UNIT-III

Introduction to JavaScript- Features of JavaScript- Introduction to ES6- Variables- Template Literals- Expressions- Datatype Conversions- Operators- Conditional Statements- Looping Statements- One Dimensional Arrays- Two dimensional Arrays- Jagged Arrays- Functions- Anonymous Functions Arrow Functions- Recursive function- Default Parameters.

Introduction to JavaScript:

What is JavaScript?

JavaScript is a lightweight, cross-platform, single-threaded, and interpreted compiled programming language. It is commonly known as the scripting language for webpages and is widely used for web development, as well as in various non-browser environments.

JavaScript is a weakly typed language (dynamically typed) and can be utilized for both client-side and server-side development. It supports both imperative and declarative programming paradigms. JavaScript features a standard library of objects, such as Array, Date, and Math, along with a core set of language elements like operators, control structures, and statements.

- **Client-side**: JavaScript provides objects to control a browser and its Document Object Model (DOM). Client-side extensions enable applications to place elements on HTML forms and respond to user events such as mouse clicks, form input, and page navigation. Useful libraries for client-side development include AngularJS, ReactJS, and VueJS.
- **Server-side**: JavaScript supplies objects relevant to running on a server. Server-side extensions allow applications to communicate with databases, maintain continuity of information across different invocations, and perform file manipulations. The most popular framework for server-side JavaScript is Node.js.
- **Imperative Language**: This type of language focuses on how tasks are accomplished, controlling the flow of computation. Procedural and object-oriented approaches fall under this category. For example, using async/await in this paradigm emphasizes what to do after the asynchronous call.
- **Declarative Programming**: In this type of programming, the emphasis is on describing the desired result without detailing the exact steps to achieve it. The main goal is to express logical computation, exemplified by constructs like arrow functions.

How to Link JavaScript File in HTML

JavaScript can be added to an HTML file in two ways:

- **Internal JS**: You can add JavaScript directly to your HTML file by writing the code inside the <script> tag. The <script> tag can be placed either inside the <head> or the <body> tag according to your requirements.
- **External JS**: You can write JavaScript code in a separate file with the .js

extension and then link this file inside the <head> tag of the HTML file where you want to add the code.

Syntax

```
<script>
    // JavaScript Code
</script>
```

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Basic Example to Describe JavaScript</title>
</head>
<body>
<!-- JavaScript code can be embedded inside head section or body section -->
 <script>
    console.log ("Welcome to GeeksforGeeks");
</script>
</body>
</html>
```

Output

The output will display on the console:

Welcome to GeeksforGeeks

Features of JavaScript

According to a recent survey conducted by Stack Overflow—a popular online platform where developers ask and answer programming-related questions—JavaScript is the most popular language on earth. With advances in browser technology and JavaScript moving into the server with Node.js and other frameworks, it is capable of so much more. Here are a few things that we can do with JavaScript:

- JavaScript was created for DOM manipulation. Earlier websites were mostly static; after JS was created, dynamic websites emerged.
- Functions in JavaScript are objects. They can have properties and methods just like other objects and can be passed as arguments to other functions.
- JavaScript can handle date and time.
- It performs form validation, even though the forms are created using HTML.
- No compiler is needed.

Introduction to ES6

ES6, or ECMAScript 2015, is the 6th version of the ECMAScript programming language. ECMAScript is the standardization of JavaScript, which was released in 2015 and subsequently renamed ECMAScript 2015. ECMAScript and JavaScript are both different.

ECMAScript vs JavaScript

- **ECMAScript**: Stands for **European Computer Manufacturers Association**. It is the specification defined in ECMA-262 for creating a general-purpose scripting language. Introduced in **1997**, it serves as a standardization for creating a scripting language. It was introduced by ECMA International and provides guidelines for how to create a scripting language.
- **JavaScript**: A general-purpose scripting language that conforms to the ECMAScript specification. It is an implementation that instructs how to use a scripting language.

ES6

JavaScript ES6 has been around for a few years now, allowing us to write code in a clever way that makes it more modern and readable. It's fair to say that with the use of ES6 features, we write less and do more; hence the term "write less, do more" definitely suits ES6.

ES6 introduced several key features like const, let, arrow functions, template literals, default parameters, and much more.

ES6 Features

- **Let and Const Keywords**: -`let` allows users to define block-scoped variables, while `const` is used to define constants that cannot be reassigned. This improves variable management compared to the traditional `var` keyword.
- **Arrow Functions**: -Arrow functions provide a more concise syntax for writing function expressions. They simplify the syntax by omitting the `function` keyword and automatically binding the `this` value from the surrounding context.
- **Multi-line Strings**: -ES6 allows for the creation of multi-line strings using back-ticks (`), making it easier to work with strings that span multiple lines.
- **Default Parameters**: -Functions can have default parameter values defined in their signatures, simplifying the assignment of default values compared to earlier approaches.
- **Template Literals**: -Template literals facilitate the inclusion of placeholders within strings, allowing for easier string interpolation.

Difference between ES6 and JavaScript:

Aspect	ECMAScript	JavaScript
Definition	A standardized specification that outlines the rules, guidelines, and features of the scripting language.	An implementation of the ECMAScript specification with additional features for web development.

Purpose	Provides a foundation for scripting languages and serves as a standard for developers.	Designed specifically for interactive web pages and applications, adding features like the Document Object Model (DOM) and browser APIs.
Features	Defines the core features of the language, including syntax, types, and operations.	Extends ECMAScript with additional functionalities such as event handling, the DOM, and other browser-specific features.
Versioning	Releases new versions (e.g., ES5, ES6) that introduce new features and improvements.	While it follows ECMAScript standards, it also includes proprietary features specific to various environments (like Node.js, browsers, etc.).

Variables in JavaScript

Variables are used to store data in JavaScript. They allow you to store reusable values, and the values of variables are allocated using the assignment operator ("=").

JavaScript Assignment Operator

The JavaScript assignment operator is equal (=), which assigns the value of the right-hand operand to its left-hand operand.

Example:

```
y = "Geek";
```

JavaScript Identifiers

JavaScript variables must have unique names, known as **identifiers**.

Basic rules to declare a variable in JavaScript:

- Identifiers are case-sensitive.
- They can only begin with a letter, underscore (_), or dollar sign ($).
- They can contain letters, numbers, underscores, or dollar signs.
- A variable name cannot be a reserved keyword.

JavaScript is a dynamically typed language, meaning the type of variables is determined at runtime. Therefore, there is no need to explicitly define the type of a variable.

Ways to Declare Variables

Variables in JavaScript can be declared in three ways:

1. **Using var**
2. **Using let**
3. **Using const**

Note: In JavaScript, variables can be declared automatically.

Syntax Examples:

```
// Declaration using var

var geek = "Hello Geek";

// Declaration using let

let $ = "Welcome";

// Declaration using const

const _example = "Gfg";
```

All three keywords serve the basic task of declaring a variable but have some differences. Initially, all variables in JavaScript were written using the var keyword, but in ES6, the keywords let and const were introduced.

Examples of Variable Declaration

Example 1: Declaring Variables Using var

```
<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <meta name="viewport" content="width=device-width, initial-scale=1.0">

   <title>JavaScript Variable Declaration Examples</title>

</head>

<body>

   <h2>Example 1: Declaring Variables Using var</h2>

   <p id="var-output"></p>

   <script>

      var a = "Hello Geeks";

      var b = 10;

      var c = 12;
```

```
        var d = b + c;
        document.getElementById("var-output").innerHTML = a + ", " + b + ", " + c + ", " + d;
    </script>
</body>
</html>
```

Output: Hello Geeks, 10, 12, 22

Example 2: Declaring Variables Using let

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JavaScript Variable Declaration Examples</title>
</head>
<body>
<h2>Example 2: Declaring Variables Using let</h2>
    <p id="let-output"></p>
    <script>
        let a = "Hello learners";
        let b = "joining";
        let c = " 12";
        let d = b + c;
        document.getElementById("let-output").innerHTML = a + ", " + b + ", " + c + ", " + d;
    </script>
</body>
```

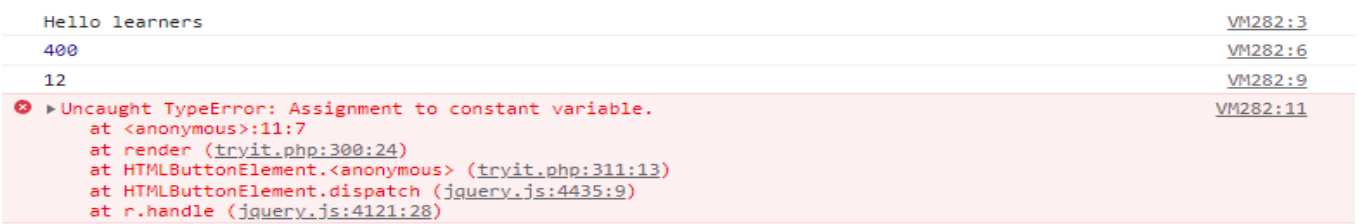
```
</html>
```

Output: Hello learners, joining, 12, joining 12

Example 3: Declaring Variables Using const

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JavaScript Variable Declaration Examples</title>
</head>
<body>
<h2>Example 3: Declaring Variables Using const</h2>
    <p id="const-output"></p>
    <script>
        const a = "Hello learners";
        const b = 400;
        const c = "12";
        document.getElementById("const-output").innerHTML = a + ", " + b + ", " + c;
    </script>
</body>
</html>
```

Output:



Explanation of const

The const keyword is used when you want to assign a value permanently to a variable. Attempting to change the value of a variable declared with the const keyword will throw an error. Variables declared with var and let are mutable, meaning their values can be changed, but those declared using const are immutable.

Example: In this example, we are trying to access the block scoped variables outside the block that's why we are getting error.

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Block Scope Example in JavaScript</title>

</head>

<body>

    <h1>Block Scope Example in JavaScript</h1>

    <h2>Example: Block Scoped Variables</h2>

    <p id="block-output"></p>

    <p id="outside-output"></p>

    <p id="error-a"></p>

    <p id="error-c"></p>

    <script>

        // Block-scoped variables

        {

            let a = 2;

            var b = 4;

            const c = "Hello";
```

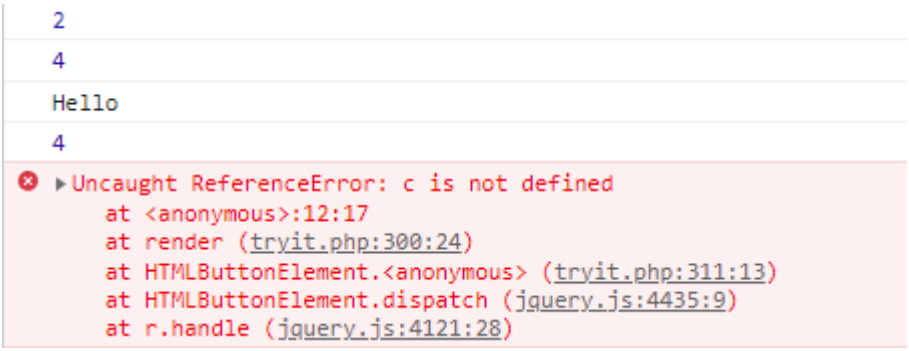
```
        document.getElementById("block-output").innerHTML = "Inside block: " + a + ", " + b + ", " + c;

    }

    // Output for b outside the block

    document.getElementById("outside-output").innerHTML = "Outside block b: " + b;

    // Attempting to access a and c outside the block will throw errors

    // Uncommenting the following lines will cause errors:

    // document.getElementById("error-a").innerHTML = "Outside block a: " + a; // This will throw an error

    // document.getElementById("error-c").innerHTML = "Outside block c: " + c; // This will throw an error

  </script>

</body>

</html>
```

Output:

When you open this HTML code in a web browser, you will see:

- Inside block: 2, 4, Hello
- Outside block b: 4

Accessing a and c outside the block will not display any errors on the page but would cause errors if you uncomment those lines in the JavaScript.



When to Use var, let, or const

- Use const if the value should not be changed.
- Use let if you want a mutable value or if you cannot use const.

- Use var only if you need to support older browsers.

Comparison of var, let, and const

Property	**var**	**let**	**const**
Scope	Function scoped	Block scoped	Block scoped
Updatable	Mutable	Mutable	Immutable
Redeclaration	Can be redeclared	Cannot be redeclared	Cannot be redeclared
Hoisting	Hoisted at top	Hoisted at top	Hoisted at top
Origins	Pre ES2015	ES2015 (ES6)	ES2015 (ES6)

JavaScript Template Literals

Template literals in ES6 provide new features to create strings that offer more control over dynamic content. Traditionally, strings are created using single quotes (‘) or double quotes (“), while template literals are created using the backtick (`) character.

Syntax

let s = `some string`; // Creates a string using backticks.

Multiline Strings

To create a multiline string, an escape sequence \n was traditionally used to indicate a new line. However, with template literals, there is no need to add \n; the string ends only when it reaches the backtick (`) character.

Example:

```
// Without template literal
console.log ('Some text that I want \non two lines!');
// With template literal
console.log (`Some text that I want on two lines! `);
```

Output:

```
Some text that I want
on two lines!
Some text that I want
on two lines!
```

Expressions

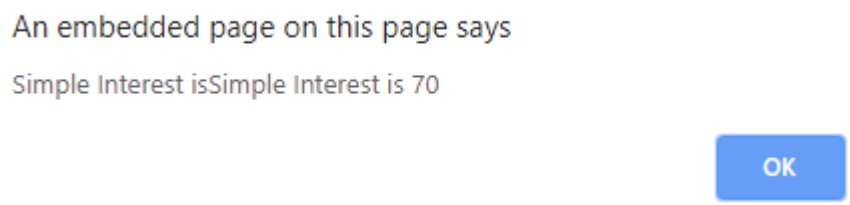
Template literals allow you to dynamically add values using expressions. The ${} syntax permits any expression inside that produces a value, whether it's a string stored in a variable or a computation.

Example:

This example calculates simple interest and embeds it directly in a string.

```
let principal = 1000;
let noofyears = 1;
let rateofinterest = 7;
let SI = `Simple Interest is ${(principal * noofyears * rateofinterest) / 100} `;
alert ("Simple Interest is " + SI);
```

Output:



Tagged Templates

Tagged template literals enable the creation of templates with added functionality. They are defined like a function, but unlike regular functions, they do not use parentheses when called. An array of strings is passed as a parameter.

Example:

This example uses tagged template to output the string "GeeksforGeeks".

```
function TaggedLiteralEg(strings) {
        document. write(strings);
}
TaggedLiteralEg`GeeksforGeeks`;
```

Output:

GeeksforGeeks

Raw Strings

The raw method of template literals allows access to raw strings as they were entered, without processing escape sequences. The String.raw() method can be used to create raw strings.

Example:

This example demonstrates how to create a raw string that ignores escape sequences

```
let s = String.raw`Welcome to GeeksforGeeks Value of expression is ${2 + 3}`;
console.log(s);
```

Output:

Welcome to GeeksforGeeks Value of expression is 5

Nested Templates

Template literals can also be nested to evaluate multiple expressions or conditions. This enhances readability and eases the developer's job.

Example:

This example finds and displays the maximum of three numbers using nested template literals.

```
function maximum(x, y, z) {
   let c = `value ${(y > x && y > z) ? 'y is greater' :
      `${x > z ? 'x is greater' : 'z is greater'}`}`;
   return c;
}
console.log(maximum(5, 11, 15)); // Output: value z is greater
console.log(maximum(15, 11, 3));  // Output: value x is greater
console.log(maximum(11, 33, 2));  // Output: value y is greater
```

Output:

value z is greater
value x is greater
value y is greater

JavaScript Expressions

JavaScript expressions are crucial building blocks of the language, combining literals, variables, operators, and functions to evaluate to a single value. This can be a number, a string, a boolean, or any other type, depending on the expression.

Types of JavaScript Expressions

1. **Primary Expressions**: - These are the simplest forms of expressions that directly yield a value.

   Examples:

   - **this Keyword**: Refers to the current execution context, often used within methods to access the object invoking the method.
   - **Async/Await Function**: Ensures that asynchronous code runs in a predictable manner, maintaining the execution flow without breaking the thread.
   - **Object Initializer**: Creates objects using key-value pairs, making it easier to manage data and functions associated with that data.
   - **Grouping Operator**: Parentheses () are used to control the order of evaluation in complex expressions, ensuring that certain operations are performed first.

- **Async Function**: Defined with the async keyword, allows the use of await for asynchronous operations, simplifying promise handling.
- **Regular Expressions**: Patterns that provide a powerful way to perform text search and manipulation.

2. **Function Expression**: -A function defined as part of an expression, allowing it to be treated like any other value. This is particularly useful for passing functions as arguments or returning them from other functions.
3. **Class Expression**: -Similar to function expressions, class expressions define a class that can be assigned to a variable. The class name is only accessible within the class definition itself, which helps in encapsulating functionality.

Example of Expression Evaluation

Consider the following example that illustrates the concept of expressions in JavaScript:

```
function* func() {
   yield 1;
   yield 2;
   yield 3;
   yield " - Geeks";
}
let obj = '';
for (const i of func()) {
   obj = obj + i;
}
console.log(obj);
```

Output

```
123 - Geeks
```

Explanation:

- **Generator Function**: The function func is a generator that yields multiple values. Each call to yield produces a value that can be iterated over.
- **Iteration**: The for...of loop iterates over the yielded values, concatenating them into the obj string.
- **Output**: The final output is 123 - Geeks, demonstrating how expressions can yield and manipulate values dynamically.

Data Type Conversion in JavaScript

JavaScript is a loosely typed language, which means it allows variables to hold values

of any data type without strict declarations. While JavaScript often performs implicit type conversions, there are instances where explicit conversions are necessary. The two primary types of data conversions in JavaScript: converting values to strings and converting values to numbers.

1. Converting Values to Strings

JavaScript provides two main methods for converting values to strings: the String() function and the .toString() method.

Syntax

```
String(value)  // or variableName.toString()
```

Examples

- **Converting Numbers to Strings**

  ```
  let v1 = 123;
  console.log("Type Of v1 before conversion: " + typeof v1);
  console.log("Type Of v1 after conversion: " + typeof String(v1).toString());
  ```

  Output:

  ```
  Type Of v1 before conversion: number
  Type Of v1 after conversion: string
  ```

  Explanation: Converts the number 123 to a string, changing its type from number to string.

- **Converting Booleans to Strings**

  ```
  console.log("Type Of false before conversion: " + typeof false + and Type of true before conversion: " + typeof true);
  console.log("Type Of false after conversion: " + typeof String(false) + and Type of true after conversion: " + typeof (true).toString());
  ```

  Output:

  ```
  Type Of false before conversion: boolean and Type of true before conversion: boolean
  Type Of false after conversion: string and Type of true after conversion: string
  ```

  Explanation: Converts false and true to strings, changing their types from boolean to string.

- **Converting Date Objects to Strings**

  ```
  const currentDate = new Date();
  const pastDate = new Date('2023-12-20');
  ```

```
console.log("Type Of currentDate before conversion: " + typeof currentDate + " and Type of pastDate before conversion: " + typeof pastDate);
console.log("Type Of currentDate after conversion: " + typeof String(currentDate) + " and Type of pastDate after conversion: " + typeof (pastDate).toString());
```

Output:

```
Type Of currentDate before conversion: object and Type of pastDate before conversion: object
Type Of currentDate after conversion: string and Type of pastDate after conversion: string
```

Explanation: Converts date objects to strings, changing their types from object to string.

2. Converting Values to Numbers

To convert values to numbers, JavaScript uses the Number() function. This function can convert numerical strings, boolean values, and date objects to numbers. If the conversion fails, it results in NaN (Not a Number).

Syntax

Examples

- **Converting Strings to Numbers**

```
let v = "144";
console.log("Type of v before conversion: " + typeof v);
console.log("Type of v after conversion: " + typeof Number(v));
```

Output:

```
Type of v before conversion: string
Type of v after conversion: number
```

Explanation: Converts the string "144" to a number, changing its type from string to number.

- **Converting Booleans to Numbers**

```
console.log("Type of true before conversion: " + typeof true);
console.log("Type of true after conversion: " + typeof Number(true));
```

Output:

```
Type of true before conversion: boolean
Type of true after conversion: number
```

Explanation: Converts the boolean true to a number, changing its type from boolean to number.

- **Converting Date Objects to Numbers**

  let d = new Date('1995-12-17T03:24:00');

  console.log("Type of d before conversion: " + typeof d);

  console.log("Type of d after conversion: " + typeof Number(d));

  Output:

  Type of d before conversion: object

  Type of d after conversion: number

  Explanation: Converts a date object to a number (timestamp), changing its type from object to number.

JavaScript Operators

JavaScript Operators are symbols used to perform specific mathematical, comparison, assignment, and logical computations on operands. They are fundamental elements in JavaScript programming, allowing developers to manipulate data and control program flow efficiently.

There are various operators supported by JavaScript.

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators
- Comma Operators
- Unary Operators
- Relational Operators
- BigInt Operators
- String Operators

Arithmetic Operators

JavaScript Arithmetic Operators are the operator that operate upon the numerical values and return a numerical value.

Arithmetic Operators list

There are many arithmetic operators as shown in the table with the description.

Operator Name	Usage	Operation
Addition (+)	a + b	Adds two numbers or concatenates strings.
Subtraction (-)	a - b	Computes the difference between two numbers.
Multiplication (*)	a * b	Multiplies two numbers.
Division (/)	a / b	Divides the left operand by the right operand.
Modulus (%)	a % b	Returns the remainder of the division of two operands.
Exponentiation (**)	a ** b	Raises the left operand to the power of the right operand.
Increment (++)	a++ or ++a	Increases a variable's value by one.
Decrement (--)	a-- or --a	Decreases a variable's value by one.
Unary Plus (+)	+a	Converts a non-number to a number.
Unary Negation (-)	-a	Converts the operand to its negative value

Example:

```
// Arithmetic Operations
let result1 = 1 + 2; // Addition
let result2 = 5 + "hello"; // Addition with string
let result3 = 10 - 7; // Subtraction
let result4 = "Hello" - 1; // Subtraction with string
let result5 = 3 * 3; // Multiplication
let result6 = -4 * 4; // Multiplication
let result7 = 5 / 2; // Division
let result8 = 1.0 / 2.0; // Division
let result9 = 9 % 5; // Modulus
let result10 = -12 % 5; // Modulus

// Output results
console.log("Addition: 1 + 2 =", result1);
console.log("Addition with string: 5 + 'hello' =", result2);
console.log("Subtraction: 10 - 7 =", result3);
console.log("Subtraction with string: 'Hello' - 1 =", result4);
console.log("Multiplication: 3 * 3 =", result5);
console.log("Multiplication: -4 * 4 =", result6);
console.log("Division: 5 / 2 =", result7);
```

```
console.log("Division: 1.0 / 2.0 =", result8);
console.log("Modulus: 9 % 5 =", result9);
console.log("Modulus: -12 % 5 =", result10);
```

Output:

```
Addition: 1 + 2 = 3
Addition with string: 5 + 'hello' = 5hello
Subtraction: 10 - 7 = 3
Subtraction with string: 'Hello' - 1 = NaN
Multiplication: 3 * 3 = 9
Multiplication: -4 * 4 = -16
Division: 5 / 2 = 2.5
Division: 1.0 / 2.0 = 0.5
Modulus: 9 % 5 = 4
Modulus: -12 % 5 = -2
```

Assignment Operators

JavaScript Assignment Operators are used to assign values to variables.

Operator Name	Usage	Operation
Assignment (=)	a = b	Assigns the value of b to a.
Addition Assignment (+=)	a += b	Adds b to a and assigns the result to a.
Subtraction Assignment (-=)	a -= b	Subtracts b from a and assigns the result to a.
Multiplication Assignment (*=)	a *= b	Multiplies a by b and assigns the result to a.
Division Assignment (/=)	a /= b	Divides a by b and assigns the result to a.
Modulus Assignment (%=)	a %= b	Computes the modulus and assigns the result to a.
Exponentiation Assignment (**=)	a **= b	Raises a to the power of b and assigns the result to a.

Comparison Operators

Comparison Operators are used to compare two values and return a boolean result.

Operator Name	Usage	Operation
Equal (==)	a == b	Checks if a is equal to b (type coercion).
Strict Equal (===)	a === b	Checks if a is strictly equal to b (no type coercion).
Not Equal (!=)	a != b	Checks if a is not equal to b (type coercion).
Strict Not Equal (!==)	a !== b	Checks if a is strictly not equal to b.
Greater Than (>)	a > b	Checks if a is greater than b.
Less Than (<)	a < b	Checks if a is less than b.

Greater Than or Equal (>=)	a >= b	Checks if a is greater than or equal to b.
Less Than or Equal (<=)	a <= b	Checks if a is less than or equal to b.

Example

```
// Comparison Operations
let x = 5;
let y = '5';
// Output results
console.log("Equal:", x == y);          // true (type coercion)
console.log("Strict Equal:", x === y);   // false (no type coercion)
console.log("Not Equal:", x != y);       // false (type coercion)
console.log("Strict Not Equal:", x !== y); // true (no type coercion)
console.log("Greater Than:", x > 3);     // true
console.log("Less Than:", x < 10);       // true
```

Logical Operators

Logical Operators are used to combine multiple boolean expressions.

Operator Name	Usage	Operation
Logical AND (&&)	a && b	Returns true if both a and b are true.
Logical OR (‖)	a || b	Returns true if least a or b are true.
Logical NOT (!)	!a	Returns true if a is false.

Example:

```
// Logical Operations
let a = true;
let b = false;
// Output results
console.log("Logical AND:", a && b);     // false
console.log("Logical OR:", a || b);      // true
console.log("Logical NOT:", !a);         // false
```

String Operators

String Operators are used to manipulate string values.

Operator Name	Usage	Operation
Concatenation (+)	a + b	Combines two strings into one.
Template Literals (`)	${expression}	Allows embedding expressions in strings.

Example:

```
let str1 = "Hello, ";
```

```
let str2 = "World!";
let greeting = str1 + str2; // "Hello, World!"
let name = "Alice";
let message = `Hello, ${name}!`; // "Hello, Alice!"
```

Conditional Statements in JavaScript

Conditional statements in JavaScript allow the execution of specific blocks of code based on certain conditions. The primary conditional statements include:

Conditional Statement	Description
if statement	Executes a block of code if a specified condition is true.
else statement	Executes a block of code if the preceding if condition is false.
else if statement	Allows for multiple conditions to be tested.
switch statement	Evaluates an expression and executes the case statement that matches its value.
ternary operator	A concise way to write if-else statements in a single line.
nested if-else	Allows for multiple conditions to be checked in a hierarchical manner.

1. Using the if Statement

The if statement evaluates a condition, executing the associated code block if true.

Syntax:

```
if (condition) {
   // Code to be executed if the condition is true
}
```

Example:

```
let num = 20;
if (num % 2 === 0) {
   console.log("Given number is even number.");
}
```

Output:

Given number is even number.

2. Using if-else Statement

The if-else statement executes a block of code based on whether the specified condition is true or false.

Syntax:

```
if (condition) {
```

```
    // Executes if condition is true
} else {
    // Executes if condition is false
}
```

Example:

```
let age = 25;
if (age >= 18) {
    console.log("You are eligible for a driving license");
} else {
    console.log("You are not eligible for a driving license");
}
```

Output:

```
You are eligible for a driving license
```

3. Using else if Statement

The else if statement checks multiple conditions, allowing for more than two options.

Syntax:

```
if (first condition) {
    // Code for first condition
} else if (second condition) {
    // Code for second condition
} else {
    // Code if all conditions are false
}
```

Example:

```
const num = 0;
if (num > 0) {
    console.log("Given number is positive.");
} else if (num < 0) {
    console.log("Given number is negative.");
} else {
    console.log("Given number is zero.");
}
```

Output:

```
Given number is zero.
```

4. Using Switch Statement

The switch statement evaluates an expression and executes the matching case statement.

Syntax:

```
switch (expression) {
   case value1:
      // Code for value1
      break;
   case value2:
      // Code for value2
      break;
   default:
      // Code if no cases match
}
```

Example:

```
const marks = 85;
let Branch;
switch (true) {
   case marks >= 90:
      Branch = "Computer science engineering";
      break;
   case marks >= 80:
      Branch = "Mechanical engineering";
      break;
   default:
      Branch = "Bio technology";
      break;
}
console.log(`Student Branch name is: ${Branch}`);
```

Output:

Student Branch name is: Mechanical engineering

5. Using Ternary Operator

The ternary operator offers a compact way to write conditional statements.

Syntax:

condition ? valueIfTrue : valueIfFalse;

Example:

```
let age = 21;
const result = (age >= 18) ? "You are eligible to vote." : "You are not eligible to vote.";
console.log(result);
```

Output:

You are eligible to vote.

6. Using Nested if-else

Nested if-else statements allow for complex conditional logic by checking multiple conditions in a hierarchy.

Syntax:

```
if (condition1) {
   if (condition2) {
      // Code block 1
   } else {
      // Code block 2
   }
} else {
   // Code block 3
}
```

Example:

```
let weather = "sunny";
let temperature = 25;
if (weather === "sunny") {
   if (temperature > 30) {
      console.log("It's a hot day!");
   } else {
      console.log("It's a warm day.");
   }
} else {
   console.log("Check the weather forecast!");
}
```

Output:

It's a warm day.

Loops in JavaScript

Loops are fundamental programming constructs that allow you to execute a block of code repeatedly, as long as a specified condition remains true. They are essential for automating tasks and handling repetitive actions efficiently.

Types of Loops in JavaScript

1. For Loop
2. While Loop
3. Do-While Loop
4. For-In Loop
5. For-Of Loop
6. Labeled Statement
7. Break Statement
8. Continue Statement
9. Infinite Loop (Loop Error)

1. For Loop

The for loop is a control flow statement that allows code to be executed repeatedly based on a condition. It includes initialization, condition checking, and increment/decrement in a single line.

Syntax:

```
for (initialization; condition; increment/decrement) {
   // Code to be executed
}
```

Example:

```
for (let x = 2; x <= 4; x++) {
   console.log("Value of x: " + x);
}
```

Output:

Value of x: 2

Value of x: 3

Value of x: 4

2. While Loop

The while loop executes a block of code repeatedly as long as a specified condition is true. It is considered an entry control loop.

Syntax:

```
while (condition) {
   // Code to be executed
}
```

Example: To Prints numbers from 1 to 5.

```
let val = 1;
while (val < 6) {
   console.log(val);
   val += 1;
}
```

Output:

1
2
3
4
5

3. Do-While Loop

The do-while loop is similar to the while loop, but it checks the condition after executing the statements, ensuring that the code runs at least once.

Syntax:

```
do {
   // Code to be executed
} while (condition);
```

Example: To Prints numbers from 1 to 5, ensuring the loop runs at least once.

```
let test = 1;
do {
   console.log(test);
   test++;
} while(test <= 5);
```

Output:

1
2
3
4
5

4. For-In Loop

The for-in loop is used to iterate over the properties of an object. It only iterates over keys with enumerable properties.

Syntax:

```
for (let variable in object) {
   // Code to be executed
}
```

Example: To Prints each key and value from the object.

```
let myObj = { x: 1, y: 2, z: 3 };
for (let key in myObj) {
   console.log(key, myObj[key]);
}
```

Output:

```
x 1
y 2
z 3
```

5. For-Of Loop

The for-of loop is used to iterate over iterable objects, such as arrays and strings, directly accessing their values.

Syntax:

```
for (let variable of iterable) {
   // Code to be executed
}
```

Example: To Prints each value from the array.

```
let arr = [1, 2, 3, 4, 5];
for (let value of arr) {
   console.log(value);
}
```

Output:

```
1
2
3
4
5
```

6. Labeled Statement

Labeled statements allow you to use labels with break and continue to control nested loops.

Syntax:

```
labelName: statement;
```

Example: To Sums values in nested loops, printing each sum until it exceeds 12.

```
let sum = 0, a = 1;
outerloop: while (true) {
   a = 1;
   innerloop: while (a < 3) {
      sum += a;
      if (sum > 12) {
         break outerloop;
      }
      console.log("sum = " + sum);
      a++;
   }
}
```

Output:

```
sum = 1
sum = 3
sum = 6
sum = 10
sum = 15
```

7. Break Statement

The break statement terminates the execution of a loop or switch statement when a specified condition is met.

Syntax:

```
break;
```

Example: To Prints numbers from 1 to 3 and stops when reaching 4.

```
for (let i = 1; i < 6; i++) {
   if (i == 4) break;
   console.log(i);
}
```

Output:

1

2

3

8. Continue Statement

The continue statement skips the current iteration of a loop and moves to the next iteration.

Syntax:

continue;

Example: To Prints odd numbers from 0 to 10 by skipping even numbers.

```
for (let i = 0; i < 11; i++) {
   if (i % 2 == 0) continue;
   console.log(i);
}
```

Output:

1

3

5

7

9

9. Infinite Loop (Loop Error)

An infinite loop occurs when the loop's terminating condition is never met, causing it to run indefinitely. This can happen due to a faulty condition or lack of an update statement.

Example: To Continuously prints 5, 3, and 1 due to an unmet loop condition.

```
for (let i = 5; i != 0; i -= 2) {
   console.log(i); // This will run indefinitely if condition is not met.
}
```

Output:

5

3

1

An "**unmet loop condition**" refers to a situation where the condition that controls a loop never evaluates to false, causing the loop to run indefinitely. In the context of the infinite loop example, the condition i != 0 never becomes false because the loop's logic prevents i from

reaching 0. As a result, the loop continues executing without end.

Arrays in Javascript

What is an Array?

An array in JavaScript is a special variable that can hold multiple values. Instead of declaring multiple variables to store related items (like car names, for example), you can use an array to store them all under a single variable name.

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Arrays</h1>
<p id="demo"></p>
<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Output:

JavaScript Arrays

Saab,Volvo,BMW

Why Use Arrays?

Using arrays is efficient for managing collections of data. If you had multiple items (e.g., 300 cars), it would be impractical to create separate variables for each one. Arrays allow you to group items together and access them using an index.

Creating an Array

There are several ways to create an array in JavaScript:

Array Literal Method:

```
const array_name = [item1, item2, ...];
```

This is the most common and preferred method.

Using the new Keyword:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h1>JavaScript Arrays</h1>
<p id="demo"></p>
<script>
const cars = new Array("Saab", "Volvo", "BMW");
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Output:

JavaScript Arrays

Saab,Volvo,BMW

This method is less common and not recommended for general use due to readability and performance.

Empty Array Initialization:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Arrays</h1>
<p id="demo"></p>
<script>
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Output:

JavaScript Arrays

Saab,Volvo,BMW

Accessing Array Elements

Array elements are accessed using their index. Remember, JavaScript arrays are **zero-indexed**, meaning the first element is at index 0.

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Arrays</h1>
<h2>Bracket Indexing</h2>
<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>
<p id="demo"></p
<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
</script>
</body>
</html>
```

Output:

JavaScript Arrays

Bracket Indexing

JavaScript array elements are accessed using numeric indexes (starting from 0).

Saab

Changing Array Elements

You can change the value of an element in an array by referencing its index:

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Arrays</h1>
<h2>Bracket Indexing</h2>

<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
const cars = ["Saab", "Volvo", "BMW"];
```

```
cars[0] = "Opel";
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Output

JavaScript Arrays

Bracket Indexing

JavaScript array elements are accessed using numeric indexes (starting from 0).

Opel,Volvo,BMW

Converting an Array to a String

The toString() method converts an array to a string of comma-separated values.

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Result:

Banana,Orange,Apple,Mango

One-Dimensional Arrays in JavaScript

A one-dimensional array in JavaScript is a linear structure that holds a collection of items, which can be of the same or different types. You can access each element in the array using its index, which starts from 0.

Examples:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>One-Dimensional Array Example</title>
</head>
<body>
  <h1>One-Dimensional Array in JavaScript</h1>
  <div id="output"></div>
  <script>
    const numbers = [10, 20, 30, 40, 50];
```

```
    const firstElement = numbers[0];  // 10
    const secondElement = numbers[1];  // 20
    numbers[2] = 100;  // Changing 30 to 100
    let outputHTML = `<p>First element: ${firstElement}</p>`;
    outputHTML += `<p>Second element: ${secondElement}</p>`;
    outputHTML += `<p>Modified array: ${numbers.join(', ')}</p>`;
    outputHTML += `<h3>All elements in the array:</h3>`;
    outputHTML += `<ul>`;
    numbers.forEach(number => {
      outputHTML += `<li>${number}</li>`;
    });
    outputHTML += `</ul>`;
    document.getElementById('output').innerHTML = outputHTML;
  </script>
</body>
</html>
```

Explanation of the Code

Declaration and Initialization: const numbers = [10, 20, 30, 40, 50];

Here, we declare a one-dimensional array named numbers and initialize it with five integers.

- **Accessing Elements**:

```
const firstElement = numbers[0];  // 10
const secondElement = numbers[1];  // 20
```

Here, numbers[0] accesses the first element (10) and numbers[1] accesses the second element (20).

- **Modifying an Element**:

```
numbers[2] = 100;  // Changing 30 to 100
```

This changes the value at index 2 from 30 to 100, resulting in the modified array: [10, 20, 100, 40, 50].

- **Generating Output**:

```
let outputHTML = `<p>First element: ${firstElement}</p>`;
outputHTML += `<p>Second element: ${secondElement}</p>`;
outputHTML += `<p>Modified array: ${numbers.join(', ')}</p>`;
```

We build a string of HTML to display the first and second elements, as well as the modified array.

- **Traversing the Array**:

```
numbers.forEach(number => {
  outputHTML += `<li>${number}</li>`;
});
```

This loop goes through each element in the numbers array and creates a list item for each number.

- **Displaying the Output**:

```
document.getElementById('output').innerHTML = outputHTML;
```

Finally, we insert the generated HTML string into the with ID output to display the results on the web page.

Output:

One-Dimensional Array in JavaScript
First element: 10
Second element: 20
Modified array: 10, 20, 100, 40, 50
All elements in the array:
- 10
- 20
- 100
- 40
- 50

Two-Dimensional Arrays in JavaScript

A two-dimensional array (2D array) in JavaScript is an array of arrays. It can be thought of as a grid or table, where each element is itself an array. This structure is useful for representing matrices, tables, or any data that requires a two-dimensional representation.

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Two-Dimensional Array Example</title>
  <style>
    table {
```

```
            border-collapse: collapse;
            width: 50%;
            margin: 20px auto;
        }
        th, td {
            border: 1px solid #000;
            padding: 10px;
            text-align: center;
        }
    </style>
</head>
<body>
    <h1>Two-Dimensional Array in JavaScript</h1>
    <div id="output"></div>
    <script>
        const matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]
        ];
        const firstRowFirstElement = matrix[0][0];  // 1
        const secondRowThirdElement = matrix[1][2];  // 6
        matrix[2][1] = 100;  // Changing 8 to 100
        let outputHTML = `<p>First row, first element: ${firstRowFirstElement}</p>`;
        outputHTML += `<p>Second row, third element: ${secondRowThirdElement}</p>`;
        outputHTML += `<p>Modified matrix:</p>`;
        outputHTML += `<table>`;
        for (let i = 0; i < matrix.length; i++) {
            outputHTML += `<tr>`;
            for (let j = 0; j < matrix[i].length; j++) {
                outputHTML += `<td>${matrix[i][j]}</td>`;
            }
            outputHTML += `</tr>`;
        }
```

```
    outputHTML += `</table>`;
    document.getElementById('output').innerHTML = outputHTML;
  </script>
</body>
</html>
```

Explanation of the Code

1. **HTML Structure**:
   - The HTML contains a header and a with the ID output where the results will be displayed. A simple CSS style is included for better table visualization.
2. **JavaScript Code**:
   - **Declaration and Initialization**:

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

Here, we declare a two-dimensional array named matrix. Each inner array represents a row in the matrix.

   - **Accessing Elements**:

```
const firstRowFirstElement = matrix[0][0];  // 1
const secondRowThirdElement = matrix[1][2];  // 6
```

     - matrix[0][0] accesses the first element in the first row (1).
     - matrix[1][2] accesses the third element in the second row (6).
   - **Modifying an Element**:

```
matrix[2][1] = 100;  // Changing 8 to 100
```

This changes the value at the second row, first column from 8 to 100, resulting in the modified matrix:

```
[
  [1, 2, 3],
  [4, 5, 6],
  [7, 100, 9]
]
```

   - **Generating Output**:

```
let outputHTML = `<p>First row, first element: ${firstRowFirstElement}</p>`;
```

```
outputHTML += `<p>Second row, third element: ${secondRowThirdElement}</p>`;
outputHTML += `<p>Modified matrix:</p>`;
outputHTML += `<table>`;
for (let i = 0; i < matrix.length; i++) {
   outputHTML += `<tr>`;
   for (let j = 0; j < matrix[i].length; j++) {
      outputHTML += `<td>${matrix[i][j]}</td>`;
   }
   outputHTML += `</tr>`;
}
outputHTML += `</table>`;
```

We build a string of HTML to display the first and second accessed elements, and then create a table to visualize the matrix.

Two nested loops are used to iterate through each row and column of the matrix, generating table rows and cells.

3. **Displaying the Output**:

```
document.getElementById('output').innerHTML = outputHTML;
```

Finally, we insert the generated HTML string into the with ID output to display the results on the web page.

Output

When you open this HTML file in a web browser, you should see the following output:

Two-Dimensional Array in JavaScript

First row, first element: 1

Second row, third element: 6

Modified matrix:

[1, 2, 3]

[4, 5, 6]

[7, 100, 9]

The modified matrix will be displayed in a table format.

Difference between the One-dimensional array and two-dimensional array in Javascript

Feature	One-Dimensional Array	Two-Dimensional Array
Structure	A single list of elements.	An array of arrays, resembling a grid or table.

Dimensions	One dimension (1D).	Two dimensions (2D).
Syntax	const array = [1, 2, 3, 4];	const array = [[1, 2, 3], [4, 5, 6]];
Indexing	Accessed with a single index (e.g., array[0]).	Accessed with two indices (e.g., array[0][1]).
Example Representation	[10, 20, 30]	[[10, 20, 30], [40, 50, 60]]
Use Cases	Suitable for linear collections (e.g., lists).	Suitable for tabular data (e.g., matrices, grids).
Memory Layout	Stored as a single contiguous block of memory.	Consists of multiple arrays, which may be non-contiguous.
Traversal	Uses a single loop to access elements.	Uses nested loops to access elements.
Performance	Generally faster for single operations.	More complex, can be slower due to nested loops.
Data Representation	Represents a single sequence of data.	Represents multiple sequences or records of data.
Memory Layout	Stored as a single contiguous block of memory.	Consists of multiple arrays, which may be non-contiguous.

Common properties and methods of arrays in JavaScript

Property/Method	Description	Example	Explanation
length	Returns the number of elements in the array.	const arr = [1, 2, 3]; console.log(arr.length);	Outputs 3, as there are three elements in the array.
push()	Adds one or more elements to the end of the array and returns the new length.	arr.push(4); console.log(arr);	Modifies arr to [1, 2, 3, 4].
pop()	Removes the last element from the array and returns that element.	const last = arr.pop(); console.log(last);	last is 4, and arr becomes [1, 2, 3].
shift()	Removes the first element from the array and returns that element.	const first = arr.shift(); console.log(first);	first is 1, and arr becomes [2, 3].
unshift()	Adds one or more elements to the beginning of the array and returns the new length.	arr.unshift(0); console.log(arr);	Modifies arr to [0, 2, 3].
concat()	Combines two or more arrays	const newArr =	newArr becomes [0, 2,

	and returns a new array.	arr.concat([4, 5]);	3, 4, 5].
slice()	Returns a shallow copy of a portion of an array into a new array.	const subArr = arr.slice(1, 3);	subArr becomes [2, 3], original array remains unchanged.
indexOf()	Returns the first index at which a given element can be found in the array.	const index = arr.indexOf(3);	Returns 2, the index of element 3.
forEach()	Executes a provided function once for each array element.	arr.forEach(num => console.log(num));	Prints each number in the array to the console.
map()	Creates a new array populated with the results of calling a provided function on every element in the calling array.	const squared = arr.map(num => num * num);	squared becomes [0, 9801] after squaring each element.
filter()	Creates a new array with all elements that pass the test implemented by the provided function.	const filtered = arr.filter(num => num > 2);	filtered becomes [3] after filtering numbers greater than 2.
reduce()	Executes a reducer function on each element of the array, resulting in a single output value.	const sum = arr.reduce((acc, num) => acc + num, 0);	sum becomes 99, the total of the elements.

Jagged Arrays in JavaScript

A **jagged array** (or ragged array) is an array of arrays where the inner arrays can have different lengths. Unlike a two-dimensional array where each row has the same number of columns, a jagged array allows for more flexibility in the number of elements in each sub-array.

Jagged Arrays in JavaScript

A **jagged array** (or ragged array) is an array of arrays where the inner arrays can have different lengths. Unlike a two-dimensional array where each row has the same number of columns, a jagged array allows for more flexibility in the number of elements in each sub-array.

Example of a Jagged Array in JavaScript

Let’s create a simple example of a jagged array and demonstrate how to access its elements.

Code Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Jagged Array Example</title>
</head>
<body>
    <h1>Jagged Array in JavaScript</h1>
    <div id="output"></div>
    <script>
        // Declaration and initialization of a jagged array
        const jaggedArray = [
            [1, 2, 3],
            [4, 5],
            [6, 7, 8, 9]
        ];
        // Accessing elements
        const firstElement = jaggedArray[0][0];  // 1
        const secondElement = jaggedArray[1][1]; // 5
        const thirdElement = jaggedArray[2][2];  // 8
        // Modifying an element
        jaggedArray[1][0] = 100;  // Changing 4 to 100
        // Generating output
        let outputHTML = `<p>First element: ${firstElement}</p>`;
        outputHTML += `<p>Second element: ${secondElement}</p>`;
        outputHTML += `<p>Third element: ${thirdElement}</p>`;
        outputHTML += `<p>Modified jagged array:</p>`;
        outputHTML += `<ul>`;
        jaggedArray.forEach(innerArray => {
            outputHTML += `<li>[${innerArray.join(', ')}]</li>`;
```

```
    });
    outputHTML += `</ul>`;
    // Displaying the output on the web page
    document.getElementById('output').innerHTML = outputHTML;
  </script>
</body>
</html>
```

Output

When you open this HTML file in a web browser, you should see the following output:

Jagged Array in JavaScript
First element: 1
Second element: 5
Third element: 8
Modified jagged array:
- [1, 2, 3]
- [100, 5]
- [6, 7, 8, 9]

This example demonstrates how to work with a jagged array in JavaScript, covering declaration, accessing and modifying elements, and displaying the results. Jagged arrays are useful for representing data where the number of items can vary, such as rows of varying lengths in a table or matrix.

Jagged arrays in JavaScript are essentially arrays of arrays, so you can use most of the standard array methods on the outer array and the inner arrays. Here's a list of common methods you can use with jagged arrays, along with explanations and examples:

Common Methods for Jagged Arrays

Method	Description	Example	Explanation
push()	Adds an element to the end of the outer array.	jaggedArray.push([10, 11]);	Adds a new sub-array to the jagged array.
pop()	Removes the last element of the outer array.	const last = jaggedArray.pop();	Removes the last sub-array and returns it.
shift()	Removes the first element of the outer array.	const first = jaggedArray.shift();	Removes the first sub-array and

			returns it.
unshift()	Adds an element to the beginning of the outer array.	jaggedArray.unshift([0]);	Adds a new sub-array to the beginning of the jagged array.
forEach()	Executes a function on each sub-array.	jaggedArray.forEach(innerArray => console.log(innerArray));	Prints each sub-array to the console.
map()	Creates a new array by applying a function to each sub-array.	const lengths = jaggedArray.map(innerArray => innerArray.length);	Returns an array of lengths of each sub-array.
flat()	Flattens the jagged array into a one-dimensional array.	const flatArray = jaggedArray.flat();	Combines all elements from inner arrays into a single array.
filter()	Creates a new array with elements that pass a test.	const filtered = jaggedArray.filter(innerArray => innerArray.length > 2);	Returns sub-arrays with more than 2 elements.
slice()	Returns a shallow copy of a portion of the outer array.	const subArray = jaggedArray.slice(1);	Returns a new jagged array starting from the second sub-array.
splice()	Changes the contents of the outer array by removing or adding sub-arrays.	jaggedArray.splice(1, 1, [99]);	Removes the second sub-array and adds [99] in its place.
reduce()	Executes a reducer function on each element, resulting in a single output value.	const totalLength = jaggedArray.reduce((acc, innerArray) => acc + innerArray.length, 0);	Returns the total number of elements across all sub-arrays.

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Jagged Array Methods Example</title>
```

```
</head>
<body>
    <h1>Jagged Array Methods</h1>
    <div id="output"></div>
    <script>
        // Declaration of a jagged array
        const jaggedArray = [
            [1, 2, 3],
            [4, 5],
            [6, 7, 8, 9]
        ];
        // Using push() to add a new sub-array
        jaggedArray.push([10, 11]);

        // Using pop() to remove the last sub-array
        const lastSubArray = jaggedArray.pop();
        // Using forEach() to print each sub-array
        let outputHTML = "<h2>Sub-arrays:</h2><ul>";
        jaggedArray.forEach(innerArray => {
            outputHTML += `<li>[${innerArray.join(', ')}]</li>`;
        });
        outputHTML += `</ul>`;
        // Using map() to get the lengths of each sub-array
        const lengths = jaggedArray.map(innerArray => innerArray.length);
        outputHTML += `<p>Lengths of each sub-array: [${lengths.join(', ')}]</p>`;
        // Displaying the output on the web page
        document.getElementById('output').innerHTML = outputHTML;
    </script>
</body>
</html>
```

Uses of Arrays in JavaScript

1. **Data Storage**: Arrays are used to store collections of related data items in a single variable, making it easier to manage and manipulate data.
2. **Indexed Access**: JavaScript arrays allow for quick access to elements using zero-based

indices, enabling efficient data retrieval.

3. **Iteration**: Arrays can be easily iterated over using loops or array methods like forEach(), allowing for operations like searching, sorting, and filtering.
4. **Dynamic Sizing**: Unlike fixed-size arrays in some programming languages, JavaScript arrays can grow or shrink in size dynamically, making them flexible for varying data needs.

Uses of One-Dimensional Arrays

- **Storing Lists**: Useful for storing lists of items, such as numbers, strings, or objects.
    - Example: let fruits = ["apple", "banana", "cherry"];
- **Managing Simple Data**: Suitable for tasks like calculating sums, averages, or filtering elements based on conditions.
- **Function Arguments**: Can be used to group related parameters when passing them to functions.
- **Tracking State**: Commonly used to track states or properties in applications, such as user inputs or selections.

Uses Two-Dimensional Arrays

- **Matrices and Grids**: Frequently used to represent mathematical matrices or grid-like structures, such as game boards.
    - Example: let matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
- **Image Processing**: Used to represent pixel data in images, where each element corresponds to a pixel's color value.
- **Tabular Data**: Suitable for representing data in a structured format, like a table, for use in applications that require data manipulation or display.
- **Game Development**: Useful for representing maps, levels, or spatial data where multiple attributes are needed.

Uses Jagged Arrays

- **Variable-Length Rows**: Ideal for scenarios where sub-arrays need to have different lengths, such as storing lists of items that vary in size.
    - Example: let jaggedArray = [[1, 2], [3, 4, 5], [6]];
- **Sparse Data Representation**: Efficient for storing data where many elements may be empty, saving memory compared to a fixed-size two-dimensional array.
- **Complex Data Structures**: Useful for representing more complex data models, like trees or nested collections, where each level may have a different number of children or elements.

- **Dynamic User Inputs**: Commonly used in applications where user-generated data can vary, such as inputting grades for students across different subjects.

JavaScript Functions

A **JavaScript function** is a reusable block of code designed to perform a specific task. Functions are essential in programming, allowing you to organize code, avoid repetition, and improve readability.

Function Definition

A function is defined using the function keyword, followed by a name, parentheses (), and a block of code within curly brackets {}. The function name can include letters, digits, underscores, and dollar signs—similar to variable naming rules.

Syntax:

```
function functionName(parameter1, parameter2) {
  // Code to be executed
}
```

Parameters and Arguments

- **Parameters**: Variables listed inside the parentheses during function definition. They act as placeholders for values.
- **Arguments**: The actual values passed to the function when it is invoked.

Inside the function, these parameters behave as local variables, scoped only to that function.

Function Invocation

A function is executed (invoked) when called from various contexts:

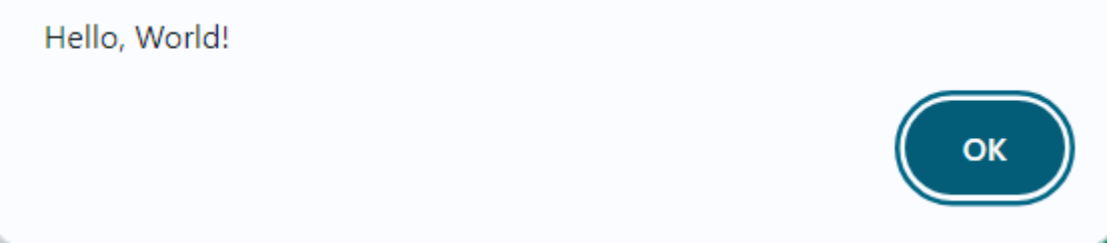
- **Event-driven**: Such as clicking a button.
- **Direct invocation**: Calling the function in JavaScript code.
- **Self-invocation**: Automatically executed without being called.

Example of invoking a function:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Functions</h1>
<p id="demo"></p>
<script>
function greet() {
  alert("Hello, World!");
```

```
}
greet(); // Calls the function and displays an alert
</script>
</body>
</html>
```

Output:



Function Return

When a function encounters a return statement, it stops executing and sends a value back to the caller. This return value can be captured in a variable for further use.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Functions</h1>

<p>Call a function which performs a calculation and returns the result:</p>

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

<script>
let x = myFunction(4, 3);
document.getElementById("demo").innerHTML = x;

function myFunction(a, b) {
  return a * b;
}
</script>
</body>
```

```
</html>
```

Output

JavaScript Functions

Call a function which performs a calculation and returns the result:

12

Why Use Functions?

Functions promote code reuse, allowing you to execute the same code with different arguments, producing varied results without rewriting the logic.

The () Operator

Using parentheses () invokes the function:

```
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Functions</h1>

<p>Invoke (call) a function that converts from Fahrenheit to Celsius:</p>
<p id="demo"></p>

<script>
function toCelsius(f) {
  return (5/9) * (f-32);
}

let value = toCelsius(77);
document.getElementById("demo").innerHTML = value;
</script>

</body>
</html>
```

Output:

JavaScript Functions

Invoke (call) a function that converts from Fahrenheit to Celsius:

25

- Accessing a function with incorrect parameters can return an incorrect answer:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Functions</h1>
<p>Accessing a function without () returns the function and not the function result:</p>
<p id="demo"></p>
<script>
function toCelsius(f) {
  return (5/9) * (f-32);
}
let value = toCelsius;
document.getElementById("demo").innerHTML = value;
</script>
</body>
      </html>
```

Output:

JavaScript Functions

Accessing a function without () returns the function and not the function result:

function toCelsius(f) { return (5/9) * (f-32); }

- Accessing a function without () returns the function and not the function result:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Functions</h1>
<p>Accessing a function without () returns the function and not the function result:</p>
<p id="demo"></p>
<script>
function toCelsius(f) {
  return (5/9) * (f-32);
}
let value = toCelsius;
document.getElementById("demo").innerHTML = value;
</script>
```

```
</body>
</html>
```

Output:

JavaScript Functions

Accessing a function without () returns the function and not the function result:

function toCelsius(f) { return (5/9) * (f-32); }

Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Functions</h1>
<p>Using a function as a variable:</p>
<p id="demo"></p>
<script>
let text = "The temperature is " + toCelsius(77) + " Celsius.";
document.getElementById("demo").innerHTML = text;
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
</script>
</body>
</html>
```

Output;

JavaScript Functions

Using a function as a variable:

The temperature is 25 Celsius.

Local Variables

Variables declared within a function are **local** to that function and cannot be accessed from outside. This scoping allows for variable name reuse in different functions without conflict. Local variables are created when the function starts and deleted once it completes.

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Functions</h1>
<p>Outside myFunction() carName is undefined.</p>
<p id="demo1"></p>
<p id="demo2"></p>
<script>
let text = "Outside: " + typeof carName;
document.getElementById("demo1").innerHTML = text;
function myFunction() {
  let carName = "Volvo";
  let text = "Inside: " + typeof carName + " " + carName;
  document.getElementById("demo2").innerHTML = text;
}
myFunction();
</script>
</body>
</html>
```

Output:

JavaScript Functions

Outside myFunction() carName is undefined.

Outside: undefined

Inside: string Volvo

Uses of Functions in JavaScript:

1. **Code Reusability**: Allows defining a block of code once and using it multiple times.
2. **Modularity**: Breaks down complex problems into smaller, manageable pieces.
3. **Abstraction**: Hides implementation details from users, enabling easier use.
4. **Event Handling**: Responds to user interactions like clicks and form submissions.
5. **Higher-Order Functions**: Accepts functions as arguments or returns them for flexible programming.
6. **Recursion**: Calls itself to solve problems that can be divided into smaller subproblems.
7. **Dynamic Behavior**: Can be assigned to variables, passed around, and returned, enabling versatile patterns.

Anonymous Functions in JavaScript

What are Anonymous Functions?

Anonymous functions are functions that do not have a name. Unlike named functions, which are declared with a specific identifier for easy reference, anonymous functions are primarily used in contexts where the function is not meant to be reused or referenced elsewhere. They are often utilized for specific tasks and can be assigned to variables, passed as arguments to other functions, or executed immediately.

Characteristics of Anonymous Functions:

- **No Name**: As the name suggests, they lack an identifier, making them anonymous.
- **Temporary Use**: They are generally used for short-lived operations, such as callbacks or as arguments in higher-order functions.
- **First-Class Citizens**: Functions in JavaScript can be treated as values, so anonymous functions can be stored in variables or passed around.

Syntax

You can define an anonymous function using traditional function declaration syntax or the more modern arrow function syntax.

Traditional Syntax:

```
function() {
   // Function Body
}
```

Examples of Anonymous Functions

Example 1: Basic Anonymous Function

In this example, we define an anonymous function that prints a message to the console. The function is assigned to the variable `greet`, which can then be called.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Anonymous Function Example</title>
</head>
<body>
    <script>
        var greet = function () {
            console.log("Welcome to GeeksforGeeks!");
        };
        greet(); // Calling the anonymous function
    </script>
</body>
</html>
```

Output:

Welcome to GeeksforGeeks!

Explanation: In this example, the anonymous function is defined and assigned to the variable `greet`. When we invoke `greet()`, it executes the function and prints the message.

Example 2: Anonymous Function with Arguments

This example demonstrates how to pass arguments to an anonymous function.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Anonymous Function with Arguments</title>
</head>
<body>
    <script>
        var greet = function (platform) {
            console.log("Welcome to " + platform);
        };
```

```
        greet("GeeksforGeeks!"); // Passing an argument to the anonymous function
    </script>
</body>
</html>
```

Output:

Welcome to GeeksforGeeks!

Explanation: Here, the anonymous function takes a parameter named `platform` and prints a welcome message. By calling `greet("GeeksforGeeks!")`, the argument is passed and included in the output.

Example 3: Anonymous Function as a Callback

In this example, an anonymous function is passed as a callback to the `setTimeout()` method, which executes the function after a specified delay.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Anonymous Function as Callback</title>
</head>
<body>
    <script>
        setTimeout(function () {
            console.log("Welcome to GeeksforGeeks!");
        }, 2000); // Executes after 2 seconds
    </script>
</body>
</html>
```

Output (after 2 seconds):

Welcome to GeeksforGeeks!

Explanation: In this case, the anonymous function is defined inline and passed to `setTimeout()`. It executes after a delay of 2000 milliseconds, demonstrating the use of anonymous functions for delayed execution.

Self-Executing Anonymous Functions

Another common application of anonymous functions is in creating self-executing functions, known as Immediately Invoked Function Expressions (IIFE).

Example 4: Self-Executing Function

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Self-Executing Function</title>
</head>
<body>
   <script>
      (function () {
         console.log("Welcome to GeeksforGeeks!");
      })(); // Self-executing function
   </script>
</body>
</html>
```

Output:

Welcome to GeeksforGeeks!

Explanation: This example demonstrates an IIFE, which executes immediately upon definition. This is useful for encapsulating code and preventing polluting the global scope.

Arrow Functions in JavaScript

What are Arrow Functions?

Arrow functions, introduced in ES6 (ECMAScript 2015), provide a more concise syntax for writing function expressions. They offer a shorter way to define functions and also have different behaviour regarding the 'this' keyword compared to traditional function expressions.

Key Features of Arrow Functions:

- **Concise Syntax**: Arrow functions reduce the code needed for writing functions, making them easier to read and write.
- **Lexical this**: Unlike regular functions, arrow functions do not bind their own this. Instead, they inherit this from the enclosing lexical context, which can be particularly useful in certain scenarios, such as when working with callbacks or methods.

Syntax

The basic syntax of an arrow function is:

```
(parameters) => {
```

```
    // Function Body
}
```

If the function has only one parameter, you can omit the parentheses:

```
parameter => {
    // Function Body
}
```

If the function body consists of a single expression, you can omit the curly braces and the return keyword:

```
(parameter) => expression
```

Examples of Arrow Functions

Example 1: Basic Arrow Function In this example, we define a simple arrow function that prints a message to the console.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Function Example</title>
</head>
<body>
    <script>
        const greet = () => {
            console.log("Welcome to GeeksforGeeks!");
        };

        greet(); // Calling the arrow function
    </script>
</body>
</html>
```

Output:

```
Welcome to GeeksforGeeks!
```

Explanation: The arrow function greet is defined and invoked, executing the function and printing the welcome message to the console.

Example 2: Arrow Function with Parameters This example demonstrates how to define an arrow function that takes parameters.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Function with Parameters</title>
</head>
<body>
    <script>
        const greet = (platform) => {
            console.log("Welcome to " + platform);
        };
        greet("GeeksforGeeks!"); // Passing an argument to the arrow function
    </script>
</body>
</html>
```

Output:

Welcome to GeeksforGeeks!

Explanation: The arrow function takes a parameter platform and prints a welcome message. By calling greet("GeeksforGeeks!"), the argument is passed and included in the output.

Example 3: Arrow Function as a Callback In this example, we pass an arrow function as a callback to the setTimeout() method, which executes it after a specified delay.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Function as Callback</title>
</head>
<body>
    <script>
        setTimeout(() => {
            console.log("Welcome to GeeksforGeeks!");
        }, 2000); // Executes after 2 seconds
    </script>
</body>
```

```
</html>
```

Output (after 2 seconds):

Welcome to GeeksforGeeks!

Explanation: The arrow function is defined inline and passed to setTimeout(), executing after a delay of 2000 milliseconds. This showcases the flexibility of arrow functions in callback scenarios.

Example 4: Self-Executing Arrow Function Here's an example of a self-executing arrow function (IIFE).

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Self-Executing Arrow Function</title>
</head>
<body>
    <script>
        (() => {
            console.log("Welcome to GeeksforGeeks!");
        })(); // Self-executing arrow function
    </script>
</body>
</html>
```

Output:

Welcome to GeeksforGeeks!

Explanation: Similar to the IIFE with anonymous functions, this arrow function executes immediately upon definition, demonstrating how to encapsulate code.

Example 5: Arrow Function with Single Expression In this example, we create an arrow function that returns a value without using curly braces.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Function Single Expression</title>
</head>
```

```
<body>
  <script>
    const square = x => x * x; // Single expression, implicit return
    console.log(square(5)); // Calling the arrow function
  </script>
</body>
</html>
```

Output:

25

Explanation: The square function calculates the square of a number. Since it's a single expression, the result is returned implicitly. Calling square(5) outputs 25.

Recursive Functions in JavaScript

A **recursive function** is a function that calls itself to solve smaller instances of the same problem until a base condition is met, which stops the recursion. This technique is commonly used to simplify complex problems by breaking them down into more manageable sub-problems.

Syntax

```
function functionName(parameters) {
  // Base case: condition to stop recursion
  if (condition) {
    // return a value or perform an action
  } else {
    // Recursive case: call the function itself
    return functionName(modifiedParameters);
  }
}
```

Example 1: Countdown Function

This example demonstrates a countdown from a specified number down to 1.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Countdown Function</title>
```

```
<script>
    function countDown(fromNumber) {
        // Base case: stop when the number is 1 or less
        if (fromNumber <= 0) {
            return; // Stop recursion
        } else {
            console.log(fromNumber); // Print the current number
            countDown(fromNumber - 1); // Recursive call
        }
    }

    // Start countdown when the window loads
    window.onload = function() {
        countDown(3);
    };
</script>
</head>
<body>
    <h1>Countdown Function Example</h1>
    <p>Open the console to see the countdown from 3 to 1.</p>
</body>
</html>
```

Explanation

- **Base Case**: The function checks if fromNumber is less than or equal to 0. If true, it returns, stopping the recursion.
- **Recursive Case**: If fromNumber is greater than 0, it prints the number and calls itself with fromNumber - 1.

When the page loads, the console will display:

3

2

1

Example 2: Sum of Natural Numbers

This example calculates the sum of natural numbers from 1 to n.

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Sum of Natural Numbers</title>
   <script>
      function sum(n) {
         // Base case
         if (n <= 1) {
            return n; // Return n when it's 1 or less
         }
         // Recursive case
         return n + sum(n - 1); // Recursive call
      }
      // Display the sum when the window loads
      window.onload = function() {
         const result = sum(3);
         console.log(result); // Output: 6
         document.getElementById("result").innerText = "Sum of numbers from 1 to 3
is: " + result;
      };
   </script>
</head>
<body>
   <h1>Sum of Natural Numbers Example</h1>
   <p id="result"></p>
</body>
</html>
```

Explanation

- **Base Case**: The function checks if n is less than or equal to 1. If so, it returns n.
- **Recursive Case**: If n is greater than 1, it returns n plus the sum of the previous number (n - 1).

When the page loads, the output displayed will be:

```
Sum of numbers from 1 to 3 is: 6
```

How Recursion Works

1. **Function Call Stack**: Each time a recursive function calls itself, a new instance of that function is placed on the call stack.
2. **Base Case**: The recursion continues until the base case is met, at which point the function returns a value.
3. **Unwinding the Stack**: Once the base case is reached, the call stack starts unwinding, resolving each function call until the final result is returned.

Default Parameters in JavaScript

Default parameters allow named parameters to be initialized with default values if no value or undefined is passed during the function call. This feature was introduced in ES6 and enhances function flexibility by providing a way to set initial values without requiring extra logic inside the function.

Syntax

```
function functionName(parameter1 = defaultValue1, parameter2 = defaultValue2) {
   // Function body
}
```

Example 1: Default Parameters

This example demonstrates a function that greets a user, with a default name set to "Guest" if no name is provided.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Default Parameters Example</title>
   <script>
      function greet(name = "Guest") {
         console.log("Hello, " + name + "!");
      }
      // Display greetings when the window loads
      window.onload = function() {
         greet(); // Uses default parameter
          greet("Alice"); // Uses provided parameter
      };
```

```
    </script>
</head>
<body>
    <h1>Default Parameters Example</h1>
    <p>Open the console to see the greeting messages.</p>
</body>
</html>
```

Explanation

- **Default Value**: The greet function has a parameter name with a default value of "Guest".
- **Function Call**: When greet() is called without arguments, it uses the default value

Outputs:

```
Hello, Guest!
```

When called with greet("Alice"), it outputs:

```
Hello, Alice!
```

Example 2: Multiple Default Parameters

This example shows a function that calculates the area of a rectangle, using default values for length and width.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Area Calculation Example</title>
    <script>
        function calculateArea(length = 5, width = 10) {
            return length * width;
        }
        // Display area calculations when the window loads
        window.onload = function() {
            console.log("Area with default parameters: " + calculateArea()); // 50
            console.log("Area with provided parameters: " + calculateArea(7, 3)); // 21
        };
    </script>
```

```
</head>
<body>
    <h1>Area Calculation Example</h1>
    <p>Open the console to see the area calculations.</p>
</body>
</html>
```

Explanation

- **Default Values**: The calculateArea function defines length and width with default values of 5 and 10, respectively.
- **Function Call**: Calling calculateArea() with no arguments returns:

Outputs:

```
Area with default parameters: 50
Calling calculateArea(7, 3) returns:
Area with provided parameters: 21
```

Benefits of Default Parameters

1. **Simplifies Code**: Reduces the need for checks and conditions within the function body.
2. **Enhances Readability**: Makes it clear what the expected default behavior is.
3. **Flexible Function Calls**: Allows functions to be called with varying numbers of arguments without errors.