2...

Node JS Modules

Objectives...

- To learn about Functions.
- To study Node JS Buffer.
- To study Module and its type.
- To understand Module.Exports.

2.1 INTRODUCTION

- In Node JS, a module is a collection of JavaScript functions and objects that can be used by external applications. It is a set of functions which we would like to include in our application.
- Node JS modules are a type of package that can be published to NPM (Node Package Manager).
- Modules in Node JS are a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node JS application.
- In the Node JS module system, each file is treated as a separate module. So, if we are creating a `demo.js` file, this implies we are creating a module in Node. Basically, modules help us encapsulating our code into manageable chunks.
- Anything that we define in our module (i.e., in our JavaScript file) remains limited to that module only, unless we want to expose it to other parts of our code. So, anything we define inside our module remains private to that module only.
- In this chapter, we will be initially dealing with the basics of functions followed by creating and using different modules.

2.2 FUNCTIONS

- A function is a block of statements, which is used to perform a specific task. It is a block of organized, reusable code that is used to perform a single, related action.
- Functions provide better modularity for the application and a high degree of code reusability.

- A function is composed of a sequence of statements called the function body. Values can be passed to a function, and the function will return a value.
- We know that, Node JS is an interpreter and environment for the JavaScript with some specific useful libraries. So, let us see how to write a function in JavaScript.

2.2.1 Defining Functions

- A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:
  - The name of the function.
  - A list of parameters to the function enclosed in parentheses and separated by commas.
  - The JavaScript statements that define the function, enclosed in curly brackets, {...}.
- A normal function structure in JavaScript is defined as follows:

```
function functionName([param[, param, [..., param]]]){
    // function body
    // optional return;
}
```

- All functions return a value in JavaScript. In the absence of an explicit return statement, a function returns undefined.
- For example, the following code defines a simple function named `calcRectArea`.

Program 2.1: Write a program to calculate area of rectangle using function.

```
function calcRectArea(width, height) {
  return width * height;
}

//invoking the function and printing the result
console.log("Area ::" + calcRectArea(7,3));
```

Output:

```
C:\BCA>node Funcl.js
Area :: 21
```

- Let us see another example, where function is not returning any value. But we know that even though not specified, every function returns the value called `undefined`.

Program 2.2: Program to create a function which is not returned any value.

```
function Display(){
  console.log("Hello World !\n");
}
console.log("Value returned : " + Display());
```

Output:

```
C:\BCA>node Func2.js
Hello world !
Value returned : undefined
```

- Primitive parameters (such as a number) are passed to functions by value. The value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function.
- If we pass an object (i.e. a non-primitive value, such as array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following program.

Program 2.3: Program to show how to pass an object as a parameter in the function.

```
function ShowData(obj) {
    obj.name='Janardan Pawar';
}
var stud = {roll: 101, name: 'Janardan', percentage:72.45};
var old_name, modified_name;

console.log ("Before Function Call:");
console.log(stud);
old_name = stud.name; // old_name gets the value "Janardan'

ShowData(stud);
Console.log("\n After Function Call:");
Modified_name = stud.name; // modified name gets the value 'Janardan Pawar'

console.log(stud);
```

Output:

```
C:\BCA>node Func3.js
Before Function Call:
{ roll: 101, name: 'Janardan, percentage: 72.45 }
After Function Call:
{ roll: 101, name: 'Janardan Pawar', percentage: 72.45 }
```

2.2.2 Function Expressions (Anonymous Function) [S-22, S-23]

- **Function Expression** allows us to create an anonymous function which doesn't have any function name. This is the main difference between Function Expression and Function Declaration.
- A function expression has to be stored in a variable and can be accessed using *variableName*.
- For example, the function `calcRectArea` could have been defined as:

Program 2.4: Program for anonymous function.

```
const calcRectArea = function(width, height) { return width * height}
var result = calcRectArea(7, 3)
console.log("Value of result = " + result);
```

Output:

```
C:BCA>node Func4.js
Value of result = 21
```

2.2.3 Function Scope

- JavaScript permits us to define a function in other function, which is nested function. Just like any other programming languages, JavaScript has local and global variables.
- Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.
- In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.
- Following program shows accessibility of the variables (local and global).

Program 2.5: Program for local and global variables.

```
// global scope
var roll = 101,
    name = 'Janardan',
    phy = 67,
    che = 72,
    maths = 65;

function totalMarks() {
   return phy+che+maths;
}
console.log("Total Marks Secured = " + totalMarks()); // Returns value 204

// A nested function example
function getScore() {
   var mar = 85,
       sub = 'Marathi';      //local scope

   function showMarks() { //this is a nested function inside the getScore()
```

```
        return name + ' scored ' + mar + ' in the subject ' + sub;
    }
    return showMarks(); //invoking the nested function
}
console.log(getScore()); //invoking the global function
```

Output:

```
C:\BCA>node Func5.js
Total Marks secured = 204
Janardan scored 85 in the subject Marathi
```

Function with Default Arguments:

- Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

Program 2.6: Program for function with default arguments.

```
function multiply(num1, num2 = 1) {
return num1 * num2;
}
console.log(multiply(7, 3));
console.log(multiply(8));
```

Output:

```
21
8
```

Solved Programs Based on Functions

Program 2.7: Write a Node JS script to accept radius from the user and calculate area of the Circle. [S-22]

```
const prompt = require('prompt-sync')();
function calcArea(radius) {
    return 3.14 * radius * radius ;
}
var rad = prompt('Enter radius of the Circle : ');
var result = calcArea(rad);
console.log("Area of the Circle having radius " + rad + " is " + result );
```

Output:

```
C:\BCA>node Area_circle.js
Enter radius of the circle : 3.42
Area of the Circle having radius 3.42 is 36.726696
```

Program 2.8: Write a Node JS script to check a given number is even or odd using the function. [S-22]

```
const prompt = require('prompt-sync')();
function oddEven(num) {
```

```
    return (num%2==0)? "Even" : "Odd";
}
var n = prompt('Enter a number : ');
console.log("The given number is " + oddEven(n));
```

Output:

```
C:\BCA>node odd_Even.js
Enter a number : 5
The given number is odd

C:\BCA>node odd_Even.js
Enter a number : 26
The given number is Even
```

Program 2.9: Write Node JS program that accepts principle, rate of interest, time and compute the simple interest.

```
const prompt = require('prompt-sync')();
function calSimpleInterest(p,n,r) {
    var SI = (p*n*r)/100;
    return SI;
}
const p_amt = prompt('Enter Principle Amount : ');
const roi = prompt('Enter Rate of Interest :');
const time = prompt('Enter duration/time :');
var answer = calSimpleInterest(p_amt,roi,time);
console.log("Simple Interest = " + answer);
```

Output:

```
C:\BCA>node simple_interest.js
Enter Principle Amount : 10000
Enter Rate of Interest :10
Enter duration/time :12
Simple Interest = 12000
```

Program 2.10: Write a Node JS script to check a given number is perfect or not using function. [S-22]

```
const prompt = require('prompt-sync')();
function isPerfectNumber(num) {
    var sum = 0;
    for(var i=1;i<=num/2;i++){
        if(num % i === 0)
            sum += i;
    }
    if(sum == num && sum != 0)
```

```
        console.log("The given number " + num + " is a perfect number.");
    else
        console.log("The given number " + num + " is Not a perfect number.");
}
var n = prompt('Enter a number : ');
isPerfectNumber(n);
```

Output:

```
C:\BCA>node Perfect_Number.js
Enter a number : 28
The given number 28 is a perfect number.

C:\BCA>node Perfect_Number.js
Enter a number : 35
The given number 35 is Not a perfect number.
```

Program 2.11: Write a Node JS program to calculate factorial of given number using function. [W-22]

```
const prompt = require('prompt-sync')();
function factorial(n) {
    var answer = i;
    if(n<0)
        return ' not possible(Negative number)';
    else if (n == 0 || n == 1)
        return answer;
    else {
        for(var i = n; i >= 1; i--) {
            answer = answer * i;
        }
      return answer;
    }
}
const n = prompt('Enter a number : ');
answer = factorial(n)
console.log("The factorial of " + n + " is " + answer);
```

Output:

```
C:\BCA>node Factorial.js
Enter a number : 5
The factorial of 5 is 120

C:\BCA>node Factorial.js
Enter a number : 0
```

```
The factorial of 0 is 1

C:\BCA>node Factorial.js
Enter a number : -2
The factorial of -2 is not possible(Negative number)
```

2.3 BUFFER [W-22]

- Buffer is an object property on Node's global object, which is heavily used in Node to deal with streams of binary data. As it is globally available, there is no need to require it in our code.
- Buffer is actually a chunk of memory allocated outside of the V8 heap. V8 is the default JavaScript engine which powers Node and Google Chrome. In Node, buffers are implemented using a JavaScript typedArray (Uint8Array), but that does not mean the memory allocated to buffer is inside of the V8 heap. It is still explicitly allocated outside the V8 heap.
- So, we can think of buffer as some kind of array which is a lower level data structure to represent a sequence of binary data, but there is one major difference: Unlike arrays, once a buffer is allocated, it cannot be resized.

2.3.1 Why we use Buffer? [S-22]

- Pure JavaScript works well with Unicode-encoded strings. In fact, it is the Unicode in your browser that states that 76 should represent L. Pure JavaScript do not handle straight binary data very well; this is fine in browsers where most data is in the form of strings. However, Node JS have to face with reading and writing to the file system and TCP streams thus it makes it necessary to deal with purely binary streams of data.
- Node has a way to handle binary data, Buffer class is the primary data structure in a node and used with most I/O operations. In node, each buffer represents to some raw memory allocated outside V8. A buffer acts like an array of integers, once allocated it cannot be resized.
- Buffers act somewhat like arrays of integers, but are not resizable and have a whole bunch of methods specifically for binary data. The "integers" in buffer represents a byte and they are limited to values from 0 to 255 (2^8-1).

2.3.2 How to create Buffers? [S-22]

- Below mentioned methods of Buffer can be used to create buffers.

1. `Buffer.from()`:This method is used to generate a buffer from a string, object, array or buffer.

   Syntax: `Buffer.from(obj, encoding);`

   Where,

`Obj`: Object by which should be filled in, Different types that we can fill are (String,Array,Buffer,arrayBuffer)

`Encoding`: The encoding of string. Default value is utf8. This is an optional parameter.

2. `Buffer.alloc()`:It takes a size (integer) as an argument and returns a new initialized buffer of the specified size (i.e., it creates a filled buffer of a certain size).

   Syntax: `Buffer.alloc(size, fill, encoding);`

   Where,

   `size`: Desired length of new Buffer. It accepts integer type of data.

   `fill`: The value to prefill the buffer. The default value is 0.It accepts any of the following: integer, string, buffer type of data.

   `encoding`: It is Optional. If buffer values are string , default encoding type is utf8.

3. `Buffer.allocUnsafe()`:This method creates new buffer object of the specified size but it will not initialize the values thus it is the no-prefill buffer. The segment of allocated memory is uninitialized; an allocated segment of memory might contain sensitive or confidential data from older buffers.

   Syntax: `Buffer.allocUnsafe(size);`

   Where, `Size` : Desired length of new Buffer. It accepts integer type of data.

Program 2.12: Program creates a zero-filled buffer of the specified length.

```
var MyBuffer1 = Buffer.alloc(15);
console.log(MyBuffer1);
```

Output:

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

- Here we have allocated the memory of 20 bytes or size of 20 bytes to buffer object. Here we have not specified the fill value thus by default it will take zero value in hexadecimal format.

Program 2.13: Program creates buffer from a given array.

```
var MyBuffer2 = Buffer.from([ 8, 6, 0, 5, 4, 12, 9, 6]);
console.log(MyBuffer2);
```

Output:

```
<Buffer 08 06 00 05 04 0c 09 06>
```

- Here, this initializes the buffer to the contents of this array. Keep in mind that the contents of the array are integers representing the bytes.
- See following code, will initialize the buffer to a binary encoding of the first string as specified by the second argument (in this case, 'utf-8'). 'utf-8' is by far the most common encoding used with Node JS.

Program 2.14: Program to initialize the buffer to a binary encoding.

```
var MyBuffer3 = Buffer.from("Indira College", "utf-8");
console.log(MyBuffer3);
```

Output:

```
<Buffer 49 6e 64 69 72 61 20 43 6f 6c 6c 65 67 65>
```

2.3.3 Writing to Buffers

- Till now, we have seen few ways to create the buffer in Node JS. Now, let us see how to write on to the buffers.
- **Syntax:**

```
BufferName.write(string, offset, length, encoding)
```

Where,

`String`: This is the string data to be written to buffer.

`Offset`: This is the index of the buffer to start writing at. Default value is 0.

`Length`: This is the number of bytes to write. Defaults to buffer.length.

`Encoding`: Encoding to use. 'utf8' is the default encoding.

Return Value

- This write() method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.
- Let us explore it a bit using Command Prompt of Node JS.

```
C:\BCA>node
Welcome to Node.js v14.15.1.
Type ".help" for more information.
> var MyBuffer = Buffer.alloc(20);
undefined
> MyBuffer.write("Hello World!",'utf-8');
12
```

- Here, we have a created a buffer of 20 bytes. MyBuffer.write() has returned the value 12. This means that we wrote to 12 bytes of data of the buffer.
- Now, let us consider following snippet. When `MyBuffer.write()` has 3 arguments, the second argument indicates an offset, or the index of the buffer to start writing at.

```
MyBuffer.write("Indira",6,'utf-8');
6
```

2.3.4 Reading from Buffers

- Probably the most common way to read buffers is to use the `toString()` method, since many buffers contain text.

- **Syntax** : `buf.toString(encoding, start, end)`

  Where,

  Encoding: This is encoding type to be used. The 'utf8' is the default encoding type.

  Start: This denotes the beginning index to start reading. The defaults value is 0.

  End: This denotes the end index to end reading. The default value is the complete buffer.

- Let us have a look at following snippet.

```
> MyBuffer.toString('utf-8');
'Hello Indira\x00\x00\x00\x00\x00\x00\x00\x00'
```

- Again, the first argument is the encoding. In this case, it can be seen that not the entire buffer was used.
- As we know, how many bytes we've written to the buffer, we can simply add more arguments to "stringify" the slice. Following code will read the contents of the buffer named `MyBuffer` from index 0 to 12.

```
> myBuffer.toString('utf-8, 0, 12);
'Hello Indira'
```

- There are so many methods and properties which belong to buffer. Few of them are used in the code given below.
- Let us see, entire command prompt of the above mentioned methods and properties of the buffer.

```
Type ".help" for more information.
> var MyBuffer = Buffer.alloc(20);
undefined
> MyBuffer.write("Hello World!",'utf-8');
12
> MyBuffer.write("Indira",6,'utf-8');
6
> MyBuffer.toString('utf-8');
'Hello Indira\x00\x00\x00\x00\x00\x00\x00\x00'
> MyBuffer.toString('utf-8',0, 12);
'Hello Indira'
> Buffer.isBuffer(MyBuffer);
true
> MyBuffer.length
20
> MyBuffer.slice(0,5);
<Buffer 48 65 6c 6c 6f>
> const data = MyBuffer.slice(0,5);
undefined
> data.toString()
'Hello'
>
```

2.3.5 Concatenate Buffers [S-22]

- Following Syntax is used to concatenate Node buffers to a single Node Buffer.

  Syntax : Buffer.concat(list, Length)

  Where,

  `List`: This is the array List of the Buffer objects which are to be concatenated.

  `Length`: Here it denotes the total length of the buffers when they are concatenated, it is optional.

Program 2.15: Program for concatenation of buffers.

```
var buffer1 =  Buffer.from('Hi ');
var buffer2 =  Buffer.from('Nirali Publication');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 content are: " + buffer3.toString());
```

Output: `buffer3 content are: Hi Nirali Publication`

2.3.6 Compare Buffers

- The compare() method compares two buffer objects and it returns a number according to their differences
- Following Syntax is used to compare two node Buffers

  Syntax : `buf.compare(buffer1,buffer2) ; buffer1.compare(buffer2)`
- **Note:** This method compares two buffer objects and returns a number defining their differences:

  Shows 0 if they are equal

  Shows 1 if buffer1 is higher than buffer2

  Shows -1 if buffer1 is lower than buffer2

Program 2.16: Program for comparing buffers.

```
var buffer1 = Buffer.from('Nirali');
var buffer2 = Buffer.from('Publication Pune');
var result = buffer1.compare(buffer2);

console.log("Value returned is : " + result);
if(result < 0) {
  console.log(buffer1 +" lower than " + buffer2);
}
else if(result === 0){
  console.log(buffer1 +" is same as " + buffer2);
}
```

```
    else {
      console.log(buffer1 +" higher than " + buffer2);
    }
```

Output:

```
Value returned is : -1
Nirali lower than Publication Pune
```

2.4 MODULE [S-22]

- In Node JS, a module is a collection of JavaScript functions and objects that can be used by external applications. It is a set of functions which we would like to include in our application.
- Module in Node JS is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node JS application.
- Collection of files can be considered as a module if its functions and data are made usable to external programs.
- In Node, the modularity is a first class concept. In the Node JS module system, each file is treated as a separate module.
- So, if you are creating, let's say, a `demo.js` file, this implies you are creating a module in Node. Basically modules help us encapsulating our code into manageable chunks.
- Anything that we define in our module (i.e. in our JavaScript file) remains limited to that module only, unless we want to expose it to other parts of our code.
- So, anything we define inside our module remains private to that module only.

2.5 MODULE TYPES [S-22]

- Node JS includes three types of modules:

1. Core Modules, 2. Local Modules, 3. Third Party Modules

2.5.1 Core Modules [W-22, S-23]

- Core Modules in Node JS are also called as built-in modules which we can use without any further installation. The core modules include basic minimum functionalities of Node JS. These core modules are compiled into its binary distribution and load automatically when Node JS process starts.
- Below mentioned are the few Core Modules in Node JS:

Module Name	Description
`http`	This module includes classes, methods and events to create Node JS http server.

Contd...

path	This module includes methods to deal with file paths.
util	This module includes utility functions useful for programmers.
fs	This module includes classes, methods, and events to work with file I/O.
timers	This module is used to execute a function after a given number of milliseconds.
zlib	This module is used to compress or decompress files.
url	This module includes methods for URL resolution and parsing.
assert	It is used to Provides a set of assertion tests.
buffer	It is used to handle binary data.
timers	It is used to execute a function after a given number of milliseconds.

How to load core modules in Node JS Application?

- We need to import the core module first in order to use it in our application. If we unable to load any module, then we cannot use functionality of the modules. Means, we cannot access the objects and functions available in the modules.
- To include the required module in our application is very easy. We just need to make use of `require()` function as:

  Syntax:

```
var variable_name = require('Module_Name');
```

- We can also use **const** instead of **var** here.
- **Return value:** The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.
- **For Example:**

```
const absolutePath = require('path');
```

- The path module provides a lot of very useful functionality to access and interact with the file system. There is no need to install it. Being part of the Node JS core, it can be used by simply requiring it.

Methods available in the path module: W-22, S-23

1. **basename():** It return the last portion of a path. A second parameter can filter out the file extension.

Program 2.17: Program for basename() method.

```
const absolutePath = require('path');

const fileNamel = absolutePath.basename('D:/Study_Data/NodeJS/Notes.docx');
console.log(fileNamel);

const fileName2=
absolutePath.basenase('D:/Study_Data/NodeJS/Notes.docx','.docx');
console.log(fileName2);
```

Output:

```
C:\BCA>node ModuleDemo.js
Notes.docx
Notes
```

- If you are the Windows OS user, then you can use the path as given below.

```
'D:\\Study_Data\\Node JS\\Notes.docx'
```

2. dirname(): It returns the directory part of the path.

Program 2.18: Program for dirname() method.

```
const absolutePath = require('path');

const dir = absolutePath.dirname('D:/Study_Data/Node JS/Notes.docx');
console.log(dir);
```

Output:

```
C:\BCA>node ModuleDemo2.js
D:/Study_Data/Node JS
```

3. extname(): It returns the extension part of a path.

Program 2.19: Program for extname() method.

```
const absolutePath = require('path');

const ext = absolutePath.extname('D:/Study_Data/Node JS/Notes.docx');
console.log("File Extension : " + ext);
```

Output:

```
C:\BCA>node ModuleDemo3.js
File Extension: .docx
```

4. **isAbsolute()**: It returns true if it's an absolute path.

Program 2.20: Program for isAbsolute() method.

```
const absolutePath = require('path');

const ans1 = absolutePath.isAbsolute('D:/Study_Data/Node JS/Notes.docx');
console.log("Answer : " ans1);

const ans2 = absolutePath.isAbsolute('Study_Data/Node JS/Notes.docx');
console.log("Answer : " ans2);
```

Output:

```
C:\BCA>node MuduleDemo4.js
Answer : true
Answer : false
```

- Other methods which are available under path module are join(), normalize(), parse(), relative(), etc.

2.5.2 Local Modules [S-22]

- Local modules are the custom modules defined and developed by the user i.e. coder. Hence, it is also called as User Defined Module. These modules are created locally in our Node JS application. These modules include different functionalities of our application in separate files and folders.

Writing Local Module:

- Creating our own customized module is very easy and consisting of following simple steps.
- Let us write simple module which perform arithmetic operations like addition, subtraction, multiplication and division of any two numbers.
- In Node JS, module should be placed in a separate JavaScript file. So, first, create a file and write the following code in it. Save this file with the name **userDefinedModule.js**.

```
//.js file containing functions which are performing arithmetic operations

// Returns addition of two numbers
exports.add = function (num1, num2) {
   return num1+num2;
};

// Returns difference of two numbers
```

```
exports.subtract = function (num1, num2) {
    return num1-num2;
};

// Returns multiplication of two numbers
exports.multiply = function (num1, num2) {
    return num1*num2;
};

// Returns division of two numbers
exports.divide = function (num1, num2) {
    return num1/num2;
};
```

- Note that, the functions we add to the custom module must be defined with "exports" in front of the function name.
- The "exports" keyword is used to ensure that the functionality defined in this file can actually be accessed by other files.

Loading Local Module:

- Here, we will actually see how to create the application which will call our custom module.
- To use local modules in our application, we need to load it using require() function in the same way as core module. However, we need to specify the path of JavaScript file of the module.

MainApplication.js

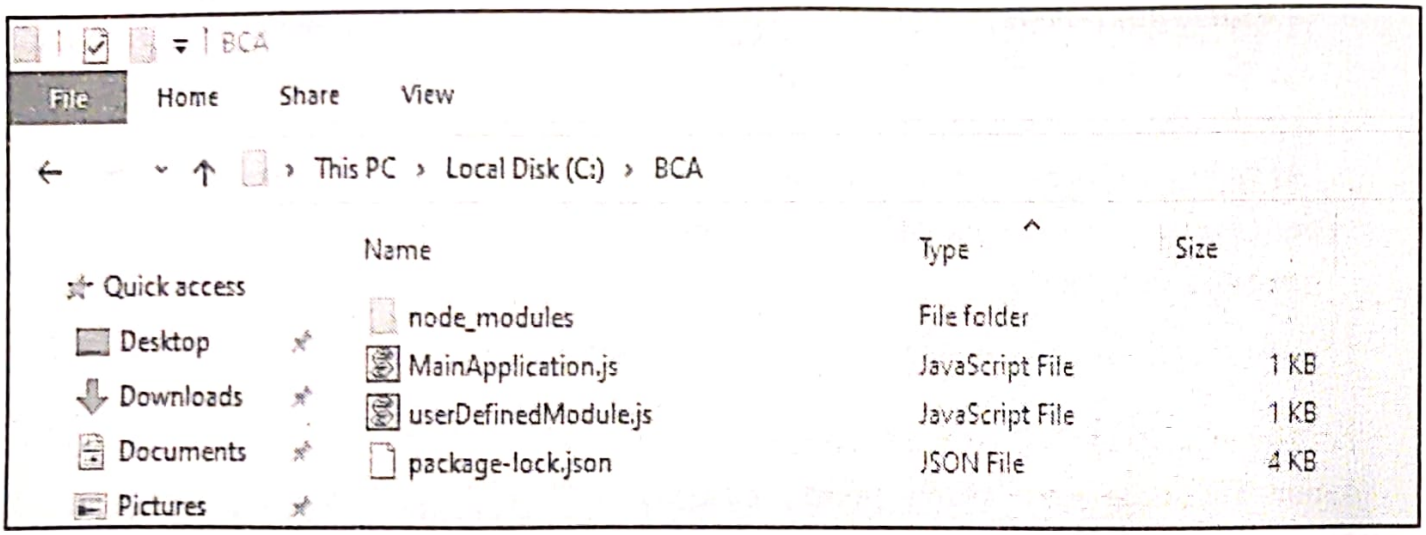
```
//Accessing user defined (custom module) named userDefinedModule

var op = require('./userDefinedModule');

var x=25, y=5;

console.log("Arithmetic Operations on ", x , "and ", y , "are :\n");
console.log("Addition : "+ op.add(x,y));
console.log("Subtraction : "+ op.subtract(x,y));
console.log("Multiplication : "+ op.multiply(x,y));
console.log("Division : "+ op.divide(x,y));
```

- Following image shows the files that we have created.



- When we execute our main application, we will get following output:

```
C:\BCA>node MainApplication.js
Arithmetic Operations on 25 and 5 are :
Addition : 30
Subtraction : 20
Multiplication : 125
Division : 5
```

2.5.3 Third Party Modules [S-22]

- **Third-party modules:** Third-party modules are modules that are available online using the Node Package Manager (NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are mongoose, express, angular, and react.

2.6 MODULE.EXPORTS [W-22, S-23]

- Now, we will see how to expose different types as a module using `module.exports`.
- The `module.exports` is a special object which is included in every JavaScript file in the Node JS application by default. The `module` is a variable that represents the current module, and `exports` is an object that will be exposed as a module. So, whatever we assign to `module.exports` will be exposed as a module.

Export Literals:

- `exports` is an object. So it exposes whatever we assigned to it as a module.
- For example, if we assign a string literal then it will expose that string literal as a module.
- Let us see this using simple program.

Program 2.21: Program for export literals.

Greetings.js

```
module.exports = 'Welcome to Indira College of Commerce & Science';
```

mainApp.js

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

Output:

```
C:\BCA>node mainApp.js
Welcome to Indira College of Commerce & science
```

- In above program, we add simple and one line code in the file `Greetings.js`. After that import this module in the file named `mainApp.js` and execute it to see the output.

Export Object:

- The `export` is an object. So, you can attach properties or methods to it.

Program 2.22: Program for export object.

Notices.js

```
exports.notice1 = 'Beware of COVID19.';
//we can also write it as following
module.exports.notice2 = 'Take Necessary Precautionary Measures.';
```

mainApp.js

```
const message = require('./Notices.js');
console.log(message.notice1);
console.log(message.notice2);
```

Output:

```
C:\BCA>node mainApp.js
Beware of COVID19.
Take Necessary Precautionary Measures.
```

- In above program, exposes an object with a string property in `Notices.js` file. Then import the module `Notices` in `mainApp.js` file and execute it to see the output.

Exposing an object with function using exports:

- Example given below exposes an object with the myFun function as a module.
- Import the module `ObjectFunction` in `mainApp.js` file and execute it to see the output.

Program 2.23: Program for exposing an object with function using exports.

ObjectFunction.js

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

mainApp.js

```
var data = require('./ObjectFunction.js');
data.myFun('Welcome to Indira College of Commerce & Science, Pune(MH)');
```

Output:

```
C:\BCA>node mainApp.js
Welcome to Indira College of Commerce & Science, Pune(MH)
```

Summary

- Functions are first class citizens in Node's JavaScript, similar to the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.
- Buffers are instances of the Buffer class in Node.js. Buffers are designed to handle binary raw data. Buffers allocate raw memory outside the V8 heap. Buffer class is a global class so it can be used without importing the Buffer module in an application.
- In Node.js, Modules are the blocks of encapsulated code that communicates with an external application on the basis of their related functionality. Modules can be a single file or a collection of multiples files/folders.
- The reason programmers are heavily reliant on modules is because of their re usability as well as the ability to break down a complex piece of code into manageable chunks.
- Modules are of three types: Core Modules, local Modules, Third-party Modules.
- **Core Modules:** Node.js has many built-in modules that are part of the platform and comes with Node.js installation. These modules can be loaded into the program by using the require function.
- **Local Modules:** Unlike built-in and external modules, local modules are created locally in your Node.js application.
- **Third-party modules:** Third-party modules are modules that are available online using the Node Package Manager(NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are mongoose, express, angular, and react.

Check Your Understanding

1. Which module is used to serve static resources in Node JS?
   (a) static (b) node-resource
   (c) http (d) node-static
2. How Node JS modules are available externally?
   (a) module.spread (b) module.exports
   (c) module.expose (d) None of the above

3. Which of the following module is required for path specific operations?
   (a) Os module (b) Path module
   (c) Fs module (d) All of above
4. Simple or complex functionality organized in a single or multiple JavaScript files which can be reused throughout your Node JS application is called _____.
   (a) Library (b) Package
   (c) Function (d) Module
5. Which is/are the core module(s) in Node.js?
   (a) http (b) url
   (c) path (d) all of the above
6. Which function is used to include modules in Node Js.
   (a) Include() (b) Require()
   (c) Attach() (d) None of the above
7. Which of following is not built-in node module?
   (a) Zlib (b) https
   (c) dgram (d) fsread
8. What does the fs module stand for?
   (a) File Service (b) File System
   (c) File Store (d) None of the above

ANSWER KEY

1. (d)	2. (b)	3. (b)	4. (d)	5. (d)
6. (b)	7. (d)	8. (b)		

Practice Questions

Q.I: Answer the following questions in short.

1. What is function in node JS?
2. Define Anonymous function?
3. List out different core module?
4. Write different types of Node JS Modules?
5. Which module is used for file based operations?
6. Define Scope of functions?
7. Write syntax to create Buffer?
8. How to read and write buffer, Write Syntax?

Q.II: Answer the following questions.

1. Explain Modules in Node JS.
2. Explain module.exports in Node JS?
3. What are some of the most popular modules of Node.js?
4. What is function? Write syntax to define function.
5. Write Steps to load Core modules in Node JS?
6. Write Steps to load Local modules in Node JS?

Q.III: Define the terms.

1. Modules
2. Function
3. Core Module
4. Local Module
5. Module.export
6. Buffer