Node.js REPL -

1. What is REPL?

- **REPL** stands for **Read–Eval–Print–Loop**.
- It's an interactive programming environment that comes with Node.js by default.
- Lets you run JavaScript code **line-by-line** directly from the terminal.

2. Meaning of REPL steps:

- **Read** → Reads user's input (JavaScript code).
- **Eval** → Evaluates the input using the V8 JavaScript engine.
- **Print** → Prints the result of the evaluation.
- **Loop** → Waits for the next command and repeats the cycle.

3. How to start REPL:

node

- Just type node in your terminal and press Enter.

4. Features of REPL:

- **Run JS code directly:**
- > 2 + 3
- 5
- **Use variables:**
- > let x = 10
- > x * 2
- 20
- **Multiline code:**
  Press Enter after each line until you complete the block.
- > if (true) {
- ... console.log("Hello");
- ... }
- Hello
- ***underscore ()* variable_** → stores the last result:
- > 5 + 5
- 10
- > _ * 2
- 20
- **Load/Save files:**
  - .load filename.js → run a JS file in REPL
  - .save filename.js → save REPL session to a file
- **Exit REPL:**
  - Press **Ctrl + C twice** or type .exit

5. Common REPL commands:

Command	Description
.help	Show all available commands
.break	Exit from multiline expression
.clear	Clear the REPL context
.save	Save session to file
.load	Load file into REPL
.exit	Exit REPL

6. Use cases:

- Quick code testing.
- Trying out Node.js APIs.
- Debugging small snippets.

Short Notes – Node REPL

REPL = Read, Eval, Print, Loop.

Starts with `node` in terminal.

Steps: Read (input) → Eval (execute) → Print (result) → Loop (repeat).

Features: run JS, store last result in `_`, multiline support, `.save` / `.load`.

Exit: Ctrl+C twice or `.exit`.

Use for quick testing, API tryouts, debugging small code.

window vs global in Node.js -

1. window object:

- Exists in **browsers only**.
- Represents the **global object** in client-side JavaScript.
- Contains browser-specific APIs like document, alert(), localStorage, etc.
- Example (in browser console):
- window.alert("Hello");
- console.log(window.location.href);
- In Node.js → **window is not defined** because Node doesn't have browser APIs.

2. global object in Node.js:

- Node.js equivalent of window.
- Represents the **global namespace** in Node environment.
- Contains Node-specific globals like:
  - process → info about the running process
  - __dirname → current directory
  - __filename → current file path
  - setTimeout(), setInterval()
- Example:
- console.log(global.process.version);
- global.myVar = 100;
- console.log(myVar); // 100

3. Key Differences Table:

Feature	window (Browser)	global (Node.js)
Environment	Browser only	Node.js only
Scope	Global scope in browser JS	Global scope in Node.js
Browser APIs	Yes (DOM, alert, fetch, etc.)	No browser APIs
Node APIs	No	Yes (process, __dirname, etc.)
Access method	Implicit or window.property	Implicit or global.property

4. Extra note:

- In modern JS (ES2020+), globalThis works in **both** browser and Node.js as a universal global object.
- console.log(globalThis === window); // true in browser

- console.log(globalThis === global); // true in Node.js

Short Notes – window vs global

window → browser global object, has browser APIs (DOM, alert, fetch).

global → Node.js global object, has Node APIs (process, __dirname).

window not available in Node, global not available in browser.

globalThis → universal global object (works in both environments).

Node.js Module -

1. What is a Module?

- A **module** in Node.js is a **reusable block of code** that is separated into its own file.
- Helps in organizing code into smaller, manageable parts.
- Node.js follows the **CommonJS module system** (by default).

2. Types of Modules in Node.js:

1. **Core (Built-in) Modules** – Provided by Node.js (no installation needed).
   Examples:
   - fs → File System
   - path → Path utilities
   - http → HTTP server
   - os → OS information
2. const fs = require('fs');
3. **Local Modules** – Your own custom files.
4. // math.js
5. function add(a, b) { return a + b; }
6. module.exports = add;
7.
8. // app.js
9. const add = require('./math');
10. console.log(add(2, 3)); // 5
11. **Third-party Modules** – Installed via **npm**.
12. npm install lodash
13. const _ = require('lodash');
14. console.log(_.random(1, 10));

3. Module Import/Export:

- **Exporting:**
- module.exports = { functionName, variableName };
- **Importing:**
- const myModule = require('./moduleName');

4. Special Module Variables in Node.js:

- __filename → Full path of the current file.
- __dirname → Directory name of the current module.
- module → Information about the current module.
- exports → Shortcut to export values.

5. Benefits of Modules:

- Reusability
- Maintainability
- Separation of concerns
- Avoids global variable conflicts

Short Notes – Node.js Module

Module = reusable block of code in Node.js.

Types: Core (fs, path, http), Local (your files), Third-party (via npm).

Export → module.exports, Import → require().

Special vars: __filename, __dirname, module, exports.

Benefits: reusable, organized, maintainable.

Multiple Exports in Node.js -

1. What is Multiple Export?

- In Node.js, you can export **more than one function, object, or variable** from the same module so other files can use them.

2. Ways to Do Multiple Exports:

A. Using module.exports as an object

// math.js

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

function subtract(a, b) { return a - b; }


module.exports = { add, subtract };

// app.js

const math = require('./math');

console.log(math.add(5, 3));      // 8

console.log(math.subtract(5, 3)); // 2
```

B. Exporting individually using exports

```
// math.js

exports.add = (a, b) => a + b;

exports.subtract = (a, b) => a - b;

// app.js

const { add, subtract } = require('./math');

console.log(add(5, 3));      // 8

console.log(subtract(5, 3)); // 2
```

3. Things to Remember:

- exports is a **shortcut** to module.exports.
- If you reassign module.exports directly, exports is ignored.
- exports = { a: 1 }; // ❌ won't work as expected
- module.exports = { a: 1 }; // ✅ works
- You can mix functions, objects, and variables in the same export.

4. Example with mixed exports:

```
// utils.js

const name = "Node.js";

function greet() { return `Hello from ${name}`; }

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

```
module.exports = { name, greet, sum };

// app.js

const { name, greet, sum } = require('./utils');

console.log(name);

console.log(greet());

console.log(sum(2, 3));
```

Short Notes – Multiple Exports

Multiple exports = exporting more than one item from a Node.js module.

Ways:

1. module.exports = { fn1, fn2 }

2. exports.fn1 = ..., exports.fn2 = ...

exports is a shortcut to module.exports (don’t reassign exports directly).

Can export functions, variables, objects together.

Named & Aggregate Export -

Note:
These terms are more common in **ES Modules** (.mjs or "type": "module" in package.json) rather than CommonJS (require/module.exports).
But Node.js supports them in modern versions.

1. Named Export

- Allows you to **export multiple variables, functions, or classes by name**.
- Importers must use **the exact same name** when importing.

Example:

```
// math.mjs

export const PI = 3.14;
```

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

export function subtract(a, b) { return a - b; }

// app.mjs

import { PI, add, subtract } from './math.mjs';


console.log(PI);

console.log(add(5, 3));

console.log(subtract(5, 3));
```

✅ You can export multiple things from the same file.
✅ Import only what you need:

```
import { add } from './math.mjs';
```

2. Aggregate Export

- Allows you to **re-export** things from another module without importing them first.
- Useful for **index.js** or **barrel files** that centralize exports.

Example:

```
// math.mjs

export const PI = 3.14;

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

// stringUtils.mjs

export function upper(str) { return str.toUpperCase(); }

// index.mjs (aggregate export)

export * from './math.mjs';

export * from './stringUtils.mjs';

// app.mjs

import { PI, add, upper } from './index.mjs';


console.log(PI);
```

console.log(add(2, 3));

console.log(upper("node"));

3. Key Points Table:

Feature	Named Export	Aggregate Export
Purpose	Export multiple items by name	Re-export from another module
Syntax	export { item1, item2 }	export * from './module.js'
Import syntax	{ itemName }	{ itemName } (same as named)
Use case	Direct export from file	Centralizing exports in one place

Short Notes – Named & Aggregate Export

Named Export → export multiple items by name, import with same names.

Syntax:

export const x = ...;

export function y() {};

import { x, y } from './file.js';

Aggregate Export → re-export from another module without importing first.

Syntax:

export * from './module.js';

Use for centralizing exports in index.js.

Node.js path Module -

1. What is the path module?

- A **built-in Node.js core module** for working with file and directory paths.
- Helps handle path differences between **Windows (\)** and **Unix (/)** systems.
- No installation needed – just require('path').

2. Importing the path module:

const path = require('path');

3. Commonly Used Methods & Properties:

Method / Property	Description	Example
path.basename(path)	Returns the last part of a path (file name).	path.basename('/folder/file.txt') → "file.txt"
path.dirname(path)	Returns directory part of path.	path.dirname('/folder/file.txt') → "/folder"
path.extname(path)	Returns file extension.	path.extname('index.html') → ".html"
path.join(...paths)	Joins paths, normalizes slashes.	path.join('folder','sub','file.txt') → "folder/sub/file.txt"
path.resolve(...paths)	Returns absolute path.	path.resolve('file.txt') → "/abs/path/file.txt"
path.parse(path)	Returns object with {root, dir, base, ext, name}.	path.parse('/folder/file.txt') → object
path.format(obj)	Opposite of parse() – makes path from object.	path.format({dir:'/folder', name:'file', ext:'.txt'})
path.isAbsolute(path)	Checks if path is absolute.	path.isAbsolute('/home') → true
path.sep	Platform-specific separator ('/' or '\\').	Useful for OS compatibility.

4. Example Usage:

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



console.log(path.basename('/users/john/file.txt')); // file.txt

console.log(path.dirname('/users/john/file.txt'));  // /users/john

console.log(path.extname('/users/john/file.txt'));  // .txt

console.log(path.join('folder', 'sub', 'file.txt')); // folder/sub/file.txt

console.log(path.resolve('file.txt')); // absolute path

console.log(path.parse('/users/john/file.txt'));

// { root: '/', dir: '/users/john', base: 'file.txt', ext: '.txt', name: 'file' }
```

5. Why use path module?

- OS-independent path handling.
- Easier to construct, normalize, and manipulate paths.
- Avoids manual string concatenation errors.

Short Notes – Node.js path Module

path → built-in module for file & directory paths.

Import: const path = require('path');

Common methods:

basename() → file name

dirname() → directory name

extname() → file extension

join() → join paths (cross-platform)

resolve() → absolute path

parse() / format() → convert between path & object

isAbsolute() → check absolute path

sep → path separator