SM
SOUVIK MITRA

February 2024

@THECODEMITTER

Mastering the Essentials: 7 Key JavaScript Concepts

Presented by :
Souvik Mitra

Swipe →

Hoisting

Hoisting in JavaScript is a mechanism where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed.

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

myFunction(); // Output: "Hello, world!"
function myFunction() {
  console.log("Hello, world!");
}
```

Despite being declared later in the code, the variable myVar is hoisted to the top of its containing scope, resulting in the output undefined when logged before its initialization. The function myFunction is hoisted to the top of its containing scope, allowing it to be invoked before its actual declaration in the code.

Closure

A closure is a combination of a function and the lexical environment within which that function was declared. It allows a function to access variables from its surrounding scope even after the outer function has finished executing.

```
function outerFunction() {
  var outerVariable = "I am from outerFunction";

  function innerFunction() {
    console.log(outerVariable); // Access outerVariable from outerFunction
  }

  return innerFunction;
}

var closureExample = outerFunction();
closureExample(); // Output: I am from outerFunction
```

outerFunction defines an inner function innerFunction.
innerFunction has access to outerVariable, even though outerVariable is declared in the outer scope of outerFunction.
When outerFunction is invoked and closureExample is assigned the returned innerFunction, closureExample becomes a closure, retaining access to outerVariable even though outerFunction has finished executing.

Scope

Scope in JavaScript refers to the visibility and accessibility of variables and functions within a particular context or region of code.

Global Scope: Variables and functions declared outside of any function or block have global scope. They are accessible from anywhere in the code, including inside functions.

```
var globalVar = "I am global";

function greet() {
  console.log(globalVar); // Output: I am global
}

greet();
console.log(globalVar); // Output: I am global
```

Local Scope: Variables declared inside a function or block have local scope. They are only accessible within the function or block in which they are declared.

```
function greet() {
  var localVar = "I am local";
  console.log(localVar); // Output: I am local
}

greet();
console.log(localVar); // Throws an error: localVar is not defined
```

Block scope: It refers to the visibility and accessibility of variables declared using the let and const keywords within a block of code, such as within curly braces {}. Unlike variables declared with var, which have function scope or global scope, variables declared with let and const have block scope, meaning they are only accessible within the block in which they are defined.

```
function example() {
  if (true) {
    let blockVar = "I am inside a block";
    console.log(blockVar); // Output: I am inside a block
  }
  console.log(blockVar); // Throws an error: blockVar is not defined
}
```

Callbacks

A callback function is a function that is passed as an argument to another function and is executed after a particular task or event has occurred. It allows for asynchronous programming by providing a way to handle actions that occur at unpredictable times, such as the completion of an asynchronous operation like fetching data from a server or handling user interactions like clicking a button.

```
function fetchData(callback) {
  // Simulating an asynchronous operation (e.g., fetching data from a server)
  setTimeout(function () {
    const data = { name: "John", age: 30 };
    callback(data); // Execute the callback function with the fetched data
  }, 1000);
}

// Define a callback function to handle the fetched data
function handleData(data) {
  console.log("Received data:", data);
}

// Call the fetchData function and pass the handleData function as a callback
fetchData(handleData);
```

Promises

Promises were introduced as a solution to callback hell where multiple nested callback functions are used to handle asynchronous operations. It allows you to handle asynchronous operations more easily and efficiently compared to traditional callback-based approaches.

A Promise can be in one of three states:

1. Pending: Initial state, neither fulfilled nor rejected.
2. Fulfilled: The operation completed successfully.
3. Rejected: The operation failed.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const randomNumber = Math.random();
    if (randomNumber > 0.5) {
      resolve(randomNumber);
    } else {
      reject(new Error("Random number is too low"));
    }
  }, 1000);
});
```

Promises have a then() method, which allows you to specify what to do when the promise is fulfilled or rejected. This method takes two callback functions as arguments: one for handling the fulfillment (onFulfilled) and one for handling the rejection (onRejected). The catch() method is used to handle any errors that occur during the Promise chain.

```
myPromise.then(
  (result) => {
    console.log("Promise fulfilled with result:", result);
  },
  (error) => {
    console.error("Promise rejected with error:", error.message);
  }
);
```

Promises provide a more structured and intuitive way to work with asynchronous code, making it easier to manage complex asynchronous operations and handle errors more effectively.

Async & Await

Async/await is a modern JavaScript feature that allows you to write asynchronous code in a synchronous-like manner, making it easier to read and understand compared to traditional Promise-based code. It is built on top of Promises and provides a more intuitive syntax for handling asynchronous operations.

```
// Asynchronous function that returns a Promise
function fetchData() {
  return new Promise((resolve) => {
    // Simulate an asynchronous operation (e.g., fetching data from an API)
    setTimeout(() => {
      const data = { name: "John", age: 30 };
      resolve(data); // Resolve the Promise with the fetched data
    }, 1000);
  });
}

// Using async/await to handle asynchronous code
async function fetchDataAsync() {
  try {
    // Await the completion of the asynchronous operation
    const data = await fetchData();
    console.log("Fetched data:", data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}

// Call the asynchronous function
fetchDataAsync();
```

Currying

Currying is a technique in JavaScript where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument. This allows you to create new functions by partially applying the original function with some of its arguments, resulting in a more flexible and reusable code.

```
// Original function with multiple arguments
function add(x, y, z) {
  return x + y + z;
}

// Curried version of the add function
function curriedAdd(x) {
  return function (y) {
    return function (z) {
      return x + y + z;
    };
  };
}

// Usage of the curriedAdd function
const add5 = curriedAdd(5); // Partially apply the add function with x = 5
const add5And6 = add5(6); // Partially apply the resulting function with y = 6
const result = add5And6(7); // Call the final function with z = 7

console.log(result); // Output: 18 (5 + 6 + 7)
```

thecodemitter.bio.link