Node

Cmd to check if node is installed

$\boxed{\text{node -v}}$

⇒ Download → install → node.js

⇒ VS code setup

to run a node file use 'node filename'
in Cmd Prompt

global scope → like window in browser

Path → Core module to work with file and dir paths

Path.basename(s) → filename

Path.dirname(s) → directory name

Path.extname → gives extension

Path.Parse() → gives all of above

1) Node Js vs Vanilla JS

⇒ Node runs on a server

⇒ The console is the terminal window

⇒ global object not window obj

⇒ has some common core modules

⇒ Common Js modules instead of ES6

import is done using require() ⇒ Some APIS X

```
console.log(os.type())
console.log(os.version())
console.log(os.homedir())
```

to access file name — dir name
for file name — file name

we can creat our own modules as well

① way
module exports = { object of functions }

② nd way
exports.add (a, b) => a+b

File Systems

Const fs = require('fs')

read file

```
fs.readfile ('Path', (err, data) => {
    if (err) throw err
    Console.log ( data.toString())
})
```

↳ returns Buffer vals

(or)

add encoding like 'utf8' Param before (err, data)

Process can enable us to interact during Process of Program

By:- We can catch exceptions.

```
Process.on (uncaughtexpressionerr, => {
    console.log ('The error is' + err)
    Process.exit (1);
}
```

accessing Paths like ("...\...\...") is bad because various conversions use various ways to identify Paths like some \ (or) /

so better to use Path module

```
fs.readfile (Path.join (__dirname, 'files', 'Starter.txt'))
```

~~ref~~ <u>write file</u> =>

```
fs.writefile ( 'Path', 'text to write', (err) => {
    'handle error'
});
```

appendFile

=> appends text to a file. If file does not exist then creates one.

fs.appendFile('path', 'text to append', (err) => {

});

rename) →

fs.rename('Path', 'newPath', (err) => {})

to use a proper flow in fs we might right functions in nested way that becomes a call back hell

to avoid this we can use fs as promises

```
Const fs = require('fs').promises
```

```
const fsPromises = require('fs').promises;  // Import promise-based fs module
const path = require('path');  // Import path module

const fileOps = async () => {
  try {
    // Read the contents of 'starter.txt'
    const data = await fsPromises.readFile(path.join(__dirname, 'files', 'starter.txt'), 'utf8');
    console.log(data);

    // Write the content to 'promisewrite.txt'
    await fsPromises.writeFile(path.join(__dirname, 'files', 'promisewrite.txt'), data);

    // Append additional text to 'promisewrite.txt'
    await fsPromises.appendFile(path.join(__dirname, 'files', 'promisewrite.txt'), '\n\nNice to meet you.');

    // Rename the file from 'promisewrite.txt' to 'promiseComplete.txt'
    await fsPromises.rename(
      path.join(__dirname, 'files', 'promisewrite.txt'),
      path.join(__dirname, 'files', 'promiseComplete.txt')
    );

    console.log('File operations completed successfully!');
  } catch (err) {
    console.error('Error during file operations:', err);
  }
};

// Call the async function
fileOps();
```

delete / unlink

```
await fsPromises.unlink(path.join(__dirname, 'files', 'starter.txt'));  // Delete 'starter.txt'
```

Streams → used to read or write data chunk by chunk so that we dont have to worry about working with large files

in readFile and writeFile we read or write entire data so space issues might arise for large files

```
Const fs = require('fs')
Const rs = fs.CreateReadStream('Path', {encoding: 'Utf8'})

Const ws = fs.CreateWriteStream('Path')
rs.on('data', (data chunk) => {
    ws.write(data chunk);
})
```

(or) rs.pipe(ws)

On → function that listens to an event and Performs a call back.

Pipe(): Methode for streaming data

Connects Readable stream to writable Stream allowing you to transfer data from 1 source to another