

Assignment: 2

Task 1 : Hoisting in Variables

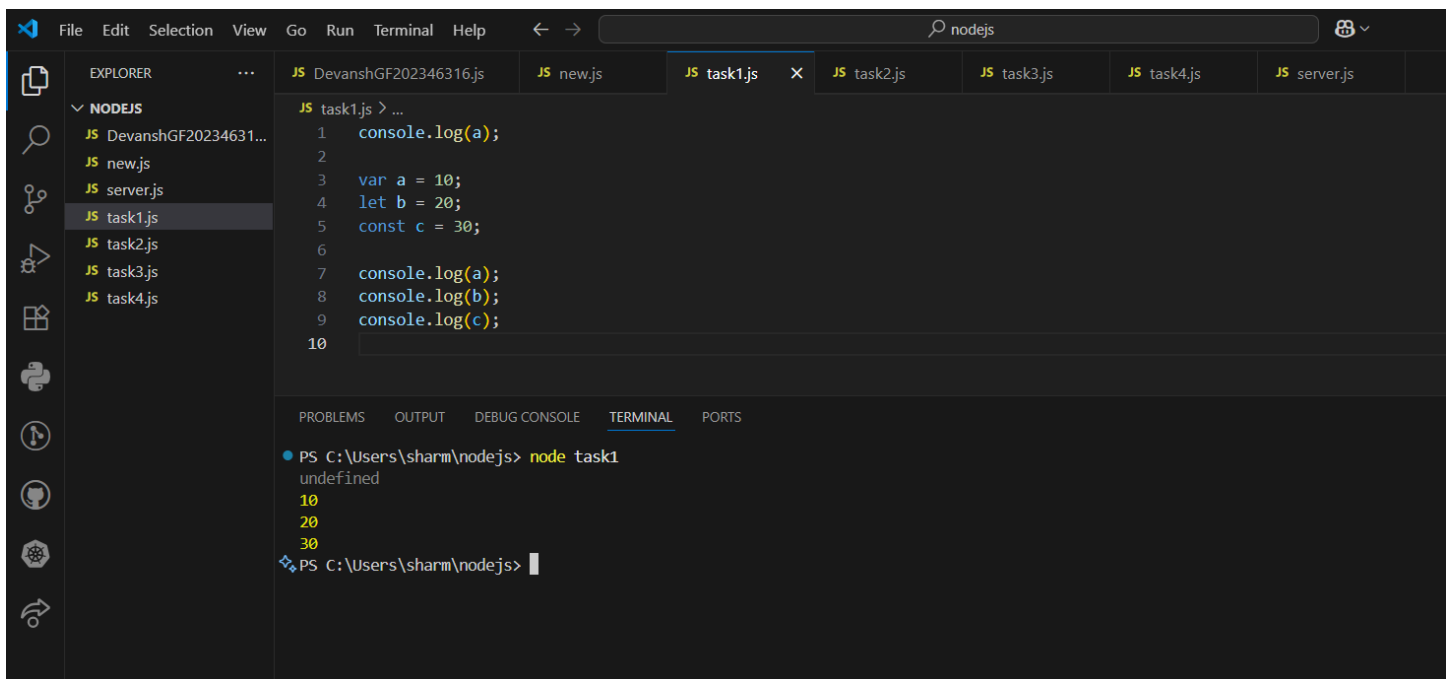
Write a Node.js program that demonstrates variable hoisting using var, let, and const.

Print a variable before it is declared

Show the difference between var, let, and const.

Explain the output.

OUTPUT:



```
File Edit Selection View Go Run Terminal Help
nodejs

EXPLORER
  NODEJS
    DevanshGF20234631...
    new.js
    server.js
    task1.js
    task2.js
    task3.js
    task4.js

JS task1.js > ...
1 console.log(a);
2
3 var a = 10;
4 let b = 20;
5 const c = 30;
6
7 console.log(a);
8 console.log(b);
9 console.log(c);
10

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\sharm\nodejs> node task1
undefined
10
20
30
PS C:\Users\sharm\nodejs>
```

Explanation:

- Here, **var** variables are set up early with a value of undefined, so using them before their line just gives undefined.
- Whereas, **let** and **const** are set up early too, but aren't usable until their line. If we use them before their line, it gives an error ("ReferenceError").

Task2 : Function Declarations vs Expressions

Create two functions in Node.js:

A function declaration (function add(a,b) {})

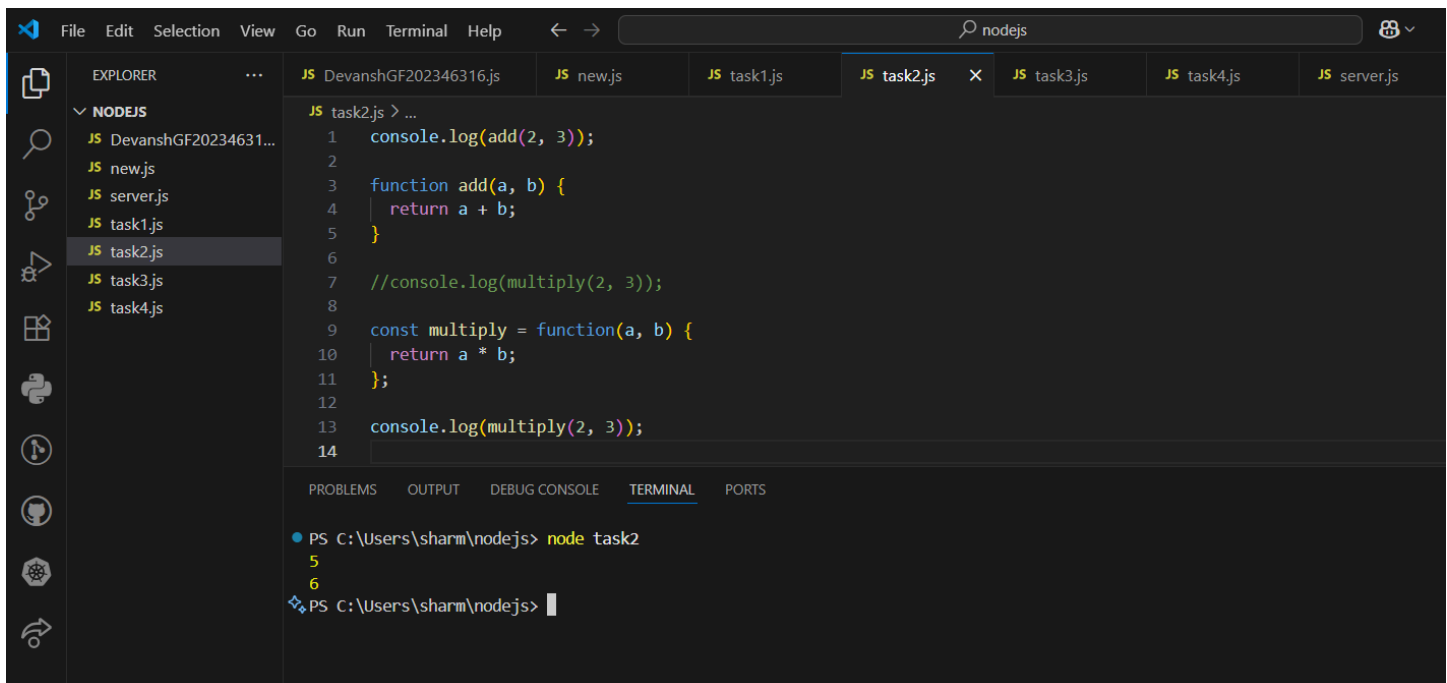
A function expression (const multiply = function(a,b) {})

Call both functions before and after their definitions.

Record what works and what fails.

Explain why.

OUTPUT :



```
JS task2.js > ...
1 console.log(add(2, 3));
2
3 function add(a, b) {
4   return a + b;
5 }
6
7 //console.log(multiply(2, 3));
8
9 const multiply = function(a, b) {
10   return a * b;
11 };
12
13 console.log(multiply(2, 3));
14
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
PS C:\Users\sharm\nodejs> node task2
5
6
PS C:\Users\sharm\nodejs>
```

- Calling add(2,3) before its declaration works, and prints 5.
- Calling multiply(2,3) before its assignment (if uncommented) fails, throwing a ("ReferenceError").
- Calling multiply(2,3) after initialization works, printing 6.

Explanation:

- The function declaration add is hoisted along with its implementation, so it can be called before its definition in the code.
- The function expression multiply using a const variable is hoisted only as a variable declaration without initialization (in the temporal dead zone), so calling it before assignment results in a ReferenceError.

Task 3: Arrow Functions vs Normal Functions

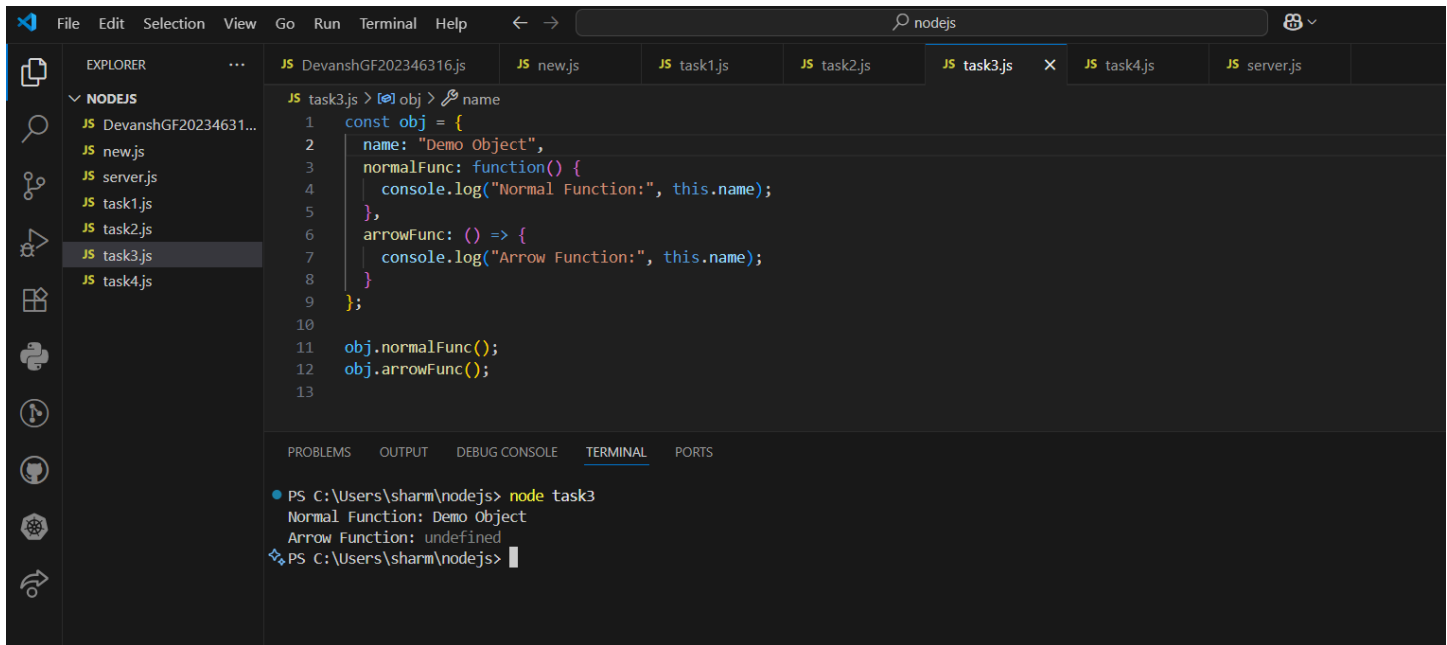
Create two functions inside an object

- One arrow function
- One normal function

Both should print this.

Compare their outputs when called as methods of the object.

OUTPUT:



The screenshot shows the VS Code editor with a file named `task3.js` open. The code defines an object `obj` with a `name` property and two methods: `normalFunc` (a normal function) and `arrowFunc` (an arrow function). Both methods use `this.name` to log the name. The `normalFunc` logs "Normal Function: Demo Object", while `arrowFunc` logs "Arrow Function: undefined". The terminal output confirms this, showing the output of `node task3`.

```
JS task3.js > [obj] obj > name
1  const obj = {
2    name: "Demo Object",
3    normalFunc: function() {
4      console.log("Normal Function:", this.name);
5    },
6    arrowFunc: () => {
7      console.log("Arrow Function:", this.name);
8    }
9  };
10
11  obj.normalFunc();
12  obj.arrowFunc();
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\sharm\nodejs> node task3
Normal Function: Demo Object
Arrow Function: undefined
PS C:\Users\sharm\nodejs>
```

Comparison:

Here,

1. `obj.normalFunc();` prints: (Normal Function: Demo Object)
2. `obj.arrowFunc();` prints: (Arrow Function: undefined)

Explanation:

- **Normal function** has its own `this` context, which is dynamically set to the object that called it (`obj`). Therefore, `this.name` correctly refers to "Demo Object".
- **Arrow function** does not have its own `this` context. Since this is the global or module scope (not the object), `this.name` is undefined.

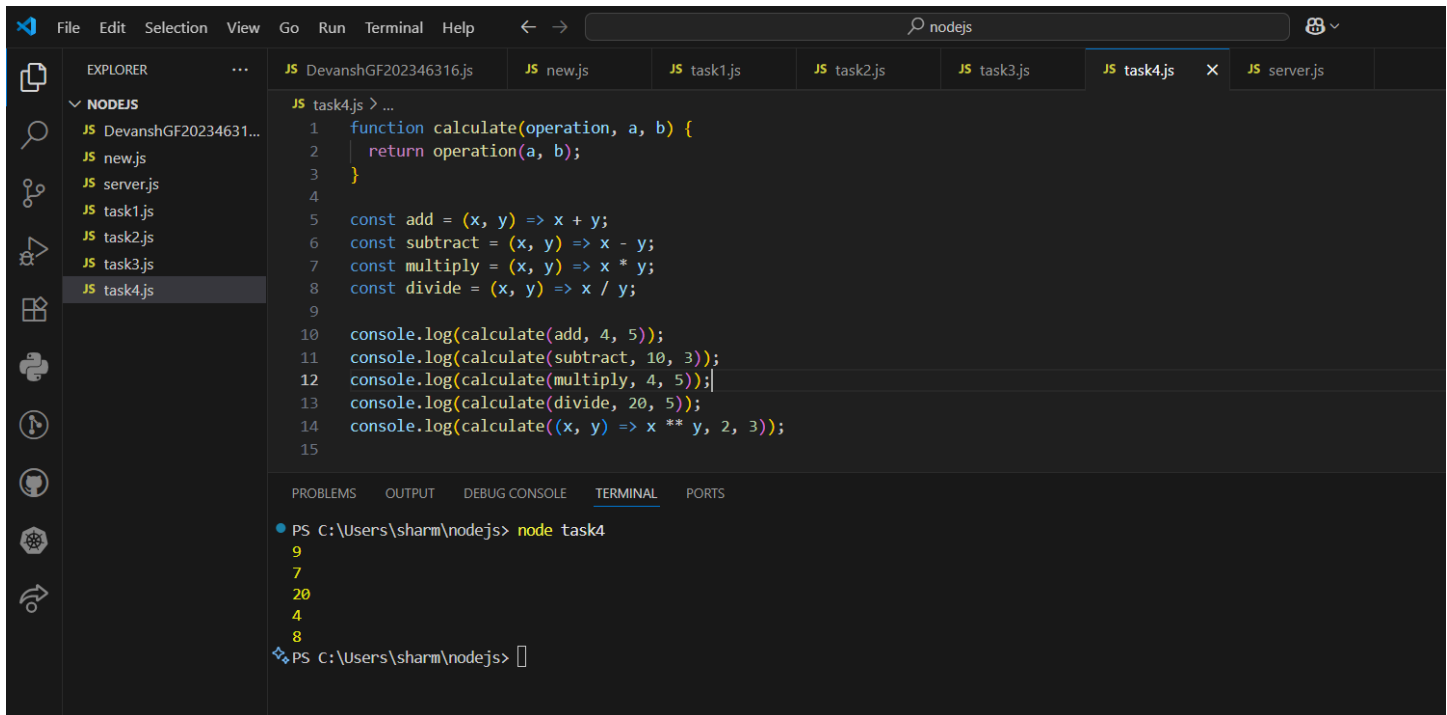
Task 4: Higher Order Functions

Write a Node js function `calculate(operation, a, b)` where `operation` is another function (like `add`, `subtract`).

Pass different functions to `calculate` and print results

Example `calculate((x,y) => x*y, 4, 5)` should return 20

OUTPUT:



The screenshot shows the Visual Studio Code editor with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'NODEJS' with files: 'DevanshGF20234631...', 'new.js', 'server.js', 'task1.js', 'task2.js', 'task3.js', and 'task4.js'. The 'task4.js' file is selected and its content is displayed in the editor. The code defines a `calculate` function that takes an `operation` function and two arguments `a` and `b`. It then defines four arrow functions: `add`, `subtract`, `multiply`, and `divide`. Finally, it logs the results of `calculate` with these operations and various arguments. The terminal at the bottom shows the command `node task4` being executed, which produces the output: 9, 7, 20, 4, 8.

```
JS task4.js > ...
1  function calculate(operation, a, b) {
2      |   return operation(a, b);
3      |
4      |
5      const add = (x, y) => x + y;
6      const subtract = (x, y) => x - y;
7      const multiply = (x, y) => x * y;
8      const divide = (x, y) => x / y;
9
10     console.log(calculate(add, 4, 5));
11     console.log(calculate(subtract, 10, 3));
12     console.log(calculate(multiply, 4, 5));
13     console.log(calculate(divide, 20, 5));
14     console.log(calculate((x, y) => x ** y, 2, 3));
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\sharm\nodejs> node task4
9
7
20
4
8
PS C:\Users\sharm\nodejs>
```