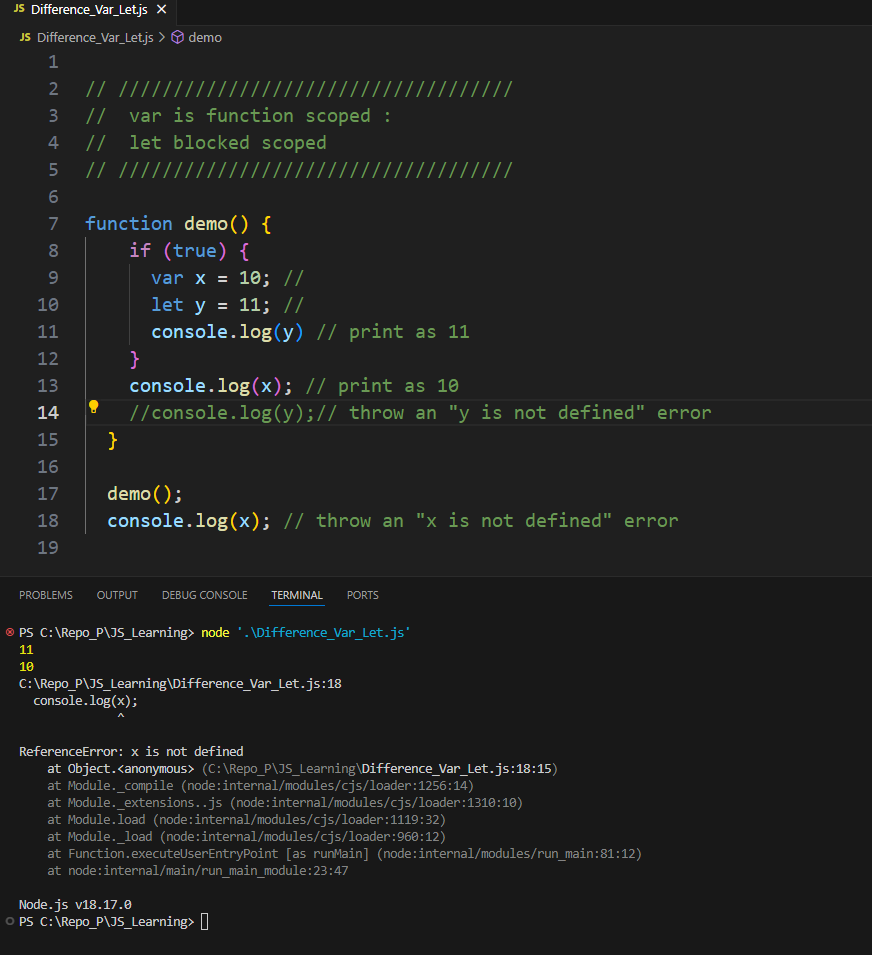
**Difference between VAR and LET**



**1. What are the key differences between var, let, and const?**

* **var:**
  + Function-scoped.
  + Can be re-declared and updated.
  + Hoisted to the top of its scope, initialized with **undefined**.
* **let:**
  + Block-scoped.
  + Cannot be re-declared in the same scope but can be updated.
  + Hoisted but not initialized (Temporal Dead Zone).
* **const:**
  + Block-scoped.
  + Cannot be re-declared or updated (immutable binding).
  + Hoisted but not initialized (Temporal Dead Zone).

**2. Explain closures in JavaScript.**

* A closure is a function that remembers its lexical scope even when the function is executed outside that scope. This enables functions to have private variables.

javascript

Copy code

function outerFunction() { let outerVariable = 'I am outside!'; function innerFunction() { console.log(outerVariable); // 'I am outside!' } return innerFunction; } const myInnerFunction = outerFunction(); myInnerFunction();

**3. What is the difference between == and ===?**

* **==** checks for value equality with type coercion.
* **===** checks for both value and type equality (strict equality).

javascript

Copy code

5 == '5'; // true 5 === '5'; // false

**4. What is the purpose of use strict?**

* **use strict** enforces stricter parsing and error handling in your JavaScript code, making it easier to write secure JavaScript. It:
  + Prevents the use of undeclared variables.
  + Throws errors for certain silent JavaScript mistakes.
  + Disallows some syntax likely to be defined in future versions of ECMAScript.

**5. How does prototypal inheritance work in JavaScript?**

* Every object in JavaScript has a prototype, which is another object that it inherits methods and properties from. This chain of prototypes ends when an object's prototype is **null**.

javascript

Copy code

function Person(name) { this.name = name; } Person.prototype.sayHello = function() { console.log(`Hello, my name is ${this.name}`); }; const alice = new Person('Alice'); alice.sayHello(); // Hello, my name is Alice

**6. Explain the concept of event delegation.**

* Event delegation is a technique in which a single event listener is added to a parent element to manage events from its child elements. It leverages the event bubbling mechanism, where events propagate up the DOM tree.

javascript

Copy code

document.getElementById('parent').addEventListener('click', function(event) { if (event.target && event.target.matches('button.className')) { // Handle the button click } });

**7. What is the difference between null and undefined?**

* **undefined** means a variable has been declared but not assigned a value.
* **null** is an assignment value that represents no value or no object.

javascript

Copy code

let a; console.log(a); // undefined let b = null; console.log(b); // null

**8. How do you handle asynchronous code in JavaScript?**

* Asynchronous code can be handled using callbacks, Promises, and async/await.

**Callbacks:**

javascript

Copy code

function fetchData(callback) { setTimeout(() => { callback('data'); }, 1000); } fetchData((data) => { console.log(data); });

**Promises:**

javascript

Copy code

function fetchData() { return new Promise((resolve, reject) => { setTimeout(() => { resolve('data'); }, 1000); }); } fetchData().then(data => { console.log(data); });

**async/await:**

javascript

Copy code

async function fetchData() { let data = await new Promise((resolve, reject) => { setTimeout(() => { resolve('data'); }, 1000); }); console.log(data); } fetchData();

**9. What is a promise in JavaScript?**

* A promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It has three states: **pending**, **fulfilled**, and **rejected**.

javascript

Copy code

let promise = new Promise((resolve, reject) => { let success = true; if (success) { resolve("Operation successful"); } else { reject("Operation failed"); } }); promise.then((message) => { console.log(message); // Operation successful }).catch((message) => { console.log(message); // Operation failed });

**10. What is the event loop in JavaScript?**

* The event loop is a mechanism that allows JavaScript to perform non-blocking operations by offloading operations to the system kernel whenever possible. It handles the execution of multiple chunks of your program over time, managing asynchronous callbacks.

javascript

Copy code

console.log('Start'); setTimeout(() => { console.log('Middle'); }, 0); console.log('End'); // Output: // Start // End // Middle

**11. What is hoisting in JavaScript?**

* Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope (script or function). Only declarations are hoisted, not initializations.

javascript

Copy code

console.log(a); // undefined var a = 5; console.log(b); // ReferenceError: Cannot access 'b' before initialization let b = 10;

**12. Explain call, apply, and bind methods.**

* **call**: Invokes a function with a given **this** value and arguments provided individually.

javascript

Copy code

function greet(greeting, punctuation) { console.log(`${greeting}, ${this.name}${punctuation}`); } const person = { name: 'Alice' }; greet.call(person, 'Hello', '!'); // Hello, Alice!

* **apply**: Invokes a function with a given **this** value and arguments provided as an array.

javascript

Copy code

greet.apply(person, ['Hi', '!!']); // Hi, Alice!!

* **bind**: Creates a new function that, when called, has its **this** value set to the provided value, with a given sequence of arguments.

javascript

Copy code

const greetAlice = greet.bind(person, 'Hey'); greetAlice('?'); // Hey, Alice?

**13. What is the difference between map, filter, and reduce?**

* **map**: Creates a new array with the results of calling a provided function on every element in the calling array.

javascript

Copy code

const numbers = [1, 2, 3]; const doubled = numbers.map(num => num \* 2); // [2, 4, 6]

* **filter**: Creates a new array with all elements that pass the test implemented by the provided function.

javascript

Copy code

const numbers = [1, 2, 3, 4]; const even = numbers.filter(num => num % 2 === 0); // [2, 4]

* **reduce**: Executes a reducer function on each element of the array, resulting in a single output value.

javascript

Copy code

const numbers = [1, 2, 3, 4]; const sum = numbers.reduce((total, num) => total + num, 0); // 10

**14. What are arrow functions and how do they differ from regular functions?**

* Arrow functions are a concise syntax for writing function expressions. They do not have their own **this**, **arguments**, **super**, or **new.target** binding.

javascript

Copy code

const add = (a, b) => a + b; console.log(add(2, 3)); // 5 function RegularFunction() { console.log(this); // refers to the calling context } const ArrowFunction = () => { console.log(this); // inherits `this` from the enclosing lexical context };

**15. What are JavaScript modules and how do you use them?**

* JavaScript modules allow you to split your code into separate files. This helps to manage dependencies and improve maintainability. Modules are imported and exported using **import** and **export** statements.

**Module file (math.js):**

javascript

Copy code

export const add = (a, b) => a + b; export const subtract = (a, b) => a - b;

**Main file (app.js):**

javascript

Copy code

import { add, subtract } from './math.js'; console.log(add(2, 3)); // 5 console.log(subtract(5, 2)); // 3

**16. What is async/await and how does it work?**

* **async/await** is syntactic sugar built on Promises, making asynchronous code easier to write and understand. An **async** function returns a Promise, and **await** pauses the execution of the async function until the Promise is resolved.

javascript

Copy code

async function fetchData() { try { const response = await fetch('https://api.example.com/data'); const data = await response.json(); console.log(data); } catch (error) { console.error('Error fetching data', error); } } fetchData();

30-May-2024

In JavaScript, callbacks, promises, and async/await are different ways to handle asynchronous operations. Here are the differences between them along with examples for each:

**Callbacks**

**Definition**: A callback is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

**Pros**:

* Simple and widely used.
* Good for handling simple asynchronous operations.

**Cons**:

* Can lead to "callback hell" or "pyramid of doom" if there are multiple nested asynchronous operations.
* Harder to maintain and debug.

**Example**:

function fetchData(callback) {

setTimeout(() => {

callback("Data received");

}, 1000);

}

fetchData((data) => {

console.log(data); // Output: Data received

});