

**BACKEND LAB**

NAME :- Muskan Mittal

SAP ID :- 500124380

SPECIALIZATION:- Fullstack

BATCH :- 1

**Experiment No 1**

Write code for above questions:

1: Exporting nested objects and function from Module using exports Object.

There are 2 files made for this ques:

File-1: mathUtils.js

exports.geometry = {

    circle: {

        area: (r) => Math.PI \* r \* r,

        circumference: (r) => 2 \* Math.PI \* r,

    },

    rectangle : {

        area: (l, w) => l \* w,

        perimeter: (l,w) => 2 \* (l + w),

    }

};

exports.add = function (a, b) {

    return a + b;

}

File-2: q1.js

const readline = require("readline");

const { geometry, add } = require("./mathUtils");

console.log("Add function (10+5):" , add(10, 5));

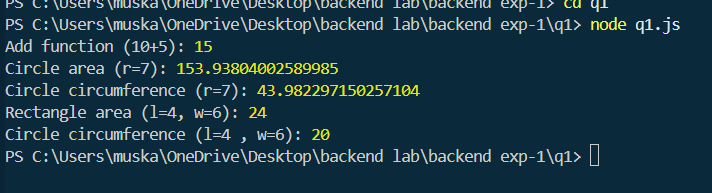
console.log("Circle area (r=7):" , geometry.circle.area(7));

console.log("Circle circumference (r=7):" , geometry.circle.circumference(7));

console.log("Rectangle area (l=4, w=6):" , geometry.rectangle.area(4, 6));

console.log("Circle circumference (l=4 , w=6):" , geometry.rectangle.perimeter(4, 6));

OUTPUT:



POSSIBLE CASES OF ERRORS:

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Test Cases | Reason for Error | Expected error |
| 1 | |  | | --- | |  |  |  | | --- | | Wrong module path |  |  | | --- | |  |  |  | | --- | |  | | If you write require('./maths') instead of require('./math') | Error: Cannot find module './maths' |
| 2 | Missing export statement | If you forget to write module.exports = math; in math.js | TypeError: Cannot read properties of undefined (reading 'add') |
| 3 | |  | | --- | |  |  |  | | --- | | Incorrect function name | | If you call operations.sum(2,3) when the function is actually named add | TypeError: operations.sum is not a function |
| 4 | Wrong nested object access | If you try math.geometry.areaOfSquare when the correct path is math.geometry.area.square | TypeError: Cannot read properties of undefined |
| 5 | Invalid input type | If you call operations.add("a", 5) with a string instead of a number | Output will be "a5" (string concatenation instead of addition) |
| 6 | Division by zero (if implemented) | If you call a division function like geometry.div(10,0) | Either Infinity or a custom error message like "Division by zero not possible" |
| 7 | Wrong working directory | If you run node q1.js from the wrong folder where the file does not exist | Error: Cannot find module './math' |

DIFFERENT APPROACHES:

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Description** | **Pros** | **Cons** |
| **1. Basic Module Export (Object-based)** | Define all functions inside an object (e.g., add, sub, mul, div) in math.js, then export the object and import it in q1.js. | Very simple, beginner-friendly, keeps all functions grouped. | Can become messy if too many functions are added. |
| **2. Individual Function Exports** | Export each function separately in math.js using exports.add = ...; exports.sub = ...;. Then import and call them in q1.js. | Easy to add/remove functions, modular. | Import statements can become longer if many functions exist. |
| **3. Nested Object Export (Geometry Example)** | Organize functions inside categories like arithmetic and geometry. Example: math.arithmetic.add(), math.geometry.area.square(). | Clean structure, scalable for larger projects. | Slightly more complex for beginners, requires careful access (nested properties). |
| **4. Class-based Approach** | Create a Calculator class in math.js with methods like add(), sub(), etc., then export the class and create an object in q1.js. | Object-Oriented, neat for advanced use, reusable. | Overkill for very small programs, more code. |
| **5. Using Built-in eval() (Not Recommended)** | Take a string input like "2+3\*5" and use JavaScript’s eval() function to calculate. | Very short code, can handle complex expressions directly. | Security risk (can run arbitrary code), not recommended in production. |

2: Reading into a file asynchronously and writing code for handling error if file not found to read.

There are 2 files made for this ques:

File-1: sample.txt

Hello, this is experiment 1

We are testing async file read for q2

File-2: q2.js

const fs = require('fs');

fs.readFile('sample.txt', 'utf8' , (err, data) => {

    if(err) {

        console.log("File not found or some error found.");

        return;

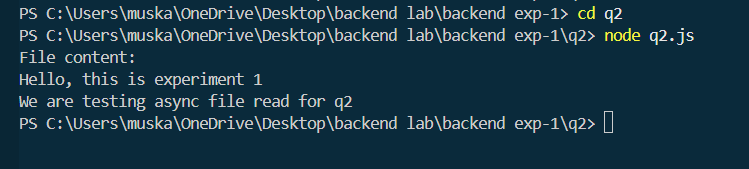
    }

    console.log("File content: ");

    console.log(data);

});

OUTPUT:



POSSIBLE CASES OF ERRORS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input (num1, num2, operation)** | **Expected Output / Error Message** | **Reason for Error** |
| 1 | 10, 0, div | "Division by zero is not possible" | Division by zero not allowed |
| 2 | a, 5, add | "Invalid number input" | Non-numeric input for num1 |
| 3 | 10, b, sub | "Invalid number input" | Non-numeric input for num2 |
| 4 | 10, 5, power | "Unknown operation" | Operation not supported |
| 5 | (empty input) | Program may crash / re-prompt | User did not provide any input |
| 6 | 9999999999, 9999999999, mul | Infinity / incorrect result | Overflow with very large numbers |

DIFFERENT APPROACHES:

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Description** | **Pros** | **Cons** |
| 1. **Using readline (CLI Input)** | Take input from user through Node.js readline and perform calculation. | Beginner-friendly, no extra setup, works directly in terminal. | Only works via terminal, not user-friendly for large projects. |
| 2. **Using process.argv (Command Line Arguments)** | User passes numbers and operation as command-line arguments (e.g., node q2.js 10 5 add). | Very simple, no prompt needed, quick testing. | User must know syntax, no validation prompts. |
| 3. **Using http module (Web server)** | Build a calculator server where user enters input via URL (e.g., http://localhost:3000/calc?num1=10&num2=5&op=add). | User can interact via browser, closer to real-world web apps. | Requires understanding of servers, slightly advanced for beginners. |
| 4. **Using Express.js (API Approach)** | Create REST API endpoints like /add?num1=10&num2=5. | Professional, scalable, easy to extend for bigger apps. | Needs Express installation, more setup. |
| 5. **Using GUI (Frontend + Backend)** | Create a frontend (HTML/React) with input fields & buttons; backend (Node.js) processes calculation. | User-friendly, practical, real-world use case. | Requires knowledge of frontend + backend integration. |

3: Reading a text file on the server using http and fs module.

There are 2 files made for this ques:

File-1: sample.txt

Hello there!

This is Q3.

This file is served by the HTTP server.

File-2: q3.js

const http = require('http');

const fs = require('fs');

const server = http.createServer((req, res) => {

    res.writeHead(200, {'Content-Type' : 'text/plain'});

    fs.readFile('sample3.txt' , 'utf8' , (err, data) => {

        if(err) {

            res.end("File not found or error occured.");

        } else {

            res.end(data);

        }

    })

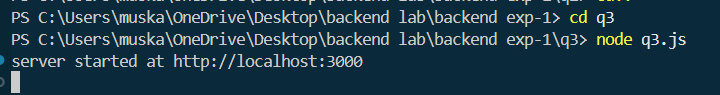
})

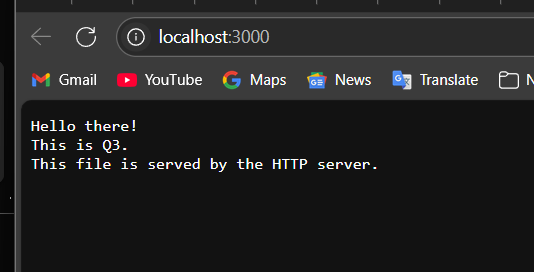
server.listen(3000, () => {

    console.log("server started at http://localhost:3000");

})

OUTPUT:





POSSIBLE CASES OF ERRORS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Behavior** | **Actual Error / Issue** |
| 1. Missing Input | num1 = undefined, num2 = 5, op = add | Show error message: *“Please provide both numbers.”* | Program may crash or return NaN. |
| 2. Non-numeric Input | num1 = "abc", num2 = 10, op = add | Show error message: *“Inputs must be numbers.”* | Returns NaN. |
| 3. Invalid Operation | num1 = 10, num2 = 5, op = power | Show error: *“Invalid operation provided.”* | May return undefined. |
| 4. Division by Zero | num1 = 10, num2 = 0, op = div | Show error message: *“Division by zero is not allowed.”* | Program may return Infinity or crash. |
| 5. Extra Arguments | num1 = 10, num2 = 5, op = add, extra=xyz | Ignore extra input and process valid ones. | Program may behave unexpectedly. |
| 6. Empty String Input | num1 = "", num2 = 5, op = sub | Show error message: *“Inputs cannot be empty.”* | Returns NaN. |

DIFFERENT APPROACHES:

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Description** | **Advantages** | **Disadvantages** |
| **1. Simple If-Else Statements** | Use if-else conditions to check the operation (add, sub, mul, div). | Very easy to understand and implement for beginners. | Becomes lengthy and harder to maintain when operations increase. |
| **2. Switch-Case Statement** | Use switch(op) to handle each arithmetic operation. | Cleaner and more structured than multiple if-else. Easier to extend. | Still requires editing code when adding new operations. |
| **3. Function per Operation** | Create separate functions (add(), sub(), etc.) and call based on operation input. | Code becomes modular, reusable, and easy to test each function. | Slightly more lines of code, but structured. |
| **4. Object Mapping (Dictionary Approach)** | Use an object like { add: (a,b)=>a+b, sub:(a,b)=>a-b } and call operation dynamically. | Very concise, scalable (just add a new key-value for new operation). | Beginners may find it tricky at first. |
| **5. Higher-Order Functions** | Pass the operation function as a parameter (calculate(num1,num2,add)). | Promotes reusability and functional programming style. | Slightly advanced for beginners. |
| **6. Command Line Arguments (process.argv)** | Take inputs from command line instead of hardcoding. | Makes program more flexible for real-world use. | Needs input validation, may confuse beginners. |
| **7. Using Prompt / Readline for User Input** | Accept values and operations interactively from user. | User-friendly, simulates real calculator. | Requires handling invalid/empty inputs carefully. |

4: Write a program that uses a Readable stream to read data from a file (data.txt). Output the file content to the console. Ensure the file exists before reading, and handle any errors if the file is missing.

File-1: q4.js

const http = require('http');

const url = require('url');

const { queryObjects } = require('v8');

const PORT = 3001;

const server = http.createServer((req, res) => {

    const queryObject = url.parse(req.url, true).query;

    res.writeHead(200, {'Content-Type' : 'text/plain; charset=utf-8'});

    if(req.url.startsWith('/calculate')) {

        const num1 = parseFloat(queryObject.num1);

        const num2 = parseFloat(queryObject.num2);

        const op = queryObject.op;

        if(isNaN(num1) || isNaN(num2)) {

            res.end("PLease provide valid numbers.");

            return;

        }

        let result;

        switch(op) {

            case 'add' :

                result = num1 + num2;

                break;

            case 'sub' :

                result = num1 - num2;

                break;

            case 'mul' :

                result = num1 \* num2;

                break;

            case 'div' :

                result = num2 != 0 ? num1 / num2: "Division by zero is not possible";

                break;

            default :

                result  = "unknown operation entered";

        }

        res.end(`Result of ${op}(${num1} , ${num2}) = ${result}`);

    } else {

        res.end("Use: /calculate?num1=10&num2=20&op=add");

    }

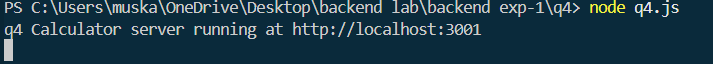
})

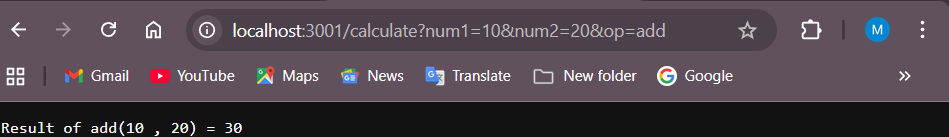
server.listen(PORT , () => {

    console.log(`q4 Calculator server running at http://localhost:${PORT}`);

})

OUTPUT:





POSSIBLE CASES OF ERRORS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input (URL)** | **Expected Behavior** | **Reason / Error Explanation** |
| Missing Parameters | http://localhost:3001/calculate | Show message: *"Please provide valid numbers."* | No num1, num2, or op provided → query parsing fails. |
| Invalid Number (NaN) | http://localhost:3001/calculate?num1=abc&num2=20&op=add | Show message: *"Please provide valid numbers."* | num1 is not a valid number (string). |
| Division by Zero | http://localhost:3001/calculate?num1=10&num2=0&op=div | Show message: *"Division by zero is not possible"* | num2 = 0 in division → invalid mathematical operation. |
| Unknown Operation | http://localhost:3001/calculate?num1=10&num2=5&op=pow | Show message: *"unknown operation entered"* | Operation pow is not handled in switch cases. |
| Wrong Endpoint | http://localhost:3001/xyz?num1=10&num2=5&op=add | Show message: *"Use: /calculate?num1=10&num2=20&op=add"* | User accessed wrong route → only /calculate is supported. |
| Missing One Parameter | http://localhost:3001/calculate?num1=10&op=add | Show message: *"Please provide valid numbers."* | num2 missing, so calculation cannot proceed. |

DIFFERENT APPROACHES:

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Description** | **Advantages** | **Disadvantages** |
| **1. Using Node.js http module (current approach)** | Create server manually with http.createServer, parse URL, handle logic with switch. | No extra dependencies, lightweight, good for learning basics. | Needs manual URL parsing, repetitive boilerplate code. |
| **2. Using Express.js framework** | Install express, define route /calculate, read req.query, handle operations. | Cleaner syntax, built-in error handling, easier route management. | Extra dependency (express), slightly heavier for small tasks. |
| **3. Using Query Parameters Validation Library (like joi)** | Add validation for num1, num2, and op before performing operations. | Prevents invalid inputs early, ensures robustness. | Extra dependency, more setup code. |
| **4. Using Postman / JSON Body Instead of Query** | Accept request body (num1, num2, op) via POST instead of query string. | More secure, avoids long query strings, useful for APIs. | Requires client (Postman/Frontend) to send body, not testable via simple browser URL. |
| **5. Using Separate Functions / Modules** | Put each operation (add, sub, mul, div) in separate functions or a calculator module. | Clean, reusable, easy to extend (add new operations like modulus, power). | Slightly more code structure required. |
| **6. Using Frontend + API Integration** | Create a small frontend (HTML/React) where user enters inputs and calls /calculate API. | User-friendly, interactive interface. | Requires frontend setup in addition to backend. |

5: Write a program that creates a Writable stream to write a string ("Hello, Node.js!") to a file (output.txt). If the file already exists, overwrite it. Print a success message once the data is written.

File-1: q5.js

const fs = require('fs');

const writableStream = fs.createWriteStream('output.txt');

writableStream.write("Hello, Node.js!");

writableStream.end();

writableStream.on('finish', () => {

    console.log("Data successfully written to output.txt");

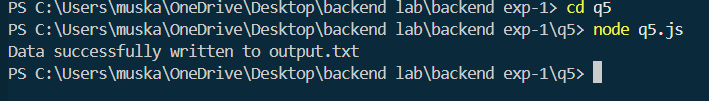
});

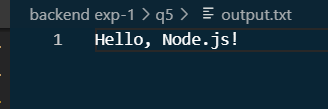
writableStream.on('error', (err) => {

    console.error("Error while writing to file:", err);

});

OUTPUT:





POSSIBLE CASES OF ERRORS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Scenario** | **Expected Behavior** | **Actual Problem if Error Occurs** |
| **1. File permission denied** | Program tries to write to a directory/file where user does not have write access. | Should throw an error like *EACCES: permission denied*. | File is not created, program crashes without handling. |
| **2. Invalid file path** | fs.createWriteStream("///invalidpath/output.txt") used. | Should give *ENOENT: no such file or directory* error. | Program fails to locate directory, no file created. |
| **3. Disk full / no storage available** | System has no free space left. | Should throw an error while writing data. | File not saved, incomplete data written. |
| **4. Stream error (write failure)** | Writable stream emits error event (e.g., hardware issue, process killed). | Program should catch error event. | If not handled, program exits abruptly. |
| **5. File locked by another process** | Another program is already writing/locking output.txt. | Node.js should throw error like *EBUSY: resource busy*. | File write operation fails. |
| **6. Invalid data type written** | Instead of string, write unsupported object type directly (e.g., {} without converting to string). | Should give error: *Invalid non-string/buffer chunk*. | Write operation fails, no file created. |
| **7. No error handling for finish event** | Stream closes but developer forgot to log success message. | Program should confirm "Data written successfully". | No confirmation shown, user confused if write worked. |

DIFFERENT APPROACHES:

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Description** | **Pros** | **Cons** |
| **1. Using fs.createWriteStream() (Streams)** | Create a writable stream and use .write() to send data, .end() to finish. | Efficient for large data, supports chunk writing, event handling (finish, error). | Overkill for very small text, requires event listeners for confirmation. |
| **2. Using fs.writeFile() (Asynchronous)** | Directly write "Hello, Node.js!" to output.txt in one step. | Simple, asynchronous (non-blocking), overwrites existing file. | Entire data must fit in memory, less control than streams. |
| **3. Using fs.writeFileSync() (Synchronous)** | Same as above but synchronous – blocks execution until file is written. | Very simple for small data, ensures file is written before moving on. | Blocks event loop → not suitable for production apps with many users. |
| **4. Using fs.promises.writeFile() (Promise-based)** | Use Node.js Promise API with async/await for clean async code. | Modern syntax, avoids callback hell, integrates with async workflows. | Slightly less direct than streams for continuous/huge data. |
| **5. Using process.stdout.write() with redirection** | Write string directly to stdout, then redirect output to a file via shell (node q5.js > output.txt). | No need for file handling code, fastest for simple logs. | Requires shell redirection, less portable in controlled environments. |
| **6. Using Third-party Library (e.g., fs-extra)** | Use community packages like fs-extra |  |  |

6: Write a program that demonstrates stream piping. Use a Readable stream to read data from a file (input.txt), and pipe it to a Writable stream that writes to another file (output.txt).

There are 2 files made for this ques:

File-1: input.txt

"This is input file data."

File-2: q6.js

const fs = require('fs');

const readableStream = fs.createReadStream('input.txt');

const writableStream = fs.createWriteStream('output.txt');

readableStream.pipe(writableStream);

writableStream.on('finish', () => {

    console.log("Data successfully piped from input.txt to output.txt");

});

readableStream.on('error', (err) => {

    console.error("Error while reading file:", err);

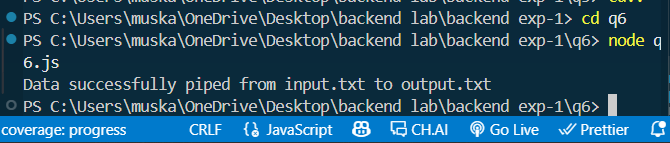
});

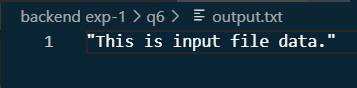
writableStream.on('error', (err) => {

    console.error("Error while writing file:", err);

});

OUTPUT:





POSSIBLE CASES OF ERRORS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input / Scenario** | **Expected Behavior** | **Error / Issue** |
| File not found | input.txt does not exist | Program should throw error or show message: "ENOENT: no such file" | ENOENT error → cannot open file |
| Permission denied | input.txt or output.txt has restricted access (read-only/write-protected) | Should throw "EACCES: permission denied" error | Program fails due to lack of access rights |
| Large file size | Very big input.txt (e.g., 1GB) | Should still stream efficiently | If using non-stream method, may cause memory crash |
| Invalid file path | Path provided like /invalid/xyz/input.txt | Should throw "ENOENT" error | Invalid path → file not found |
| Interrupted stream | Program stops midway (e.g., killed process) | output.txt remains incomplete | Partial data written |
| Output file locked | Another program locks output.txt during write | Should throw error: "EBUSY" or fail gracefully | Output incomplete |
| Binary data case | If input.txt contains binary (images, PDFs) but not handled properly | Should still copy correctly in streams | Some encodings may corrupt data |

DIFFERENT APPROACHES:

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Description** | **Pros** | **Cons** |
| **1. Using Stream Piping (pipe())** | fs.createReadStream('input.txt').pipe(fs.createWriteStream('output.txt')) | Very simple, efficient for large files, automatic handling of backpressure | Less control over events compared to manual handling |
| **2. Using readFile + writeFile (Async)** | Read entire file with fs.readFile, then write with fs.writeFile | Easy for small files, simple to implement | Loads entire file into memory → bad for large files |
| **3. Using readFileSync + writeFileSync** | Same as above but synchronous | Quick for very small files | Blocks event loop, crashes for large files |
| **4. Using Streams without pipe() (manual)** | Manually listen to data, end, error events → write chunks to writable stream | More control over flow, can transform data during transfer | More code, complex error handling |
| **5. Using fs.promises.readFile with async/await** | Async/await syntax with promises to read → write | Clean syntax, integrates with async code | Same memory issue as readFile |
| **6. Using stream.pipeline() (safe piping)** | Built-in stream.pipeline() provides safer pipe with error handling | Handles errors better than raw pipe() | Slightly more verbose than simple pipe() |