('/myData/myValue/..'));  //Also removes duplicate '//' slashes  //logs on Unix: /myData/myValue  //logs on Windows: \myData\myValue  console.log(path.normalize('/myData//myValue/bas/..'));  // logs on Unix: myData/myValue/bas  // logs on Windows: myData\myValue\bas  console.log(path.join('myData', '/myValue', 'bas'));  **Output:**  \myData  \myData\myValue  myData\myValue\bas |

**dirname, basename, and extname**

* path.dirname gives you the directory portion of a specific path string (OS independent).
* path.basename returns the name of the file.
* path.extname gives you the file extension.

|  |
| --- |
| **Example:**  **var** path = require('path');  **var** completePath = '/myData/myValue/test.html';  console.log(path.dirname(completePath));  console.log(path.basename(completePath));  console.log(path.extname(completePath));  **Output:**  /myData/myValue  test.html  .html |

**4. fs Module**

* The fs module provides access to the file system.
* Use require('fs') to load this module.
* The fs module has functions for renaming files, deleting files, reading files, and writing to files.
* fs module has asynchronous as well as synchronous functions with the -Sync postfix for dealing with the file system.

The following code shows how to write to the file system and read from the file system.

|  |
| --- |
| **Example:**  **var** fs = require('fs');  fs.writeFileSync('myData1.js', 'Hello!!! Iam trying to implement fs!');  console.log(fs.readFileSync('myData1.js').toString());  **Output:**  **//In colsole**  Hello!!! Iam trying to implement fs!  //In myData1.js |

* To delete a file you can use unlink or unlinkSync.
* The async version takes a callback and is passed the error object if there is one.

|  |
| --- |
| **Example:**  **var** fs = require('fs');  **try** {  fs.unlinkSync('./myData1.js');  console.log('myData1.js successfully deleted');  } **catch** (err) {  console.log('Error:', err);  }  fs.unlink('./myData2.js', **function** (err) {  **if** (err) {  console.log('Error:', err);  } **else** {  console.log('myData2.js successfully deleted');  }  }); |

**5. os Module**

* The os module provides a few basic operating-system related utility functions and properties.
* You can access it using a require('os') call.
* For example, to get the current system memory usage, use os.totalmem() and os.freemem() functions.
* A vital facility provided by the os module is information about the number of CPUs available.

|  |
| --- |
| **Example:**  **var** os = require('os');  **var** gigaByte = 1 / (Math.pow(1024, 3));  console.log('Total Memory\t\t:', os.totalmem() \* gigaByte, 'GBs');  console.log('Available Memory\t:', os.freemem() \* gigaByte, 'GBs');  console.log('Percent consumed\t:', 100 \* (1 - os.freemem() / os.totalmem()));  console.log('This machine has', os.cpus().length, 'CPUs');  **output:**  Total Memory : 3.9093589782714844 GBs  Available Memory : 0.8172607421875 GBs  Percent consumed : 79.09515375945902  This machine has 4 CPUs |

**6. util Module**

* The util module contains a number of useful functions.
* We can access the util module using a require('util') call.
* To log out something to the console with a timestamp, you can use the util.log function.

|  |
| --- |
| **Example:**  **var** util = require('util');  util.log('sample message');  **Output:**  23 Dec 10:03:30 - sample message |

**a. util.format**

* util.format function is similar to the C/C++ printf function.
* The first argument is a string that contains zero or more placeholders.
* Each placeholder is then replaced using the remaining arguments based on the meaning of the placeholder.
* Popular placeholders are %s (used for strings) and %d (used for numbers).
* util has a few functions to check if something is of a particular type (isArray, isDate, isError).

|  |
| --- |
| **Example:**  **var** util = require('util');  **var** name = "Tom's";  **var** a = 33;  console.log(util.format('%s age is %d.', name, a));  **Output:**  Tom's age is 33. |

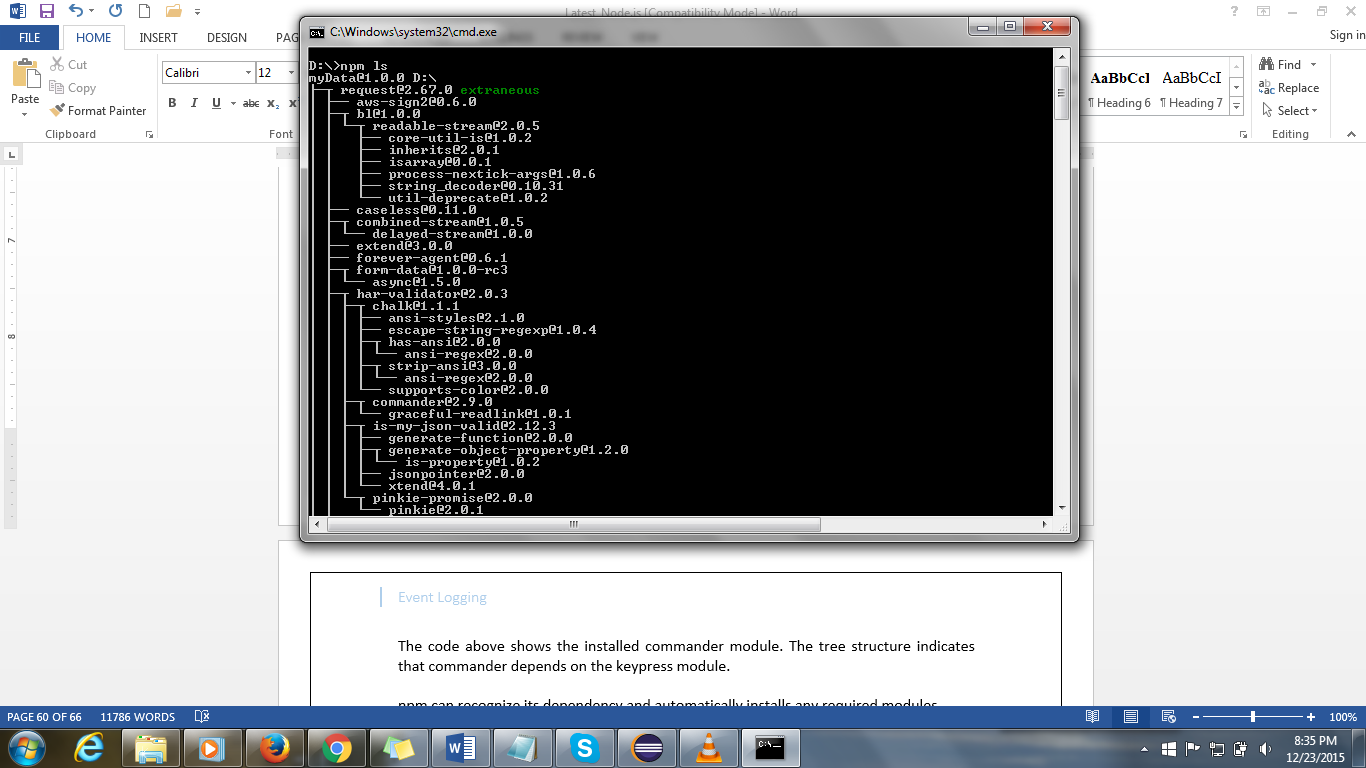
|  |
| --- |
| **Example:**  **var** util = require('util');  console.log(util.isArray([])); // true  console.log(util.isArray({ length: 0 })); // false  console.log(util.isDate(**new** Date())); // true  console.log(util.isDate({})); // false  console.log(util.isError(**new** Error('This is an error'))); // true  console.log(util.isError({ message: 'I have a message' })); // false  **Output:**  true  false  true  false  true  false |

**Node.js Packages**

* Node.js comes with its own package management system called Node Package Manager (NPM).
* There are three kinds of Node.js modules: file-based modules, core modules, and external node\_modules.
* If the module name passed into the require function is prefixed with './' or '../' or '/', then it is assumed to be a file-based module and the file is loaded.
* Otherwise, we look for core modules with the same name, for example, util­if the call was require('util').
* If no core module matching this name is found, we look for a node\_module called ­util.

**a. Package Locations**

* Packages are stored in a subdirectory named node\_modules within your current directory.
* To determine the location, use the command npm root.
* To view all the installed modules using the npm ls command.
* After installing the commander module, you can verify it using npm ls.



* The code above shows the installed commander module. The tree structure indicates that commander depends on the keypress module.
* npm can recognize its dependency and automatically installs any required modules.
* You can see the installed modules by browsing the node\_modules subdirectory.

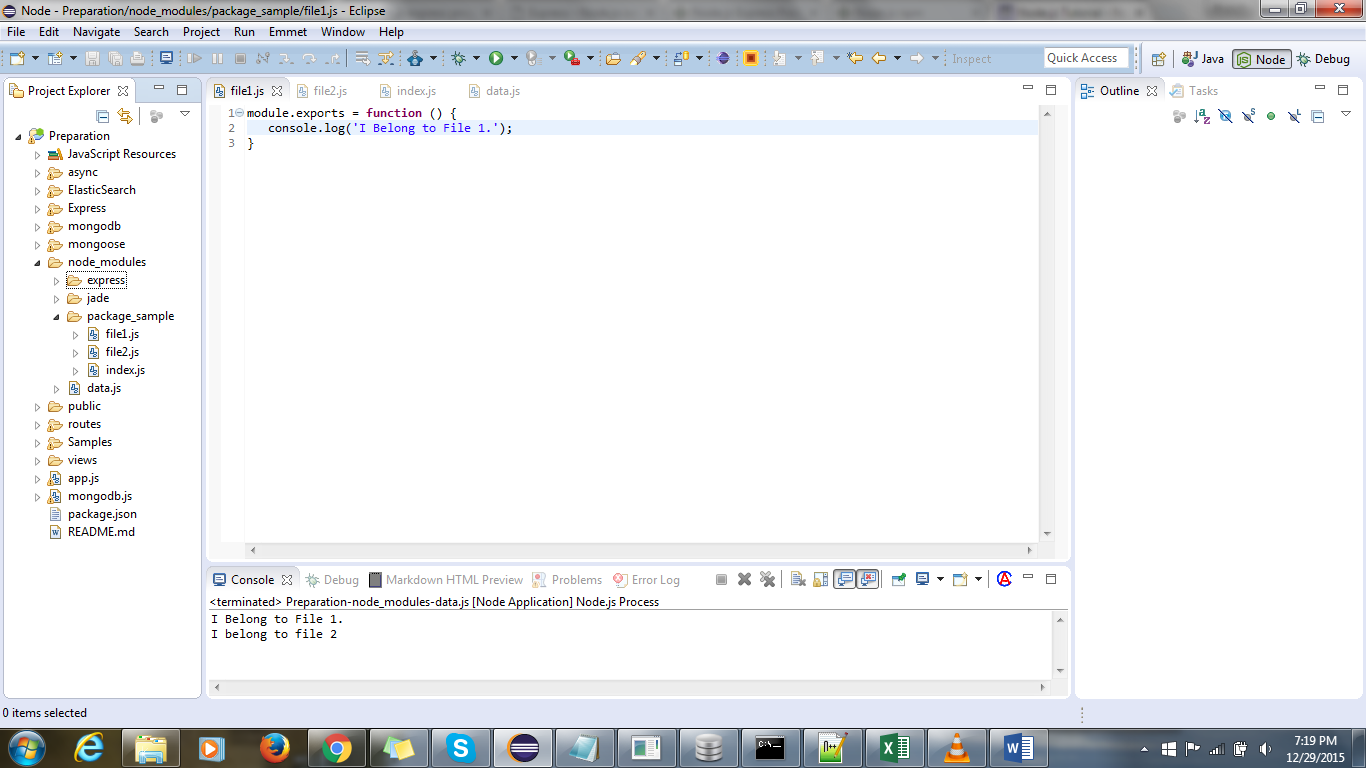
**b. Scanning for node\_modules**

|  |
| --- |
| **Example:**  **Basic.js**  module.exports = **function** () {  console.log('hello node\_modulesssss!');  }  **Hello.js**  **var** myModule = require('./Basic');  myModule();  **Output:**  hello node\_modulesssss! |

The only difference between file-based modules and node\_modules is the way in which the file system is scanned to load the JavaScript file. All other behavior is the same.

**Folder Based Modules:**

Our example follows the below mentioned folder structure.



|  |
| --- |
| **Example:**  **file1.js**  module.exports = **function** () {  console.log('I Belong to File 1.');  }  **file2.js**  module.exports = **function** () {  console.log('I belong to file 2..');  }  **index.js**  exports.f1 = require('./file1');  exports.f2 = require('./file2');  **data.js**  **var** data = require('package\_sample');  data.f1();  data.f2();  **Output:**  I Belong to File 1.  I belong to file 2.. |

**Node.js NPM JSON**

* NPM uses JSON files for configuring modules.
* JSON is a standard format transfer data over the network.
* JSON is a subset of JavaScript object literals.
* For JSON you must use quotation marks for JavaScript object keys.
* JSON limits a value for a given key.
* The values can only be a string, number, boolean (true or false), array, null, or another valid JSON object.

|  |
| --- |
| **Json Example:**  {  "firstName": "CSS",  "lastName": "HTML",  "isAlive": true,  "age": 5,  "height\_cm": 111.12,  "address": {  "streetAddress": "1234 Main Street",  "city": "New York",  "state": "NY"  },  "phoneNumbers": [  { "type": "home", "number": "222 555-1234" },  { "type": "fax", "number": "666 555-4567" }  ],  "additionalInfo": null  } |

* The firstName value is a string, age is a number, isAlive is a boolean, phoneNumbers is an array of valid JSON objects, additionalInfo is null, and address is another valid JSON object.
* The last property in JSON must not have an extra comma.

**a. Loading JSON in Node.js**

* We can load a JSON object from the file system the same way we load a JavaScript module.
* Every single time within the module loading sequence, Node.js looks for a file.json.
* If it is found, it returns a JavaScript object representing the JSON object.

|  |
| --- |
| **Example:**  file.json  {  "myData": "This an example for loading JSON in node.js"  }  **File.js:**  **var** config = require('./file');  console.log(config.myData);  **Output:**  This an example for loading JSON in node.js |

**JSON Converter**

* JSON object has functions for converting a string representation of JSON to JavaScript objects and converting JavaScript objects into a JSON string.
* To convert a JavaScript object to a JSON string, call JSON.stringify passing in the JavaScript object.
* This function returns the JSON string representation of the JavaScript object.
* To convert a JSON string into a JavaScript object, use the JSON.parse function, which simply parses the JSON string and returns a JavaScript object.

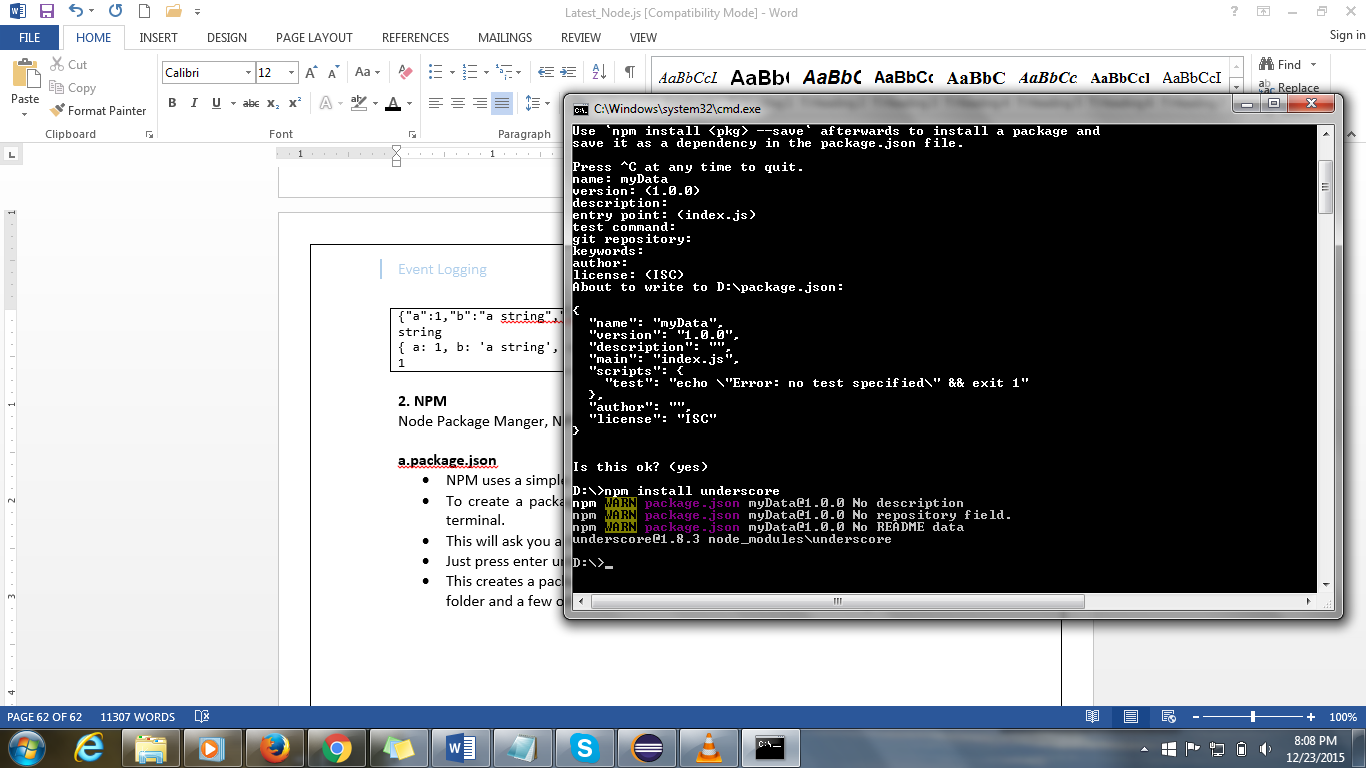
|  |
| --- |
| **Example:**  **var** myData = {  a: 1,  b: 'a string',  c: **true**  };  // convert a JavaScript object to a string  **var** json = JSON.stringify(myData);  console.log(json);  console.log(**typeof** json); // string  // convert a JSON string to a JavaScript object  **var** backToJs = JSON.parse(json);  console.log(backToJs);  console.log(backToJs.a); // 1  **Output:**  {"a":1,"b":"a string","c":true}  string  { a: 1, b: 'a string', c: true }  1 |

**2. NPM**

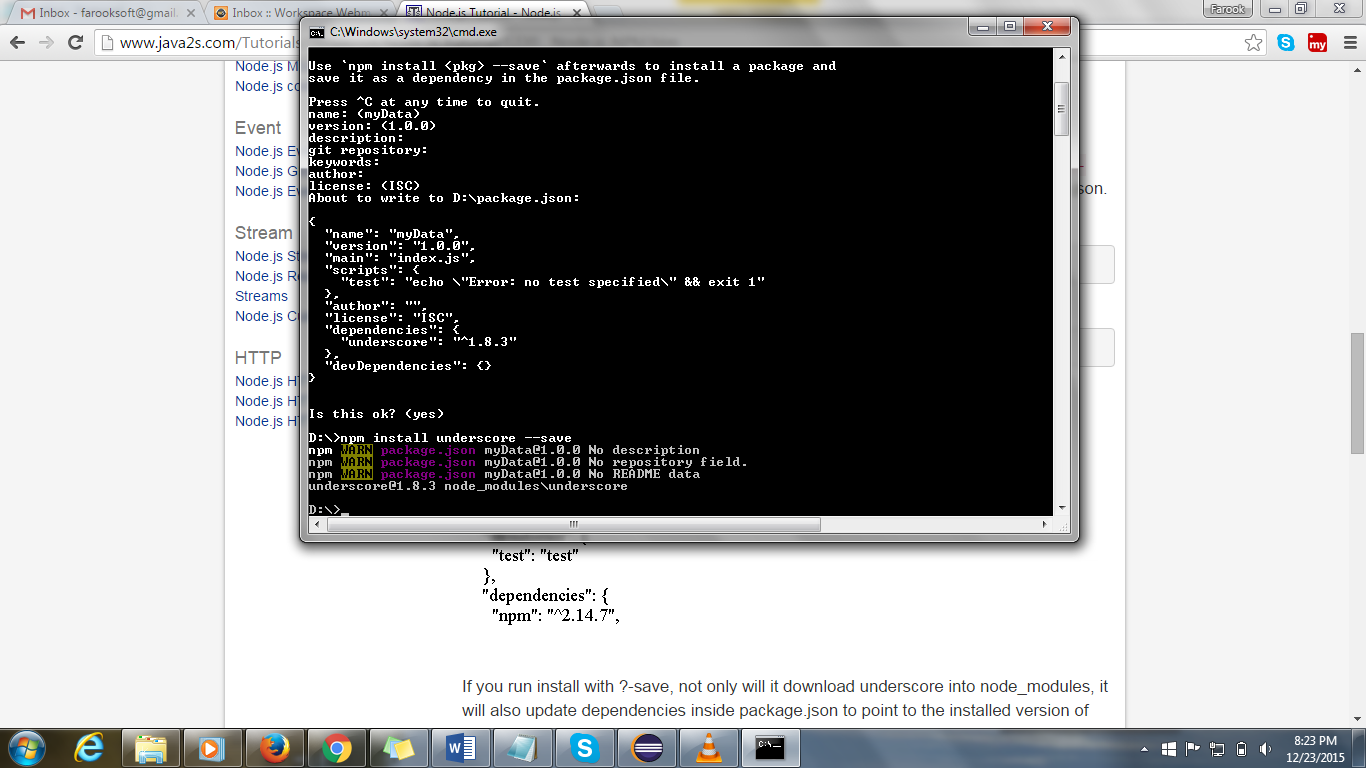
Node Package Manger, NPM, is a way to share node\_modules with the community.

**a.package.json**

* NPM uses a simple JSON file called package.json to share module information.
* To create a package.json file in the current folder, just run "npm init" in your terminal.
* This will ask you a few questions such as the name of the module and its version.
* Just press enter until the end.
* This creates a package.json in the current folder with the name set to the current folder and a few other reasonable defaults which look something like this.



* As well try to install underscore module as mentioned in the snapshot. Whenever you run npm install, you have an optional command line flag available --save that tells NPM to write the information about what you installed into package.json.



|  |
| --- |
| **Example:**  **var** \_ = require('underscore');  console.log(\_.max([879, 098, 878]));  **Output:**  879 |

**b. Listing All Dependencies**

To see which packages you have installed, run npm ls command.

**c. Removing a Dependency**

* To remove a package, use either the **npm uninstall** or **npm rm** command, and specify one or more package names.
* **npm rm underscore --save** deletes the underscore folder from **node\_modules** locally and modifies the dependencies section of your package.json.
* You can remove global packages by providing the **-g** option.

**d. package.json Online Dependency Tracking**

* npm install request is used to install online dependency

NPM not only installs request but also brings down a number of other packages that request depends upon.

**Child Processes**

As Node.js runs in a single thread mode but it uses an event-driven paradigm to handle concurrency. It also facilitates creation of child processes to leverage parallel processing on multi-core cpu based systems.

Child processes always have three streams child.stdin, child.stdout, and child.stderr which may be shared with the stdio streams of the parent process.

Node provides child\_process module which has following three major ways to create child process.

* **exec** - child\_process.exec method runs a command in a shell/console and buffers the output.
* **spawn** - child\_process.spawn launches a new process with a given command
* **fork** - The child\_process.fork method is a special case of the spawn() to create child processes.

**The exec() method**

child\_process.exec method runs a command in a shell and buffers the output. It has the following signature:

**child\_process.exec(command[, options], callback)**

**Parameters**

Here is the description of the parameters used:

* **command** String The command to run, with space-separated arguments
* **options** Object may comprise one or more of the following options:
  + **cwd** String Current working directory of the child process
  + **env** Object Environment key-value pairs
  + **encoding** String (Default: 'utf8')
  + **shell** 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** Function gets three arguments error, stdout and stderr which is called with the following output when process terminates

The exec() method returns a buffer with a max size and waits for the process to end and tries to return all the buffered data at once.

**Example:**

|  |
| --- |
| **Support.js**  console.log("Child Process " + process.argv[2] + " executed." );  **master.js**  **const** fs = require('fs');  **const** child\_process = require('child\_process');  **for**(**var** i=0; i<3; i++) {  **var** workerProcess = child\_process.exec('node support.js '+i,  **function** (error, stdout, stderr) {  **if** (error) {  console.log(error.stack);  console.log('Error code: '+error.code);  console.log('Signal received: '+error.signal);  }  console.log('stdout: ' + stdout);  console.log('stderr: ' + stderr);  });  workerProcess.on('exit', **function** (code) {  console.log('Child process exited with exit code '+code);  });  }  **Output:**  Child process exited with exit code 0  Child process exited with exit code 0  stdout: Child Process 1 executed.  stderr:  stdout: Child Process 2 executed.  stderr:  Child process exited with exit code 0  stdout: Child Process 0 executed.  stderr: |

**The spawn() method**

child\_process.spawn method launches a new process with a given command. It has the following signature:

**child\_process.spawn(command[, args][, options])**

The spawn() method returns streams (stdout & stderr) and it should be used when the process returns large amount of data. spawn() starts receiving the response as soon as the process starts executing.

**Example:**

|  |
| --- |
| **Spawn\_support.js**  console.log("Child Process " + process.argv[2] + " executed." );  **spawn\_master.js**  **const** fs = require('fs');  **const** child\_process = require('child\_process');    **for**(**var** i=0; i<3; i++) {  **var** workerProcess = child\_process.spawn('node', ['spawn\_support.js', i]);  workerProcess.stdout.on('data', **function** (data) {  console.log('stdout: ' + data);  });  workerProcess.stderr.on('data', **function** (data) {  console.log('stderr: ' + data);  });  workerProcess.on('close', **function** (code) {  console.log('child process exited with code ' + code);  });  }  **Output**  stdout: Child Process 0 executed.  stdout: Child Process 1 executed.  child process exited with code 0  child process exited with code 0  stdout: Child Process 2 executed.  child process exited with code 0 |

**The fork method**

child\_process.fork method is a special case of the spawn() to create Node processes. It has the following signature:

**child\_process.fork(modulePath[, args][, options])**

The fork method returns object with a built-in communication channel in addition to having all the methods in a normal ChildProcess instance.

**Example:**

|  |
| --- |
| **Fork\_support.js:**  console.log("Child Process " + process.argv[2] + " executed." );  **fork\_master.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** (code) {  console.log('child process exited with code ' + code);  });  }  **Output:**  Child Process 0 executed.  Child Process 1 executed.  Child Process 2 executed.  child process exited with code 0  child process exited with code 0  child process exited with code 0 |

**Cryptography**