Practical-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.

Step1: Program

variables.js > ...

```javascript

console.log("Using var:");
console.log(myVar); // Output: undefined
var myVar = 10;
console.log(myVar); // Output: 10

console.log("\nUsing let:");
try {
    console.log(myLet); // Error: Cannot access 'myLet' before initialization
} catch (error) {
    console.log("Error:", error.message);
}
let myLet = 20;
console.log(myLet); // Output: 20


console.log("\nUsing const:");
try {
    console.log(myConst); // Error: Cannot access 'myConst' before initialization
} catch (error) {
    console.log("Error:", error.message);
}
const myConst = 30;
console.log(myConst); // Output: 30
```

Step 2: The difference between var let and const .

```
o/variables.js .js"
Using var:
undefined
10

Using let:
Error: Cannot access 'myLet' before initialization
20

Using const:
Error: Cannot access 'myConst' before initialization
30
```

Step 3: Output Explain.

- Var- Variables declared with var are hoisted to the top of their scope and initialized with undefined. That's why printing my Var before declaration gives undefined (not an error).
- Let and const -These are also hoisted, but they stay in a "temporal dead zone" (TDZ) until the declaration is reached. Accessing them before declaration causes a Reference Error.

Task 2: 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) (f)
Call both functions before and after their definitions.
Record what works and what fails.
Explain why.

Step1:Two function. Call both function

JS Decalartion.js > ...

```javascript
console.log("Function Declaration:");
try {
    console.log(add(2, 3));
} catch (error) {
    console.log("Error:", error.message);
}

function add(a, b) {
    return a + b;
}
console.log(add(5, 7));

// Function Expression
console.log("\nFunction Expression:");
try {
    console.log(multiply(2, 3)); // Error: Cannot access 'multiply' before initialization
} catch (error) {
    console.log("Error:", error.message);
}

const multiply = function (a, b) {
    return a * b;
};

console.log(multiply(4, 6));
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
deDemo/Decalartion.js .js"
Function Declaration:
5
12

Function Expression:
Error: Cannot access 'multiply' before initialization
24
```

Code
Code
Code

Step2. Explaining why this happens .

1.Function Declaration (function add(a,b){})

- Function declarations are **hoisted completely** (both name and body).

That's **Function Expression (const multiply = function(a,b){})**

- Function why calling add(2,3) **before** its definition still works.

2. Function Expression (`const multiply = function(a,b){}`)

- Function expressions behave like normal variables declared with let or const.
- They are hoisted but kept in the **temporal dead zone (TDZ)** until the line of initialization.

- That's why calling multiply(2,3) **before** definition throws an error, but works fine **after** definition.

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.

Step1. Create two functions .

JS arrow function .js > ...

```js
const obj = {
    name: "Ishan ",

    // Normal Function
    normalFunc: function () {
        console.log("Normal Function this.name:", this.name);
        console.log("Normal Function this:", this);
    },

    // Arrow Function
    arrowFunc: () => {
        console.log("Arrow Function this.name:", this.name);
        console.log("Arrow Function this:", this);
    }
};

console.log("Calling normal function:");
obj.normalFunc();

console.log("\nCalling arrow function:");
obj.arrowFunc();
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Normal Function this.name: Ishan
Normal Function this: {
  name: 'Ishan ',
  normalFunc: [Function: normalFunc],
  arrowFunc: [Function: arrowFunc]
}

Calling arrow function:
Arrow Function this.name: undefined
Arrow Function this: {}
```

Step2. Compare there output .

1.Normal Function

- In a method call like obj.normalFunc(), this refers to the object obj.
- So this.name prints "Tanmay".

2. **Arrow Function**

- Arrow functions **do not have their own this**.
- They use the this from the surrounding scope (lexical this).
- In Node.js (in strict mode), the top-level this is undefined.
- That’s why this.name is undefined and this is not the object.

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.

Step1. Function Calculate & print result .

JS Higher order functions .js > calculate

```javascript
function calculate(operation, a, b) {
    return operation(a, b);
}
function add(x, y) {
    return x + y;
}
function subtract(x, y) {
    return x - y;
}
function multiply(x, y) {
    return x * y;
}
function divide(x, y) {
    return y !== 0 ? x / y : "Cannot divide by zero";
}
// Testing with different operations
console.log("Add:", calculate(add, 10, 5));         // 15
console.log("Subtract:", calculate(subtract, 10, 5)); // 5
console.log("Multiply:", calculate(multiply, 4, 5));  // 20
console.log("Divide:", calculate(divide, 20, 4));     // 5
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
deDemo/Higher order functions .js"
Add: 15
Subtract: 5
Multiply: 20
Divide: 5
```