Modules

Process

Each Node.js script runs in a **process**. It includes process object to get all the information about the current process of Node.js application.

The process is a global object that can be accessed from anywhere.

Property	Description
arch	returns process architecture: 'arm', 'ia32', or 'x64'
argv	returns commands line arguments as an array
env	returns user environment
pid	returns process id of the process
platform	returns platform of the process: 'darwin', 'freebsd', 'linux', or 'win32'
release	returns the metadata for the current node release
version	returns the node version
versions	returns the node version and its dependencies

Process

Methods	Description
cwd()	returns path of current working directory
memoryUsage()	returns an object having information of memory usage.
process.kill (pid[,signal])	is used to kill the given pid.
uptime()	returns the Node.js process uptime in seconds

Modules

- A simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.
- Each JavaScript file is treated as a separate module.
- Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope.
- Node.js includes three types of modules:
  - Core/Built-it Modules – Modules provided by Node.js.
  - Local Modules – Modules that we create in our application.
  - Third Party Modules – Modules written by other developers that we can use in our application.

Core / Built-in modules

- The core modules include bare minimum functionalities of Node.js.
- The core modules are compiled into its binary distribution and loaded automatically when Node.js process starts.
- Few important core modules below:

Module	Description
buffer	To handle binary data
os	Provides information about the operation system
timers	To execute a function after a given number of milliseconds
util	To access utility functions
events	To handle events
fs	To handle the file system
http	To make Node.js act as an HTTP server
stream	To handle streaming data
readline	To handle readable streams one line at the time

The buffer module

buffer

The buffer module provides a way of handling streams of binary data.

Buffer object is mainly used to store binary data, while reading from a file or receiving packets over the network.

The Buffer object is a global object in Node.js, and it is not necessary to import it using the require keyword.

Source code: https://github.com/nodejs/node/blob/v21.5.0/lib/buffer.js

The buffer module

Important properties & methods:

Properties & Methods	Description
alloc()	Creates a Buffer object of the specified length
byteLength()	Returns the numbers of bytes in a specified object
compare()	Compares two Buffer objects
concat()	Concatenates an array of Buffer objects into one Buffer object
copy()	Copies the specified number of bytes of a Buffer object
equals()	Compares two Buffer objects, and returns true if it is a match, otherwise false
fill()	Fills a Buffer object with the specified values
from()	Creates a Buffer object from an object (string/array/buffer)
includes()	Checks if the Buffer object contains the specified value. Returns true if there is a match, otherwise false

The buffer module

Properties & Methods	Description
indexOf()	Checks if the Buffer object contains the specified value. Returns the first occurrence, otherwise -1
length	Returns the length of a Buffer object, in bytes
slice()	Slices a Buffer object into a new Buffer objects starting and ending at the specified positions
toString()	Returns a string version of a Buffer object
toJSON()	Returns a JSON version of a Buffer object
values()	Returns an array of values in a Buffer object
write()	Writes a specified string to a Buffer object

The os module

os

The os module provides information about the computer's operating system.

Use this syntax to include OS module in your Node.js application:

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

The os module

OS – important methods

Method	Description
arch()	Returns the operating system CPU architecture
cpus()	Returns an array containing information about the computer's CPUs
freemem()	Returns the number of free memory of the system
hostname()	Returns the hostname of the operating system
platform()	Returns information about the operating system's platform
release()	Returns information about the operating system's release
tmpdir()	Returns the operating system's default directory for temporary files
totalmem()	Returns the number of total memory of the system
type()	Returns the name of the operating system
uptime()	Returns the uptime of the operating system, in seconds
userInfo()	Returns information about the current user

The timers module

timers

The timers module provides a way scheduling functions to be called later at a given time.

The Timers object is a global object in Node.js, and it is not necessary to import it using the require keyword.

Method	Description
clearInterval()	Cancels an Interval object
clearTimeout()	Cancels a Timeout object
setImmediate()	Executes a given function immediately.
setInterval()	Executes a given function at every given milliseconds
setTimeout()	Executes a given function after a given time (in milliseconds)

The util module

util

The util module provides access to some utility functions.

Use this syntax to include util module in your Node.js application:

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

Method	Description
debuglog()	Writes debug messages to the error object
deprecate()	Marks the specified function as deprecated
format()	Formats the specified string, using the specified arguments
inherits()	Inherits methods from one function into another
inspect()	Inspects the specified object and returns the object as a string

Local modules

- Local modules are modules created locally in your Node.js application.
- These modules include different functionalities of your application in separate files or folders.
- You can also package it and distribute it via NPM, so that Node.js community can use it.
- For example - if you need to connect to the database and fetch data then you can create a module for it, which can be reused in your application.

Local modules

- To use local modules in your application, you need to load it using require() function in the same way as core modules.
- However, you need to specify the relative path of JavaScript file of the module.

```
var calc = require('./calculator.js');
```

Module files present in another folder – 'utility'

```
var calc = require('../utility/calculator.js');
```

You must specify ./ as a path of root folder to import a local module.

The '.' denotes a root folder.

However, you do not need to specify the path to import Node.js core modules or NPM modules in the require() function.

Export Modules

- The *module.exports* is a special object which is included in every JS file in the Node.js application by default.
- The *module.exports* or *exports* is used to expose a function, object or variable in Node.js.
- The *module* is a variable that represents the current module, and *exports* represents an object that will be exposed and can be reused in another module.
- So, whatever you assign to module.exports will be exposed from that module.

Export Literals

- The following example exposes an object with a string property/literal present in message.js file.

Alternate syntax:

```
var message = 'Hello Node.js';
module.exports = message;
```

message.js

```
module.exports = 'Hello Node.js';
```

app.js

```
var msg = require('./message.js');
console.log(msg);
```

Specifying the extension is optional

Export named function

messageFunction.js

```
module.exports.greet = function (msg) {
    console.log(msg);
};
```

app.js

```
var msgObj = require('./messageFunction.js');
msgObj.greet('Named function…');
```

Export named function

Alternate syntax:

messageFunction.js

```
function greet(msg) {
    console.log(msg);
}
module.exports = greet;
```

OR

```
module.exports = function greet(msg) {
    console.log(msg);
};
```

app.js

```
var greet = require('./messageFunction.js');
greet('Named function…');
```

Export anonymous function

messageFunctionAnonymous.js

```
module.exports = function (msg) {
    console.log(msg);
};
```

app.js

```
var msg = require('./messageFunctionAnonymous.js');
msg('Anonymous function…');
```

Export simple objects

<u>messageObject.js</u>

```
module.exports.myMessage = 'Hello Node.js’;
```

OR

```
exports.myMessage = 'Hello Node.js';
```

<u>app.js</u>

```
var msgObj = require('./messageObject.js');
console.log(msgObj.myMessage);
```

Export complex objects

messageComplexObject.js

```
module.exports = {
    college: 'BMCC',
    course: 'BBA-CA',
    subject: 'Node.js'
}
```

app.js

```
var student = require('./messageComplexObject.js');
console.log(`${student.college} ${student.course}
${student.subject}`);
```

Export function as a Class

- In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.
- The new keyword is used to create an instance/object of the class.

student.js

```
module.exports = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.getFullName = function () {
        return this.firstName + ' ' + this.lastName;
    }
}
```

app.js

```
var student = require('./student.js');
var s1 = new student('First Name', 'Last Name');
console.log(s1.getFullName());
```

Export function as a Class

calculator.js

```
module.exports = {
    add: function (a, b) {
        return a + b;
    },
    subtract: function (a, b) {
        return a - b;
    },
    multiply: function (a, b) {
        return a * b;
    },
    divide: function (a, b) {
        return a / b;
    }
}
```

calculatorApp.js

```
var calculator = require('../utility/calculator.js');

console.log('----Calculator ----');
console.log('Addition is: ' + calculator.add(8, 4));
console.log('Subtraction is: ' + calculator.subtract(8, 4));
console.log('Multiplication is: ' + calculator.multiply(8, 4));
console.log('Division is: ' + calculator.divide(8, 4));
```

Export function as a Class

Alternate syntax:

```
function add(a, b) {
    return a + b;
}
function subtract(a, b) {
    return a - b;
}
function multiply(a, b) {
    return a * b;
}
function divide(a, b) {
    if (b != 0) {
        return a / b;
    }
    return "Divided by zero!";
}
```

calculator.js

Short hand syntax

```
module.exports = {
    add,
    subtract,
    multiply,
    divide
};
```

OR

```
module.exports = {
    add: add,
    subtract: subtract,
    multiply: multiply,
    divide: divide
};
```

Key: value syntax

Export function as a Class

Yet another syntax:

```
function add(a, b) {
    return a + b;
}
function subtract(a, b) {
    return a - b;
}
function multiply(a, b) {
    return a * b;
}
function divide(a, b) {
    if (b != 0) {
        return a / b;
    }
    return "Divided by zero!";
}
```

```
module.exports.add = add;
module.exports.subtract = subtract;
module.exports.multiply = multiply;
module.exports.divide = divide;
```

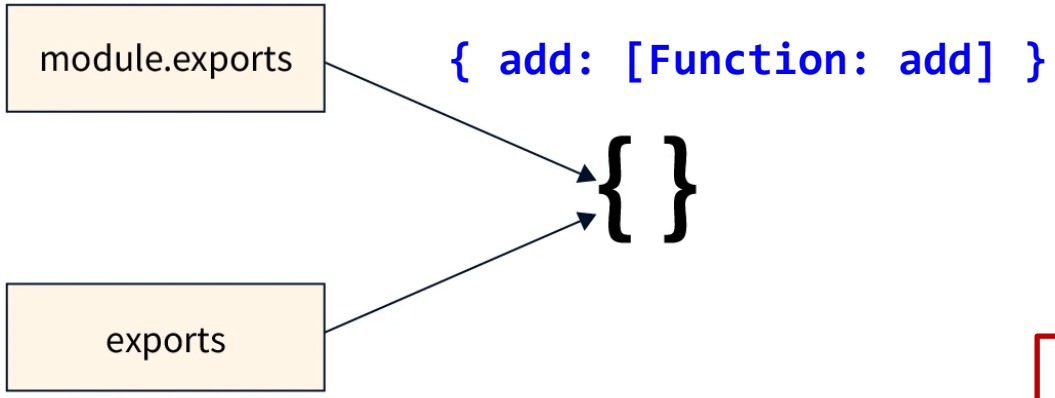
calculator.js

module.exports and exports

- The *module* is a plain JavaScript Object representing the current module.
- The module object has *exports* property which is a JavaScript variable, i.e. module.exports.
- The *exports* is a variable which is available in JavaScript files in Node.js.
- When you assign something (literal, variable or function) to *exports* directly, you are not assigning it to the *exports* property of the *module* object.
- So *exports* is just a convenience feature so that the module authors (developers) can write less code.

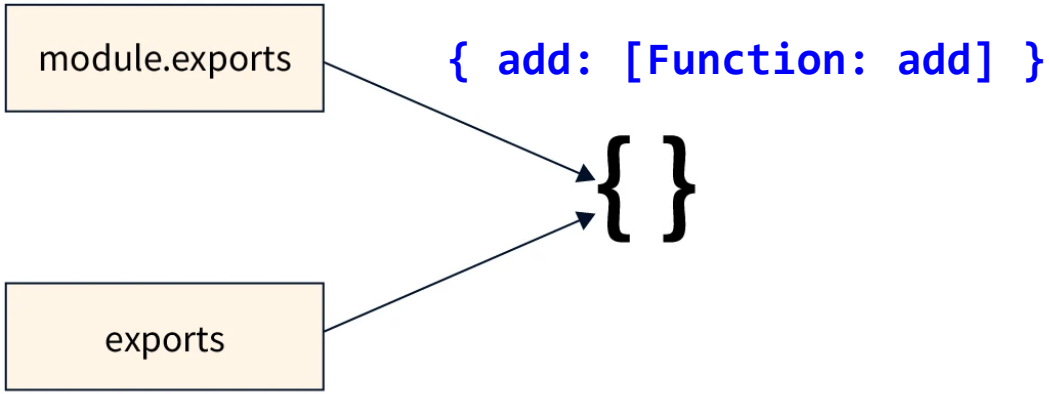
Be very careful while using 'exports' directly!

module.exports and exports




```
function add(a, b) {
    return a + b;
}
```

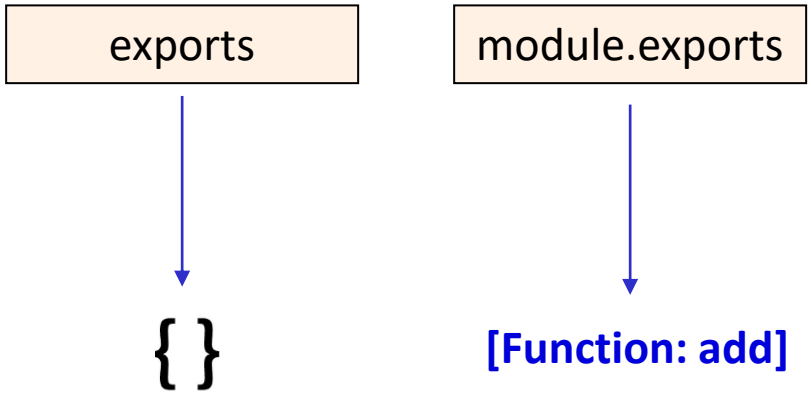
```
module.exports.add = add;
```




```
function add(a, b) {
    return a + b;
}
```

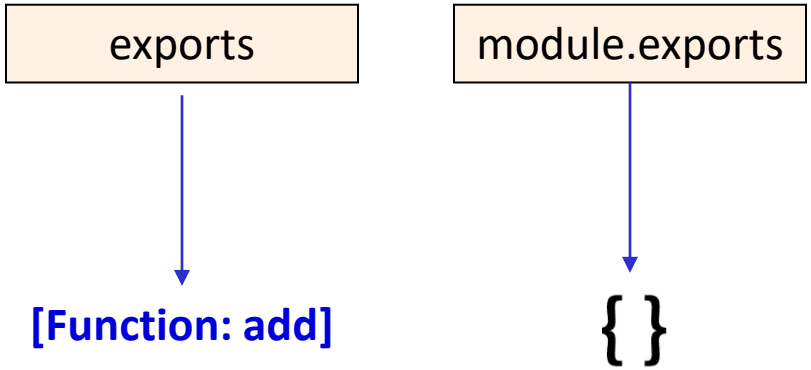
```
exports.add = add;
```

module.exports and exports




```
function add(a, b) {
    return a + b;
}

module.exports = add;
```




```
function add(a, b) {
    return a + b;
}

exports = add;
```

Home Work

- Explain the use of process object along with its important properties/methods
- Explain the use of Buffer object/module along with its important properties/methods
- Explain the use of os object/module along with its important properties/methods
- Explain the use of timers object/module along with its important properties/methods
- Explain the use of util object/module along with its important properties/methods

Home Work

- What is Module? What are the different types of modules?
- What are local modules? How can we use local modules?
- What is the difference between exports and module.exports?

Home Work

- Explain with code snippet:
  - Export literals
  - Export named function
  - Export anonymous function
  - Export simple object
  - Export complex object
  - Export function as a class

Practice Programs

Create Node.js application(s)

1) To convert kilometers to miles and vice versa.
2) To convert Celsius to Fahrenheit and vice versa.
3) To generate a random number between 1 to 100.
4) To find the factorial of a given number.
5) To print Fibonacci series.
6) To replace character of a string by another character.
7) To reverse a string.
8) To sort words in a string in alphabetical order.
9) To check number of occurrences of a character in a string.
10) To demonstrate calculator. Accept decimal numbers and operation from the user and perform respective operation.

Note: Use command line arguments and Node.js modules

Practice Programs

Create Node.js application(s)

1) To check if a string starts with a given character.
2) To replace all occurrences of a word in a string by another word.
3) To display current date/time as: Sat Dec 23 2023, 10:30:00 AM
4) To format the current date in different formats as:
    - 12/23/2023, 23/12/2023, 23-12-2023, 23-Dec-2023
5) To find the days between 2 dates.
6) To remove duplicates from an array.
7) To print table of 8, with 2 seconds delay in between.
8) To count the number of vowels in a string.
9) To check if the given word is a palindrome or not.
10) To convert first letter of each word in a string to uppercase

<u>Note</u>: Use command line arguments and Node.js modules