CipherSchools

Class 2 NodeJS

os Module

The os module provides operating system-related utility methods and properties.

Importing

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

Common os Methods

1. os.platform()
   Returns the OS platform: 'linux', 'darwin' (macOS), 'win32'

```
console.log(os.platform()); // → 'linux'
```

2. os.type()
   Returns the OS name: 'Linux', 'Windows_NT', etc.

```
console.log(os.type()); // → 'Linux'
```

3. os.arch()
   CPU architecture: 'x64', 'arm', etc.

```
console.log(os.arch()); // → 'x64'
```

4. os.userInfo()
   Returns the user Information.

```
console.log(os.userInfo());
/*{
  uid: 1000,
  gid: 1000,
  username: 'nitesh20',
  homedir: '/home/nitesh20',
  shell: '/bin/bash'
}*/
```

5. os.homedir()
   Returns the directory part.

```
console.log(os.homedir());
```

6. os.uptime()
   System uptime in seconds.

```
console.log(os.uptime());
// 12234.63
```

7. os.freemem() and os.totalmem()
   Memory usage (in bytes).

```
console.log('Free:', os.freemem() / 1024 / 1024, 'MB');
console.log('Total:', os.totalmem() / 1024 / 1024, 'MB');
```

8. os.cpus()
   Array of CPU info.

```
console.log(os.cpus());
```

9. os.networkInterfaces()
   Info about network interfaces (like IP addresses).

```
console.log(os.networkInterfaces());
```

Summary:

Method	Purpose
platform()	OS platform (linux, win32)
type()	OS name (Linux, Darwin)
arch()	CPU architecture

userInfo()	Logged-in user info
homedir()	Home directory path
uptime()	System uptime in seconds
freemem()	Free RAM in bytes
totalmem()	Total RAM in bytes
cpus()	CPU core info
networkInterfaces()	Network info (IP, MAC, etc.)

Real-Life Use Cases

1. *Logging System Info for Debugging*const os = require('os');

```
console.log(`Platform: ${os.platform()}`);
console.log(`CPU Cores: ${os.cpus().length}`);
console.log(`Free Memory: ${os.freemem()}`);
```

Useful for collecting debug info when bugs occur in production.

2. *Writing Large Files with Memory Check*

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

if (os.freemem() > 100 * 1024 * 1024) { // 100MB
  // Proceed to write large file
} else {
  console.error("Not enough memory!");
}
```

Prevents server crash by ensuring memory is sufficient before processing.

path Module

The path module provides utilities for working with file and directory paths. It handles differences across OSes (e.g., Windows uses \, Linux uses /).

Importing

```
const path = require(‘path’);
```

Common path Methods

1. path.join([...paths])
   Joins path segments using the correct platform separator.

```
const fullPath = path.join('folder', 'subfolder', 'file.txt');
console.log(fullPath); // → folder/subfolder/file.txt (cross-platform)
```

2. path.resolve([...paths])
   Resolves absolute path based on current directory.

```
const absPath = path.resolve('folder', 'file.txt');
console.log(absPath); // → /home/..../MERN/node/folder/file.txt
```

3. path.basename(path)
   Returns the file name from the path.

```
const name = path.basename('/folder/file.txt');
console.log(name); // → file.txt
```

4. path.extname(path)
   Returns the file extension.

```
const ext = path.extname('index.html');
console.log(ext); // → .html
```

5. path.dirname(path)
   Returns the directory part.

```
const dir = path.dirname('/folder/file.txt');
console.log(dir); // → /folder
```

6. path.parse(path)
   Returns an object with: root, dir, base, name, ext

```
console.log(path.parse('/folder/index.html'));
// {
//   root: '/',
//   dir: '/folder',
//   base: 'index.html',
//   ext: '.html',
//   name: 'index'
// }
```

7. path.format(obj)
   Opposite of parse() — builds a path from parts.

```
console.log(path.format({
  dir: '/folder',
  name: 'index',
  ext: '.html'
})); // → /folder/index.html
```

Summary:

Method	Purpose
join()	Concatenate path segments
resolve()	Resolve full absolute path
basename()	Get file name
dirname()	Get directory name
extname()	Get file extension
parse()	Break path into parts
format()	Create path from parts

Real-Life Use Cases

3. *File Upload Handling*

```
const path = require('path');
const uploadPath = path.join(__dirname, 'uploads',
userId.toString());
```

Ensures the correct path is generated regardless of OS (Windows uses \, Linux uses /).

events Module

The events module in Node.js provides a way to create, fire (emit), and listen for custom events.

It implements the **Observer pattern**, where one object emits an event and others "observe" (or listen to) it.

Node.js is built on **event-driven architecture**, making this module a core component of how things work behind the scenes (e.g., http, fs, net, streams all use events).

Importing the module

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
```

Basic Usage

1. Create and listen to an event

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('greet', () => {
  console.log('Hello there!');
});

emitter.emit('greet'); // Output: Hello there!
```

2. You can emit the same event multiple times:

```
emitter.emit('greet');
emitter.emit('greet');
```

3. Pass data with events

```
emitter.on('orderPlaced', (orderId, customer) => {
  console.log(`Order ${orderId} placed by ${customer}`);
});

emitter.emit('orderPlaced', 'A102', 'John');
```

4. Listen only once with once()

```
emitter.once('login', () => {
  console.log('First login only!');
});

emitter.emit('login'); // Triggered
emitter.emit('login'); // Ignored
```

5. Remove a listener

```
function sayHello() {
  console.log('Hello!');
}

emitter.on('hello', sayHello);
emitter.removeListener('hello', sayHello);
emitter.emit('hello'); // No output
```

6. Or use:

```
emitter.off('hello', sayHello); // Same as removeListener in latest versions
```

Real-Life Use Cases

User Signup Flow

```
// Trigger events after a user registers:
emitter.on('userRegistered', (user) => {
  sendWelcomeEmail(user.email);
  logActivity(user.id);
});

registerUser({ id: 1, email: 'test@mail.com' });
```

```
emitter.emit('userRegistered', { id: 1, email: 'test@mail.com' });
```

Custom Order Management Event

```
const EventEmitter = require("events");

class OrderManager extends EventEmitter {
  placeOrder(order) {
    console.log(`Order placed: ${order}`);
    this.emit("order", order);
  }
}

const manager = new OrderManager();

manager.on("order", (order) => {
  console.log("Processing order:", order);
});

manager.placeOrder("T-shirt");
```

Other Scenarios:

- Email/sms notifications after database insert
- Logging user actions across microservices
- Event-driven microservices with Kafka/NATS/RabbitMQ (inspired by events module)

Task:

You are building an e-commerce backend system. When a user places an order (e.g., buying a T-shirt), the system should do multiple things:

- Log the order.
- Send the order to the warehouse for processing.
- Notify the shipping team.
- Send an email or SMS to the customer.
- Record the order in analytics.

Rather than calling all these services one after the other in a tight, coupled function, you want to emit an event (like order) and let different modules subscribe and react independently.

streams Module

A stream is an abstract interface for handling streaming data - data that’s not available all at once, but arrives in chunks over time.

Think of a stream like a water pipe - water (data) flows through it gradually instead of coming all at once.

Why Streams?

Without streams, you'd have to load the entire file/data into memory, which:

- Wastes memory
- Slows performance
- Crashes for large files

With streams:

- You process data chunk-by-chunk
- Efficient for big files, real-time logs, video, HTTP requests, etc.

Four Types of Streams

Type	Description	Example
Readable	Data can be read from it	Reading a file
Writable	Data can be written to it	Writing to a log file
Duplex	Both read and write	TCP sockets
Transform	Modify data while reading/writing	Compressing, encrypting

Key Concepts

1. Chunk-based Processing
   Streams break the data into chunks, process them individually, and keep memory usage low.

2. Event-Driven
   Streams emit events like:
   - data – when a chunk is available
   - end – when data finishes
   - error – if something goes wrong
   - finish – for writable streams

Real World Problems Solved by Streams

1. Reading Large Files (Efficiently)
   Problem: Reading a 5GB log file with fs.readFile() will crash the server (memory overload).
   Solution: Use a Readable Stream.

```
const fs = require('fs');
const stream = fs.createReadStream('bigfile.txt', {
encoding: 'utf8',
highWaterMark: 4 //Chunk size
});

stream.on('data', chunk => {
  console.log('Chunk received:', chunk);
});
```

Efficient: Only a small part of the file is read at a time.

How Streams Work Internally

When you use:

```
const stream = fs.createReadStream('bigfile.txt', { encoding: 'utf8'
});
```

You're telling Node.js:

"Hey, don't load the whole file into memory. Just give me a piece (chunk) at a time."

What Is a Chunk?

A chunk is a portion of the data (by default, ~64 KB for files). Node reads the file bit by bit and emits a data event for each chunk.
So this:

```
stream.on('data', chunk => {
  console.log('Chunk received:', chunk.length);
});
```

...will log multiple lines if the file is big enough.

What is .pipe() in Streams?

.pipe() is a shortcut for connecting streams
Imagine you have:

- A Readable Stream (like reading from a file)
- A Writable Stream (like writing to a file or response)

Instead of manually writing:

```
readStream.on('data', chunk => {
  writeStream.write(chunk);
});
```

You simply do:

```
readStream.pipe(writeStream);
```

💡 It connects the output of one stream directly to the input of another.

Example: Copying a File

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

const readStream = fs.createReadStream('input.txt', {
encoding: 'utf-8',
highWaterMark: 4 //chunk size
});
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);
```

This copies input.txt → output.txt using a stream pipe. Very memory-efficient, even for large files.

Example2: Transforming Data – Compression or Encryption

```
const fs = require('fs');
const zlib = require('zlib');
const gzip = zlib.createGzip();
fs.createReadStream('input.txt')
  .pipe(gzip)         // transform stream
  .pipe(fs.createWriteStream('input.txt.gz'));
```

Compress files without loading the entire file into memory.

Use Cases

Scenario	Stream Type(s) Used
Reading huge files	Readable
File upload/download	Readable + Writable
Compression	Transform
Live chat app	Duplex / Transform
Audio/video streaming	Readable + Pipe

What is a Buffer in Node.js?

A Buffer is a temporary memory storage used to handle binary data (not strings or objects) directly.

Why?
JavaScript (in browsers) traditionally deals with strings, not binary data.
But in Node.js - for things like file I/O, streams, TCP/UDP sockets, image/video files - we often deal with raw bytes.

Buffers allow Node.js to read and write binary data directly, efficiently, and without converting to strings unless needed.

When Do We Need Buffers?

Use Case	Reason to Use Buffer
Reading files	Files are binary
Working with streams	Streams emit Buffers
Sending/receiving network data	Sockets work with bytes
Working with binary formats	Images, PDFs, etc
Encoding/decoding	Base64, UTF-8, Hex

Creating Buffers

1. From a String

```
const buf = Buffer.from('Hello');
console.log(buf);           // <Buffer 48 65 6c 6c 6f>
```

Each letter is converted into its ASCII byte.

2. Allocating Raw Memory

```
const buf = Buffer.alloc(10); // 10 bytes of zero
console.log(buf);             // <Buffer 00 00 00 00 00 00 00 00 00
00>
```

Reading & Writing Data in Buffers

1. Write to Buffer

```
const buf = Buffer.alloc(5);
buf.write('Hi');
console.log(buf); // <Buffer 48 69 00 00 00>
```

2. Read from Buffer

```
console.log(buf.toString()); // 'Hi'
```

Example

Encode a Password

```
const password = 'secret';
const buf = Buffer.from(password);
console.log(buf.toString('base64')); // c2VjcmV0
```

Useful for authentication and encoding.

.Common Buffer Methods

Method	Description
Buffer.from(str)	Creates buffer from string
Buffer.alloc(size)	Allocates clean buffer
Buffer.isBuffer(val)	Checks if value is a buffer
buf.toString(encoding)	Converts buffer to string
buf.write(str)	Writes string into buffer
buf.slice(start, end)	Returns a slice of the buffer