JAVASCRIPT

JavaScript is a high-level, interpreted programming language that is widely used for web development. It allows developers to create dynamic and interactive content on websites. JavaScript is an essential part of the web technology stack, along with HTML and CSS.

Introduction

JavaScript is popular because it:

- Runs in the browser, allowing for interactive web pages.
- Has a vast ecosystem with numerous libraries and frameworks.
- Is versatile and can be used for both client-side and server-side development (e.g., Node.js).
- Has a large and active community, providing extensive support and resources.

JavaScript Where To?

In HTML, JavaScript code is inserted between <script> and </script> tags.

Example:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript in Body</h2>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
</body>
</html>
```

JavaScript output

JavaScript can "display" data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

Using innerHTML

To access an HTML element, JavaScript can use the document.getElementById(id) method.

The id attribute defines the HTML element. The innerHTML property defines the HTML content:

```
<html>
<body>
<h2>My First Web Page</h2>
<p>My First Paragraph.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

Using document.write()

The document.write() method in JavaScript writes directly to the HTML output stream.

Example:

```
<!DOCTYPE html>

<html>

<body>

<script>

document.write("hello");

</script>

</body>

</html>
```

Using window.alert()

The window.alert() method in JavaScript is used to display a modal dialog box with a specified message and an "OK" button. It is typically used to give feedback to users or display a simple message. The dialog box will block user interaction with the rest of the page until the user clicks "OK."

Example:

```
<!DOCTYPE html>

<html>

<body>

<h2>My First Web Page</h2>

<p>My first paragraph.</p>

<script>

window.alert("hello");

</script>

</body>

</html>
```

Using Console.log()

The console.log() method in JavaScript is used to output messages to the browser's **console**. It is primarily used for debugging purposes by displaying values, expressions, or messages to help developers understand the state of their code.

Example:

```
<!DOCTYPE html>
<html>
<body>
<p>Then select "Console" in the debugger menu.</p>
<script>
console.log("hello");
</script>
</body>
</html>
```

JavaScript Working

1.JavaScript Engine

The JavaScript engine is the core component that interprets and executes JavaScript code. Some well-known JavaScript engines include:

V8: Used in Google Chrome and Node.js.

SpiderMonkey: Used in Mozilla Firefox.

Chakra: Used in Microsoft Edge.

JavaScriptCore: Used in Safari.

2.Loading and Parsing

When a web page containing JavaScript is loaded:

HTML Parsing: The browser starts by parsing the HTML document to construct the DOM (Document Object Model).

Script Loading: When the browser encounters a <script> tag, it loads the JavaScript file.

Parsing: The JavaScript engine parses the loaded JavaScript code to convert it into an abstract syntax tree (AST).

3.Compilation and Execution

JavaScript engines use a Just-In-Time (JIT) compiler to optimize performance:

Parsing: The JavaScript engine parses the code into an abstract syntax tree (AST).

Bytecode Generation: The AST is converted into bytecode, an intermediate representation of the code.

Execution: The bytecode is executed by the interpreter.

Optimization: Frequently executed code (hot code) is optimized further by compiling it into machine code.

4.Execution Context and Call Stack

JavaScript executes code within the context of an execution context, which can be global or function-specific. The call stack manages function execution:

Global Execution Context: Created when the script starts. It includes global variables and functions.

Function Execution Context: Created each time a function is called. It includes the function's local variables, arguments, and scope.

5.Event Loop and Asynchronous Programming

JavaScript is single-threaded but can handle asynchronous operations using the event loop:

Call Stack: Manages the execution of function calls.

Web APIs: Handles asynchronous operations like network requests, timers, and DOM events.

Task Queue: Holds callback functions from asynchronous operations.

Event Loop: Continuously checks if the call stack is empty. If it is, it moves the first task from the task queue to the call stack for execution.

Example: Execution of a Simple Script

```
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Example</title>
  <script>
    console.log('Start');
    setTimeout(() => {
      console.log('Timeout');
    }, 1000);
    console.log('End');
  </script>
</head>
<body>
</body>
</html>
```

Execution Steps:

- HTML Parsing: The browser parses the HTML and encounters the <script> tag.
- Script Loading: The JavaScript file is loaded and parsed by the JavaScript engine.
- Global Execution Context: Created and pushed onto the call stack.
- Console.log('Start'): Executes and logs 'Start'.
- setTimeout: Schedules the callback function and moves it to the Web APIs, then continues execution.
- Console.log('End'): Executes and logs 'End'.
- Event Loop: After 1000 ms, moves the timeout callback to the task queue.
- Call Stack: When empty, the event loop moves the callback from the task queue to the call stack.
- Timeout Callback: Executes and logs 'Timeout'.

JavaScript Statements

A **computer program** is a list of "instructions" to be "executed" by a computer. In a programming language, these programming instructions are called **statements**. A **JavaScript program** is a list of programming **statements**.

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

```
<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Statements</h2>

<p>In HTML, JavaScript statements are executed by the browser.</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = "Hello Dolly.";

</script>

</body>

</html>
```

Data Types in JavaScript

In JavaScript, data types are divided into two main categories: primitive types and non-primitive types (also known as reference types).

1. Primitive Data Types

Primitive data types are immutable and hold single values (not objects). JavaScript has the following primitive data types:

a) Number

Represents numeric values, both integers and floating-point numbers.

```
let integerNum = 10;     // Integer

let floatNum = 3.14;     // Floating-point
```

let negativeNum = -50; // Negative integer

b) String

Represents sequences of characters (text).

Strings can be enclosed in single ('), double ("), or backticks (`) for template literals (which allow embedded expressions and multiline strings).

let greeting = "Hello, World!";

let char = 'A';

c) Boolean

Represents a logical value: either true or false.

let isJavaScriptFun = true;

let isSkyBlue = false;

d) Undefined

A variable that has been declared but has not yet been assigned a value.

let a;

console.log(a); // undefined

e) Null

Represents an explicitly empty or non-existent value.

let emptyValue = null;

f) BigInt (introduced in ES11)

Represents integers larger than Number.MAX_SAFE_INTEGER (which is 253−12^{53} - 1253−1).

let bigIntValue = 1234567890123456789012345678901234567890n;

console.log(bigIntValue + 10n); // BigInt arithmetic

2)Non- primitive data ypes

Non-primitive data types, also known as reference types, are objects and derived data types. They can store collections of values or more complex entities. The two key non-primitive data types in JavaScript are:

a) Object

The most important reference type in JavaScript, representing collections of key-value pairs.

```
let person = {
   name: "Alice",
   age: 25,
   isStudent: false
};
console.log(person.name); // Accessing property
```

b) Array

Arrays are objects used to store ordered collections of values (of any type).

```
let numbers = [1, 2, 3, 4, 5];
console.log(numbers[0]); // Access first element: 1
```

c) Function

Functions are also objects in JavaScript. They can be passed as arguments to other functions, and can return values.

```
function greet() {
   return "Hello!";
}
console.log(greet()); // Calls the function
```

d) Date

The Date object is used to work with dates and times.

```
let currentDate = new Date();
console.log(currentDate); // Current date and time
```

e) RegExp

Represents regular expressions and is used for pattern matching in strings.

```
let pattern = /hello/i;
console.log(pattern.test("Hello, World!")); // true
```

Variables in JavaScript

In JavaScript, variables are used to store data values. You can declare variables using the keywords var, let, or const.

Declaring Variables

Variables are declared using var, let, or const.

1)Var

- The **var** is the oldest keyword to declare a variable in JavaScript. It has the Global scoped or function scoped. This means:
- If you create a variable outside of a function, you can use it anywhere in your code.
- If you create a variable inside a function, you can only use it within that function.

```
var a = 10
function f() {
    var b = 20
    console.log(a, b)
}
f();
console.log(a);
```

2) Let

The let keyword is an improved version of the var keyword. These variables has the block scope. It can't be accessible outside the particular code block.

```
let a = 10;
function f() {
    let b = 9
    console.log(b);
    console.log(a);
}
f();
```

3)Const

The const keyword has all the properties that are the same as the let keyword, except the user cannot update it and have to assign it with a value at the time of declaration. These variables also have the block scope. It is mainly used to create constant variables whose values can not be changed once they are initialized with a value.

```
const a = 10;
function f() {
    a = 9
    console.log(a)
}
f();
```

Temporal Dead Zone (TDZ)

The Temporal Dead Zone refers to the time between the start of a block scope (where let and const variables are hoisted) and the actual declaration of the variable. During this period, accessing the variable results in a ReferenceError.

Example:

function example() {

  console.log(a); // ReferenceError: Cannot access 'a' before initialization

  let a = 10;

}

example();

In this example, a is in the TDZ from the start of the block until its declaration is encountered. Accessing a during this period results in an error.

JavaScript Operators

Javascript operators are used to perform different types of mathematical and logical computations.

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- String Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators

- Type Operators

1. JavaScript Arithmetic Operators

Arithmetic Operators are used to perform arithmetic on numbers:

Arithmetic Operator	Name	Example
+	Addition	a + b
-	Subtraction	a – b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b
++	Increment Operator	a++
--	Decrement Operator	a--

2.Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Explanation
=	a=b	a=b
+=	a+=b	a= a+b
-=	a-=b	a = a-b
*=	a*=b	a =a*b
/=	a/=b	a = a/b
%=	a%=b	a =a%b

3.Comparison operators

Operator	Name	Example	Result
>	Greater than	x > 10	false
>=	Greater than or equal to	x >= 5	true
<	Less than	x < -50	false
<=	Less than or equal to	x <= 100	true
==	Equal to	x == "5"	true
!=	Not equal to	x != "b"	true
===	Equal value and type	x === "5"	false
!==	Not equal value or equal type	x !== "5"	true

4.String Operators

All the comparison operators above can also be used on strings.

5. Logical Operators

Logical Operators	Name	Example
&&	Logical AND operator	a && b is true if a and b are both true
||	Logical OR Operator	a || b is true if either a or b is true
!	Logical NOT Operator	!a is true if a is not true

6.Bitwise Operators

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Meaning
<<	Shifts the bits to left
>>	Shifts the bits to right
~	Bitwise inversion (one's complement)
&	Bitwise logical AND
|	Bitwise logical OR
^	Bitwise exclusive or

7. Ternary Operators

In JavaScript, the **ternary operator** (also known as the **conditional operator**) provides a shorthand way to write simple if-else statements. It has the form:

condition ? expressionIfTrue : expressionIfFalse;

condition: This is the condition that will be evaluated. If it's true, the first expression (expressionIfTrue) is executed.

expressionIfTrue: This is the code that runs if the condition is true.

expressionIfFalse: This is the code that runs if the condition is false.

8.Type Operators

In JavaScript, type operators allow you to determine or manipulate the types of values. Here's an overview of the key type-related operators:

a) typeof Operator

The typeof operator is used to find the **type of a variable or expression**. It returns a string indicating the type.

```
console.log(typeof 42);            // Output: "number"
console.log(typeof 'Hello');       // Output: "string"
console.log(typeof true);          // Output: "boolean"
console.log(typeof undefined);     // Output: "undefined"
console.log(typeof {name: 'John'}); // Output: "object"
console.log(typeof function(){});  // Output: "function"
```

b) instanceof Operator

The instanceof operator checks whether an object is an **instance of a particular constructor or class**. It returns a boolean (true or false).

```
let arr = [1, 2, 3]; console.log(arr instanceof Array);            // Output: true
```

Post-Increment and Pre-Increment

Post-Increment (x++)

The post-increment operator increments the value of the variable after returning its current value.

Syntax:

```
let x = 5;
let y = x++; // y is assigned the value of x (5), then x is incremented to 6.
console.log(x); // 6
console.log(y); // 5
```

Pre-Increment (++x)

The pre-increment operator increments the value of the variable before returning its new value.

Syntax:

let x = 5;

let y = ++x; // x is incremented to 6, then y is assigned the value of x (6).

console.log(x); // 6

console.log(y); // 6

JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: **{}**

function *name*(*parameter1, parameter2, parameter3*) {
  // *code to be executed*
}

Function **parameters** are listed inside the parentheses () in the function definition.

Function **arguments** are the **values** received by the function when it is invoked.

Inside the function, the arguments (the parameters) behave as local variables

Function Invocation

The code inside the function will execute when "something" invokes (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Function return

When JavaScript reaches a return statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

```
let x = myFunction(4, 3);

document.getElementById("demo").innerHTML = x;

function myFunction(a, b) {

  return a * b;

}
```

Types of functions

1. Named Functions

These are functions that are defined with a name, making them reusable and easy to reference within the code.

```
function functionName(parameters) {

   // Function body

}
```

2. Anonymous Functions

These functions are declared without a name and are usually assigned to a variable or passed as an argument to another function (such as callbacks).

```
let myFunction = function(parameters) {

   // Function body

};
```

3. Arrow Functions (ES6)

Introduced in ES6, arrow functions offer a shorter syntax for writing functions and have a different way of handling this context.

```
let multiply = (a, b) => a * b;

console.log(multiply(4, 5)); // Outputs: 20
```

4. IIFE (Immediately Invoked Function Expression)

An IIFE is a function that is executed immediately after it is defined. It is commonly used to create a private scope and avoid polluting the global namespace.

```
(function() {
   let message = "IIFE executed!";
   console.log(message);
})();  // Outputs: IIFE executed!
```

Executes immediately.

Useful for creating private variables and avoiding global scope pollution.

5. Constructor Functions

These functions are used to create objects and are typically used with the new keyword. They serve as templates for creating multiple instances of similar objects.

```
function Person(name, age) {
   this.name = name;
   this.age = age;
}
let person1 = new Person('arun', 25);
console.log(person1.name); // Outputs: arun
```

6. Generator Functions (ES6)

A generator function is a special type of function that can pause its execution and later resume. It is defined with an asterisk (*) and uses the yield keyword to yield values.

```
function* countNumbers() {
   let num = 1;
   while (true) {
      yield num++;
   }
}
```

```
let counter = countNumbers();

console.log(counter.next().value); // Outputs: 1

console.log(counter.next().value); // Outputs: 2
```

7. Callback Functions

A callback function is a function that is passed as an argument to another function and is executed after some operation is completed

```
function greet(name) {
   console.log(`Hello, ${name}`);
}

function processUserInput(callback) {
   let name = 'adil';
   callback(name);
}

processUserInput(greet);  // Outputs: Hello, adil
```

8. Higher-Order Functions

A higher-order function is a function that takes another function as an argument or returns a function as its result.

```
function greet() {
   console.log('Hello');
}

function execute(fn) {
   fn();
}

execute(greet);  // Outputs: Hello
```

9. Recursive Functions

A recursive function is a function that calls itself to solve a problem. It is useful for tasks that can be broken down into smaller, repetitive tasks.

```
function factorial(n) {
   if (n === 0) {
      return 1;
   }
   return n * factorial(n - 1);
}
console.log(factorial(5)); // Outputs: 120
```

JavaScript Scope

JavaScript Scope is the area where a variable (or function) exists and is accessible.

1.Global Scope

Those variables which are declared outside the function or blocks or you can say curly braces({}) are having a global scope.

In a JavaScript program, global variables can be accessed from anywhere.

```
let globalVar = 'global'; // Global variable


function showGlobalVar() {
  console.log(globalVar); // Accessible
}
```

2.Function Scope

Those variables that are declared inside a function have local or function scope which means variables that are declared inside the function are not accessible outside that function.

```
function localScope() {
  let localVar = 'local'; // Local variable
  console.log(localVar);  // Accessible inside the function
}


console.log(localVar);  // ReferenceError: localVar is not defined
```

3.Block Scope

Variables declared with let and const inside a block (e.g., within {}) are block-scoped and not accessible outside the block.

```
if (true) {
  let blockVar = 'block scoped';
  console.log(blockVar);  // Accessible inside the block
}
console.log(blockVar);  // ReferenceError: blockVar is not defined
```

4.Lexical Scope

The variable is declared inside the function and can only be accessed inside that block or nested block is called lexical scope.

```
let outerVar = 'outer';

function outerFunc() {
  let innerVar = 'inner';

  function innerFunc() {
    console.log(outerVar);  // Accessible because of lexical scoping
  }
  innerFunc();
}
outerFunc();
```

Scope chaining

Scope chaining in JavaScript refers to the process by which the JavaScript engine determines where to find variables when they are referenced in your code. It's closely related to the concept of **lexical scoping**, where nested functions have access to variables declared in their outer scope.

```
function outerFunction() {
   let outerVar = "I am outer";
   function innerFunction() {
      let innerVar = "I am inner";
      console.log(innerVar);  // Accessible: local to innerFunction
      console.log(outerVar);  // Accessible: found in outerFunction scope
      console.log(globalVar); // Accessible: found in global scope
   }
   innerFunction();
}
outerFunction();
```

JavaScript Arrays

JavaScript arrays are objects used to store multiple values in a single variable. JavaScript provides a variety of built-in methods to perform operations on arrays.

Array Methods in JavaScript

1. push()

- Adds one or more elements to the end of an array.
- Returns the new length of the array.

```
let arr = [1, 2, 3];
arr.push(4);  // arr becomes [1, 2, 3, 4]
```

2. pop()

- Removes the last element from an array.
- Returns the removed element.

```
let arr = [1, 2, 3];
arr.pop();  // arr becomes [1, 2], returns 3
```

3. shift()

- Removes the first element from an array.

- Returns the removed element.

```
let arr = [1, 2, 3];

arr.shift();  // arr becomes [2, 3], returns 1
```

4. unshift()

- Adds one or more elements to the beginning of an array.
- Returns the new length of the array.

```
let arr = [2, 3];

arr.unshift(1);  // arr becomes [1, 2, 3]
```

5. concat()

- Merges two or more arrays.
- Returns a new array.

```
let arr1 = [1, 2];

let arr2 = [3, 4];

let newArr = arr1.concat(arr2);  // newArr becomes [1, 2, 3, 4]
```

6. slice()

- Extracts a section of an array and returns it as a new array.
- Does not modify the original array.

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

let slicedArr = arr.slice(1, 3);  // slicedArr becomes [2, 3]
```

7. splice()

- Adds/removes elements from an array.
- Modifies the original array and returns the removed elements (if any).

```
let arr = [1, 2, 3, 4];

arr.splice(1, 2);  // arr becomes [1, 4], removes [2, 3]
```

8. indexOf()

- Returns the first index at which a specified element is found in the array.

- Returns -1 if the element is not found

```
let arr = [1, 2, 3];

arr.indexOf(2);  // returns 1
```

9. includes()

- Determines whether an array contains a specified element.
- Returns true or false.

```
let arr = [1, 2, 3];

arr.includes(2);  // returns true
```

10. reverse()

- Reverses the order of the elements in an array.
- Modifies the original array.

```
let arr = [1, 2, 3];

arr.reverse();  // arr becomes [3, 2, 1]
```

11. sort()

- Sorts the elements of an array.
- Modifies the original array.

```
let arr = [3, 1, 2];

arr.sort();  // arr becomes [1, 2, 3]
```

12. forEach()

- Executes a provided function once for each array element.

```
let arr = [1, 2, 3];

arr.forEach((element) => console.log(element));  // Logs 1, 2, 3
```

13. map()

- Creates a new array populated with the results of calling a provided function on every element in the original array

```
let arr = [1, 2, 3];

let doubledArr = arr.map(x => x * 2);  // doubledArr becomes [2, 4, 6]
```

14. filter()

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

```
let arr = [1, 2, 3, 4];

let evenArr = arr.filter(x => x % 2 === 0);  // evenArr becomes [2, 4]
```

15. reduce()

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

```
let arr = [1, 2, 3, 4];

let sum = arr.reduce((accumulator, currentValue) => accumulator + currentValue, 0);  // sum becomes 10
```

16. find()

- Returns the value of the first element that satisfies the provided testing function.

```
let arr = [1, 2, 3];

let found = arr.find(x => x > 2);  // found becomes 3
```

17. findIndex()

- Returns the index of the first element that satisfies the provided testing function.

```
let arr = [1, 2, 3];

let index = arr.findIndex(x => x > 2);  // index becomes 2
```

JavaScript Strings

Strings in JavaScript are used for storing and manipulating text. JavaScript provides several built-in methods for working with strings.

String Methods in JavaScript

1. length

- Returns the length of a string (number of characters).

```
let str = "Hello";

console.log(str.length);  // Output: 5
```

2. toUpperCase()

- Converts a string to uppercase letters.

```
let str = "hello";

console.log(str.toUpperCase());  // Output: "HELLO"
```

3. toLowerCase()

- Converts a string to lowercase letters.

```
let str = "HELLO";

console.log(str.toLowerCase());  // Output: "hello"
```

4. charAt()

- Returns the character at a specified index in a string.

```
let str = "Hello";

console.log(str.charAt(1));  // Output: "e"
```

5. indexOf()

- Returns the index of the first occurrence of a specified value in a string.
- Returns -1 if the value is not found.

```
let str = "Hello world";

console.log(str.indexOf("world"));  // Output: 6
```

6. lastIndexOf()

- Returns the index of the last occurrence of a specified value in a string.

```
let str = "Hello world, world";

console.log(str.lastIndexOf("world"));  // Output: 13
```

7. slice()

- Extracts a part of a string and returns a new string.

```
let str = "Hello world";

let slicedStr = str.slice(0, 5);  // Output: "Hello"
```

8. substring()

- Similar to slice(), but cannot accept negative indices.

```
let str = "Hello world";
```

```
let subStr = str.substring(0, 5);  // Output: "Hello"
```

9. substr()

- Similar to slice(), but the second parameter specifies the length of the extracted part.

```
let str = "Hello world";

let subStr = str.substr(6, 5);  // Output: "world"
```

10. replace()

- Replaces a specified value with another value in a string.

```
let str = "Hello world";

let newStr = str.replace("world", "everyone");  // Output: "Hello everyone"
```

11. split()

- Splits a string into an array of substrings based on a specified delimiter.

```
let str = "Hello world";

let arr = str.split(" ");  // Output: ["Hello", "world"]
```

12. trim()

- Removes whitespace from both sides of a string.

```
let str = "   Hello world   ";

console.log(str.trim());  // Output: "Hello world"
```

13. includes()

- Checks if a string contains a specified value.
- Returns true or false.

```
let str = "Hello world";

console.log(str.includes("world"));  // Output: true
```

14. startsWith()

- Checks if a string starts with a specified value.

```
let str = "Hello world";

console.log(str.startsWith("Hello"));  // Output: true
```

15. endsWith()

- Checks if a string ends with a specified value.

```
let str = "Hello world";

console.log(str.endsWith("world"));  // Output: true
```

16. concat()

- Joins two or more strings

```
let str1 = "Hello";

let str2 = "world";

console.log(str1.concat(" ", str2));  // Output: "Hello world"
```

17. repeat()

- The **repeat()** method returns a new string with a specified number of copies of the original string concatenated together.

```
let text = "hello";

let repeatedText = text.repeat(3);

console.log(repeatedText);  // Output: "hellohellohello"
```

Execution Context

An **execution context** is an environment in which JavaScript code is executed. It determines which variables, functions, and objects are accessible and how the code is interpreted.

Types of Execution Context:

1. **Global Execution Context**:
    - Created when the script starts running.
    - Manages global variables and functions.
    - There is only one global execution context in a JavaScript program.
2. **Function Execution Context**:
    - Created whenever a function is invoked.
    - Each function call creates its own execution context with local variables and parameters.
3. **Eval Execution Context**:

- Created when JavaScript's eval() function is executed (rarely used).

Execution Context Phases:

1. **Creation Phase**:
    - Variables, functions, and the this keyword are set up.
    - Variables are initialized as undefined, and functions are stored in memory.
2. **Execution Phase**:
    - The code inside the execution context is run, and variables are assigned their actual values.

Hoisting

Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during the creation phase of the execution context, before the code is executed.

Function Declarations:

- Entire function declarations are hoisted, meaning you can call a function before it's defined in the code.

```
greet(); // Output: "Hello!"

function greet() {
  console.log("Hello!");
}
```

Variable Declarations:

- Variables declared using var are hoisted, but they are initialized with undefined until their actual assignment.
- Variables declared with let and const are also hoisted, but they are not initialized and cannot be accessed before their declaration (Temporal Dead Zone).

```
console.log(a);  // Output: undefined

var a = 5;
```

Call stack

The **call stack** is a data structure that keeps track of the execution of function calls. It follows a **LIFO** (Last In, First Out) order, meaning the last function called is the first to be completed and removed from the stack.

How the Call Stack Works:

1. When a function is called, its execution context is placed on top of the call stack.
2. If a function calls another function, the new function's execution context is added to the stack.
3. When a function completes, its execution context is removed from the call stack.
4. JavaScript continues this process until the call stack is empty.

```
function first() {
  console.log("First");
  second();
}
function second() {
  console.log("Second");
  third();
}
function third() {
  console.log("Third");
}
first();
```

Call Stack Process:

- **Step 1**: first() is called. Its execution context is added to the stack.
- **Step 2**: Inside first(), second() is called. The execution context of second() is pushed onto the stack.
- **Step 3**: Inside second(), third() is called. The execution context of third() is added to the stack.
- **Step 4**: third() completes and is removed from the stack.

- **Step 5**: second() completes and is removed from the stack.
- **Step 6**: first() completes and is removed from the stack.

Currying

Currying is a technique used in functional programming that allows you to transform a function with multiple arguments into a sequence of functions, each taking only one argument at a time.

Consider a function called add that takes two arguments and returns their sum:

```
function add(x, y) {
  return x + y;
}
console.log(add(3, 4)); // Output: 7
```

The add function takes two arguments, x and y, and returns their sum. Now, let's curry this function. We can use a technique called partial application to achieve currying:

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

console.log(add(3)(4)); // Output: 7
```

In the curried version, the add function now takes one argument, x, and returns another function that takes the second argument, y, and performs the addition. This allows us to call add(3) and obtain a new function that can be called with the remaining argument 4.

Currying provides several benefits. One advantage is that it allows us to create specialized functions from a more generic one. We can create a reusable addOne function by partially applying the add function:

```
const addOne = add(1);
console.log(addOne(5)); // Output: 6
console.log(addOne(10)); // Output: 11
```

Here, addOne is a new function derived from add that always adds 1 to its argument. We can reuse this function throughout our codebase without duplicating the logic.

Another advantage of currying is the ability to create higher-order functions that enhance composability. Consider a function called multiply that takes three arguments and returns their product:

```
function multiply(x, y, z) {
  return x * y * z;
}

console.log(multiply(2, 3, 4)); // Output: 24
```

Now, let's curry the multiply function:

```
function multiply(x) {
  return function(y) {
    return function(z) {
      return x * y * z;
    };
  };
}

console.log(multiply(2)(3)(4)); // Output: 24
```

Closures

Closures are a fundamental concept in JavaScript where a function retains access to its lexical scope, even when the function is executed outside that scope. In other words, a closure is a function that has access to its own scope, the outer function's scope, and the global scope.

```
function createCounter() {

let count = 0;

return function() {

count++; return count;

};

} let counter = createCounter();

 console.log(counter()); // Output: 1

console.log(counter()); // Output: 2

console.log(counter()); // Output: 3
```

● createCounter defines a count variable and returns a function that increments and returns count.

● Thereturned function is a closure because it retains access to the count variable even after createCounter has finished executing.

Object cloning (reference copying, shallow copy and deep copy)

Object cloning in JavaScript refers to creating a copy of an object. There are different types of object cloning, including reference copying, shallow copying, and deep copying. Each type of cloning has its own use cases and behaviors.

1. Reference Copying Reference copying means creating a new variable that points to the same object in memory. Modifying the object through any of the references will affect the original object.

```
let original = { name: "Alice", age: 25 };

let referenceCopy = original;

referenceCopy.age = 30;

console.log(original.age); // Output: 30 (since both reference the same object)
```

2. Shallow Copy A shallow copy creates a new object with the same properties as the original object. However, if the original object has nested objects, the shallow copy will contain references to the same nested objects, rather than creating new copies of them.

Methods to Create a Shallow Copy

- **Using Object.assign():**

```
let original = { name: "Alice", age: 25, address: { city: "Wonderland" } };

let shallowCopy = Object.assign({}, original);

shallowCopy.address.city = "New Wonderland";

console.log(original.address.city); // Output: "New Wonderland"
```

- **Using Spread Operator (...):**

```
let original = { name: "Alice", age: 25, address: { city: "Wonderland" } };
let shallowCopy = { ...original };
shallowCopy.address.city = "New Wonderland";
console.log(original.address.city); // Output: "New Wonderland
```

3. Deep Copy

A deep copy creates a new object and recursively copies all nested objects, ensuring

that no references to the original objects are retained.

JavaScript DOM manipulation

DOM(Document Object Model) manipulation is a crucial aspect of JavaScript programming that involves interacting with and changing the content, structure, and style of web pages. Here's an overview of some common DOM manipulation techniques:

1. Selecting Elements You can select elements from the DOM using various methods provided by the document

● getElementById: Selects an element by its ID.

```
let element = document.getElementById("myId");
```

● getElementsByClassName: Selects elements by their class name.

```
let elements = document.getElementsByClassName("myClass");
```

● getElementsByTagName: Selects elements by their tag name.

```
let elements = document.getElementsByTagName("div");
```

● querySelector: Selects the first element that matches a CSS selector.

```
let element = document.querySelector(".myClass");
```

● querySelectorAll: Selects all elements that match a CSS selector.

```
let elements = document.querySelectorAll(".myClass");
```

2. Modifying Element Content

● innerHTML: Gets or sets the HTML content inside an element.

```
let element = document.getElementById("myId");

element.innerHTML = "New content";
```

● text Content: Gets or sets the text content inside an element.

```
let element = document.getElementById("myId");

element.textContent = "New text content";
```

Memoization

Memoization in JavaScript is an optimization technique used to **cache the results** of expensive function calls to avoid redundant computations. It stores the results of a function based on its arguments, and if the function is called again with the same arguments, the cached result is returned instead of recomputing the result.

How Memoization Works:

- When a function is invoked, it checks if the result for the given arguments is already stored (cached).
- If the result exists in the cache, it is returned immediately.
- If not, the function computes the result, stores it in the cache, and returns the result.

Memoization is particularly useful for **expensive calculations** like recursive functions (e.g., Fibonacci series, factorial), where the same inputs might be processed multiple times.

Example: Memoizing a Fibonacci Function

The Fibonacci sequence is a great example where memoization can significantly reduce the number of redundant calculations.

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
console.log(fibonacci(10)); // Output: 55
```

In the above example, fibonacci(10) will call fibonacci(9), fibonacci(8), etc., repeatedly, leading to redundant calculations.

Optimized Fibonacci with Memoization:

```
function fibonacci(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];  // Return cached result if it exists
  // Compute and store the result in the memo object
  memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
```

```
  return memo[n];
}

console.log(fibonacci(10)); // Output: 55

console.log(fibonacci(50)); // Output: 12586269025
```

How It Works:

- The memo object is used to store previously calculated results.
- Before computing fibonacci(n), the function checks if the result is already in memo. If it is, the cached result is returned.
- If it's not cached, the function computes the result and stores it in memo for future calls.

Benefits of Memoization:

1. **Performance Improvement**:
    - By avoiding redundant computations, memoization reduces the time complexity of recursive algorithms. For instance, without memoization, the Fibonacci function has an exponential time complexity O(2^n). With memoization, it reduces to linear time complexity O(n).
2. **Optimizes Recursive Functions**:
    - Memoization is highly effective for problems that involve **overlapping subproblems**, such as dynamic programming problems (e.g., Fibonacci, factorial, knapsack, etc.).

Asynchronous Programming

Asynchronous programming is a key concept in JavaScript, allowing for non-blocking operations, which means the execution of other code can continue while waiting for an asynchronous operation to complete.

Callback Functions

A callback function 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.

```
function fetchData(callback) {

setTimeout(() => {

callback("Data loaded"); },
```

```
2000);
}
fetchData((message) => {
console.log(message); // Output after 2 seconds: "Data loaded"
});
```

Callback Hell

Callback Hell refers to a situation where multiple nested callback functions make the code difficult to read and maintain.

```
function fetchData1(callback) {
setTimeout(() => {
console.log("Data 1 loaded");
callback(); }, 1000);
}
function fetchData2(callback) {
setTimeout(() => { console.log("Data 2 loaded");
 callback(); }, 1000);
}
function fetchData3(callback) {
setTimeout(() => { console.log("Data 3 loaded");
callback(); }, 1000);
}
// Callback Hell Example
fetchData1(() => {
fetchData2(() => {
fetchData3(() => {
 console.log("All data loaded");
});
```

});

});

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation. Promises provide a cleaner, more readable alternative to callbacks and help avoid callback hell.

Creating a Promises

let promise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data loaded"); }, 2000);

});

promise.then((message) => { console.log(message); // Output after 2 seconds: "Data loaded" }).catch((error) => {

console.error(error);

})

Promise Chaining

Promise chaining allows you to execute a sequence of asynchronous operations where each operation starts when the previous one succeeds.

let fetchData1 = () => new Promise((resolve) => {

setTimeout(() => {

console.log("Data 1 loaded");

resolve();

}, 1000);

});

let fetchData2 = () => new Promise((resolve) => {

setTimeout(() => { console.log("Data 2 loaded");

resolve(); }, 1000);

});

```
let fetchData3 = () => new Promise((resolve) => {
setTimeout(() => {
console.log("Data 3 loaded");
resolve(); }, 1000);
}); fetchData1()
.then(fetchData2) .
then(fetchData3) .
then(() => {
console.log("All data loaded");
});
```

Async/Await

Async/Await is syntactic sugar built on top of Promises, making asynchronous code look and behave more like synchronous code. The async keyword before a function declaration makes the function return a Promise, and the await keyword pauses the function execution until the Promise settles.

```
let fetchData1 = () => new Promise((resolve) => {
setTimeout(() => {
console.log("Data 1 loaded");
resolve();
}, 1000);
 });
let fetchData2 = () => new Promise((resolve) => {
setTimeout(() => {
console.log("Data 2 loaded");
resolve();
}, 1000); });
 let fetchData3 = () => new Promise((resolve) => {
setTimeout(() => {
```

```
console.log("Data 3 loaded");

resolve();

}, 1000);

});

async function fetchAllData() {

 await fetchData1();

await fetchData2();

await fetchData3();

console.log("All data loaded");

 }

fetchAllData();
```

Error handling in promise and error handling using try catch

Error handling in JavaScript is crucial for building robust applications. There are two primary ways to handle errors: using try...catch blocks and handling errors in Promises. Each has its use cases and benefits.

Error Handling in Promises

Promises provide a built-in mechanism for handling errors. If a promise is rejected, it will call the catch method, which allows you to handle errors in asynchronous operations.

```
let promise = new Promise((resolve, reject) => {

let success = false; // Simulate an operation

if (success) {

resolve("Operation succeeded");

} else {

reject("Operation failed");

}

});

promise .then((result) => {
```

```
console.log(result);
}) .catch((error) => {
 console.error("Error:", error);
});
```

Chained Promises

When chaining promises, any error in the chain will be caught by the nearest catch block.

```
let promise = new Promise((resolve, reject) => {
resolve("Start"); });
promise .then((result) => {
console.log(result);
return new Promise((resolve, reject) => {
reject("Something went wrong");
});
})
 .then((result) => {
console.log("This won't be logged");
 })
 .catch((error) => {
 console.error("Caught an error:", error);
 });
```

Error Handling with try...catch

try...catch is used for handling exceptions in synchronous code. When an error occurs in the try block, control is transferred to the catch block.

```
try {
let result = riskyOperation(); // This may throw an error
console.log(result);
}
```

```
catch (error) {
console.error("Error caught:", error);
}
```

Using try...catch with Asynchronous Code

When using async/await, you can combine try...catch with asynchronous code for error handling.

```
async function asyncOperation() {
return new Promise((resolve, reject) => {
setTimeout(() => {
let success = false; // Simulate an operation if (success) {
 resolve("Operation succeeded");
 } else {
 reject("Operation failed");
}
}, 1000); });
}
async function performAsyncOperation() {
try {
let result = await asyncOperation();
console.log(result);
}
catch (error) {
 console.error("Error caught:", error);
} }
performAsyncOperation();
```

Combining Promises and try...catch

You can combine both approaches for comprehensive error handling in complex applications.

```
async function fetchData() {
```

```
 try {
let response = await fetch("https://api.example.com/data");
if (!response.ok) {
throw new Error("Network response was not ok");
}
 let data = await response.json();
return data;
}
catch (error) {
 console.error("Fetch error:", error);
throw error; // Re-throw the error to be caught by a promise chain
}
 }
fetchData() .then((data) => {
console.log("Data received:", data);
}) .catch((error) => {
console.error("Caught an error in promise chain:", error);
 });
```

Object in JavaScript

Objects in JavaScript are collections of key-value pairs, where the **keys** are strings (or Symbols), and the **values** can be of any data type, including other objects, functions, or arrays.

Objects are a fundamental part of JavaScript, allowing us to structure and group related data together.

Creating an Object

Using Object Literal Syntax

- The simplest way to create an object is using the literal syntax: {}.

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

Using the new Object() Syntax

- You can also create an object using the Object constructor, though the literal syntax is preferred due to its simplicity.

```
let car = new Object();
car.brand = "Toyota";
car.model = "Corolla";
```

Accessing Object Properties

- You can access object properties using **dot notation** or **bracket notation**.

Dot Notation

- The most common and readable way.

```
console.log(person.name); // "John"
```

Bracket Notation

- Useful when the key is dynamic or contains special characters.

```
console.log(person["age"]); // 30
```

Adding, Modifying, and Deleting Properties

Adding/Modifying Properties

- Properties can be added or modified after object creation.

```
person.gender = "male";  // Adding
person.age = 31;         // Modifying
```

Deleting Properties

- Use the delete keyword to remove a property from an object.

```
delete person.isStudent;
```

5. Methods in Objects

- **Methods** are functions that are properties of an object. They allow the object to perform actions.

Defining Methods

- A function can be added as a property, either using the function keyword or shorthand.

```
let person = {
  name: "John",
  greet: function() {
    console.log("Hello, " + this.name);
  },
};
person.greet();  // "Hello, John"
```

6. The this Keyword

- Inside an object method, this refers to the object itself. This allows methods to access other properties of the same object.

```
let car = {
  brand: "Toyota",
  model: "Corolla",
  getDetails() {
    return `${this.brand} ${this.model}`;
  },
};
console.log(car.getDetails());  // "Toyota Corolla"
```

7. Nested Objects

- An object can have other objects as properties, allowing for more complex data structures.

```
let user = {
  name: "Alice",
```

```
  address: {
    city: "New York",
    zip: 10001,
  },
};


console.log(user.address.city);  // "New York"
```

Object Methods

JavaScript provides built-in methods to work with objects.

Object.keys()

- Returns an array of the object's own property names (keys)

```
console.log(Object.keys(person));  // ["name", "age", "gender"]
```

Object.values()

- Returns an array of the object's values.

```
console.log(Object.values(person));  // ["John", 30, "male"]
```

Object.entries()

- Returns an array of key-value pairs.

```
console.log(Object.entries(person));  // [["name", "John"], ["age", 30], ["gender", "male"]]
```