Node.js Global Objects (Globals in Node.js)

In Node.js, **global objects** are available in all modules without needing to import them. These objects provide essential functionality for various tasks, such as file handling, process management, and module loading.

Key Global Objects in Node.js

Global Object	Description
`Global`	The global namespace object (similar to `window` in browsers).
`Process`	Provides information and control over the Node.js process.
`__filename`	The absolute path of the current module file.
`__dirname`	The absolute directory path of the current module file.
`setTimeout()`	Executes a function after a delay.
`setInterval()`	Executes a function repeatedly at fixed intervals.
`setImmediate()`	Executes a function immediately after I/O operations.
`Console`	Used for logging messages to the console.
`require()`	Used to import modules.
`Module`	Contains information about the current module.
`Exports`	Used to export functions or objects from a module.

1. `global` Object

In Node.js, **global variables** are properties of the `global` object.

```
global.myVariable = "Hello, World!";
console.log(global.myVariable); // Output: Hello, World!
```

Use Case: Storing global configurations (not recommended for large apps).

2. `process` Object

The `process` object provides details about the running Node.js process.

```
console.log(process.pid); // Prints the process ID
console.log(process.version); // Node.js version
console.log(process.platform); // OS platform
console.log(process.cwd()); // Current working directory
```

Use Case: Process monitoring, debugging, and environment management.

3. `__filename` and `__dirname`

- `__filename` → Full path of the current file.
- `__dirname` → Directory path of the current file.

```
console.log(__filename); // Absolute file path
console.log(__dirname);  // Directory name
```

Use Case: Loading files dynamically based on directory structure.

4. `setTimeout()` - Delayed Execution

Executes a function **after** a specified delay.

```
setTimeout(() => {
    console.log("Executed after 2 seconds");
}, 2000);
```

Use Case: Scheduling tasks, animations, or delays.

5. `setInterval()` - Repeating Execution

Runs a function **repeatedly** at a given interval.

```
setInterval(() => {
    console.log("This runs every 3 seconds");
}, 3000);
```

Use Case: Polling APIs, updating dashboards.

6. `setImmediate()` - Immediate Execution

Executes a function **right after I/O events** (but before `setTimeout(0)`).

```
setImmediate(() => {
    console.log("Executed immediately after I/O operations.");
});
console.log("This prints first.");
```

Output:

```
This prints first.
Executed immediately after I/O operations.
```

Use Case: Deferring execution until the current event loop cycle is complete.

7. `console` - Logging and Debugging

The `console` object is used for logging output.

```
console.log("Normal log");
console.warn("Warning message");
console.error("Error message");
console.table([{ name: "Alice", age: 25 }, { name: "Bob", age: 30 }]);
```

Use Case: Debugging and structured logging.

8. `require()` and `module.exports` - Module System

- `require()` → Imports a module.
- `module.exports` → Exports functions/variables from a module.

Example: Exporting a function from `math.js`

```
// math.js
module.exports.add = (a, b) => a + b;
```

Example: Importing the function in `app.js`

```
const math = require("./math");
console.log(math.add(5, 3)); // Output: 8
```

Use Case: Modularizing code for reusability.

Summary

Node.js provides several global objects for system-level tasks.
`process`, `__filename`, `__dirname`, and `global` are essential for managing Node.js applications.
`setTimeout`, `setInterval`, and `setImmediate` help with timing-related operations.
`console`, `require()`, and `module.exports` are crucial for debugging and modular programming.

Modules

1. Working with Modules

Explanation:

- Modules in Node.js help in organizing and reusing code.
- A module is a self-contained block of code that can be exported and imported in other files.
- Node.js has built-in, third-party, and custom modules.

Imagine a large e-commerce application where we want to separate user-related logic into a module.

```
// user.js (User module)
class User {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }

  getUserDetails() {
    return `User: ${this.name}, Email: ${this.email}`;
  }
}

// Export the module
module.exports = User;
```

```
// app.js (Main application)
const User = require('./user');

const user1 = new User('John Doe', 'john@example.com');
console.log(user1.getUserDetails()); // Output: User: John Doe, Email:
john@example.com
```

2. Modules in Node.js

Explanation:

- Node.js provides three types of modules:
    1. **Core Modules** (e.g., `fs`, `http`, `path`)
    2. **Local Modules** (custom-created modules)
    3. **Third-Party Modules** (installed via `npm`)

Using a core module (`fs`) to read a file:

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

// Reading a file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

3. Loading a Module

Explanation:

- We use `require()` to load modules.
- Local modules require the relative path (`./moduleName`).
- Core modules can be loaded directly (`require('http')`).

Loading the `path` module to manipulate file paths:

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

const filePath = '/home/user/docs/file.txt';
console.log('Directory:', path.dirname(filePath)); // Output:
/home/user/docs
console.log('File Extension:', path.extname(filePath)); // Output: .txt
```

4. package.json Usage

Explanation:

- `package.json` is the configuration file that manages dependencies, scripts, metadata, and project information.
- It allows dependency management and project automation.

To create a `package.json` file:

```
npm init -y
```

This generates a `package.json` file with default values.

5. Creating package.json

Explanation:

- Run `npm init` for an interactive setup.
- Run `npm init -y` for a default `package.json`.

Example `package.json` file:

```
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "A simple Node.js project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {}
}
```

6. Node Package Manager (NPM)

Explanation:

- `npm` is used to install, update, and manage packages in Node.js.
- It allows using third-party libraries to speed up development.

Real-World Example:

Installing `lodash`, a utility library:

```
npm install lodash
```

Using lodash in the project:

```
const _ = require('lodash');

const numbers = [10, 5, 8, 3, 7];
console.log('Sorted Numbers:', _.sortBy(numbers)); // Output: [3, 5, 7, 8,
10]
```

7. Loading a Third-Party Module (Installed via NPM)

Explanation:

- After installing a package via `npm install <package-name>`, it can be used in the project.

Real-World Example:

Using `moment.js` for date formatting:

```
npm install moment
```

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

console.log('Current Date:', moment().format('MMMM Do YYYY, h:mm:ss a'));
```

8. Creating and Exporting a Module

Explanation:

- A module is created using `module.exports`.
- It can export objects, functions, or classes.

Real-World Example:

Creating a `math.js` module:

```
// math.js
module.exports.add = (a, b) => a + b;
module.exports.subtract = (a, b) => a - b;
```

Using the module in another file:

```
// app.js
const math = require('./math');

console.log(math.add(5, 3)); // Output: 8
console.log(math.subtract(10, 4)); // Output: 6
```

9. REPL Terminal

Explanation:

- REPL stands for **Read-Eval-Print Loop**.
- It allows testing Node.js code interactively in a terminal.

To start REPL, run:

```
node
```

Then try:

```
> 5 + 10
15

> const fs = require('fs');
> fs.readFileSync('example.txt', 'utf8');
"Hello, World!"
```

Buffers in Node.js

Buffers in Node.js are used to handle **binary data** directly in memory. They are useful for working with files, network packets, and other raw data, especially when working with **streams**.

1. Creating Buffers

There are multiple ways to create a buffer:

(a) Using `Buffer.alloc(size)`

Creates a buffer of a specified size, filled with zeroes.

```
const buf1 = Buffer.alloc(10); // 10-byte buffer initialized with zeroes
console.log(buf1);
```

Output:

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

(b) Using `Buffer.from(data)`

Creates a buffer from an existing string, array, or another buffer.

```
const buf2 = Buffer.from('Hello');
console.log(buf2);
```

Output:

```
<Buffer 48 65 6c 6c 6f>
```

`48 65 6c 6c 6f` are the ASCII/UTF-8 values for **H e l l o**.

2. Writing to Buffers

We use `buffer.write(string[, offset[, length[, encoding]]])`.

```
const buf = Buffer.alloc(10);
buf.write('Node.js', 0, 'utf8'); // Writing to buffer
console.log(buf.toString()); // Output: Node.js
```

Writes `'Node.js'` starting at position `0` with UTF-8 encoding.

3. Reading from Buffers

We can read buffer content using `toString()`.

```
const buf = Buffer.from('Hello, Node.js');
console.log(buf.toString()); // Output: Hello, Node.js
console.log(buf.toString('utf8', 0, 5)); // Output: Hello (Reads first 5
bytes)
```

4. Convert Buffer to JSON

Buffers can be converted into JSON using `JSON.stringify()`.

```
const buf = Buffer.from('Hello');
const json = JSON.stringify(buf);
console.log(json);
```

Output:

```
{"type":"Buffer","data":[72,101,108,108,111]}
```

The buffer is stored as an object with `type: "Buffer"` and `data` as an array of ASCII values.

5. Concatenating Buffers

We can merge multiple buffers using `Buffer.concat()`.

```
const buf1 = Buffer.from('Hello ');
const buf2 = Buffer.from('World!');
const combined = Buffer.concat([buf1, buf2]);
console.log(combined.toString()); // Output: Hello World!
```

6. Comparing Buffers

Buffers can be compared using `buffer.compare()`.

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('DEF');
const result = buf1.compare(buf2);

if (result < 0) console.log('buf1 comes before buf2');
else if (result > 0) console.log('buf1 comes after buf2');
else console.log('Buffers are equal');
```

- Returns `-1` if `buf1 < buf2`, `1` if `buf1 > buf2`, `0` if they are equal.

7. Copying Buffers

The `buffer.copy()` method copies data from one buffer to another.

```
const buf1 = Buffer.from('Hello');
const buf2 = Buffer.alloc(5);

buf1.copy(buf2); // Copy contents
console.log(buf2.toString()); // Output: Hello
```

8. Slicing Buffers

Buffers can be sliced like arrays.

```
const buf = Buffer.from('Node.js');
const slicedBuf = buf.slice(0, 4);
console.log(slicedBuf.toString()); // Output: Node
```

9. Buffer Length

The `.length` property returns the size of the buffer in bytes.

```
const buf = Buffer.from('Hello');
console.log(buf.length); // Output: 5
```

Length in bytes, **not** characters.

10. Class Methods of Buffer

Method	Description
`Buffer.alloc(size)`	Allocates a new buffer of `size` bytes
`Buffer.from(string/array)`	Creates a buffer from a string or array

Method	Description
`Buffer.concat([buf1, buf2])`	Combines multiple buffers into one
`Buffer.compare(buf1, buf2)`	Compares two buffers
`buf.copy(targetBuffer)`	Copies buffer contents
`buf.slice(start, end)`	Extracts a part of the buffer
`buf.length`	Returns buffer size in bytes

Example: Reading a File as a Buffer

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

fs.readFile('example.txt', (err, data) => {
  if (err) throw err;
  console.log('Buffer data:', data);
  console.log('Converted:', data.toString());
});
```

Output:

```
Buffer data: <Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64>
Converted: Hello World
```

This reads the file as a buffer and converts it to a string.

Summary

Buffers handle binary data in memory.
Efficient for I/O operations (files, streams, networking).
Use `Buffer.alloc()` for new buffers, `Buffer.from()` for existing data.
Concatenate, copy, compare, and slice buffers as needed.

Streams in Node.js

1. What are Streams?

Streams in Node.js are used to process **data in chunks** instead of reading or writing the entire data at once. This makes them efficient for handling **large files, network operations, and real-time data processing**.

Types of Streams

1. **Readable Streams** – For reading data (e.g., file reading, HTTP requests).
2. **Writable Streams** – For writing data (e.g., file writing, HTTP responses).
3. **Duplex Streams** – Both readable and writable (e.g., sockets).
4. **Transform Streams** – A special type of Duplex Stream that transforms data (e.g., compression).

2. Reading from a Stream (Readable Stream)

Example: **Reading a file using streams**

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

// Create a readable stream
const readStream = fs.createReadStream('input.txt', 'utf8');

// Listen for data events
readStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk);
});

// Handle errors
readStream.on('error', (err) => {
  console.error('Error reading file:', err);
});

// End event when stream is finished
readStream.on('end', () => {
  console.log('File reading complete.');
});
```

- **Explanation:** Instead of loading the entire file into memory, we read it in chunks.

```
{ encoding: 'utf8', highWaterMark: 16 }
```

3. Writing to a Stream (Writable Stream)

Example: **Writing data to a file using a writable stream**

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

// Create a writable stream
```

```
const writeStream = fs.createWriteStream('output.txt');

// Write data to the stream
writeStream.write('Hello, this is a stream example.\n');
writeStream.write('Data is written in chunks.\n');

// End the stream
writeStream.end(() => {
  console.log('Data successfully written to output.txt');
});

// Handle errors
writeStream.on('error', (err) => {
  console.error('Error writing file:', err);
});
```

- **Explanation:** This writes data in chunks instead of loading everything into memory.

4. Piping the Streams

Piping is a way to **connect a readable stream to a writable stream**.

Example: **Copying a file using pipe**

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

// Create read and write streams
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');

// Pipe the read stream to the write stream
readStream.pipe(writeStream);

console.log('File copied successfully using pipe.');
```

- **Explanation:**

  - `readStream.pipe(writeStream)` automatically reads and writes in chunks.
  - This is much more memory-efficient than manually reading and writing.

5. Chaining Streams (Transform Streams)

Transform streams allow us to modify data while streaming.

Example: **Compressing a file using zlib**

```
const fs = require('fs');
const zlib = require('zlib'); // Compression library

// Create a read and write stream
```

```
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('input.txt.gz');

// Create a transform stream for compression
const gzip = zlib.createGzip();

// Chain the streams
readStream.pipe(gzip).pipe(writeStream);

console.log('File compressed successfully.');
```

◇ **Explanation:**

- `createGzip()` compresses the data while reading.
- `pipe(writeStream)` writes the compressed data to a file.

Real-World Example: Streaming a Large File via HTTP

```
const http = require('http');
const fs = require('fs');

http.createServer((req, res) => {
  const readStream = fs.createReadStream('bigfile.txt');
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  readStream.pipe(res);
}).listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

◇ **Explanation:**

- The server **streams the file** instead of loading it all into memory, making it efficient for large files.

Summary

Streams process data in chunks, making them memory efficient.
Readable streams let us read data; writable streams let us write data.
Piping connects streams efficiently.
Chaining allows transformations like compression.

Node.js File System (fs) Module

The `fs` module in Node.js provides functions to interact with the **file system**, including reading, writing, deleting files, and working with directories.

To use it, first import the module:

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

1. Synchronous vs Asynchronous File Operations

Synchronous (`fs.*Sync()`)

- **Blocking**: The program waits until the operation is finished.
- **Example:**

```
const data = fs.readFileSync('example.txt', 'utf8'); // Blocking
console.log(data);
console.log('This runs *after* reading the file');
```

Asynchronous (`fs.*()` with callbacks or Promises)

- **Non-blocking**: Code executes without waiting.
- **Example:**

```
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('This runs *before* file reading completes');
```

2. Opening a File (`fs.open`)

Opens a file, returns a **file descriptor**.

Syntax:

```
fs.open('example.txt', 'r', (err, fd) => {
  if (err) throw err;
  console.log('File opened successfully', fd);
  fs.close(fd, () => {}); // Close the file after use
});
```

Common File Flags

Flag	Description
`'r'`	Read mode
`'w'`	Write mode (creates if not exists)
`'a'`	Append mode

3. Getting File Information (`fs.stat`)

Gets details about a file (size, creation date, etc.).

```
fs.stat('example.txt', (err, stats) => {
  if (err) throw err;
  console.log(stats);
});
```

4. Writing to a File (`fs.writeFile`)

Writes data to a file (creates if not exists).

```
fs.writeFile('example.txt', 'Hello, World!', (err) => {
  if (err) throw err;
  console.log('File written successfully');
});
```

To **append** to a file instead of overwriting:

```
fs.appendFile('example.txt', '\nNew Line!', (err) => {
  if (err) throw err;
  console.log('Content appended');
});
```

5. Reading from a File (`fs.readFile`)

Reads content from a file.

```
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

6. Closing a File (`fs.close`)

After opening a file using `fs.open()`, close it:

```
fs.open('example.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
    console.log('File closed');
  });
});
```

7. Truncating a File (`fs.truncate`)

Trims the file to a specific length.

```
fs.truncate('example.txt', 10, (err) => {
  if (err) throw err;
  console.log('File truncated to 10 bytes');
});
```

8. Deleting a File (`fs.unlink`)

Deletes a file.

```
fs.unlink('example.txt', (err) => {
  if (err) throw err;
  console.log('File deleted');
});
```

9. Creating a Directory (`fs.mkdir`)

Creates a new folder.

```
fs.mkdir('new-folder', (err) => {
  if (err) throw err;
  console.log('Directory created');
});
```

To create **nested directories**:

```
fs.mkdir('parent/child/grandchild', { recursive: true }, (err) => {
  if (err) throw err;
  console.log('Nested directories created');
});
```

10. Reading a Directory (`fs.readdir`)

Lists files and folders inside a directory.

```
fs.readdir('.', (err, files) => {
  if (err) throw err;
  console.log('Files in directory:', files);
});
```

11. Removing a Directory (`fs.rmdir`)

Deletes an **empty** directory.

```
fs.rmdir('new-folder', (err) => {
  if (err) throw err;
  console.log('Directory removed');
});
```

To delete a **non-empty** directory:

```
fs.rm('parent', { recursive: true, force: true }, (err) => {
  if (err) throw err;
  console.log('Directory deleted');
});
```

12. Uploading Files (Using `multer`)

To upload files in Node.js, use the `multer` package.

1 **Install Multer**

```
npm install multer
```

using **Express** and **Multer** allows users to upload files to a specific directory (`D:/uploads/`). Below is a breakdown of the code:

1 Import Required Modules

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

- **`express`**: Web framework for handling HTTP requests.
- **`multer`**: Middleware for handling file uploads.
- **`path`**: Helps manage file paths and extensions.

2 Initialize Express App

```
const app = express();
```

- Creates an instance of an Express application.

3 Configure Multer for File Storage

```
const storage = multer.diskStorage({
  destination: 'D:/uploads/', // Folder where uploaded files will be saved
  filename: (req, file, cb) => {
    cb(null, file.fieldname + '-' + Date.now() +
path.extname(file.originalname));
  }
});
```

Example: If a user uploads `image.jpg`, the saved filename will be something like:

```
file-1709063245871.jpg
```

4 Initialize Multer Middleware

```
const upload = multer({ storage: storage });
```

- Creates a middleware instance with the **storage configuration**.

5 Define the Upload Route

```
app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
  }
  res.send(`File uploaded: ${req.file.filename}`);
});
```

6 Start the Server

```
app.listen(3000, () => console.log('Server running on port 3000'));
```

Create an HTML Form for File Upload

You can use a simple HTML file (`index.html`) to upload files to this server.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>File Upload</title>
</head>
<body>
    <h2>Upload a File</h2>
    <form action="http://localhost:3000/upload" method="POST"
enctype="multipart/form-data">
        <input type="file" name="file" required>
        <button type="submit">Upload</button>
    </form>
</body>
</html>
```

Summary of File System Operations

Operation	Function
Open a file	`fs.open()`
Get file info	`fs.stat()`
Read a file	`fs.readFile()`
Write a file	`fs.writeFile()`
Append to a file	`fs.appendFile()`
Truncate file	`fs.truncate()`
Delete file	`fs.unlink()`
Create directory	`fs.mkdir()`
Read directory	`fs.readdir()`
Remove directory	`fs.rmdir()`
Upload file	`multer`

Assignment: Building a Simple File Management System in Node.js

Problem Statement:

You have been hired as a backend developer to create a simple **File Management System** for a company. The system should allow users to:

1. Create and write to files.
2. Read and display file contents.
3. Append data to an existing file.
4. Delete files.
5. Log all operations performed using the `console` object.
6. Use `setTimeout` to simulate a delay before reading a file.
7. Use `setInterval` to log system status every 5 seconds.
8. Use `process` to display environment details.

Assignment Requirements:

1. **Use Global Objects**
   - Use `__filename` and `__dirname` to display the current file path.
   - Use the `process` object to log the current Node.js version, OS platform, and process ID.
2. **Use Core Modules (`fs` and `path`)**
   - Use the `fs` module to perform file operations (create, write, read, append, and delete).
   - Use the `path` module to manipulate file paths dynamically.
3. **Use Timing Functions**
   - Use `setTimeout()` to delay reading the file by 3 seconds after writing.
   - Use `setInterval()` to log a message every 5 seconds indicating that the system is running.

4. **Use Buffers**
    - Store file data in a `Buffer` before writing it to a file.
    - Read the file as a buffer and convert it to a string before displaying it.
5. **Use Modules**
    - Implement the file handling logic inside a separate module (`fileManager.js`).
    - Import and use it in `app.js` using `require()`.

Implementation Steps:

1. **Create a module (`fileManager.js`) to handle file operations.**
2. **Create a main file (`app.js`) to use the module and execute operations.**
3. **Test the system by creating, writing, reading, and deleting a file.**

Expected Output (Console Logs):

```
File created successfully.
File data written successfully.
Reading file in 3 seconds...
(System status logged every 5 seconds)
File Contents: "Hello, this is a File Management System!"
Appending data...
Data appended successfully.
Deleting file...
File deleted successfully.
Node.js Version: v18.0.0
Platform: win32
Process ID: 12345
```