QUAIL JAVASCRIPT

ARIHARASUDHAN

arihara-sudhan.github.io/books.html

INTRODUCTION

According to WIKI, JavaScript, often abbreviated as JS, is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. As of 2023, 98.7% of websites use JavaScript on the client side for webpage behavior, often incorporating 3rd party libraries. All major web browsers have a dedicated JavaScript engine to

execute the code on users' devices. JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard. It has dynamic typing, prototype-based object-orientation, and first-class functions. Java and JavaScript are similar in name, syntax, and respective standard libraries but distinct by design.

VARIABLES

Variables are used to store values under a given name. We can declare variables using the var, let, or const keywords.

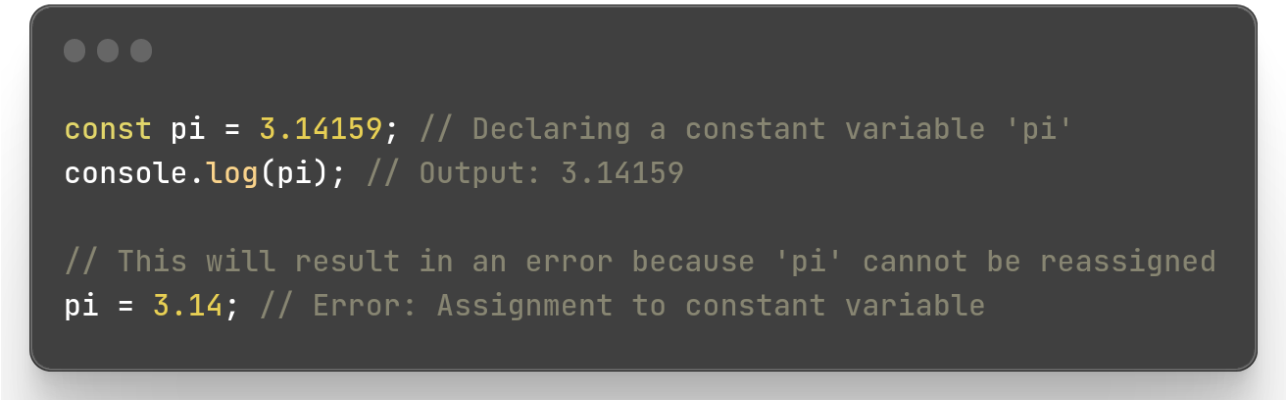
★ var declares a variable globally or locally to a function, regardless of block scope.

```
var age = 25; // Declaring a variable 'age' and assigning a value
console.log(age); // Output: 25

// Variables declared with 'var' can be redeclared
var age = 30;
console.log(age); // Output: 30
```

★ let declares a variable with block scope, which means it is limited to the block where it's defined.

```
let name = "Alice"; // Declaring a variable 'name' and assigning a value
console.log(name); // Output: Alice

// Variables declared with 'let' can be reassigned but not redeclared
name = "Bob";
console.log(name); // Output: Bob
```

★ const declares a variable with block scope, just like let, but its value cannot be reassigned after initialization.



```
const pi = 3.14159; // Declaring a constant variable 'pi'
console.log(pi); // Output: 3.14159

// This will result in an error because 'pi' cannot be reassigned
pi = 3.14; // Error: Assignment to constant variable
```

It's a good practice to use const by default for variables that won't be reassigned and use let when you need a variable that can change its value. Avoid using var in modern JavaScript because it doesn't have block scope and can lead to unexpected behavior.

Remember that variable names are case-sensitive, can include letters, digits, underscores, and the dollar sign, and must begin with a letter, underscore, or dollar sign.

DATATYPES

Datatype represents the type of the data. In JavaScript, there are several data types that we can use to store and manipulate different kinds of data.

Here are some of the most common data types in JavaScript along with examples and explanations:

★ Number: Used to represent both integer and floating-point numbers.

```
let age = 25;  // Integer
let temp = 98.6;  // Float
```

★ String: Used to represent textual data enclosed in single or double quotes.

```
// Using Double quotes
let name = "John";
// Using Single quotes
let city = 'New York';
```

★ Boolean: Represents a binary value, either true or false.

```
let isStudent = true;
let isWorking = false;
```

★ Array: An ordered collection of values, which can be of different data types (in JS).

```
let fruits = ["apple", "banana", "orange"];
let numbers = [1, 2, 3, 4, 5];
```

★ Object: A collection of key-value pairs, where keys are strings (properties) and values can be of any data type.

```
let person = {
  name: "Alice",
  age: 30,
  isStudent: false
};
```

★ Undefined: Represents that a variable has been declared but has no assigned value.

```
let x;
console.log(x); // undefined
```

★ Null: Represents an intentional absence of any object value or no value at all.

```
let car = null;
```

★ Function: A callable object that can execute a block of code.

```
function greet(name) {
  return "Hello, "+name+"!"; }
```

★ Symbol: A unique and immutable data type often used as object property keys.

```
const uniq = Symbol("hey");
```

★ BigInt: A data type that can represent arbitrarily large integers.

```
const big = 134567890123450;
```

★ Date: Used to work with dates and times.

```
const curDate = new Date();
```

★ RegExp: Represents regular expressions for pattern matching.

```
const pattern = /ab+c/;
```

OPERATORS

Operators are symbols that allow us to perform operations on variables and values.

Here are some of the most commonly used operators in JavaScript along with examples and explanations:

★ Arithmetic Operators:

+ (Addition)

```
let sum = 5 + 3;    // 8
```

- (Subtraction)

```
let difference = 10 - 3;    // 7
```

(Multiplication)

```
let product = 4  6;    // 24
```

/ (Division)

```
let quotient = 20 / 5;   // 4
```

% (Modulus)

```
let remainder = 10 % 3;   // 1
```

★ Assignment Operators:

= (Assignment)

```
let x = 5;
```

+=, -=, =, /=, %= (Compound Assignment): Performs an operation and assigns.

```
let y = 10;
y += 2; // y is now 12
```

★ Comparison Operators:

== (Equality): Compares two values for equality, but performs type coercion.

```
console.log(5 == "5"); // true
```

(coerced to the same value)

=== (Strict Equality): Compares two values for equality without type coercion.

```
console.log(5 === "5"); // false
```

(different types)

!= (Inequality): Compares two values for inequality with type coercion.

```
console.log(5 != "5"); // false
```

(coerced to the same value)

!== (Strict Inequality): Compares two values for inequality without type coercion.

```
console.log(5 !== "5"); // true
```

(different types)

★ Logical Operators:

&& (Logical AND): Returns true if both operands are true.

```
let isTrue= true && true; //true
```

|| (Logical OR): Returns true if at least one operand is true.

```
let isTrue = true || false; //true
```

! (Logical NOT): Returns the opposite boolean value of the operand.

```
let isFalse = !true; //false
```

★ Unary Operators:

++ (Increment): Increases a variable's value by 1.

```
let count = 5;

count++; // count is now 6
```

-- (Decrement): Decreases a variable's value by 1.

```
let count = 5;

count--; // count is now 4
```

EXPRESSIONS

Expressions are combinations of values, operators, and variables that can be evaluated to produce a result.

★ Arithmetic Expressions

```
let additionResult = 5+3;

let subtractionResult = 10-2;

let multiplicationResult = 46;

let divisionResult = 12/2;
```

★ String Expressions

```
let greeting = "Hello, ";

let name = "John";

let message = greeting + name;
```

★ Comparison Expressions

```
let isEqual = 5 === 5; // true

let isNotEqual = 5!==10; // true
```

★ Logical Expressions

```
let True = true;

let False = false;

let res= True && False; // false
```

★ Ternary Expressions

```
let age = 18;

let can =(age>=18)? "Y":"N";
```

★ Function Call Expressions

```
function add(x, y) {
        return x + y; }

let sum = add(3, 4);
```

★ Array Expressions

```
let numbers = [1, 2, 3, 4, 5];

let firstNumber = numbers[0];
```

★ Object Property Access Expressions

```
let person = {
    name: "Alice",
    age: 30 };
let personName = person.name;
person.age = 31;
```

★ Template Literals

```
let item = "apple";
let quantity = 3;
let orderDetails = You ordered
${quantity} ${item}s.;
```

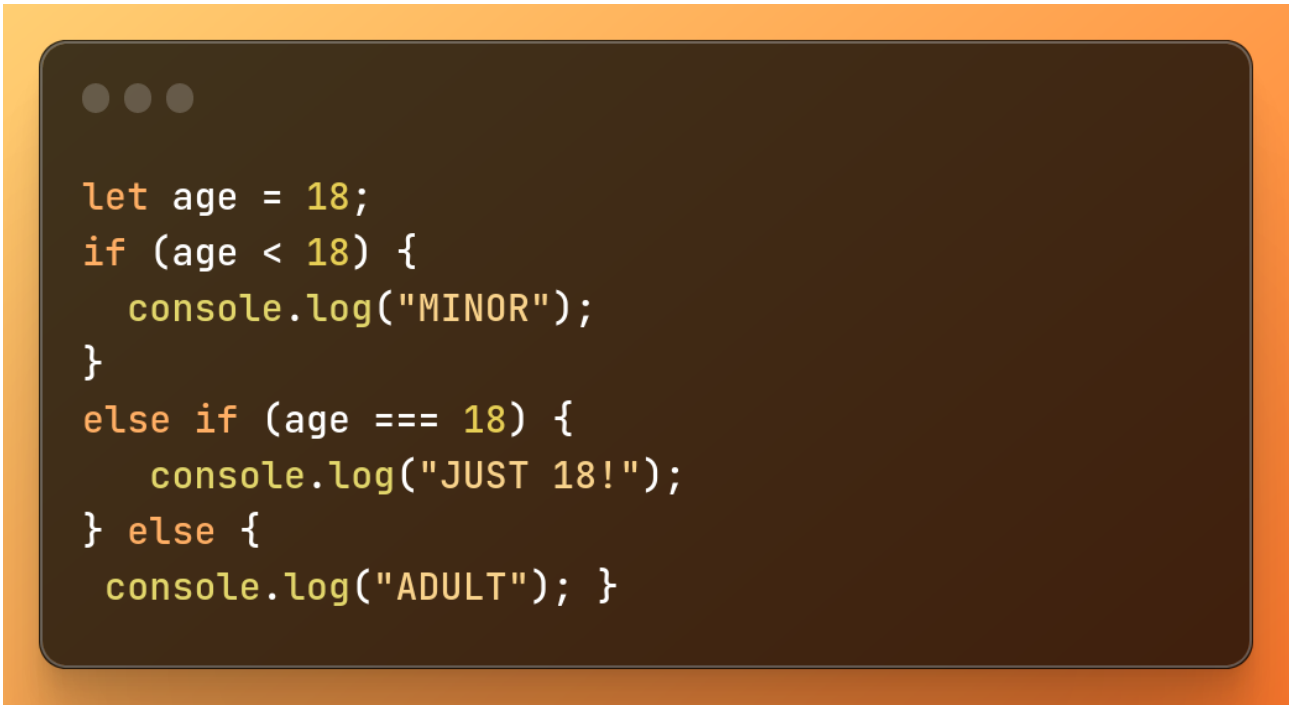
Some may seem a bit harder to get. We don't need to worry as we are about to explore them later in this petite book.

CONDITIONAL CONSTRUCTS

Conditional statements like if, else if, else, and switch are used in JavaScript to control the flow of our code based on certain conditions.

★ if Statement

The if statement is used to execute a block of code if a specified condition is true. It can be followed by an optional else if and else statement for more complex conditional logic.



```
let age = 18;
if (age < 18) {
  console.log("MINOR");
}
else if (age === 18) {
   console.log("JUST 18!");
} else {
 console.log("ADULT"); }
```

★ switch Statement

The switch statement is used when we have multiple conditions to check against a single value. It provides an alternative to a long series of else if statements.

```
let day = "Monday";
switch (day) {
    case "Monday":
        console.log("It's the start of the week.");
        break;
    case "Friday":
        console.log("It's almost the weekend!");
        break;
    default:
        console.log("It's another day.");
}
```

In this example, the code will execute the block corresponding to the value of day. These conditional statements are essential for implementing decision-making logic in our JavaScript programs. They allow us to control the execution of code based on various conditions, making our programs more versatile and responsive.

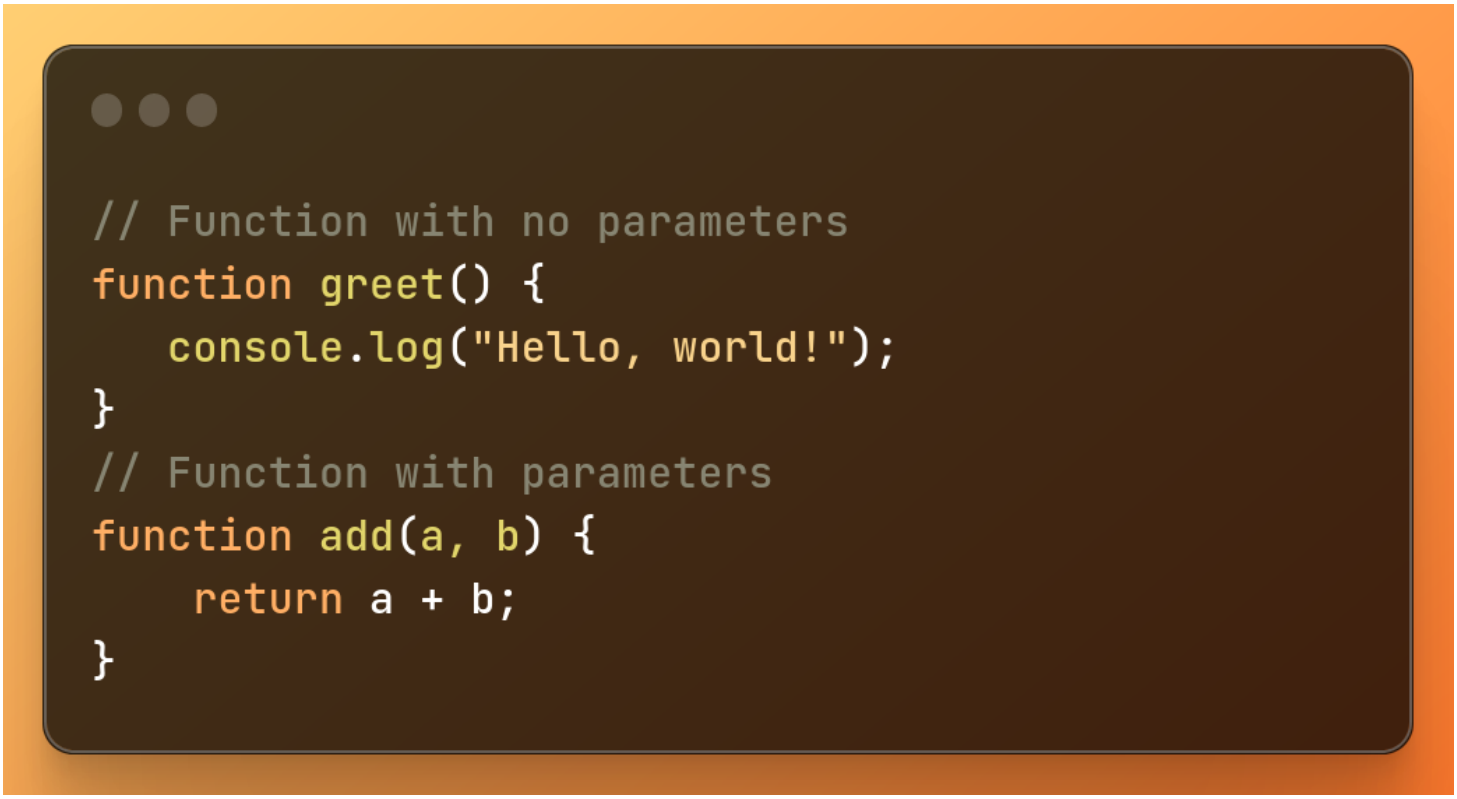
FUNCTIONS

Functions are reusable blocks of code that perform a specific task or set of tasks. They are fundamental to programming and allow us to organize and modularize our code.

★ Defining a Function

We can define a function using the function keyword followed by the function's name, a list of parameters enclosed in parentheses, and the function

body enclosed in curly braces {}. Parameters are optional, and a function can also have no parameters.



```
// Function with no parameters
function greet() {
   console.log("Hello, world!");
}
// Function with parameters
function add(a, b) {
    return a + b;
}
```

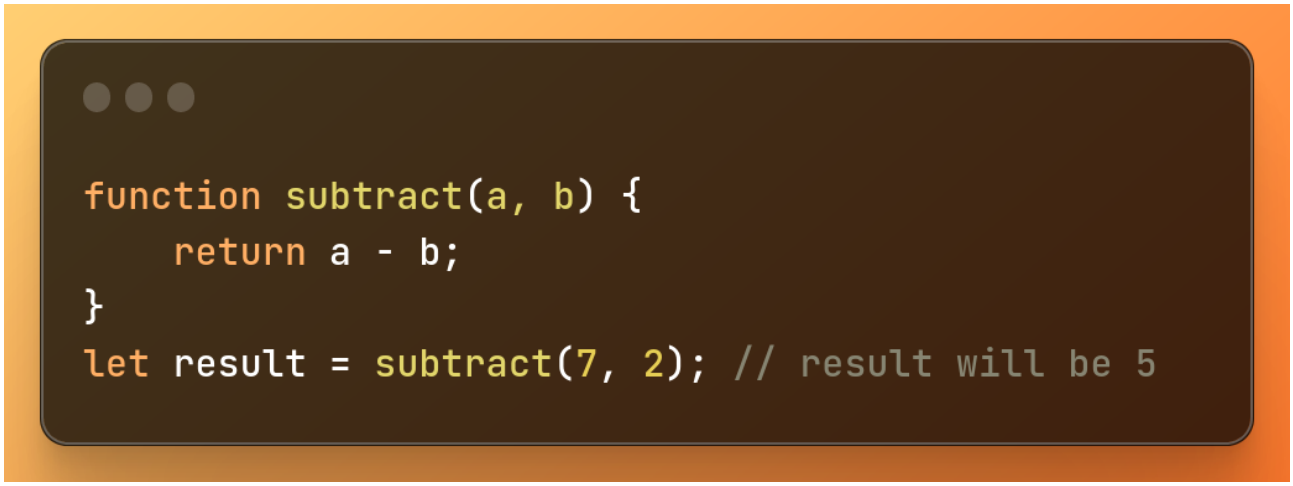
★ Calling a Function

To execute a function and make it perform its defined tasks, we call it by using its name followed by parentheses. If the function has parameters, we provide arguments within the parentheses.

```
greet(); // Calls the greet function
let sum = add(3, 4); // Calls the add function with arguments 3 and 4
```

★ Returning Values

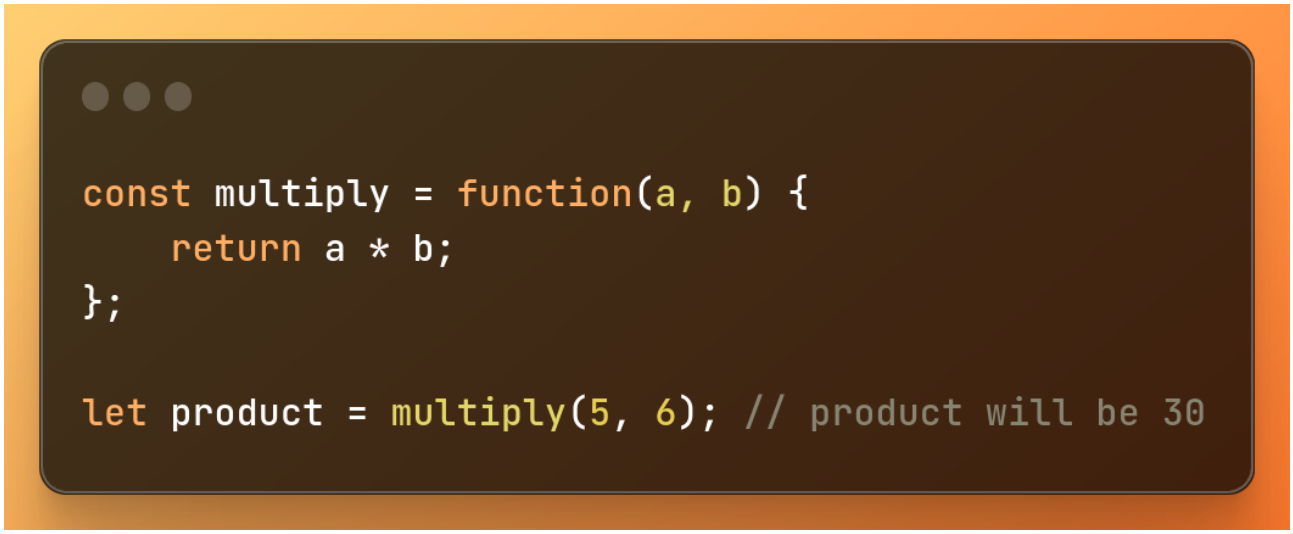
Functions can return values using the return statement. This allows you to get results or data from a function.



```
function subtract(a, b) {
    return a - b;
}
let result = subtract(7, 2); // result will be 5
```

★ Function Expressions

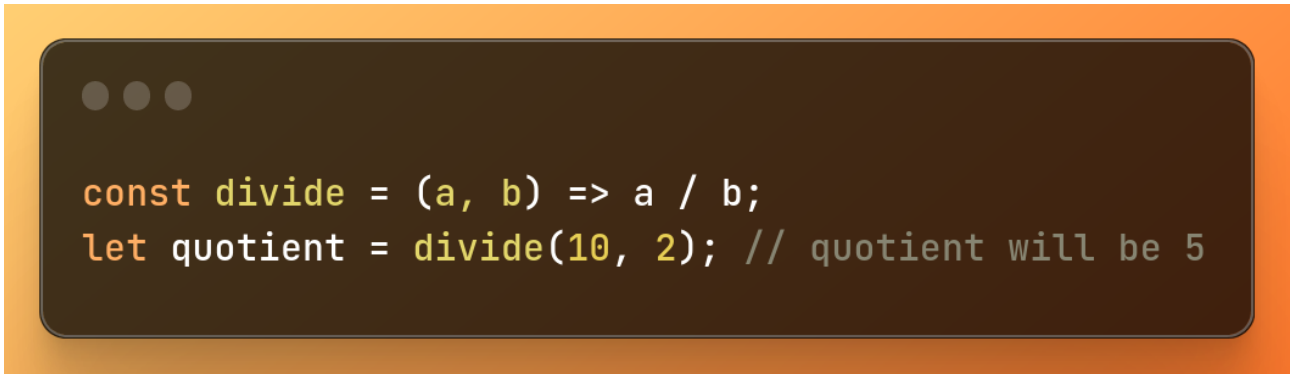
We can also define functions as expressions. In this case, we don't need to provide a function name (anonymous function), and we can assign the function to a variable.



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

let product = multiply(5, 6); // product will be 30
```

★ Arrow Functions (ES6)

Arrow functions provide a more concise way to write functions, especially for simple one-liners. They use the => syntax.



```
const divide = (a, b) => a / b;
let quotient = divide(10, 2); // quotient will be 5
```

★Immediately Invoked Function Expressions (IIFE)

An IIFE is a function that is defined and executed immediately after its creation.

The (function() { / code here / })(); syntax is commonly used to create an Immediately Invoked Function Expression (IIFE) in JavaScript. An IIFE is a function that is declared and executed immediately after its creation. It's often used to create a private scope for variables and functions, preventing them from polluting the global scope. Here's an example:

```
(function() {
    // This code is inside the IIFE and has its own private scope.
    let counter = 0;
    function incrementCounter() {
        counter++;
        console.log("Counter incremented to: " + counter);
    }

    function resetCounter() {
        counter = 0;
        console.log("Counter reset to: " + counter);
    }

    // Usage of the functions within the IIFE
    incrementCounter();
    incrementCounter();
    resetCounter();
})();
```

In this example, we've created an IIFE that encapsulates a private scope. Inside the IIFE, we have a counter variable and twofunctions,incrementCounter and resetCounter.

These functions can modify and access the counter variable but are not accessible from outside the IIFE. When we run this code, we'll see that it maintains its own counter, and the functions can be used to manipulate it. However, the counter variable and the functions themselves are not accessible in the global scope, helping to keep our code clean and prevent naming conflicts

with other parts of your program. IIFE is less commonly used in modern JavaScript development due to the introduction of block-scoped variables with let and const, as well as modules. However, it's still a useful technique in certain situations, especially for compatibility with older browsers or for creating isolated environments.

KUTTY TASK : 1 [A SIMPLE CALCULATOR]

```javascript
// Function to perform addition
function add(a, b) {
    return a + b;
}

// Function to perform subtraction
function subtract(a, b) {
    return a - b;
}

// Function to perform multiplication
function multiply(a, b) {
    return a * b;
}

// Function to perform division
function divide(a, b) {
    if (b === 0) {
        return "Cannot divide by zero!";
    }
    return a / b;
}

// Get user input for numbers and operation
const num1 = parseFloat(prompt("Enter the first number:"));
const num2 = parseFloat(prompt("Enter the second number:"));
const operation = prompt("Enter the operation (+, -, *, /):");

let result;

// Perform the selected operation
switch (operation) {
    case "+":
        result = add(num1, num2);
        break;
    case "-":
        result = subtract(num1, num2);
        break;
    case "*":
        result = multiply(num1, num2);
        break;
    case "/":
        result = divide(num1, num2);
        break;
    default:
        result = "Invalid operation";
}

// Display the result
console.log(`Result: ${result}`);
```

ARRAYS

Arrays are used to store and manage collections of values, which can be of any data type. JavaScript provides several built-in methods for manipulating arrays efficiently.

★ Creating Arrays

```
let fruits = ["apple", "banana"];
```

★ Accessing Elements

We can access elements in an array using their index, where the index starts from 0.

let firstFruit = fruits[0]; // "apple"

★ Modifying Elements

We can modify elements in an array by assigning new values to specific indexes.

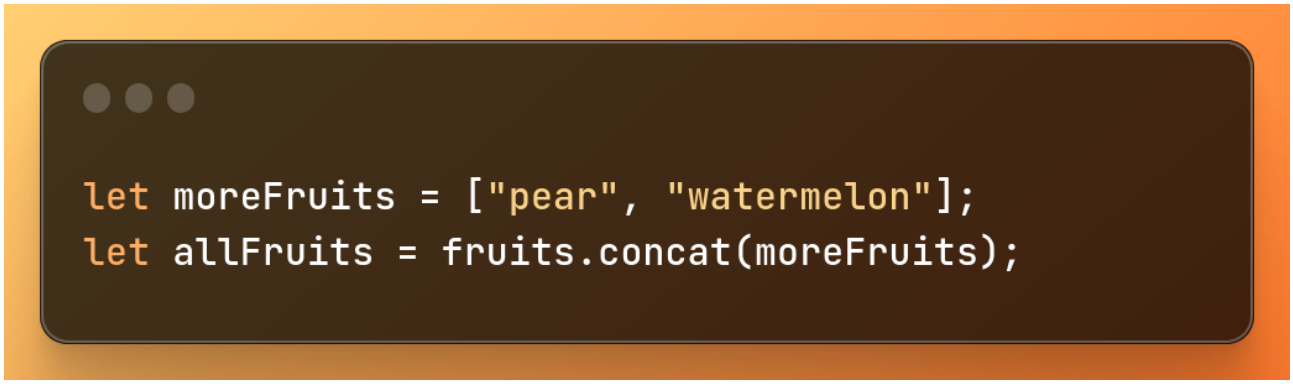
fruits[1] = "orange"; // Changes "banana" to "orange"

★Array Methods

1. push() and pop() : These methods add and remove elements from the end of an array.

2. unshift() and shift() : These methods add and remove elements from the beginning of an array.

3. concat() : This method combines two or more arrays and returns a new array.



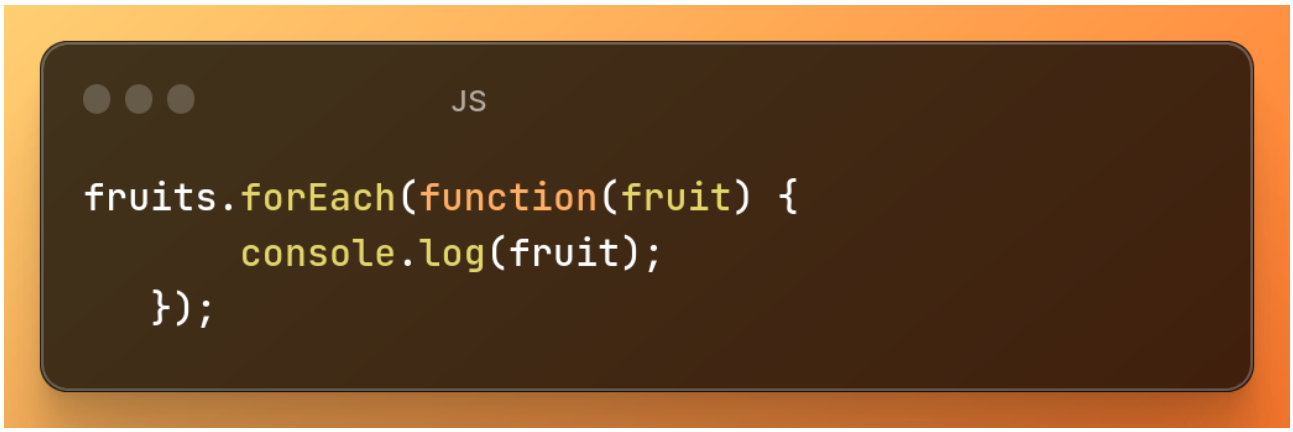
4. slice() : This method creates a new array by copying a portion of an existing array.

5. splice() : This method can add, remove, or replace elements at any position in the array.

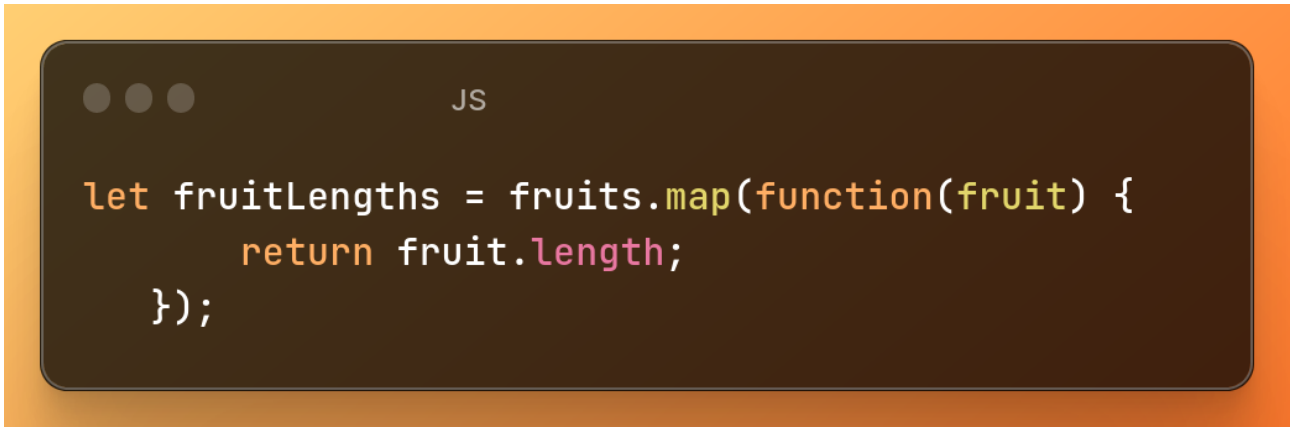
6. indexOf() & lastIndexOf()

These methods return the index of the first and last occurrence of a specified element in an array.

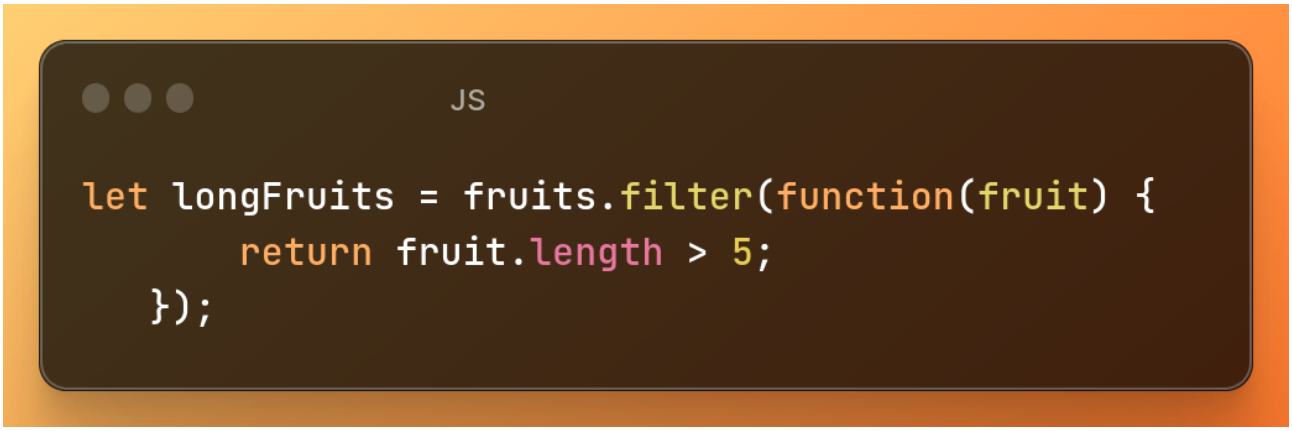
7. forEach() : This method iterates over each element in an array and applies a function to it.



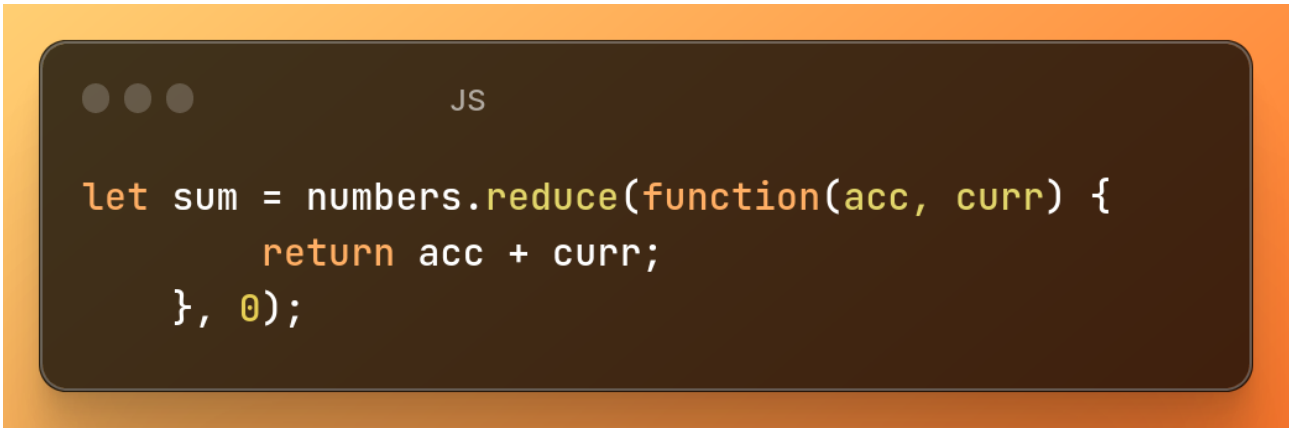
8. map(): This method creates a new array by applying a function to each element of the original array.



9. filter(): This method creates a new array with elements that pass a certain test (provided by a function).



10. reduce() and reduceRight(): These methods reduce an array to a single value by applying a function to each element in the array.



The reduce() method in JavaScript is used to iterate over an array and accumulate a single result (or value) by applying a provided function to each element of the array. This method is often used when we want to perform some operation on all the elements in an array and reduce them to a single

value, such as calculating the sum of all elements, finding the maximum or minimum value, or even constructing a new data structure. The basic syntax of the reduce() method is as follows:

```
array.reduce(callback[,initialVal])
```

callback : A function that is executed for each element in the array. This function takes four parameters:

accumulator : The accumulator stores the accumulated result. It starts with the initial value (if provided) or the first element of the array and is updated after each iteration.

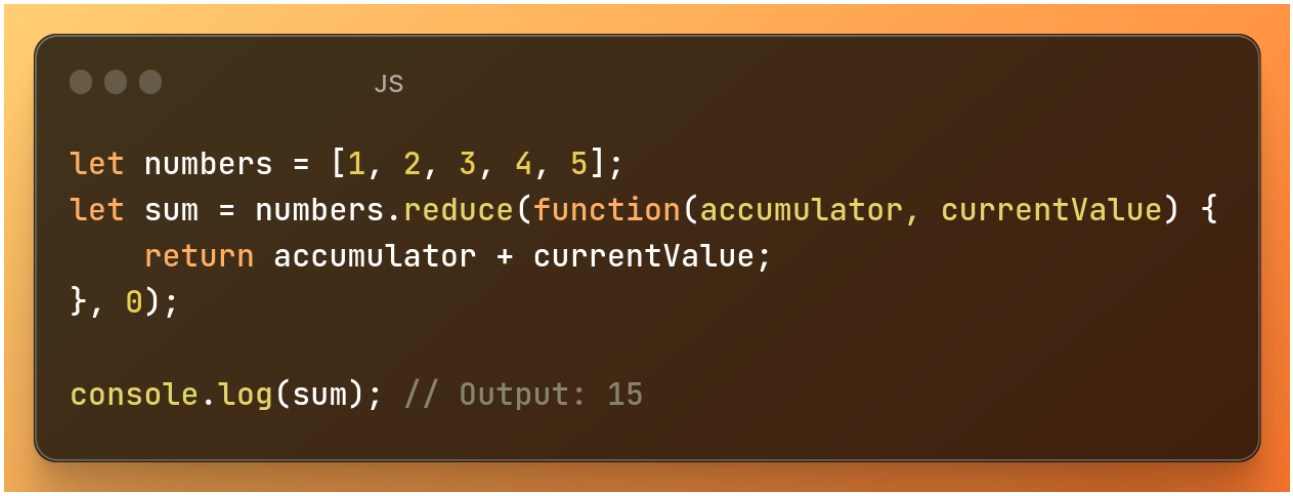
currentValue : The current element being processed in the array.

currentIndex(optional): The index of the current element being processed.

array(optional): The array on which reduce() was called.

initialVal(optional): An optional initial value for the accumulator. If not provided, the first element of the array will be used as the initial accumulator value, and the iteration will start from the second element. Here's an example that demonstrates how reduce() can be used to

calculate the sum of all elements in an array:



OBJECTS

Objects are a fundamental data type used to store and organize data. An object is a collection of key-value pairs, where each key (also called a property) is

associated with a value. These properties can be of various data types, including numbers, strings, functions, other objects, and more. Objects in JavaScript are used to represent real-world entities, and they are versatile for organizing and manipulating data.

★ Creating Objects

We can create an object using either object literal notation or the Object constructor.

Object Literal Notation:

```
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
};
```

Using Object Constructor:

```
let person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 30;
```

★ Accessing Properties

We can access the properties of an object using dot notation or square bracket notation.

Dot Notation:

```
console.log(person.firstName);
console.log(person.age); // 30
```

Square Bracket Notation:

```
console.log(person["lastName"]); // "Doe"
```

★ Modifying Properties

We can modify the values of object properties by assigning new values.

```
person.age = 31;
person["isStudent"] = true;
```

★ Adding Properties

We can add new properties to an existing object simply by assigning values to them.

```
person.city = "New York";
```

★ Removing Properties

We can delete properties from an object using the delete keyword.

```
delete person.city;
```

★ Object Methods

Objects can also have methods, which are functions associated with the object. In the following example, fullName is a method of the person object

that concatenates the first name and last name properties.

```JS
let person = {
    firstName: "John",
    lastName: "Doe",
    fullName: function() {
        return this.firstName + " " + this.lastName;
    }
};
console.log(person.fullName()); // "John Doe"
```

★ Nested Objects

Objects can contain other objects as properties, creating a nested structure.

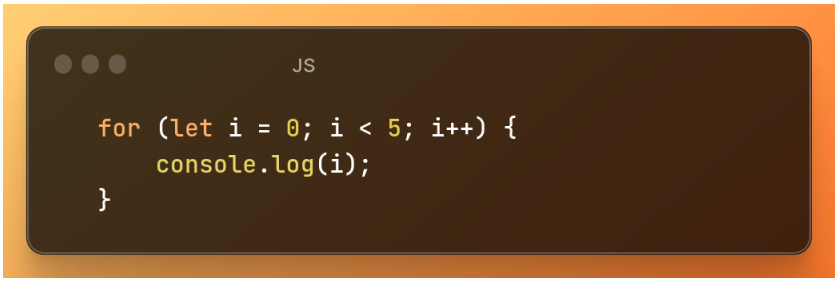
```
let address = {
    street: "123 Main St",
    city: "Cityville"
};
let person = {
    firstName: "John",
    lastName: "Doe",
    address: address
};
console.log(person.address.street
); // "123 Main St"
```

LOOPING CONSTRUCTS

Loops in JavaScript are used to repeatedly execute a block of code until a specified condition is met. JavaScript provides several types of loops to cater to different looping scenarios.

★ for Loop

The for loop is used for iterating over a range of values, typically based on a counter variable.

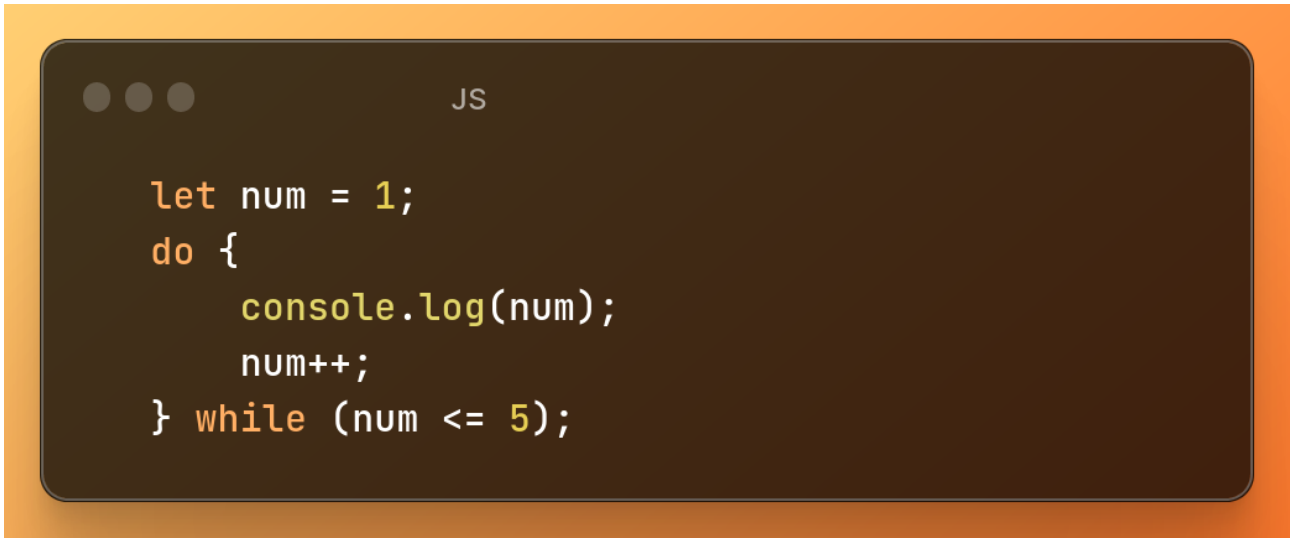


★ while Loop

The while loop repeatedly executes a block of code as long as a specified condition evaluates to true.

★ do...while Loop

Similar to the while loop, but it guarantees at least one execution of the block of code before checking the condition.

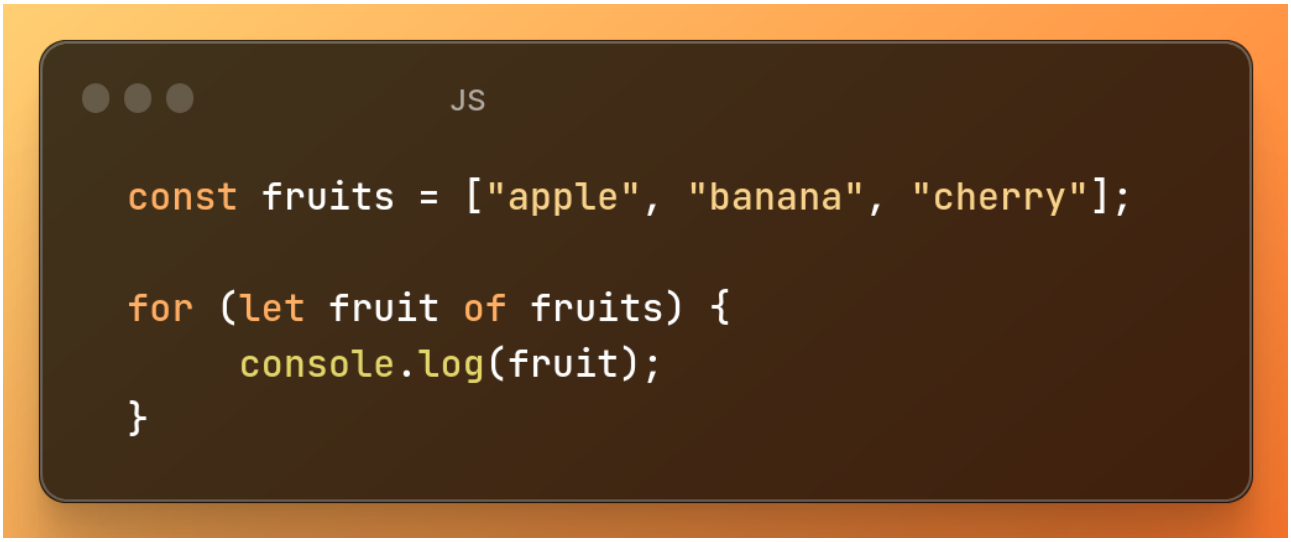


★ for...in Loop

The for...in loop is used to iterate over the properties of an object.

★ for...of Loop (ES6):

The for...of loop is used to iterate over the values of iterable objects like arrays and strings.



★ Nested Loops :

We can nest loops inside each other to handle more complex looping scenarios.

Certainly, let's explore higher-order functions and callbacks without involving the Document Object Model (DOM).

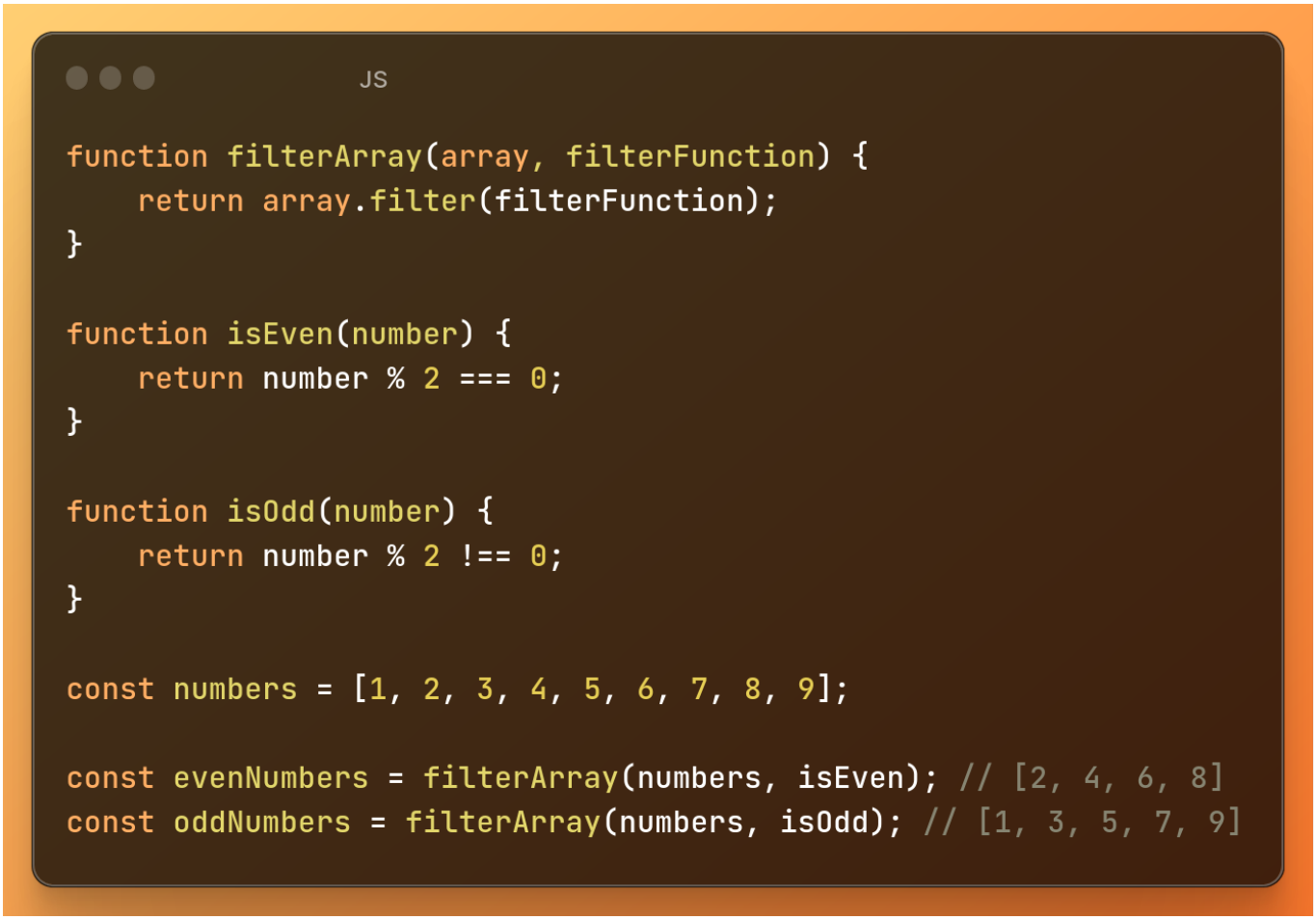
ADVANCED FUNCTIONS

★ Higher-Order Functions

Higher-order functions can be used in various contexts, not just related to web development. A higher-order function is a function that can accept one or more functions as arguments and/or can return a function as its result. Here are the key characteristics of higher-order functions: Higher-order functions can take other functions as

parameters. These functions are often referred to as "callbacks" because they are called back by the higher-order function at a specific point in its execution. Higher-order functions can also create and return new functions. These returned functions can capture and "remember" the context in which they were created, which is useful for creating closures and maintaining state.

In this example, filterArray is a higher-order function that accepts an array and a filter function. It uses the filter method to apply the filter function to the array.



```JS
function filterArray(array, filterFunction) {
    return array.filter(filterFunction);
}

function isEven(number) {
    return number % 2 === 0;
}

function isOdd(number) {
    return number % 2 !== 0;
}

const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

const evenNumbers = filterArray(numbers, isEven); // [2, 4, 6, 8]
const oddNumbers = filterArray(numbers, isOdd); // [1, 3, 5, 7, 9]
```

★ Callbacks

Callbacks can be used in various scenarios where we need to perform asynchronous or delayed operations. Here's an example using a callback with a simulated delay:

KUTTY TASK : 2 [TO DO LIST]

JS

```
// Define an array to store tasks
const tasks = [];

// Function to add a task
function addTask(taskText) {
    const newTask = {
        text: taskText,
        completed: false,
    };
    tasks.push(newTask);
}

// Function to read all tasks
function readTasks() {
    return tasks;
}

// Function to update a task's text
function updateTaskText(index, newText) {
    if (index >= 0 && index < tasks.length) {
        tasks[index].text = newText;
    }
}

// Function to mark a task as completed
function completeTask(index) {
    if (index >= 0 && index < tasks.length) {
        tasks[index].completed = true;
    }
}

// Function to unmark a task as completed
function uncompleteTask(index) {
    if (index >= 0 && index < tasks.length) {
        tasks[index].completed = false;
    }
}

// Function to remove a task
function removeTask(index) {
    if (index >= 0 && index < tasks.length) {
        tasks.splice(index, 1);
    }
}
```

JS

```
// Example usage:

addTask("Buy groceries");
addTask("Walk the dog");
addTask("Write code");

console.log("Initial Tasks:");
console.log(readTasks());

updateTaskText(0, "Buy vegetables");
completeTask(1);

console.log("Updated Tasks:");
console.log(readTasks());

removeTask(2);

console.log("Final Tasks:");
console.log(readTasks());
```

DOM CONCEPTS

The DOM (Document Object Model) is a structured representation of an HTML document's content. It creates a hierarchical tree of objects, where each object represents a part of the document, such as elements, attributes, and text. JavaScript can interact with this tree-like structure to manipulate the content and behavior of a webpage.

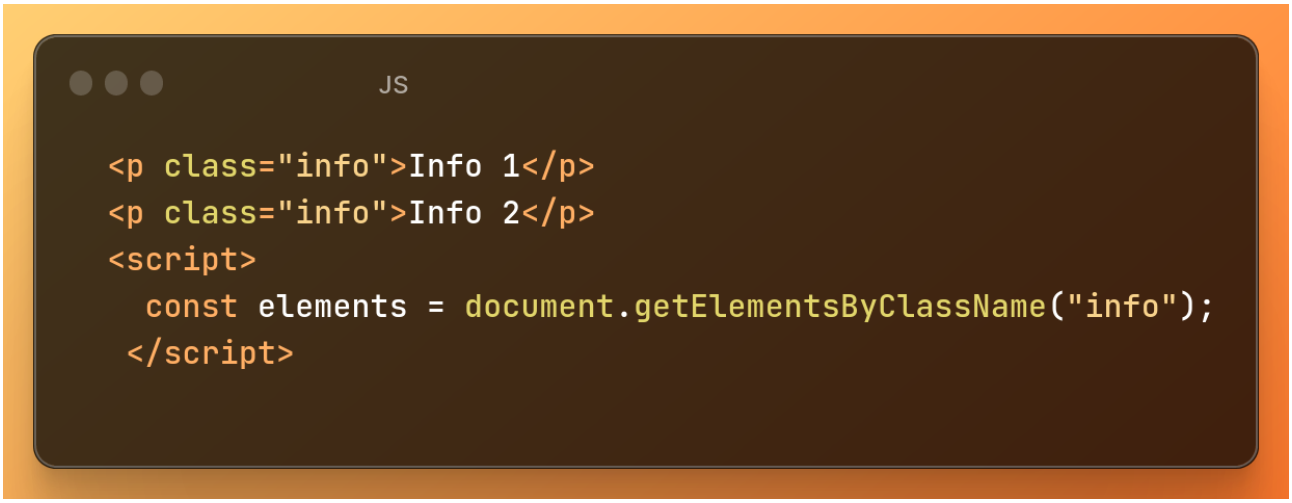
★ Selecting DOM Elements

To interact with elements on a webpage, we first need to select them. Here are some common methods for selecting DOM elements:

document.getElementById(id)

This method selects an element by its unique id attribute.

document.getElementsByClassName(className) This method selects elements by their class name. It returns an HTMLCollection (a live collection of elements).



document.getElementsByTagName(tagName)

This method selects elements by their HTML tag name. It returns an HTMLCollection.

document.querySelector(selector)

This method selects the first element that matches a CSS selector.

```JS
<div class="container">
    <p>Text 1</p>
    <p>Text 2</p>
</div>
<script>
  const element = document.querySelector(".container p");
</script>
```

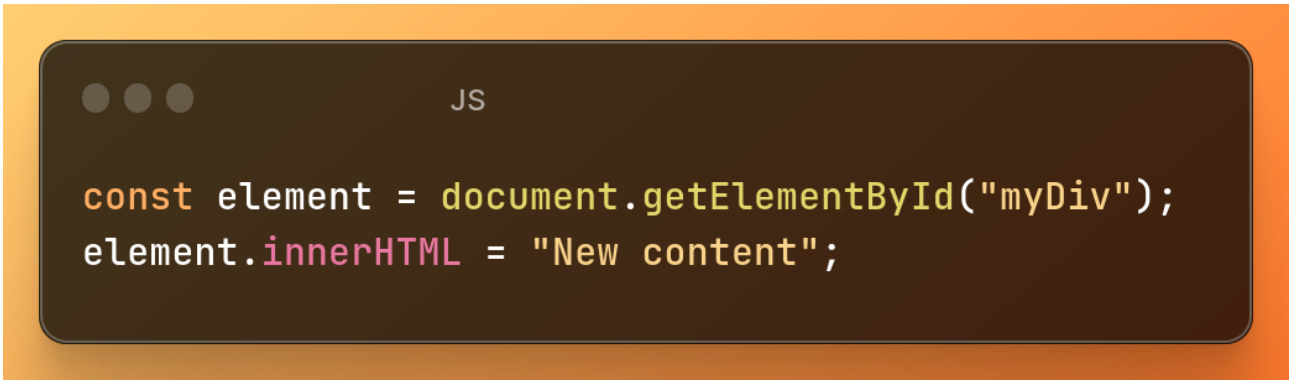
document.querySelectorAll(selector)

This method selects all elements that match a CSS selector. It returns a NodeList.

```JS
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
</ul>
<script>
  const elements = document.querySelectorAll("ul li");
</script>
```

★ Modifying DOM Elements

Once we've selected DOM elements, we can modify their content, attributes, and styling. element.innerHTML : Gets or sets the HTML content within an element.



element.textContent : Gets or sets the text content within an element (ignores HTML tags).

```JS
const element = document.getElementById("myDiv");
element.textContent = "New text content";
```

element.setAttribute(name,val) : Sets an attribute of an element.

```JS
const element = document.getElementById("myLink");
element.setAttribute("href", "https://example.com");
```

element.style.property : Sets a CSS property of an element.

```JS
const element = document.getElementById("myDiv");
element.style.color = "blue";
```

element.classList : Allows adding, removing, and toggling CSS classes on an element.

```JS
const element = document.getElementById("myDiv");
element.classList.add("highlight");
```

★ Event Handling

Events are interactions or occurrences on a webpage, like clicks, keypresses, or mouse movements.

Event handling allows us to specify what should happen when an event occurs. The following example illustrates attaching an event listener to a button element :

We select a button element with the ID "myButton." We add a "click" event listener

using the addEventListener method. The callback function inside addEventListener is executed when the button is clicked, displaying an alert. There are many other events as following :

1. Click Event: click
2. Mouse Over Event: mouseover
3. Mouse Out Event: mouseout
4. Mouse Down Event: mousedown
5. Mouse Up Event: mouseup
6. Key Down Event: keydown
7. Key Up Event: keyup
8. Input Event (for form elements): input
9. Submit Event (for forms): submit
10. Change Event (for form elements): change

11. Focus Event: focus
12. Blur Event: blur
13. Scroll Event: scroll
14. Load Event (for images and documents): load
15. Unload Event (when the page unloads): unload

We can use these event types with the addEventListener method to handle various interactions and behaviors.

KUTTY TASK : 3 [RANDOM QUOTE PAGE]

Create a simple "Quote of the Day" web page where users can click a button to get a new random quote displayed on the page. using DOM

HTML

```
// Array of quotes
const quotes = [
    "The only way to do great work is to love what you do. - Steve Jobs",
    "In the end, we will remember not the words of our enemies, but the silence of our
friends. - Martin Luther King Jr.",
    "Your time is limited, don't waste it living someone else's life. - Steve Jobs",
    "The only thing necessary for the triumph of evil is for good men to do nothing. - Edmund
Burke",
    "Success is not final, failure is not fatal: It is the courage to continue that counts. -
Winston Churchill",
];

// Get DOM elements
const quoteContainer = document.getElementById("quote-container");
const quoteElement = document.getElementById("quote");
const newQuoteButton = document.getElementById("new-quote-button");

// Function to generate a random quote
function getRandomQuote() {
    const randomIndex = Math.floor(Math.random() * quotes.length);
    return quotes[randomIndex];
}

// Function to display a new random quote
function displayNewQuote() {
    const randomQuote = getRandomQuote();
    quoteElement.textContent = randomQuote;
}

// Event listener for the "New Quote" button
newQuoteButton.addEventListener("click", displayNewQuote);

// Display an initial random quote
displayNewQuote();
```

ASYNCHRONOUS JS

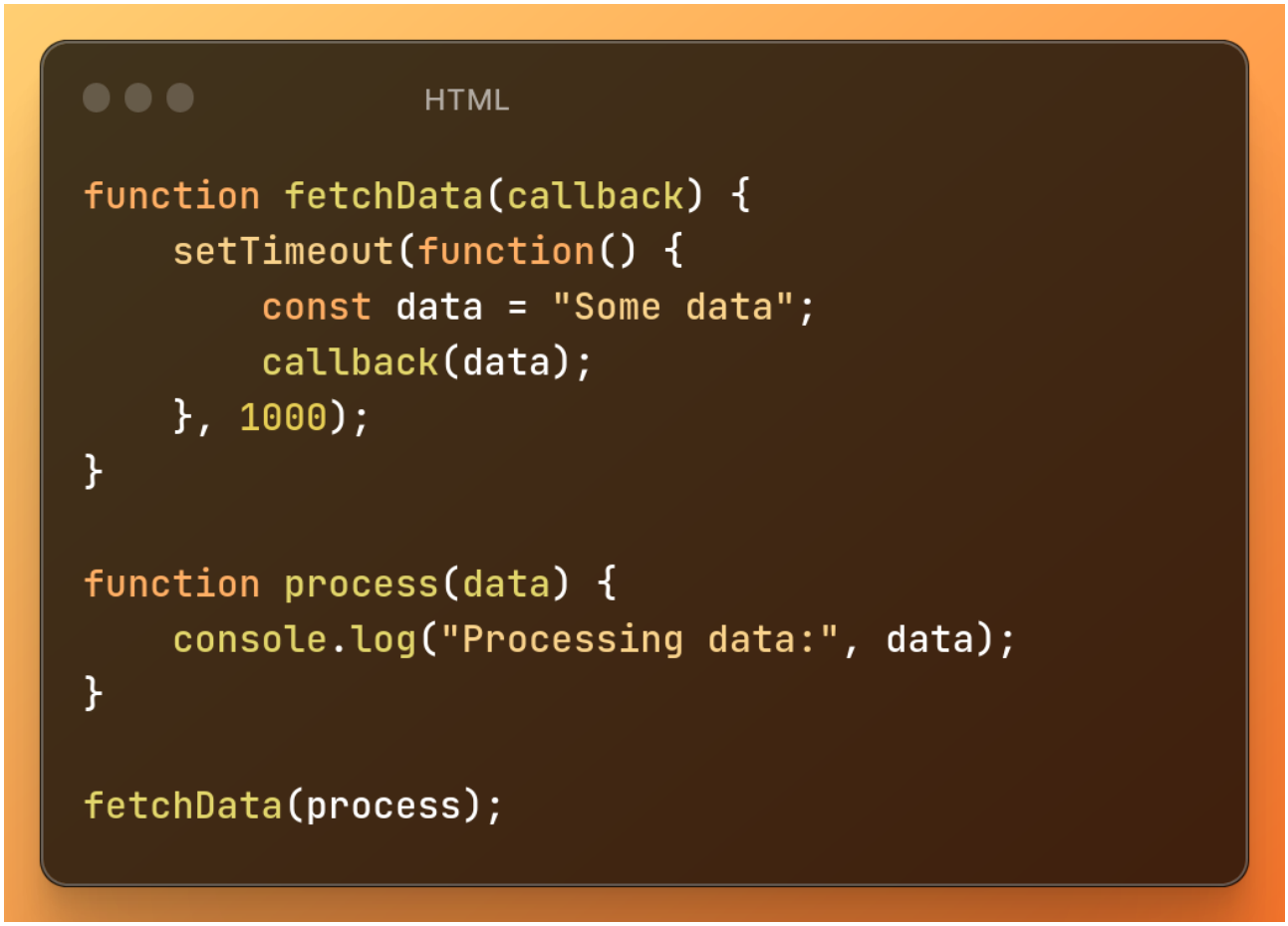
Understanding asynchronous programming in JavaScript is crucial for dealing with tasks that may take time to complete, such as network requests or reading files. Asynchronous programming in JavaScript allows us to execute code without blocking the main thread. It's essential for tasks like making API requests, reading files, or handling user

interactions without freezing the UI. JavaScript uses a non-blocking, event-driven model for handling asynchronous operations.

★ Callbacks & Asynchronous Functions

Callbacks are functions passed as arguments to other functions and are executed once the asynchronous task is complete. Callbacks are a fundamental way to work with asynchronous

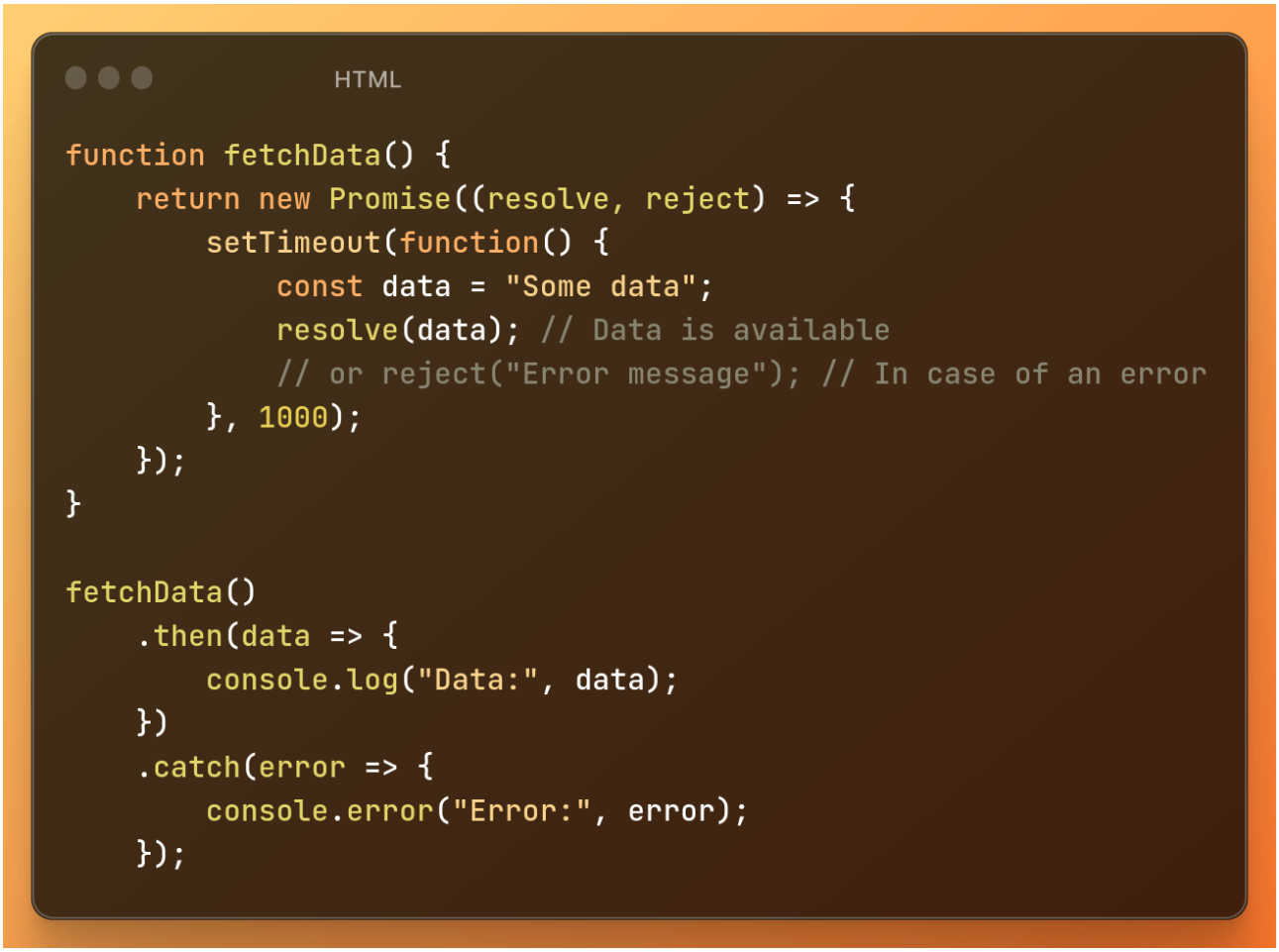
operations in JavaScript. For example:



★ Promises

Promises provide a more structured way to handle asynchronous operations.

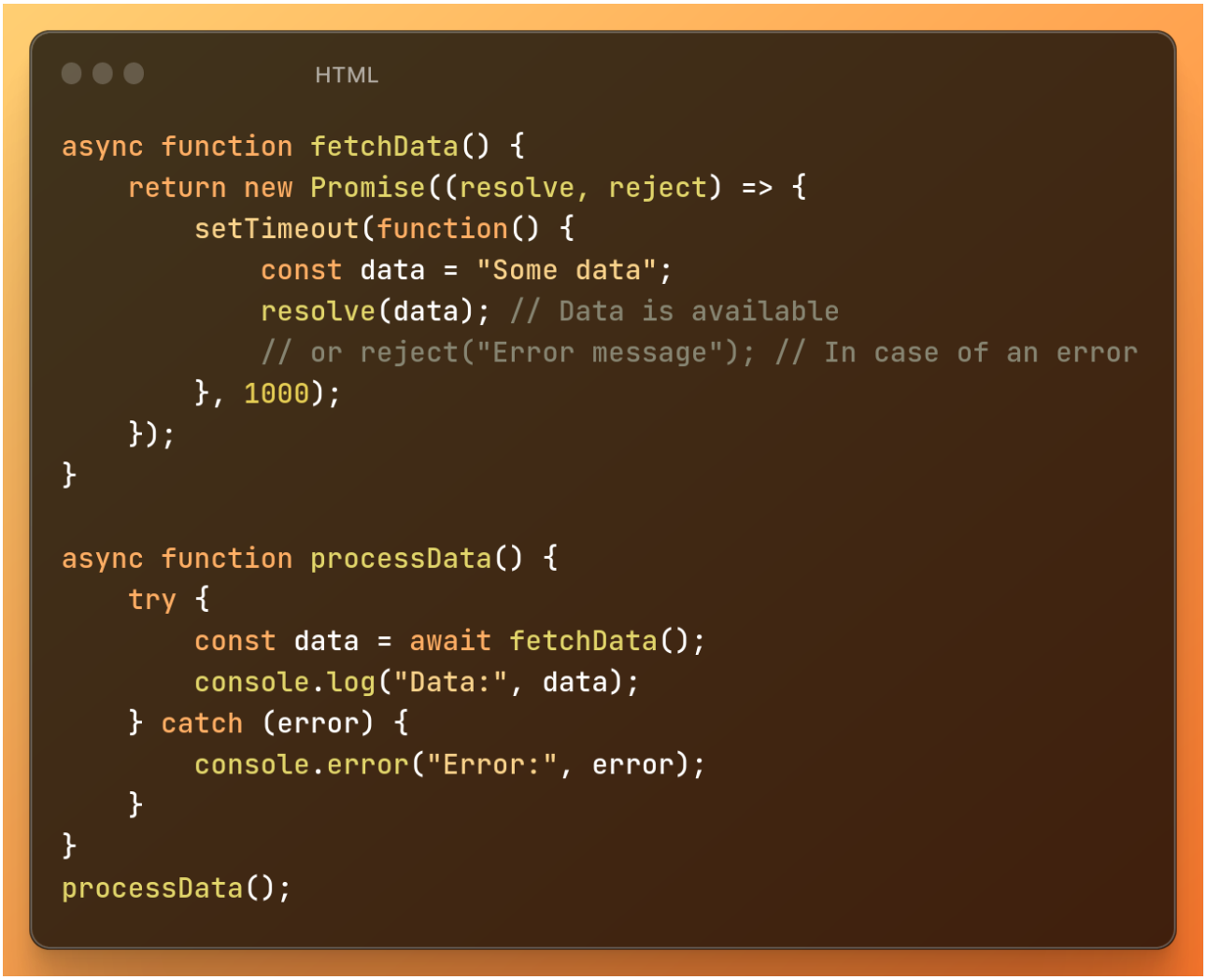
They represent a value that may not be available yet but will be resolved at some point, either successfully with a value or with an error. Promises have three states: pending, resolved (fulfilled), or rejected. They offer cleaner and more maintainable code compared to callbacks.



★ Async/Await

Async/await is a more recent addition to JavaScript and provides a way to write asynchronous code in a synchronous style.

It's built on top of promises and simplifies asynchronous code, making it easier to read and maintain.



KUTTY TASK : 4 [WEATHER APP]

Develop a weather application that uses an API to fetch real-time weather data based on the user's location or city input.

HTML

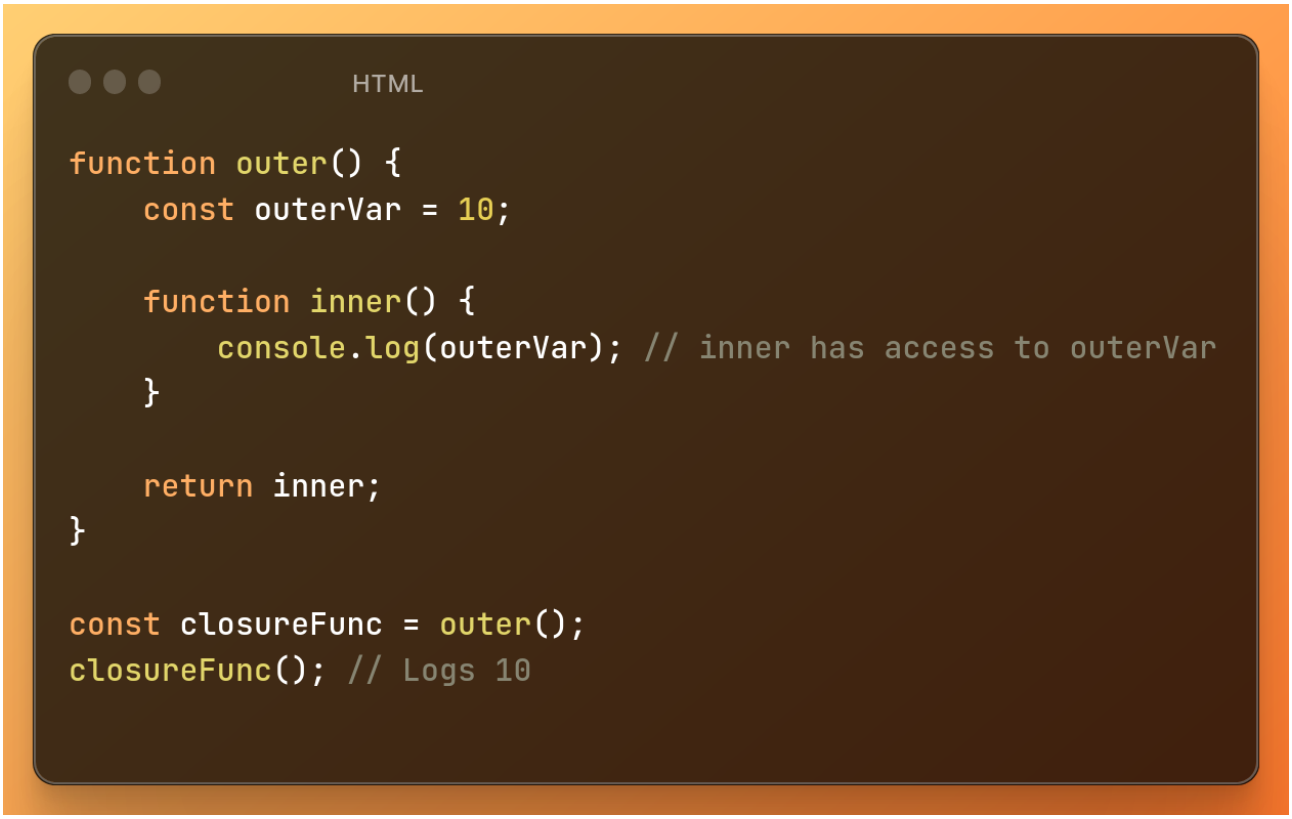
```
// Get DOM elements
const cityInput = document.getElementById("cityInput");
const searchButton = document.getElementById("searchButton");
const weatherInfo = document.getElementById("weatherInfo");

// API Key (Sign up at https://openweathermap.org/api to get your API key)
const apiKey = "YOUR_API_KEY_HERE";

// Function to fetch weather data
async function fetchWeatherData(cityName) {
    try {
        const response = await fetch(`https://api.openweathermap.org/data/2.5
/weather?q=${cityName}&appid=${apiKey}`);
        const data = await response.json();
        return data;
    } catch (error) {
        throw error;
    }
}

// Function to display weather information
function displayWeather(data) {
    const cityName = data.name;
    const temperature = (data.main.temp - 273.15).toFixed(2); // Convert temperature to
Celsius
    const description = data.weather[0].description;

    weatherInfo.innerHTML = `
        <h2>Weather in ${cityName}</h2>
        <p>Temperature: ${temperature}°C</p>
        <p>Description: ${description}</p>
    `;
}

// Event listener for the search button
searchButton.addEventListener("click", async () => {
    const cityName = cityInput.value.trim();
    if (cityName) {
        try {
            const weatherData = await fetchWeatherData(cityName);
            displayWeather(weatherData);
        } catch (error) {
            weatherInfo.innerHTML = "<p>Error fetching weather data. Please try again.</p>";
        }
    }
});
```

ADVANCED JS

These are additional important topics in JavaScript and web development.

★Closures and Lexical Scoping

Closures occur when a function is defined inside another function and has access to the outer function's variables. This allows data encapsulation and private variables in JavaScript.



Lexical Scoping refers to how variable scope is determined in nested functions. Variables are resolved based on their location in the source code, not where they are executed.

★ ES6+ Features

Arrow Functions: Arrow functions provide a concise syntax for writing functions. They capture the “this” value from the surrounding context.

```
const add = (a, b) => a + b;
```

Classes: Classes provide a more structured way to create objects with constructors and methods.

Modules: ES6 introduced a module system that allows you to split your code into separate files and import/export functionality between them.

HTML

```
export function add(a, b) {
    return a + b;
}
```

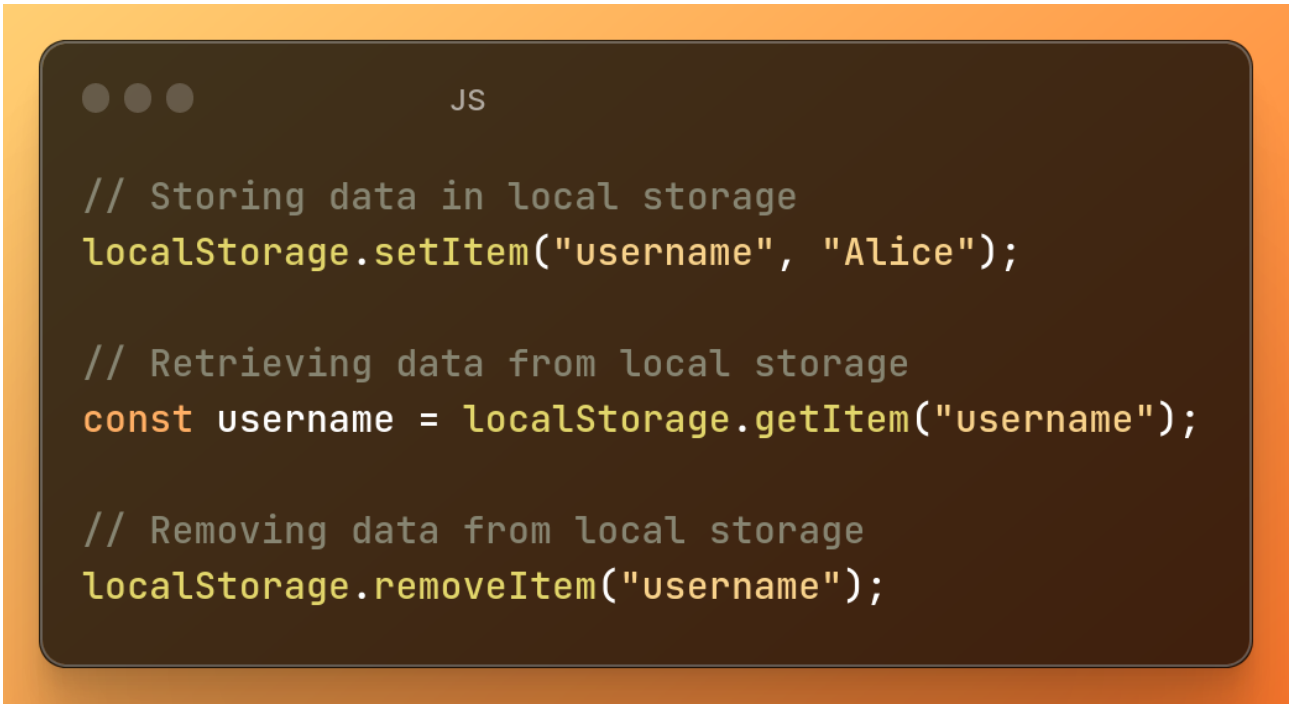
main.js

```
// In main.js
import { add } from "./math.js";
```

Error Handling: Error handling is crucial for handling exceptions gracefully in your code. The try...catch statement allows us to catch and handle exceptions without crashing our program.

```js
try {
    // Code that may throw an error
    throw new Error("Something went wrong");
} catch (error) {
    // Handle the error
    console.error(error.message);
} finally {
    // Optional: Code that always runs, whether there's an error or not
}
```

Local Storage and Client-Side Storage: Local Storage is a client-side storage mechanism in web browsers that allows you to store key-value pairs persistently. It's useful for caching data, saving user preferences, or implementing offline functionality.



Local Storage is simple to use and provides a way to store data on the client side, but it has limitations in terms of storage size and security. For more complex data management, we might consider other client-side storage options like IndexedDB.

KUTTY TASK : 5 [SIMPLE CHAT APPLICATION]

Build a simple chat application that allows users to send and receive messages in real-time using WebSocket.

HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>WebSocket Chat</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h1>WebSocket Chat</h1>
        <div id="chat-box">
            <div id="chat-messages"></div>
        </div>
        <input type="text" id="message-input" placeholder="Type your message...">
        <button id="send-button">Send</button>
    </div>
    <script src="script.js"></script>
</body>
</html>
```

```css
body {
    font-family: Arial, sans-serif;
}

.container {
    max-width: 400px;
    margin: 0 auto;
    padding: 20px;
    border: 1px solid #ccc;
    border-radius: 5px;
    box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);
}

h1 {
    text-align: center;
    margin-bottom: 20px;
}

#chat-box {
    border: 1px solid #ccc;
    border-radius: 5px;
    padding: 10px;
    max-height: 300px;
    overflow-y: auto;
}

.message {
    margin: 5px 0;
}

.timestamp {
    color: #777;
    margin-right: 5px;
}
```

```js
// Get DOM elements
const chatMessages = document.getElementById("chat-messages");
const messageInput = document.getElementById("message-input");
const sendButton = document.getElementById("send-button");

// Create a WebSocket connection (replace 'your_server_url' with the WebSocket server URL)
const socket = new WebSocket("ws://your_server_url");

// Event listener for WebSocket open connection
socket.addEventListener("open", () => {
    // Enable the input and send button
    messageInput.disabled = false;
    sendButton.disabled = false;
});

// Event listener for WebSocket incoming messages
socket.addEventListener("message", (event) => {
    const message = JSON.parse(event.data);
    displayMessage(message);
});

// Event listener for send button click
sendButton.addEventListener("click", () => {
    const messageText = messageInput.value.trim();
    if (messageText) {
        sendMessage(messageText);
        messageInput.value = "";
    }
});

// Function to send a message via WebSocket
function sendMessage(text) {
    const message = {
        text: text,
        timestamp: new Date().toLocaleTimeString(),
    };
    socket.send(JSON.stringify(message));
    displayMessage(message);
}

// Function to display a message in the chat box
function displayMessage(message) {
    const messageElement = document.createElement("div");
    messageElement.className = "message";
    messageElement.innerHTML = `<span class="timestamp">${message.timestamp}</span> -
${message.text}`;
    chatMessages.appendChild(messageElement);

    // Scroll chat box to the bottom
    chatMessages.scrollTop = chatMessages.scrollHeight;
}
```

MERCI