1. [JavaScript Basics](https://www.geeksforgeeks.org/javascript/" \l "javascript-basics)

9. [JavaScript First Program Hello world](https://www.geeksforgeeks.org/javascript-course-printing-hello-world-in-javascript/)

Involves displaying the phrase “Hello World” using simple methods like console.log(), document.write(), or alert(), introducing beginners to fundamental JavaScript syntax and output techniques.

**Using console.log() method**

The [console.log() method](https://www.geeksforgeeks.org/javascript-console-log-method/) prints the message to the browser console. This approach is mainly used for debugging purposes and checking outputs while developing code.

**Example:** In this example, we will print the legendary “Hello World” in the window console.

// Using console.log

console.log('Hello World');

**Output**

Hello World

**Note** To see the output in browser open file .html file -> right click in webpage -> inspect element or F12 -> go to console tab – here you will find the output “Hello World”.

**Using document.write() Method**

Using the [document.write() method](https://www.geeksforgeeks.org/html-dom-write-method/" \t "_blank) in JavaScript allows you to display “Hello World” directly on the webpage. This method is straightforward and places the text within the HTML document. However, it’s generally only suitable for simple demos or initial page load.

**Example:** In this example, we will print the “Hello World” in the HTML document.

// Using document.write

document.write('Hello World');

**Output:**

Hello world

**Using alert() Method**

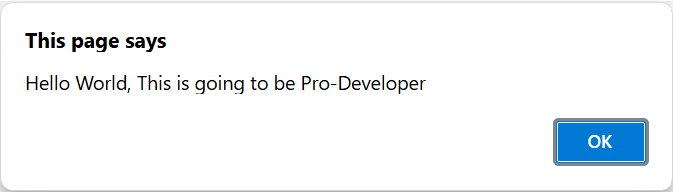
The [alert() method](https://www.geeksforgeeks.org/html-dom-window-alert-method/) displays a popup alert box with the message. It’s useful for simple notifications or warnings but should be used sparingly as it disrupts user interaction.

**Example: I**n this example, we will print the “Hello World” on the browser window with some message or warning.

// Using alert

alert("Hello World, This is going to be Pro-Developer");

**Output:**



Each of the above methods has different ways of outputting the content. Though ‘document.write()’ is used when we want to print the content onto the document which is the HTML Document. Also ‘console.log()’ is mainly used when we are debugging JavaScript code and the ‘alert()’ is used to show an alert box on the browser window with some message or warning.

**10. JavaScript Data Types:**

JavaScript supports multiple data types. JavaScript data types are broadly categorized into **primitive and non-primitive types. The primitive data types include Number, String, Boolean, Null, Undefined, and Symbol. Non-primitive types include Object, Array, and Function.**

The latest ECMAScript standard defines eight data types Out of which seven data types are **Primitive(predefined)** and one **complex or Non-Primitive**.

**Primitive Data Types**

The predefined data types provided by JavaScript language are known as primitive data types. Primitive data types are also known as in-built data types.

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Number** | JavaScript numbers are always stored in double-precision 64-bit binary format IEEE 754. |
| **String** | JavaScript Strings are made up of a list of characters, essentially an array of characters. |
| **Boolean** | Represents a logical entity and can have two values: true or false. |
| **Null** | This type has only one value: null. |
| **Undefined** | A variable that has not been assigned a value is undefined. |
| **Symbol** | Symbols return unique identifiers that can be used as property keys in objects without colliding with other keys. |
| **BigInt** | BigInt is a built-in object providing a way to represent whole numbers larger than 253-1. |

This table summarizes the basic data types in JavaScript along with their descriptions.

**Non-Primitive Data Types:**

The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.

* [**Object:**](https://www.geeksforgeeks.org/objects-in-javascript) It is the most important data type and forms the building blocks for modern JavaScript.

**JavaScript Primitive Data Types Examples:**

**Number:**

The number type in JavaScript contains both integer and floating-point numbers. Besides these numbers, we also have some ‘special-numbers’ in javascript that are: **‘Infinity’, ‘-Infinity’, and ‘NaN’. Infinity basically represents the mathematical ‘?’. The ‘NaN’ denotes a computational error.**

let num = 2; **// Integer**

let num2 = 1.3; **// Floating point number**

let num3 = Infinity; **// Infinity**

let num4 = 'something here too'/2; **// NaN**

**String:**

**A String in JavaScript is basically a series of characters that are surrounded by quotes.** There are three types of quotes in JavaScript, which are:

let str = **"Hello There";**

let str2 = **'Single quotes works fine';**

let phrase = **`can embed ${str}`;**

There’s no difference between ‘single’ and “double” quotes in JavaScript.

Backticks provide extra functionality as with their help of them we can embed variables inside them.

let name = **"Mukul";**

// embed a variable

alert( **`Hello, ${name}!`** ); // Hello, Mukul!

**Boolean:**

**The Boolean type has only two values: true and false.** This data type is used to store yes/no type of values: True means “Yes, Correct”, and False means “No, Incorrect”.

let isCoding = True; // yes

let isOld = False; // no

**NULL:**

The special NULL value does not belong to any of the default data types. It forms a separate type of its own which contains only the null value:

let age = null;

The ‘null’ data type basically defines a special value that represents ‘nothing’, ’empty’, or ‘value unknown’.

**Undefined** Just like null, Undefined makes its own type. The meaning of undefined is ‘value is not assigned’.

let x;

console.log(x); // undefined

|  |  |
| --- | --- |
| **NULL** | **UNDEFINED** |
| Null is an assignment value, meaning that a variable has been declared and given the value of null. | Undefined means a variable has been declared but has not yet been assigned a value. |
| **let** y = null;  console.**log**(y); *// logs 'null'* | **let** x;  console.**log**(x); *// logs 'undefined'* |
| console.log(typeof a); *//****logs 'object'*** | console.log(typeof z);*//****logs 'undefined'*** |
| Explicitly we need to assign to null. | Automatically assigned to undefined. |
| null values are preserved during JSON serialization (e.g., {“key”: null} ). | undefined values are omitted during serialization. |

**Symbol:**

**Symbols are new primitive built-in object types introduced as part of ES6.** Symbols return unique identifiers that can be used to add unique property keys to an object that won’t collide with keys of any other code that might add to the object. They are used as object properties that cannot be recreated. It basically helps us to enable encapsulation or information hiding.

let symbol1 = Symbol("Geeks")

let symbol2 = Symbol("Geeks")

// Each time Symbol() method

// is used to create new global Symbol

console.log(symbol1 == symbol2); // False

**BigInt:**

BigInt is a built-in object in JavaScript that provides a way to represent whole numbers larger than 253-1. The largest number that JavaScript can reliably represent with the Number primitive is 253-1, which is represented by the MAX\_SAFE\_INTEGER constant.

let bigBin = BigInt("0b1010101001010101001111111111111111");

// 11430854655n

console.log(bigBin);

**JavaScript Non-Primitive Data Types Examples:**

**Object:**

JavaScript objects are fundamental data structures used to store collections of data. They consist of key-value pairs and can be created using curly braces {} or the new keyword. Understanding objects is crucial, as everything in JavaScript is essentially an object.

**Object creation:**

**Using the “object constructor” syntax:**

let person = new Object();

**Using the “object literal” syntax:**

let person = {}; //

Both these methods are correct, though it’s totally your call what to choose. We can also put properties inside an Object.

**11. JavaScript Variables**

**Variables**are used to store data in JavaScript. Variables are used to store reusable values. The values of the variables are allocated using the assignment operator(“=”).

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

y = "Hello"

**JavaScript Identifiers**

JavaScript variables must have unique names. These names are called Identifiers.

**Basic rules to declare a variable in JavaScript:**

* These are case-sensitive
* Can only begin with a letter, underscore(“\_”) or “$” symbol
* It can contain letters, numbers, underscore, or “$” symbol
* A variable name cannot be a reserved keyword.

**Operators in JavaScript with Example:**

**1. Arithmetic Operators**

* **Addition (+)**:

let a = 5;

let b = 10;

let sum = a + b; // 15

* **Subtraction (-)**:

let diff = b - a; // 5

* **Multiplication (\*)**:

let product = a \* b; // 50

* **Division (/)**:

let quotient = b / a; // 2

* **Modulus (%)**:

let remainder = b % a; // 0

* **Increment (++)**:

let x = 5;

x++; // x becomes 6

* **Decrement (--)**:

x--; // x becomes 5 again

* **Exponentiation (\*\*)**:

let power = 2 \*\* 3; // 8

**2. Assignment Operators**

* **Assign (=)**:

let c = 5;

* **Add and assign (+=)**:

c += 5; // c becomes 10

* **Subtract and assign (-=)**:

c -= 2; // c becomes 8

* **Multiply and assign (\*=)**:

c \*= 2; // c becomes 16

* **Divide and assign (/=)**:

c /= 4; // c becomes 4

* **Modulus and assign (%=)**:

c %= 3; // c becomes 1

* **Exponentiation and assign (\*\*=)**:

c \*\*= 3; // c becomes 1 (because 1 raised to any power is still 1)

**3. Comparison Operators**

* **Equal to (==)**:

let isEqual = (5 == "5"); // true (type coercion happens)

* **Strict equal to (===)**:

let isStrictEqual = (5 === "5"); // false (no type coercion)

* **Not equal to (!=)**:

let isNotEqual = (5 != "5"); // false

* **Strict not equal to (!==)**:

let isStrictNotEqual = (5 !== "5"); // true

* **Greater than (>)**:

let isGreater = (10 > 5); // true

* **Less than (<)**:

let isLess = (10 < 5); // false

* **Greater than or equal to (>=)**:

let isGreaterOrEqual = (10 >= 10); // true

* **Less than or equal to (<=)**:

let isLessOrEqual = (10 <= 5); // false

**4. Logical Operators**

* **Logical AND (&&)**:

let andResult = (true && false); // false

* **Logical OR (||)**:

let orResult = (true || false); // true

* **Logical NOT (!)**:

let notResult = !true; // false

**5. Bitwise Operators**

* **Bitwise AND (&)**:

let andBitwise = 5 & 1; // 1 (binary: 0101 & 0001 = 0001)

* **Bitwise OR (|)**:

let orBitwise = 5 | 1; // 5 (binary: 0101 | 0001 = 0101)

* **Bitwise XOR (^)**:

let xorBitwise = 5 ^ 1; // 4 (binary: 0101 ^ 0001 = 0100)

* **Bitwise NOT (~)**:

let notBitwise = ~5; // -6 (binary: ~0101 = 1010, which is -6 in two's complement)

* **Left shift (<<)**:

let leftShift = 5 << 1; // 10 (binary: 0101 << 1 = 1010)

* **Right shift (>>)**:

let rightShift = 5 >> 1; // 2 (binary: 0101 >> 1 = 0010)

* **Unsigned right shift (>>>)**:

let unsignedRightShift = -5 >>> 1; // 2147483645 (shifts the bits to the right and fills with zeros)

**6. String Operators**

* **Concatenation (+)**:

let str1 = "Hello";

let str2 = "World";

let greeting = str1 + " " + str2; // "Hello World"

**7. Ternary Operator**

* **Ternary (? :)**:

let age = 18;

let canVote = (age >= 18) ? "Yes" : "No"; // "Yes"

**8. Type Operators**

* **typeof**:

let type = typeof 123; // "number"

* **instanceof**:

let date = new Date();

let isDate = date instanceof Date; // true

**9. Unary Operators**

* **Unary Plus (+)**:

let num = +"123"; // Converts the string "123" to the number 123

* **Unary Negation (-)**:

let neg = -123; // -123

* **Logical NOT (!)**:

let isFalse = !true; // false

* **Bitwise NOT (~)**:

let complement = ~5; // -6

* **delete**:

let obj = {name: "Alice"};

delete obj.name; // Deletes the "name" property from obj

* **void**:

void 0; // Returns undefined

**10. Relational Operators**

* **in**:

let obj = {name: "Alice"};

let hasName = "name" in obj; // true

* **instanceof**:

let arr = [];

let isArray = arr instanceof Array; // true

**11. Comma Operator**

* **Comma (,)**:

let result = (5, 10, 15); // 15 (returns the last value)

These examples demonstrate the various operators in JavaScript and how they can be used in code.

**12. Nullish Coalescing Assignment (??=)**

This operator is represented by **x ??= y**and it is called Logical nullish assignment operator. Only if the value of **x** is **nullish** then the value of y will be assigned to **x**thatmeansif the value of x is **null**or**undefined**then the value of y will be assigned to x.

logical nullish assignment is represented as **x ??= y**, this is derived by two operators nullish coalescing operator and assignment operator we can also write it as **x ?? (x = y)**. Now javascript checks the **x**first, if it is **nullish** then the value of **y** will be assigned to **x**.

**Syntax :**

x ??= y // Means : x ?? (x = y)

**Example 1 :**

let x = 12;

let y = null;

let z = 13;

// The value of x will become

// unchanged because x is not nullish.

x ??= z;

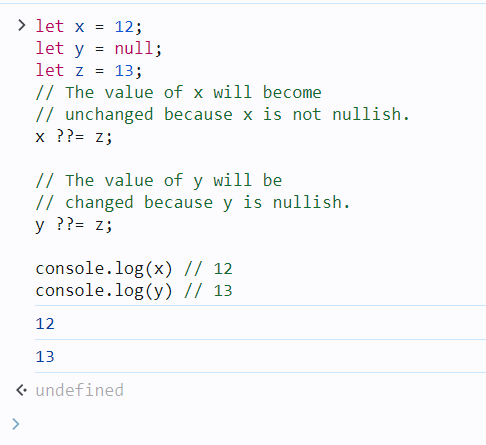
// The value of y will be

// changed because y is nullish.

y ??= z;

console.log(x) // 12

console.log(y) // 13



**Example 2:**

let x = {

name : "Ram"

}

// The value of name will remain

// unchanged because x.name is not nullish

x.name ??= "Shyam";

// There is no any property named age in object x .

// So the value of x.age will be

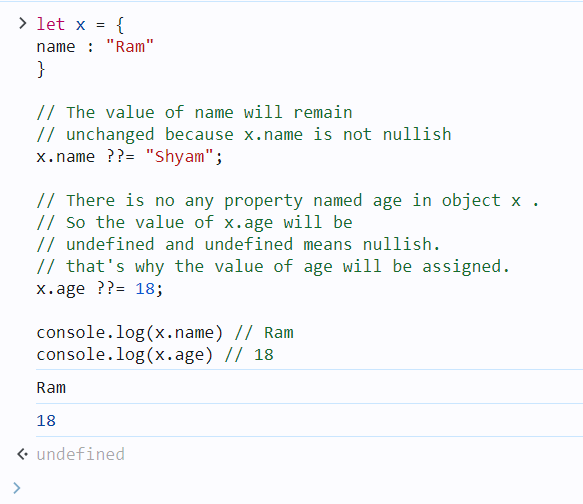
// undefined and undefined means nullish.

// that's why the value of age will be assigned.

x.age ??= 18;

console.log(x.name) // Ram

console.log(x.age) // 18

****

**Example 3:**

<h1>Hello Coalescing</h1>

<p id="print\_arr"></p>

<script>

    let arr = [1, 2, "apple", null, undefined, []]

    // Replace each nullish values with "RAM"

    arr.forEach((item, index)=>{

    arr[index] ??= "RAM"

        })

        document.getElementById("print\_arr")

    .innerText = arr.toString();

        //console.log(arr)

</script>



**Variable Declaration:**

**Var :**

The var keyword in JavaScript is used to declare a variable. It was the primary way to declare variables before the introduction of let and const in ES6. Variables declared with var have some unique characteristics that distinguish them from let and const.

**Characteristics of var:**

1. **Function Scope**: var is function-scoped, meaning that if a variable is declared inside a function, it is only accessible within that function. However, it is not block-scoped (i.e., var ignores block scopes such as loops and if statements).
2. **Hoisting**: Variables declared with var are hoisted to the top of their scope. This means the declaration is moved to the top of its scope, but the initialization stays in place.
3. **Redeclaration**: You can redeclare a variable with var without causing an error.
4. **Global Object Property**: In the global scope, var declarations create properties on the global object (e.g., window in browsers).

**Syntax:**

var variableName = value;

**Examples:**

**1. Basic Declaration and Assignment:**

var x = 10; // Declare and initialize a variable

console.log(x); // 10

**2. Function Scope:**

function testVar() {

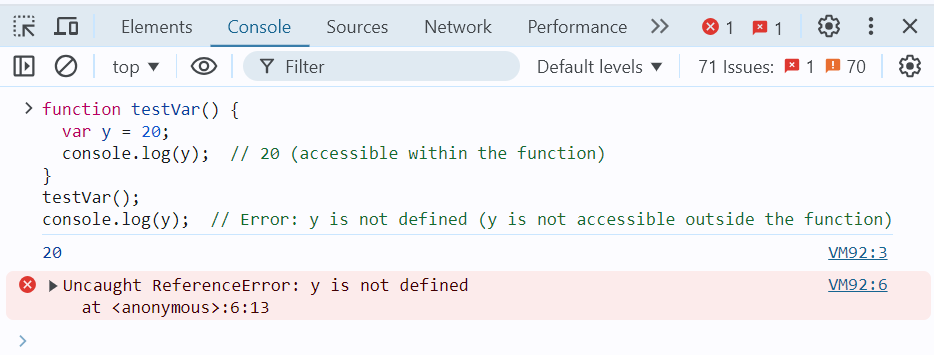
var y = 20;

console.log(y); // 20 (accessible within the function)

}

testVar();

console.log(y); // Error: y is not defined (y is not accessible outside the function)



**3. Hoisting:**

console.log(a); // undefined (the declaration is hoisted, but not the assignment)

var a = 5;

console.log(a); // 5

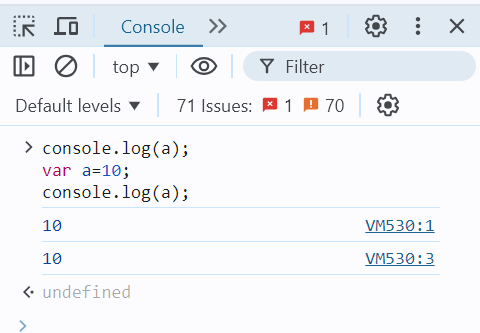
This code behaves as though it was written like this:

var a;

console.log(a); // undefined

a = 5;

console.log(a); // 5



**4. Ignoring Block Scope:**

if (true) {

var z = 30;

}

console.log(z); // 30 (accessible outside the block, because var is not block-scoped)

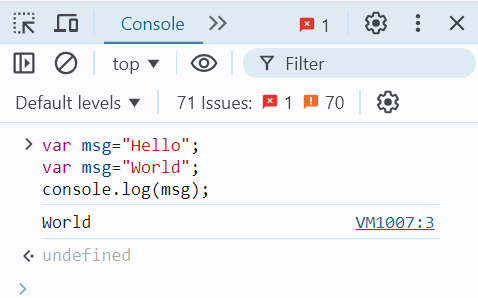


**5. Redeclaration:**

var message = "Hello";

var message = "World";

console.log(message); // "World" (no error on redeclaration)



**Let:**

The let keyword in JavaScript was introduced in ES6 (ECMAScript 2015) and is used to declare variables. Unlike var, let provides block-scoping, which makes it a more predictable and safer way to declare variables in modern JavaScript.

**Characteristics of let:**

1. **Block Scope**: Variables declared with let are confined to the block in which they are defined. A block is typically defined by {} (e.g., within an if, for, or function).
2. **No Hoisting with Initialization**: Although let variables are hoisted to the top of their block, they are not initialized until their declaration is encountered in the code. This leads to a **"Temporal Dead Zone (TDZ)"** where accessing the variable before its declaration results in an error.
3. **No Redeclaration**: Variables declared with let cannot be redeclared within the same scope. This helps prevent accidental overwriting of variables.

**Syntax:**

let variableName = value;

**Examples:**

**1. Basic Declaration and Assignment:**

let x = 10; // Declare and initialize a variable

console.log(x); // 10

**2. Block Scope:**

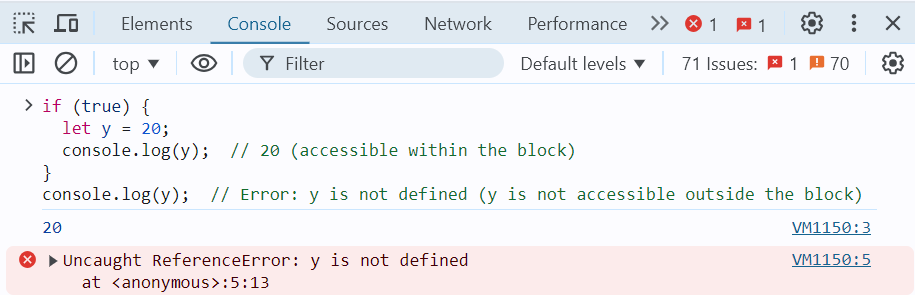
if (true) {

let y = 20;

console.log(y); // 20 (accessible within the block)

}

console.log(y); // Error: y is not defined (y is not accessible outside the block)



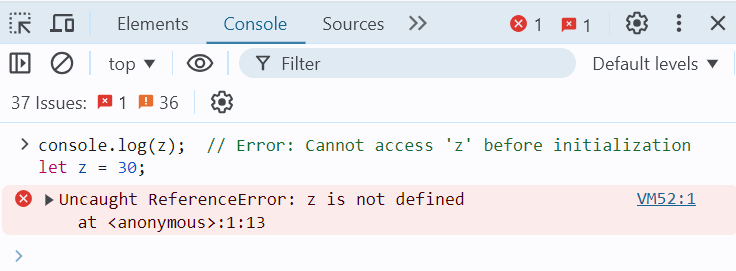
**3. No Hoisting with Initialization:**

Unlike var, let variables are hoisted but not initialized, leading to a temporal dead zone (TDZ).

console.log(z); // Error: Cannot access 'z' before initialization

let z = 30;

The above code throws an error because z is in the temporal dead zone until the let z = 30; line is executed.



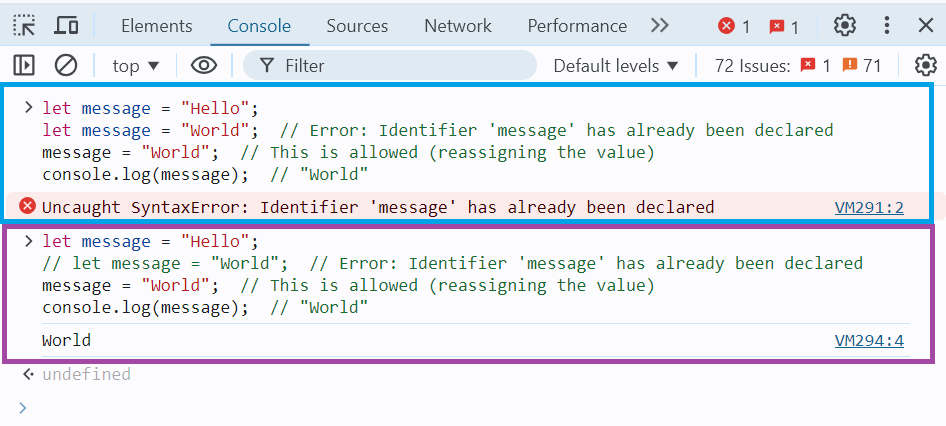
**4. No Redeclaration:**

let message = "Hello";

// let message = "World"; // Error: Identifier 'message' has already been declared

message = "World"; // This is allowed (reassigning the value)

console.log(message); // "World"



**5. Using let in Loops:**

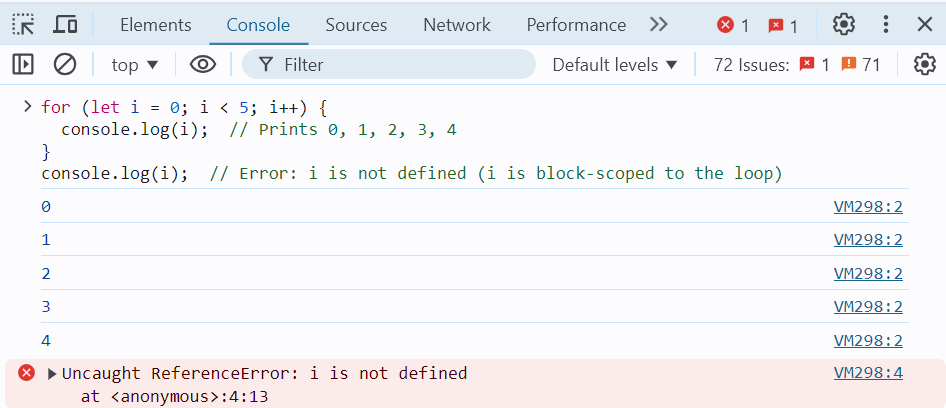
let is commonly used in loops, especially in scenarios where each iteration needs its own scope.

for (let i = 0; i < 5; i++) {

console.log(i); // Prints 0, 1, 2, 3, 4

}

console.log(i); // Error: i is not defined (i is block-scoped to the loop)



The let keyword is generally preferred over var in modern JavaScript because of its block-scoping behaviour, lack of redeclaration, and safer hoisting. It reduces the risk of errors in code by limiting the scope of variables and ensuring that they are not accidentally redeclared. This makes let a more predictable and reliable way to declare variables.

**Const:**

The const keyword in JavaScript, introduced in ES6 (ECMAScript 2015), is used to declare variables that are **constant**. Once a variable is assigned a value with const, it cannot be reassigned. const is typically used to declare variables that should not change throughout the program, such as constants or configuration values.

**Characteristics of const:**

1. **Block Scope**: Similar to let, const is block-scoped, meaning it is only accessible within the block in which it is declared.
2. **No Reassignment**: Variables declared with const cannot be reassigned a new value after they are initialized.
3. **Must be Initialized**: A const variable must be initialized at the time of declaration. Unlike let, you cannot declare a const variable without assigning a value.
4. **Mutable Objects**: While you cannot reassign a const variable, if the variable holds an object (including arrays), the contents of the object or array can still be modified.

**Syntax:**

const variableName = value;

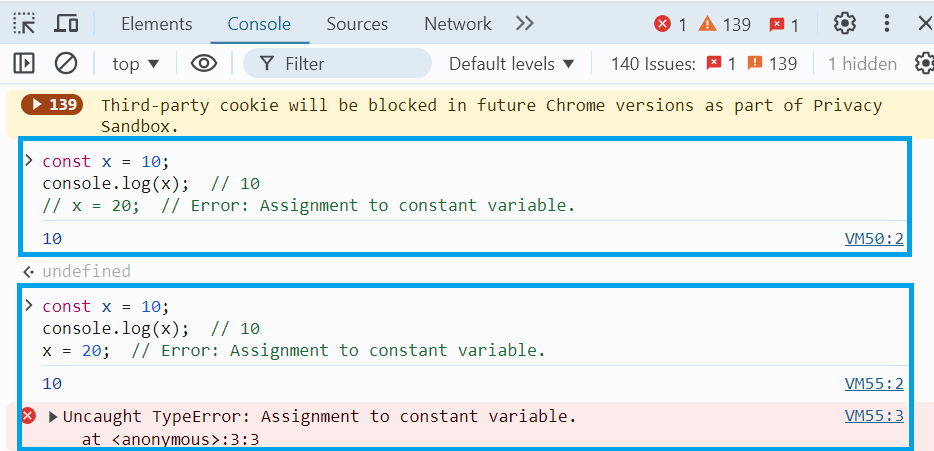
**Examples:**

**1. Basic Declaration and Assignment:**

const x = 10;

console.log(x); // 10

// x = 20; // Error: Assignment to constant variable.



**2. Block Scope:**

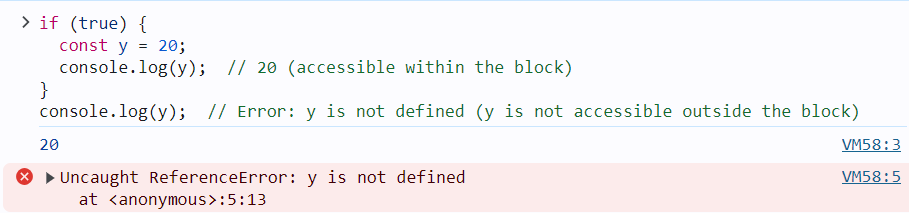
if (true) {

const y = 20;

console.log(y); // 20 (accessible within the block)

}

console.log(y); // Error: y is not defined (y is not accessible outside the block)



**3. Mutable Objects and Arrays:**

Even though const prevents reassignment, the contents of objects and arrays can still be modified:

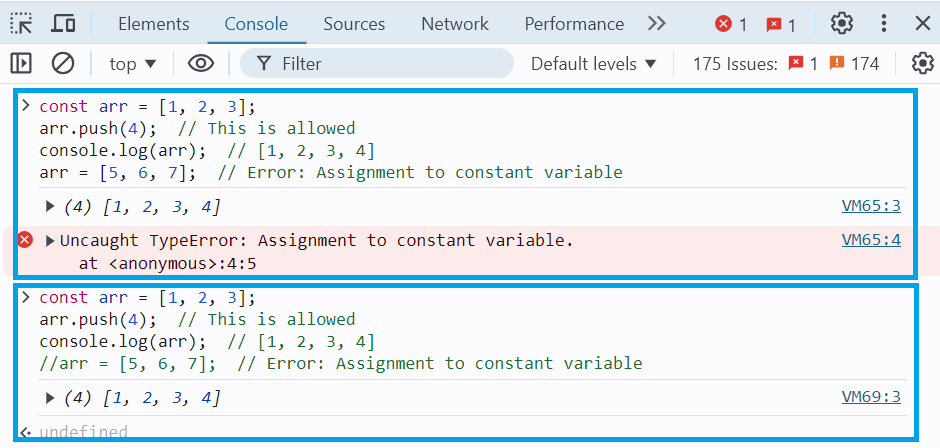
* **Array Example**:

const arr = [1, 2, 3];

arr.push(4); // This is allowed

console.log(arr); // [1, 2, 3, 4]

// arr = [5, 6, 7]; // Error: Assignment to constant variable



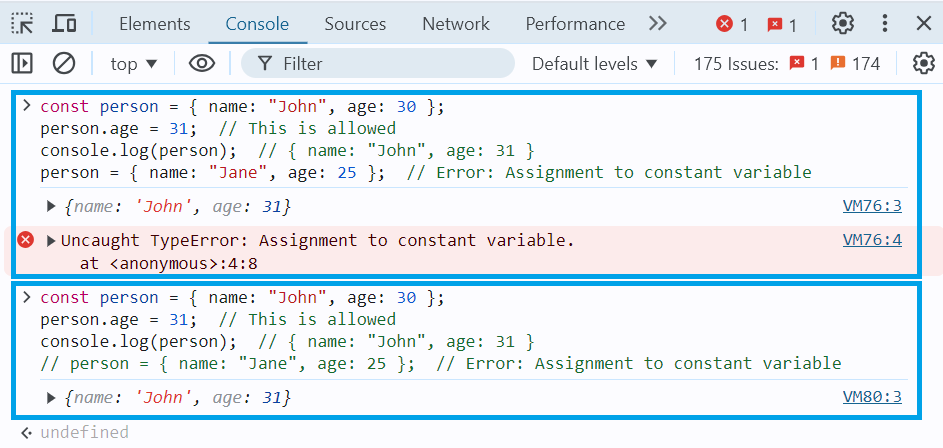
* **Object Example**:

const person = { name: "John", age: 30 };

person.age = 31; // This is allowed

console.log(person); // { name: "John", age: 31 }

// person = { name: "Jane", age: 25 }; // Error: Assignment to constant variable



**4. Const in Loops:**

You can use const in loops, but only if the variable does not need to be reassigned. For example, const can be used inside a loop for iteration variables in a for...of loop:

const arr = [10, 20, 30];

for (const num of arr) {

console.log(num); // Prints 10, 20, 30

}

However, const is not suitable for traditional for loops where the loop counter is expected to change:

// This will throw an error because i is being reassigned in every iteration

// for (const i = 0; i < 5; i++) {

// console.log(i);

// }



The const keyword is ideal for declaring variables that should not be reassigned. However, it does not make the variable itself immutable if it holds an object or an array — only the reference to the variable is constant. For values that need to remain constant throughout the program, const is the best choice, and it is generally preferred for readability and clarity in modern JavaScript.

1. JavaScript Control Flow Statements
2. [JS return Statement](https://www.geeksforgeeks.org/javascript-return-statement/)

The return statement is used to exit a function and optionally pass a value back to the caller. Once a return statement is executed, the function stops executing, and the control is returned to the calling code.

**Example 1: Basic return statement**

function add(a, b) {

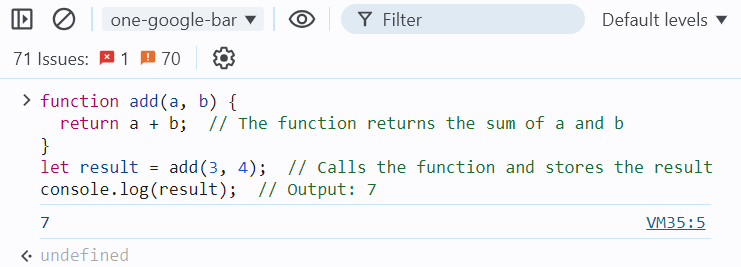
return a + b; // The function returns the sum of a and b

}

let result = add(3, 4); // Calls the function and stores the result

console.log(result); // Output: 7

In this example, the add function adds two numbers and returns the result using the return statement.



**Example 2: return without a value**

function greet(name) {

if (!name) {

return; // If no name is provided, return (undefined)

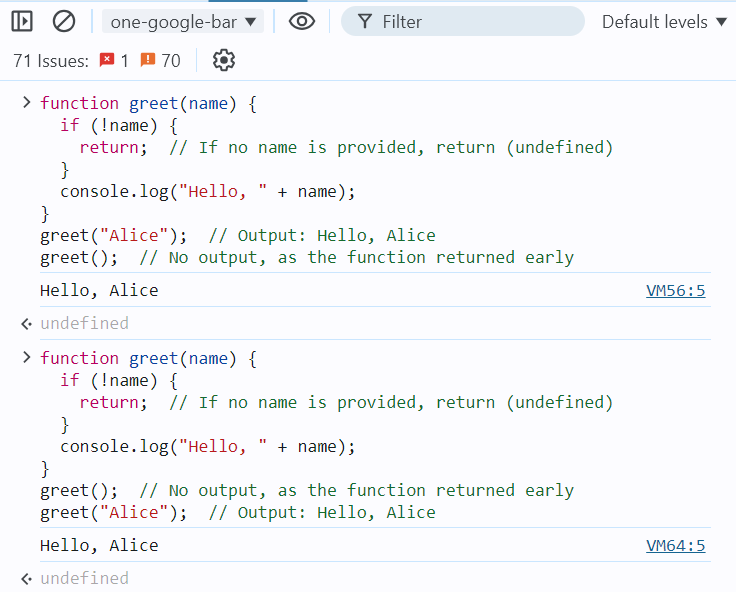
}

console.log("Hello, " + name);

}

greet("Alice"); // Output: Hello, Alice

greet(); // No output, as the function returned early



Here, the function returns early without any value if no name is provided. When return is used without a value, the function returns undefined by default.

**Example 3: Returning an object**

function createPerson(firstName, lastName) {

return {

firstName: firstName,

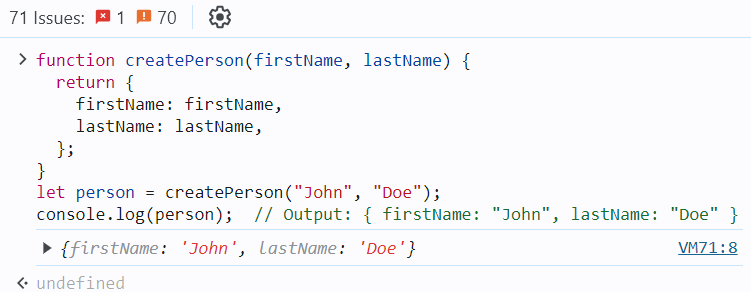
lastName: lastName,

};

}

let person = createPerson("John", "Doe");

console.log(person); // Output: { firstName: "John", lastName: "Doe" }



In this example, the function createPerson returns an object containing the provided firstName and lastName.

**Example 4: Returning an object**

The code defines a function Language() that returns an object containing three properties: first, second, and Third, each storing a string value. Then, it uses object destructuring to assign these properties to variables first, second, and Third. Finally, it logs the values of these variables.

function Language() {

let first = 'HTML',

second = 'CSS',

Third = 'Javascript'

return {

first,

second,

Third

};

}

let { first, second, Third } = Language();

console.log(first+ " " + second + " " + Third);

**I.Q.:**

Can a JavaScript function have multiple return statements?

What happens if there is no return statement in a function?

Can you return multiple values from a function?

What is the difference between return and console.log?

Does the return statement exit the function?

Can you use return outside a function in JavaScript?

*No, the return statement can only be used inside functions. Using it outside a function will result in a syntax error.*

1. [JS break Statement](https://www.geeksforgeeks.org/javascript-break-statement/)

The break statement is used to terminate a loop or switch statement. When a break is encountered, the program immediately exits the loop or switch block, and the control moves to the statement following the loop or switch.

**Example 1: Using break in a loop**

for (let i = 0; i < 10; i++) {

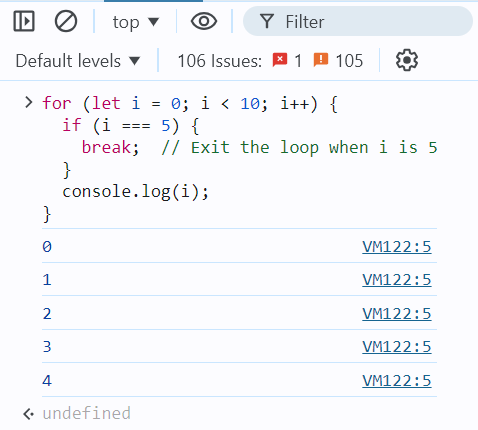
if (i === 5) {

break; // Exit the loop when i is 5

}

console.log(i);

}



In this example, the loop iterates from 0 to 9, but when i reaches 5, the break statement is executed, causing the loop to terminate early.

**Example 2: Using break in a while loop**

let i = 0;

while (i < 10) {

console.log(i);

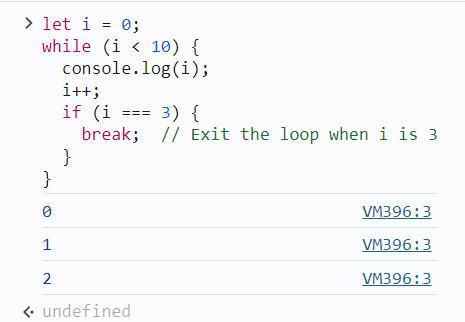
i++;

if (i === 3) {

break; // Exit the loop when i is 3

}

}



Here, the while loop runs until i equals 3, at which point the break statement is encountered, terminating the loop.

**Example 3: Using break in a switch statement**

let fruit = "apple";

switch (fruit) {

case "apple":

console.log("This is an apple.");

break;

case "banana":

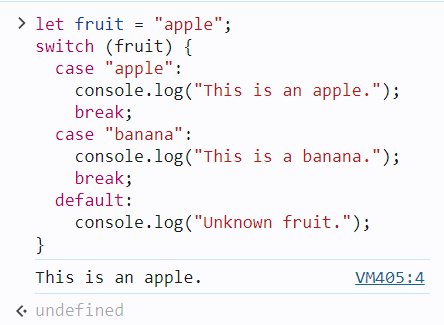
console.log("This is a banana.");

break;

default:

console.log("Unknown fruit.");

}



In this switch statement, the break statement prevents the execution from "falling through" to the next case. Without the break, all subsequent cases would be executed regardless of the match.

**Important Notes:**

* In a for, while, or do...while loop, break immediately exits the loop.
* In a switch statement, break stops the execution of further cases and exits the switch block.

1. [JS continue Statement](https://www.geeksforgeeks.org/javascript-continue-statement/)

Continue statement is used to skip the current iteration of a loop and move to the next iteration. Unlike the break statement, which completely exits the loop, continue only skips the current iteration and resumes execution at the next iteration of the loop.

**Example 1: Using continue in a for loop**

for (let i = 0; i < 5; i++) {

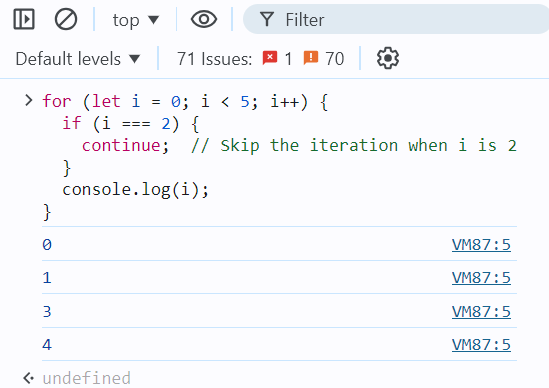
if (i === 2) {

continue; // Skip the iteration when i is 2

}

console.log(i);

}

****

In this example, when i is 2, the continue statement is executed, which skips that iteration. Therefore, 2 is not printed, but the loop continues with the next iteration.

**Example 2: Using continue in a while loop**

let i = 0;

while (i < 5) {

i++;

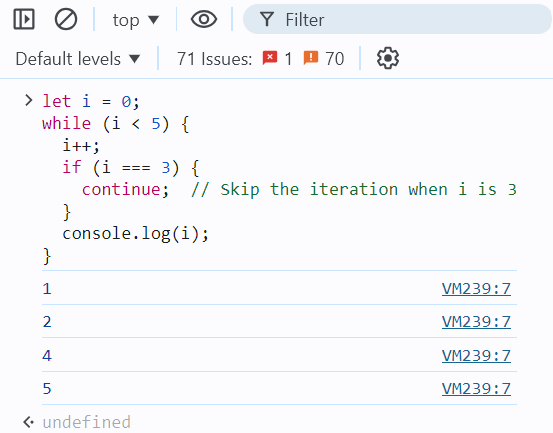
if (i === 3) {

continue; // Skip the iteration when i is 3

}

console.log(i);

}



Here, when i equals 3, the continue statement causes the loop to skip that iteration, so 3 is not printed. The loop then continues with the next value of i.

**Example 3: Using continue in a nested loop**

for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

if (j === 1) {

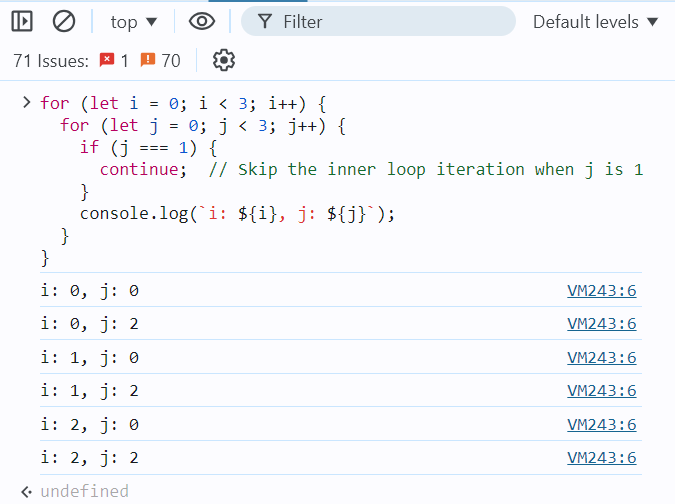
continue; // Skip the inner loop iteration when j is 1

}

console.log(`i: ${i}, j: ${j}`);

}

}



In this example, the continue statement inside the inner loop skips the iteration when j equals 1. The loop continues with the next value of j, but j = 1 is skipped for each value of i.

**Important Notes:**

* The continue statement is useful when you want to skip certain iterations of a loop based on a condition.
* It can be used in any type of loop: for, while, do...while.

1. [JS throw Statement](https://www.geeksforgeeks.org/javascript-errors-throw-and-try-to-catch/)

In JavaScript, handling errors is an essential part of writing robust code. JavaScript provides the throw statement to raise an error and the try...catch block to handle it.

**1. Throwing Errors**

You can use the throw statement to create custom errors in your code. When you throw an error, the normal flow of execution stops, and control is passed to the nearest catch block that can handle the error.

Here's how you can use the throw statement:

function divide(a, b) {

if (b === 0) {

throw new Error("Division by zero is not allowed.");

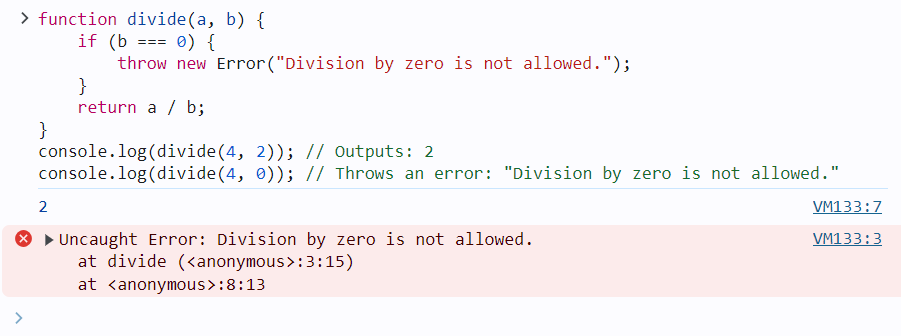
}

return a / b;

}

console.log(divide(4, 2)); // Outputs: 2

console.log(divide(4, 0)); // Throws an error: "Division by zero is not allowed."



**2. Handling Errors with Try...Catch**

To handle errors that might occur during execution, you use the try...catch statement. This block allows you to "try" to execute code that might throw an error, and if an error occurs, the control is passed to the catch block.

Here’s an example:

function divide(a, b) {

if (b === 0) {

throw new Error("Division by zero is not allowed.");

}

return a / b;

}

try {

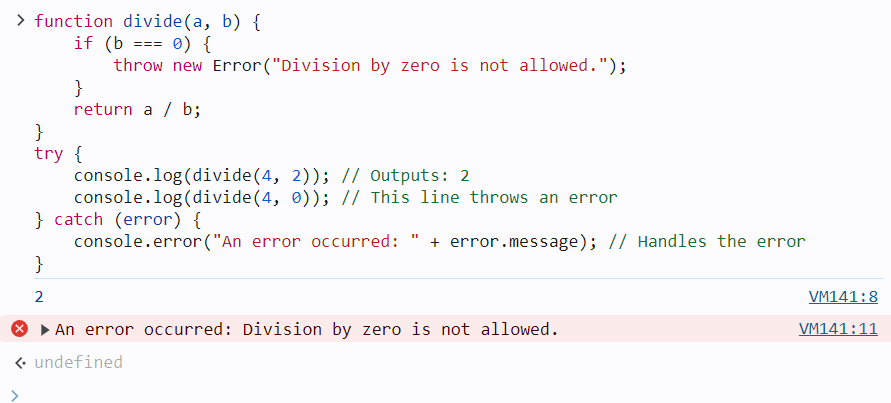
console.log(divide(4, 2)); // Outputs: 2

console.log(divide(4, 0)); // This line throws an error

} catch (error) {

console.error("An error occurred: " + error.message); // Handles the error

}



**3. The finally Block**

You can also include a finally block after the try...catch blocks. The code inside the finally block will run regardless of whether an error was thrown or not.

function divide(a, b) {

if (b === 0) {

throw new Error("Division by zero is not allowed.");

}

return a / b;

}

try {

console.log(divide(4, 2)); // Outputs: 2

console.log(divide(4, 0)); // Throws an error

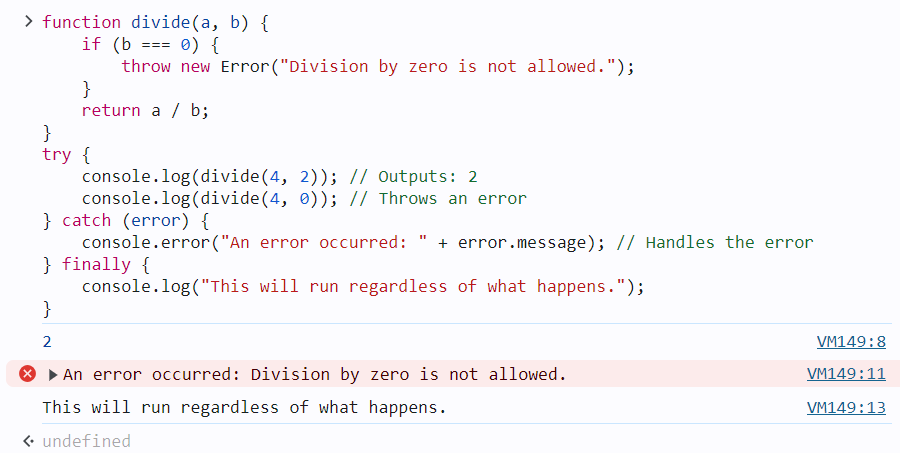
} catch (error) {

console.error("An error occurred: " + error.message); // Handles the error

} finally {

console.log("This will run regardless of what happens.");

}



In this example, the message inside the finally block will always be printed, whether or not an error occurred.

**4. Custom Error Types**

You can also create your own custom error types by extending the Error class. This is useful when you want to throw and catch specific types of errors.

class DivisionByZeroError extends Error {

constructor(message) {

super(message);

this.name = "DivisionByZeroError";

}

}

function divide(a, b) {

if (b === 0) {

throw new DivisionByZeroError("Division by zero is not allowed.");

}

return a / b;

}

try {

console.log(divide(4, 0)); // Throws a DivisionByZeroError

} catch (error) {

if (error instanceof DivisionByZeroError) {

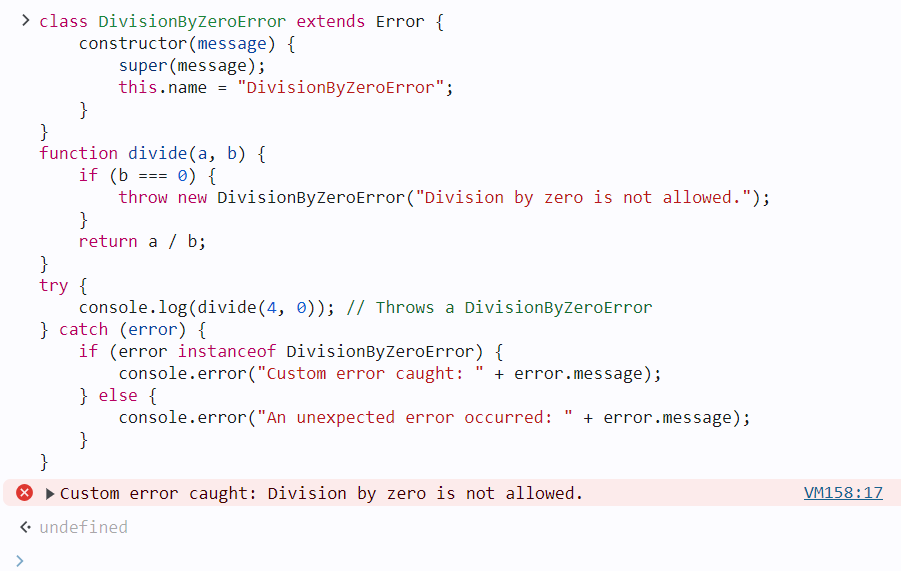
console.error("Custom error caught: " + error.message);

} else {

console.error("An unexpected error occurred: " + error.message);

}

}



This example shows how to create and handle custom error types, allowing for more fine-grained error handling.

**Summary:**

* **throw**: Used to create and throw an error.
* **try...catch**: Used to handle errors that occur within the try block.
* **finally**: Code inside this block runs whether an error is thrown or not.
* **Custom Errors**: Extend the Error class to create specific error types for better error handling.

This approach helps to manage exceptions in your code effectively, ensuring that errors are caught and handled appropriately.

1. [JS if…else Statement](https://www.geeksforgeeks.org/conditional-statements-in-javascript/)

JavaScript conditional statements allow you to execute specific blocks of code based on conditions. If the condition is met, a particular block of code will run; otherwise, another block of code will execute based on the condition.

|  |  |
| --- | --- |
| **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 same condition of the preceding if statement is false. |
| else if statement | Adds more conditions to the if statement, allowing for multiple alternative conditions to be tested. |
| switch statement | Evaluates an expression, then executes the case statement that matches the expression’s value. |
| ternary operator | Provides a concise way to write if-else statements in a single line. |
| Nested if else statement | Allows for multiple conditions to be checked in a hierarchical manner. |

**Conditional Statements Examples:**

**1. Using if Statement**

The if statement is used to evaluate a particular condition. If the condition holds true, the associated code block is executed.

**Syntax:**

if ( condition ) {

// If the condition is met,

//code will get executed.

}

**Example:**This JavaScript code determines if the variable `num` is even or odd using the modulo operator `%`. If `num` is divisible by 2 without a remainder, it logs “Given number is even number.” Otherwise, it logs “Given number is odd number.”

let num = 20;

if (num % 2 === 0) {

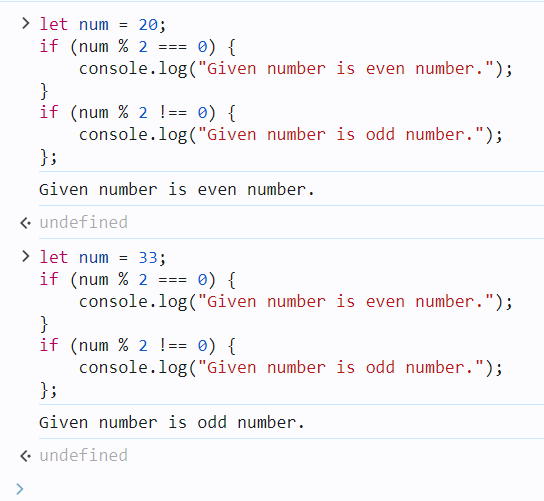
console.log("Given number is even number.");

}

if (num % 2 !== 0) {

console.log("Given number is odd number.");

};



**2. Using if-else Statement**

The if-else statement will perform some action for a specific condition. Here we are using the else statement in which the else statement is written after the if statement and it has no condition in their code block.

**Syntax:**

if (condition1) {

// Executes when condition1 is true

if (condition2) {

// Executes when condition2 is true

}

}

**Example:**This JavaScript code checks if the variable `age` is greater than or equal to 18. If true, it logs “You are eligible for a driving license.” Otherwise, it logs “You are not eligible for a driving license.” This indicates eligibility for driving based on age.

let age = 25;

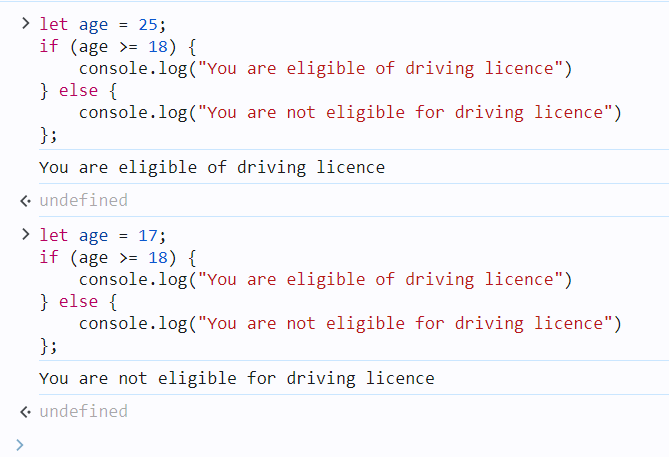
if (age >= 18) {

console.log("You are eligible of driving licence")

} else {

console.log("You are not eligible for driving licence")

};



**3. else if Statement**

The else if statement in JavaScript allows handling multiple possible conditions and outputs, evaluating more than two options based on whether the conditions are true or false.

**Syntax:**

if (1st condition) {

// Code for 1st condition

} else if (2nd condition) {

// ode for 2nd condition

} else if (3rd condition) {

// Code for 3rd condition

} else {

// ode that will execute if all

// above conditions are false

}

**Example:**This JavaScript code determines whether the constant `num` is positive, negative, or zero. If `num` is greater than 0, it logs “Given number is positive.” If `num` is less than 0, it logs “Given number is negative.” If neither condition is met (i.e., `num` is zero), it logs “Given number is zero”.

const num = 33;

if (num > 0) {

console.log("Given number is positive.");

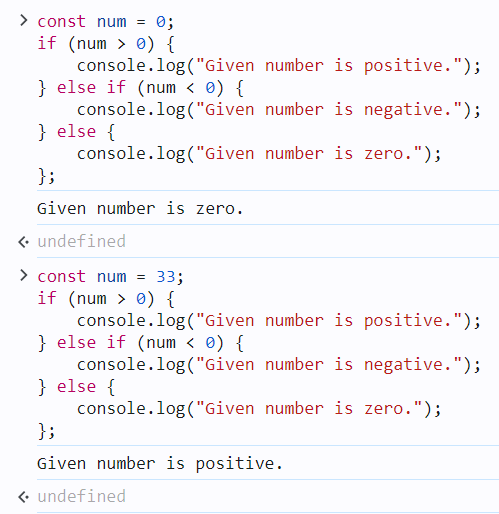
} else if (num < 0) {

console.log("Given number is negative.");

} else {

console.log("Given number is zero.");

};



**4. Using Switch Statement (JavaScript Switch Case)**

As the number of conditions increases, you can use multiple else-if statements in JavaScript. but when we dealing with many conditions, the switch statement may be a more preferred option.

**Syntax:**

switch (expression) {

case value1:

statement1;

break;

case value2:

statement2;

break;

. . .

case valueN:

statementN;

break;

default:

statementDefault;

};

**Example:**This JavaScript code assigns a branch of engineering to a student based on their marks. It uses a switch statement with cases for different mark ranges. The student’s branch is determined according to their marks and logged to the console.

**const** marks = 93;

**let** Branch;

**switch** (**true**) {

**case** marks >= 90:

Branch = "Computer science engineering";

**break**;

**case** marks >= 80:

Branch = "Mechanical engineering";

**break**;

**case** marks >= 70:

Branch = "Chemical engineering";

**break**;

**case** marks >= 60:

Branch = "Electronics and communication";

**break**;

**case** marks >= 50:

Branch = "Civil engineering";

**break**;

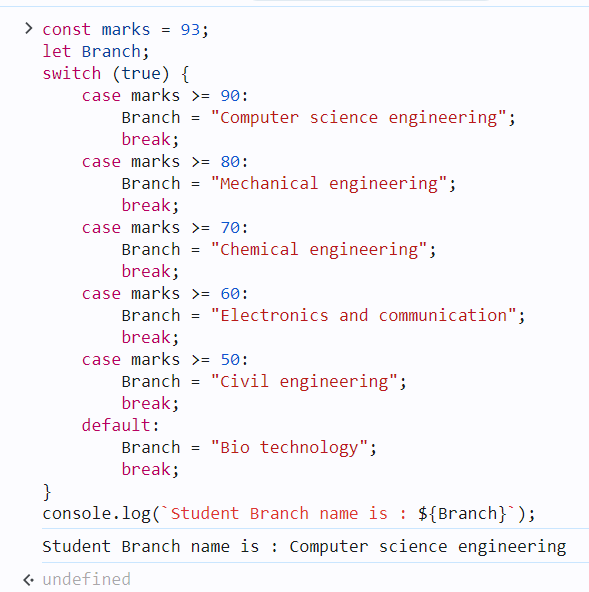
**default**:

Branch = "Bio technology";

**break**;

}

console.log(`Student Branch name is : **${**Branch**}**`);



**5. Using Ternary Operator ( ?: )**

The conditional operator, also referred to as the ternary operator (?:), is a shortcut for expressing conditional statements in JavaScript.

**Syntax:**

condition ? value if true : value if false

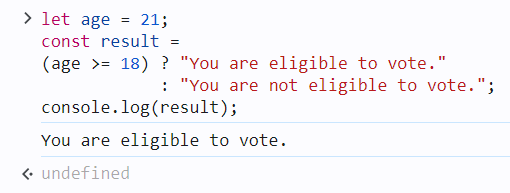
**Example:**This JavaScript code checks if the variable `age` is greater than or equal to 18. If true, it assigns the string “You are eligible to vote.” to the variable `result`. Otherwise, it assigns “You are not eligible to vote.” The value of `result` is then logged to the console.

let age = 21;

const result =

(age >= 18) ? "You are eligible to vote."

: "You are not eligible to vote.";

console.log(result);  


**6. Nested if…else**

Nested if…else statements in JavaScript allow us to create complex conditional logic by checking multiple conditions in a hierarchical manner. Each if statement can have an associated else block, and within each if or else block, you can nest another if…else statement. This nesting can continue to multiple levels, but it’s important to maintain readability and avoid excessive complexity.

**Syntax:**

if (condition1) {

// Code block 1

if (condition2) {

// Code block 2

} else {

// Code block 3

}

} else {

// Code block 4

}

**Example:**In this example, the outer if statement checks the weather variable. If it’s “sunny,” it further checks the temperature variable to determine the type of day it is (hot, warm, or cool). Depending on the values of weather and temperature, different messages will be logged to the console.

**let** weather = "sunny";

**let** temperature = 25;

**if** (weather === "sunny") {

**if** (temperature > 30) {

console.log("It's a hot day!");

} **else** **if** (temperature > 20) {

console.log("It's a warm day.");

} **else** {

console.log("It's a bit cool today.");

}

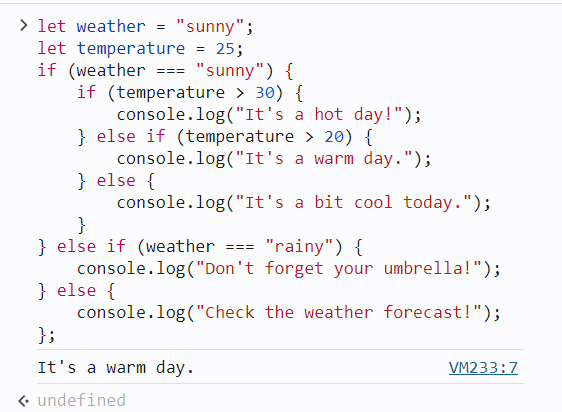
} **else** **if** (weather === "rainy") {

console.log("Don't forget your umbrella!");

} **else** {

console.log("Check the weather forecast!");

};



1. [JS switch Statement](https://www.geeksforgeeks.org/switch-case-in-javascript/)

The switch statement in JavaScript is used to perform different actions based on different conditions. It is an alternative to using multiple if...else if statements when you need to compare a value against multiple possible outcomes.

**Syntax**

switch(expression) {

case value1:

// Code to run if expression === value1

break;

case value2:

// Code to run if expression === value2

break;

// Add more cases as needed

default:

// Code to run if no case matches

}

**How It Works**

* The expression is evaluated once and compared with each case.
* If a match is found, the code block corresponding to that case is executed.
* The break statement prevents the code from running into the next case. Without break, the execution will continue to the next case, even if it doesn't match.
* The default case is optional but will execute if no matching case is found.

**Example: Days of the Week**

let day = 3;

let dayName;

switch (day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday"; // This case matches, so this block runs

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day"; // Runs if none of the above cases match

}

console.log(dayName); // Outputs: Wednesday



**Example: Grading System**

Here’s another example that assigns grades based on a score:

let score = 90;

let grade;

switch (true) {

case (score >= 90):

grade = "A";

break;

case (score >= 80):

grade = "B"; // This case matches, so this block runs

break;

case (score >= 70):

grade = "C";

break;

case (score >= 60):

grade = "D";

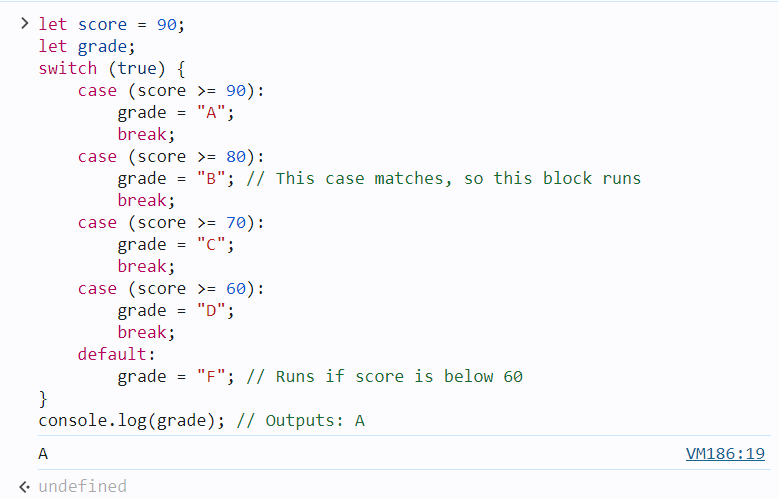
break;

default:

grade = "F"; // Runs if score is below 60

}

console.log(grade); // Outputs: A



**Key Points**

* **switch** is useful when you need to compare a single expression against multiple possible values.
* **break** is important to prevent fall-through, where multiple cases might be executed unintentionally.
* **default** is optional but provides a fallback when no case matches.

The switch statement makes code easier to read and manage, especially when dealing with multiple possible outcomes for a single expression.

1. [JS try…catch Statement](https://www.geeksforgeeks.org/javascript-errors-throw-and-try-to-catch/)
2. [JavaScript Loops](https://www.geeksforgeeks.org/javascript/#javascript-loops)
3. [JS for Loop](https://www.geeksforgeeks.org/javascript-for-loop/)

The for loop in JavaScript is used to repeatedly execute a block of code a certain number of times. It's one of the most commonly used loops in programming for iterating over arrays, performing repetitive tasks, and more.

**Syntax**

for (initialization; condition; update) {

// Code to execute in each iteration

}

**Explanation**

* **Initialization**: This step is executed only once before the loop starts. It typically initializes one or more loop counters.
* **Condition**: Before each iteration, the loop checks this condition. If it evaluates to true, the loop continues. If false, the loop stops.
* **Update**: This step is executed after each iteration. It typically increments or decrements the loop counter(s).

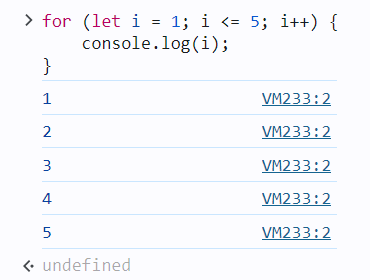
**Example: Basic for Loop**

Here’s a simple example of a for loop that prints numbers from 1 to 5:

for (let i = 1; i <= 5; i++) {

console.log(i);

}

****

**Example: Iterating Over an Array**

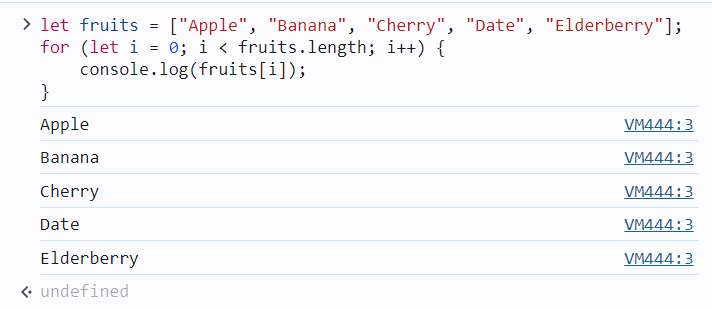
You can also use a for loop to iterate over the elements of an array:

let fruits = ["Apple", "Banana", "Cherry", "Date", "Elderberry"];

for (let i = 0; i < fruits.length; i++) {

console.log(fruits[i]);

}

****

**Example: Summing Numbers**

Here’s an example where a for loop is used to calculate the sum of numbers from 1 to 10:

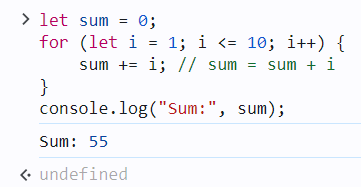
let sum = 0;

for (let i = 1; i <= 10; i++) {

sum += i; // sum = sum + i

}

console.log("Sum:", sum);

****

**Example: Nested for Loop**

You can also use nested for loops, which are loops inside other loops. This is useful when working with multi-dimensional arrays or performing operations on grid-like structures.

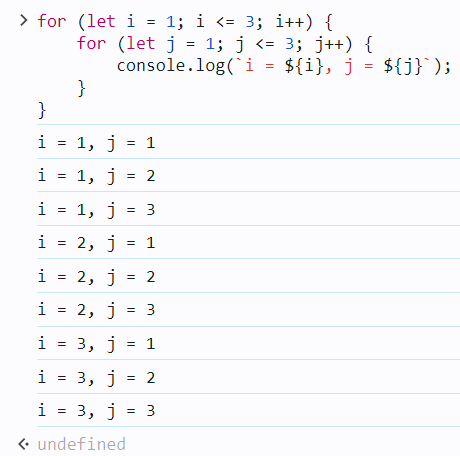
for (let i = 1; i <= 3; i++) {

for (let j = 1; j <= 3; j++) {

console.log(`i = ${i}, j = ${j}`);

}

}

****

**Summary**

* The for loop is used to repeat a block of code a specified number of times.
* **Initialization** is done once at the start.
* **Condition** is checked before each iteration, and the loop runs as long as this condition is true.
* **Update** happens after each iteration, typically used to modify the loop counter.
* It’s commonly used for iterating over arrays, summing values, and performing repetitive tasks.

1. [JS do…while Loop](https://www.geeksforgeeks.org/javascript-do-while-loop/)

The **do...while** loop in JavaScript is a control flow statement that executes a block of code **at least once**, and then continues to execute the block as long as a specified condition is true. The condition is evaluated **after** the code block is executed, which ensures that the code block runs at least once, regardless of the condition's value.

**Syntax:**

do {

// Code block to execute

} while (condition);

**Key Points:**

* The code inside the do block will run **at least once**.
* After the code block runs, the condition is checked.
* If the condition is true, the loop repeats; if false, the loop terminates.

**Example 1: Basic do...while Loop**

let i = 0;

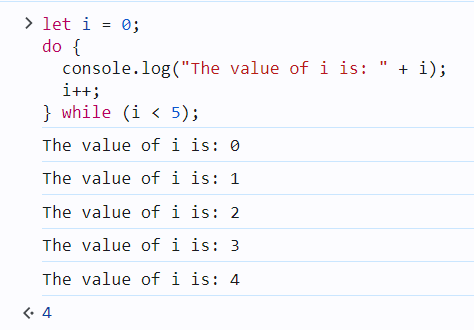
do {

console.log("The value of i is: " + i);

i++;

} while (i < 5);

**Output:**



**Explanation:**

* Initially, i is 0.
* The loop prints the value of i, then increments it by 1.
* The loop continues as long as i < 5.

**Example 2: Loop that Runs Once Even if Condition is false**

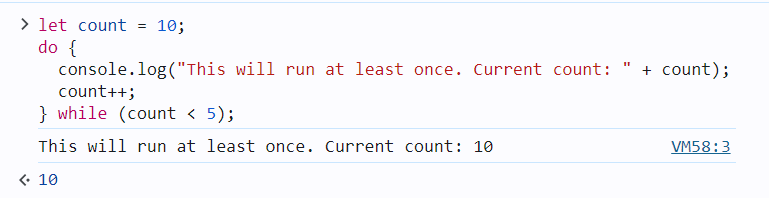
let count = 10;

do {

console.log("This will run at least once. Current count: " + count);

count++;

} while (count < 5);

****

**Explanation:**

* Although the condition count < 5 is false from the beginning, the code inside the do block still runs **once** because the condition is checked only after the code block is executed.

**Example 3: Using do...while for User Input Validation**

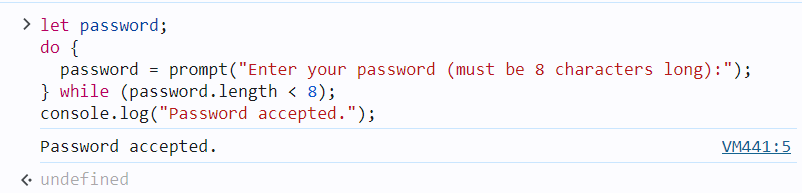
let password;

do {

password = prompt("Enter your password (must be 8 characters long):");

} while (password.length < 8);

console.log("Password accepted.");



**Explanation:**

* The loop prompts the user for a password and repeats until a password of at least 8 characters is entered.

In summary, the do...while loop guarantees that the code inside the loop will execute at least once, regardless of the condition. This is useful in scenarios where you need to ensure an action is performed before a condition is checked, such as input validation or ensuring initialization steps are executed.

1. [JS while Loop](https://www.geeksforgeeks.org/javascript-while-loop/)

The**while loop**executes a block of code as long as a specified condition is true. In JavaScript, this loop evaluates the condition before each iteration and continues running as long as the condition remains true. The loop terminates when the condition becomes false, enabling dynamic and repeated operations based on changing conditions.

**Syntax**

while (condition) {  
 Code block to be executed  
}

Example: Here’s an example of a while loop that counts from 1 to 5.

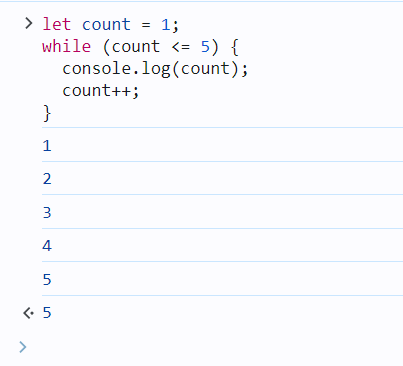
**let** count = 1;

**while** (count <= 5) {

console.log(count);

count++;

}

****

**Do-While loop**

A [**Do-While loop**](https://www.geeksforgeeks.org/javascript-do-while-loop)is another type of loop in JavaScript that is similar to the while loop, but with one key difference: the do-while loop guarantees that the block of code inside the loop will be executed at least once, regardless of whether the condition is initially true or false .

**Syntax**

do {   
 // code block to be executed   
 } while (condition);

**Example**: Here’s an example of a do-while loop that counts from 1 to 5.

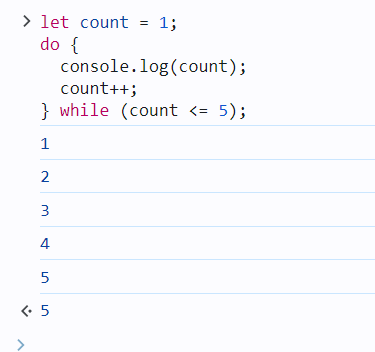
**let** count = 1;

**do** {

console.log(count);

count++;

} **while** (count <= 5);

****

**Comparison between the while and do-while loop:**

The do-while loop executes the content of the loop once before checking the condition of the while loop. While the while loop will check the condition first before executing the content.

|  |  |
| --- | --- |
| **While Loop** | **Do-While Loop** |
| It is an entry condition looping structure. | It is an exit condition looping structure. |
| The number of iterations depends on the condition mentioned in the while block. | Irrespective of the condition mentioned in the do-while block, there will a minimum of 1 iteration. |
| The block control condition is available at the starting point of the loop. | The block control condition is available at the endpoint of the loop. |

1. [JS for…in Loop](https://www.geeksforgeeks.org/javascript-for-in-loop/)

The for...in loop in JavaScript is used to iterate over the **enumerable properties** of an object. It allows you to loop through the keys (or property names) of an object or the indices of an array.

**Syntax:**

for (variable in object) {

// code block to execute

}

* variable: The variable that will be assigned the current property/key name on each iteration.
* object: The object or array whose properties/indices will be iterated over.

**Example with an Object:**

const person = {

name: "John",

age: 30,

city: "New York"

};

for (let key in person) {

console.log(key + ": " + person[key]);

}

**Output:**

name: John

age: 30

city: New York

**Explanation:**

* for (let key in person): Loops through each key of the person object.
* person[key]: Accesses the value associated with the current key.

**Example with an Array:**

const fruits = ["Apple", "Banana", "Mango"];

for (let index in fruits) {

console.log(index + ": " + fruits[index]);

}

**Output:**

0: Apple

1: Banana

2: Mango

**Explanation:**

* for (let index in fruits): Loops through each index of the fruits array.
* fruits[index]: Accesses the value at the current index.

**Important Notes:**

* The for...in loop is designed for **objects**, but it can also iterate over arrays. However, for arrays, it is generally recommended to use a for loop or for...of loop to avoid potential issues with inherited properties.
* If the object has properties inherited from its prototype, for...in will also iterate over those properties. To avoid this, you can use the hasOwnProperty() method.

**Example using hasOwnProperty():**

const person = {

name: "John",

age: 30

};

Object.prototype.gender = "male"; // Adding a property to the prototype

for (let key in person) {

if (person.hasOwnProperty(key)) {

console.log(key + ": " + person[key]);

}

}

**Output:**

name: John

age: 30

In this case, we prevent the prototype property gender from being printed.

Example :

For-in loop iterates over the properties of an object and its prototype chain’s properties. If we want to display both properties of the “student1” object which belongs to that object only and the prototype chain, then we can perform it by for in loop.

const courses = {

// Declaring a courses object

firstCourse: "C++ STL",

secondCourse: "DSA Self Paced",

thirdCourse: "CS Core Subjects"

};

// Creating a new empty object with

// prototype set to courses object

const student1 = Object.create(courses);

// Defining student1 properties and methods

student1.id = 123;

student1.firstName = "Prakhar";

student1.showEnrolledCourses = function () {

console.log(courses);

}

// Iterating over all properties of

// student1 object

for (let prop in student1) {

console.log(prop + " -> "

+ student1[prop]);

}

**Output**

id -> 123

firstName -> Prakhar

showEnrolledCourses -> function () {

console.log(courses);

}

firstCourse -> C++ STL

secondCourse -> DSA Self Paced

thirdCourse -> CS Core Subjects

1. [JS for…of Loop](https://www.geeksforgeeks.org/javascript-for-of-loop/)

The for...of loop in JavaScript is used to iterate over **iterable objects**, such as arrays, strings, maps, sets, and more. It allows you to loop through the **values** of an iterable object rather than its property names or indices.

**Syntax:**

for (variable of iterable) {

// code block to execute

}

* variable: A variable that holds the value of each iteration.
* iterable: An iterable object (such as an array, string, map, set, etc.).

**Example with an Array:**

const fruits = ["Apple", "Banana", "Mango"];

for (let fruit of fruits) {

console.log(fruit);

}

**Output:**

Apple

Banana

Mango

**Explanation:**

* for (let fruit of fruits): Loops through each element (value) in the fruits array.
* Each iteration assigns the value of the current element to fruit and logs it.

**Example with a String:**

const word = "Hello";

for (let char of word) {

console.log(char);

}

**Output:**

H

e

l

l

o

**Explanation:**

* for (let char of word): Loops through each character in the string "Hello".
* Each iteration assigns the current character to char and logs it.

**Example with a Set:**

const uniqueNumbers = new Set([1, 2, 3, 4]);

for (let number of uniqueNumbers) {

console.log(number);

}

**Output:**

1

2

3

4

**Example with a Map:**

const map = new Map();

map.set("name", "John");

map.set("age", 30);

for (let [key, value] of map) {

console.log(key + ": " + value);

}

**Output:**

name: John

age: 30

**Difference between for...in and for...of:**

* **for...in**: Iterates over **enumerable property names** (keys) of an object or array.
* **for...of**: Iterates over **values** of an iterable object.

**Example to demonstrate the difference:**

const arr = ["a", "b", "c"];

console.log("for...in:");

for (let index in arr) {

console.log(index); // Logs index

}

console.log("for...of:");

for (let value of arr) {

console.log(value); // Logs value

}

**Output:**

for...in:

0

1

2

for...of:

a

b

c

**Conclusion:**

* Use for...of when you want to iterate through the **values** of an iterable (arrays, strings, maps, sets).
* Use for...in when you want to iterate over the **keys** of an object or array.

Example:

**Iterating Over a Map using for…of Loop**

Maps are a new data structure in ES6 that store key-value pairs. The for…of loop can be used to iterate over the entries of a map.

const map = new Map([

["name", "Akash"],

["age", 25],

["city", "Noida"]

]);

for (let [key, value] of map) {

console.log(`${key}: ${value}`);

}

**Output**

name: Akash

age: 25

city: Noida

**Code Explanation**

* First, we create a Map object where we want to iterate over.
* Initiates the for...of loop, where [key, value] represents each key-value pair in the Map during each iteration.
* Inside the loop, console.log(`${key}: ${value}`); prints each key-value pair to the console during each iteration of the loop.

1. [JS labeled Statement](https://www.geeksforgeeks.org/javascript-label-statement/)

JavaScript **label statement**is used to label a block of code. A labeled statement can be used with loops and control flow statements to provide a target for the break and continue statements.

**Syntax:**

Label: statement (loop or block of code)

**Keywords to be used:**

* **Label:**Aunique string that is Used to define the name of the block or loop.
* **Statement:**It can be a loop or block.
* **Break:**Used to terminate the loop or block of code.
* **Continue:**Used to terminate or jump from the current iteration of the loop.

**Label statement with for loops:**In this section, the user will learn to assign a unique label to multiple loops. Also, we will use the break and continue keywords with the multiple loops. The below examples will demonstrate the use of labels using loops.

**Example 1:**Using the break keyword with labeled loops. Users can terminate the outer loop from the inner loop using the label.

JavaScript

**let** sum = 0, a = 1;

*// Label for outer loop*

outerloop: **while** (**true**) {

a = 1;

*// Label for inner loop*

innerloop: **while** (a < 3) {

sum += a;

**if** (sum > 12) {

*// Break outer loop from inner loop*

**break** outerloop;

}

console.log("sum = " + sum);

a++;

}

}

**Output**

sum = 1

sum = 3

sum = 4

sum = 6

sum = 7

sum = 9

sum = 10

sum = 12

**Example 2:**Using the continue keyword with labeled loops. Users can jump to the outer loop from the inner loop using the label. When the ‘ a=2 and sum < 12’ condition executes true, it doesn’t print the sum as we are terminating that iteration of the inner loop using the ‘continue’ keyword. When condition inside if statement executes true, it will jump to the outer loop.

JavaScript

**let** sum = 0, a = 1;

*// Label for outerloop*

outerloop: **while** (sum < 12) {

a = 1;

*// Label for inner loop*

innerloop: **while** (a < 3) {

sum += a;

**if** (a === 2 && sum < 12) {

*// Jump to outer loop from inner loop*

**continue** outerloop;

}

console.log("sum = " + sum + " a = " + a);

a++;

}

}

**Output**

sum = 1 a = 1

sum = 4 a = 1

sum = 7 a = 1

sum = 10 a = 1

sum = 12 a = 2

**Example 3:**Using the label statement with a block of code. Users can terminate the execution of a labeled block using the break keyword. You can observe that code after the break keyword is not executed

JavaScript

blockOfCode: {

console.log('This part will be executed');

**break** blockOfCode;

console.log('this part will not be executed');

}

console.log('out of the block');

**Output**

This part will be executed

out of the block

**Example 4:**labeled function declaration. myLabel is the label assigned to the function declaration. myLabeledFunction is the name of the function.

JavaScript

myLabel: **function** myLabeledFunction() {

console.log("This is a labeled function.");

}

*// Calling the labeled function*

myLabeledFunction();

**Output**

This is a labeled function.

1. [JS break Statement](https://www.geeksforgeeks.org/javascript-break-statement/)

JavaScript **break statement** is used to terminate the execution of the loop or the switch statement when the condition is true.

* In a switch, code breaks out and the execution of code is stopped.
* In a loop, it breaks out to the loop but the code after the loop is executed.

**Syntax:**

break;

**Using Labels**

A label reference can be used by the break statement to exit any JavaScript code block. Only a loop or a switch can be used with the break in the absence of a label.

break labelName;

**Example 1:**In this example, the switch case is executed if the condition is true then it breaks out and the next case is not checked.

* JavaScript

const fruit = "Mango";

**switch** (fruit) {

**case** "Apple":

        console.log("Apple is healthy for our body");

**break**;

**case** "Mango":

        console.log("Mango is a National fruit of India");

**break**;

**default**:

        console.log("I don't like fruits.");

}

**Output**

Mango is a National fruit of India

**Example 2:** In this example, the fruit name is apple but the given output is for the two cases. This is because of the break statement. In the case of Apple, we are not using a break statement which means the block will run for the next case also till the break statement not appear.

* JavaScript

const fruit = "Apple";

**switch** (fruit) {

**case** "Apple":

        console.log("Apple is healthy for our body");

**case** "Mango":

        console.log("Mango is a National fruit of India");

**break**;

**default**:

        console.log("I don't like fruits.");

}

**Output**

Apple is healthy for our body

Mango is a National fruit of India

**Example 3:**In this example, the loop iterate from 1 to 6 when it is equal to 4 then the condition becomes true, and code breaks out to the loop.

* Javascript

**for** (let i = 1; i < 6; i++) {

**if** (i == 4) **break**;

    console.log(i);

}

**Output**

1

2

3

**Example 4:** In this example, break statement can is used with while and do-while loop.

* JavaScript

// Using break in a while loop

let i = 1;

**while** (i <= 5) {

    console.log(i);

**if** (i === 3) {

**break**;

    }

    i++;

}

// Using break in a do-while loop

let j = 1;

**do** {

    console.log(j);

**if** (j === 3) {

**break**;

    }

    j++;

} **while** (j <= 5);

**Output**

1

2

3

1

2

3

1. [JS continue Statement](https://www.geeksforgeeks.org/javascript-continue-statement/)

The **continue statement**in Javascript is used to break the iteration of the loop and follow with the next iteration. The break in the iteration is possible only when the specified condition going to occur.

The major difference between the continue and break statement is that the break statement breaks out of the loop completely while continue is used to break one statement and iterate to the next statement.

**How does the continue statement work for different loops?**

* In a For loop, iteration goes to an updated expression which means the increment expression is first updated.
* In a While loop, it again executes the condition.

**Syntax:**

continue;

**Example 1:**In this example, we will use the continue statement in the *for loop*.

* Javascript

**for** (let i = 0; i < 11; i++) {

**if** (i % 2 == 0) **continue**;

    console.log(i);

}

**Output:**In the above example, the first increment condition is evaluated and then the condition is checked for the next iteration.

1  
3  
5  
7  
9

**Example 2:**In this example, we will use the continue statement in the while loop.

* Javascript

let i = 0;

**while** (i < 11) {

    i++;

**if** (i % 2 == 0) **continue**;

    console.log(i);

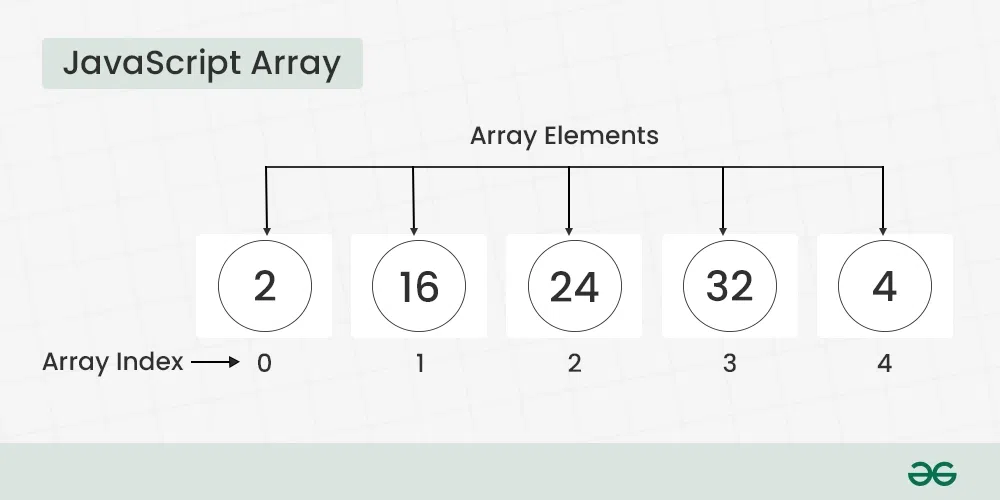
}

**Output:**In the above example, the first condition is checked, and if the condition is true then the while loop is again executed.

1  
3  
5  
7  
9  
11

1. [JS Expression and Operators](https://www.geeksforgeeks.org/javascript/#js-expression-and-operators) : The topic already coverd
   1. [JS Assignment operators](https://www.geeksforgeeks.org/javascript-assignment-operators/)
   2. [JS Comparison operators](https://www.geeksforgeeks.org/javascript-comparison-operators/)
   3. [JS Arithmetic operators](https://www.geeksforgeeks.org/javascript-arithmetic-operators/)
   4. [JS Bitwise operators](https://www.geeksforgeeks.org/javascript-bitwise-operators/)
   5. [JS Logical operators](https://www.geeksforgeeks.org/javascript-logical-operators/)
   6. [JS BigInt Operators](https://www.geeksforgeeks.org/javascript-bigint/)
   7. [JS String operators](https://www.geeksforgeeks.org/javascript-string-operators/)
   8. [JS Ternary operator](https://www.geeksforgeeks.org/javascript-ternary-operator/)
   9. [JS Comma operator](https://www.geeksforgeeks.org/javascript-comma-operator/)
   10. [JS Unary operators](https://www.geeksforgeeks.org/javascript-unary-operators/)
   11. [JS Relational operators](https://www.geeksforgeeks.org/javascript-relational-operators/)
2. [JavaScript Objects](https://www.geeksforgeeks.org/javascript/#javascript-objects)
   1. [JS Array](https://www.geeksforgeeks.org/javascript-arrays/)

An array in JavaScript is a data structure used to store multiple values in a single variable. It can hold various data types and allows for dynamic resizing. Elements are accessed by their index, starting from 0.



You have two ways to create JavaScript Arrays: **using the Array constructor**or the **shorthand array literal syntax**, which is just square brackets. Arrays are flexible in size, so they can grow or shrink as you add or remove elements.

**Table of Content**

* [Basic Terminologies of JavaScript Array](https://www.geeksforgeeks.org/javascript-arrays/#basic-terminologies-of-javascript-array)
* [Declaration of an Array](https://www.geeksforgeeks.org/javascript-arrays/#declaration-of-an-array)
* [Basic Operations on JavaScript Arrays](https://www.geeksforgeeks.org/javascript-arrays/#basic-operations-on-javascript-arrays)
* [Difference Between JavaScript Arrays and Objects](https://www.geeksforgeeks.org/javascript-arrays/#difference-between-javascript-arrays-and-objects)
* [When to use JavaScript Arrays and Objects?](https://www.geeksforgeeks.org/javascript-arrays/#when-to-use-javascript-arrays-and-objects)
* [Recognizing a JavaScript Array](https://www.geeksforgeeks.org/javascript-arrays/#recognizing-a-javascript-array)
* [JavaScript Array Complete Reference](https://www.geeksforgeeks.org/javascript-arrays/#javascript-array-complete-reference)
* [JavaScript Array Examples](https://www.geeksforgeeks.org/javascript-arrays/#javascript-array-examples)
* [JavaScript CheatSheet](https://www.geeksforgeeks.org/javascript-arrays/#javascript-cheatsheet)

**Basic Terminologies of JavaScript Array**

* **Array:** A data structure in JavaScript that allows you to store multiple values in a single variable.
* **Array Element:** Each value within an array is called an element. Elements are accessed by their index.
* **Array Index:** A numeric representation that indicates the position of an element in the array. JavaScript arrays are zero-indexed, meaning the first element is at index 0.
* **Array Length:** The number of elements in an array. It can be retrieved using the length property.

**Declaration of an Array**

There are basically two ways to declare an array i.e. Array Literal and Array Constructor.

**1. Creating an Array using Array Literal**

Creating an array using array literal involves using square brackets [] to define and initialize the array. This method is concise and widely preferred for its simplicity.

**Syntax:**

let arrayName = [value1, value2, ...];

**Example:**

JavaScript

*// Creating an Empty Array*

**let** names = [];

console.log(names);

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React"];

console.log(courses);

**Output**

[]

[ 'HTML', 'CSS', 'Javascript', 'React' ]

**2. Creating an Array using JavaScript new Keyword (Array Constructor)**

The “**Array Constructor**” refers to a method of creating arrays by invoking the Array constructor function. This approach allows for dynamic initialization and can be used to create arrays with a specified length or elements.

**Syntax:**

let arrayName = new Array();

**Example:**

javascript

*// Creating and Initializing an array with values*

**let** courses = **new** Array("HTML", "CSS", "Javascript", "React");

console.log(courses);

**Output**

[ 'HTML', 'CSS', 'Javascript', 'React' ]

***Note:****Both the above methods do exactly the same. Use the array literal method for efficiency, readability, and speed.*

**Basic Operations on JavaScript Arrays**

**1. Accessing Elements of an Array**

Any element in the array can be accessed using the index number. The index in the arrays starts with 0.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React"];

*// Accessing Array Elements*

console.log(courses[0]);

console.log(courses[1]);

**Output**

HTML

CSS

**2. Accessing the First Element of an Array**

The array indexing starts from 0, so we can access first element of array using the index number.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

*// Accessing First Array Elements*

**let** firstItem = courses[0];

console.log("First Item: ", firstItem);

**Output**

First Item: HTML

**3. Accessing the Last Element of an Array**

We can access the last array element using [array.length – 1] index number.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

*// Accessing Last Array Elements*

**let** lastItem = courses[courses.length - 1];

console.log("First Item: ", lastItem);

**Output**

First Item: React

**4. Modifying the Array Elements**

Elements in an array can be modified by assigning a new value to their corresponding index.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React"];

console.log(courses);

courses[1]= "Bootstrap";

console.log(courses);

**Output**

[ 'HTML', 'CSS', 'Javascript', 'React' ]

[ 'HTML', 'Bootstrap', 'Javascript', 'React' ]

**5. Adding Elements to the Array**

Elements can be added to the array using methods like [push()](https://www.geeksforgeeks.org/javascript-array-push-method/) and [unshift()](https://www.geeksforgeeks.org/javascript-array-unshift-method/).

* The push() method add the element to the end of the array.
* The unshift() method add the element to the starting of the array.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React"];

*// Add Element to the end of Array*

courses.push("Node.js");

*// Add Element to the beginning*

courses.unshift("Web Development");

console.log(courses);

**Output**

[ 'Web Development', 'HTML', 'CSS', 'Javascript', 'React', 'Node.js' ]

**6. Removing Elements from an Array**

To remove the elements from an array we have different methods like [pop()](https://www.geeksforgeeks.org/javascript-array-pop-method/), [shift()](https://www.geeksforgeeks.org/javascript-array-shift-method/), or [splice()](https://www.geeksforgeeks.org/javascript-array-splice-method/).

* The pop() method removes an element from the last index of the array.
* The shift() method removes the element from the first index of the array.
* The splice() method removes or replaces the element from the array.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React", "Node.js"];

console.log("Original Array: " + courses);

*// Removes and returns the last element*

**let** lastElement = courses.pop();

console.log("After Removing the last elements: " + courses);

*// Removes and returns the first element*

**let** firstElement = courses.shift();

console.log("After Removing the First elements: " + courses);

*// Removes 2 elements starting from index 1*

courses.splice(1, 2);

console.log("After Removing 2 elements starting from index 1: " + courses);

**Output**

Original Array: HTML,CSS,Javascript,React,Node.js

After Removing the last elements: HTML,CSS,Javascript,React

After Removing the First elements: CSS,Javascript,React

After Removing 2 elements starting from index 1: CSS

**7. Array Length**

We can get the length of the array using the [array length property](https://www.geeksforgeeks.org/javascript-array-length-property/).

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React", "Node.js"];

**let** len = courses.length;

console.log("Array Length: " + len);

**Output**

Array Length: 5

**8. Increase and Decrease the Array Length**

We can increase and decrease the array length using the JavaScript length property.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "Javascript", "React", "Node.js"];

*// Increase the array length to 7*

courses.length = 7;

console.log("Array After Increase the Length: ", courses);

*// Decrease the array length to 2*

courses.length = 2;

console.log("Array After Decrease the Length: ", courses)

**Output**

Array After Increase the Length: [ 'HTML', 'CSS', 'Javascript', 'React', 'Node.js', <2 empty items> ]

Array After Decrease the Length: [ 'HTML', 'CSS' ]

**9. Iterating Through Array Elements**

We can iterate array and access array elements using [for loop](https://www.geeksforgeeks.org/javascript-for-loop/) and forEach loop.

**Example:** It is an example of for loop.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

*// Iterating through for loop*

**for** (**let** i = 0; i < courses.length; i++) {

console.log(courses[i])

}

**Output**

HTML

CSS

JavaScript

React

**Example:** It is the example of [Array.forEach()](https://www.geeksforgeeks.org/javascript-array-foreach-method/" \t "_blank)loop.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

*// Iterating through forEach loop*

courses.forEach(**function** myfunc(elements) {

console.log(elements);

});

**Output**

HTML

CSS

JavaScript

React

**10. Array Concatenation**

Combine two or more arrays using the concat() method. It returns new array containing joined arrays elements.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

**let** otherCourses = ["Node.js", "Expess.js"];

*// Concatenate both arrays*

**let** concateArray = courses.concat(otherCourses);

console.log("Concatenated Array: ", concateArray);

**Output**

Concatenated Array: [ 'HTML', 'CSS', 'JavaScript', 'React', 'Node.js', 'Expess.js' ]

**11. Conversion of an Array to String**

We have a builtin method **[toString()](https://www.geeksforgeeks.org/javascript-array-tostring-method/" \t "_blank)**to converts an array to a string.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

*// Convert array ot String*

console.log(courses.toString());

**Output**

HTML,CSS,JavaScript,React

**12. Check the Type of an Arrays**

The JavaScript [typeof](https://www.geeksforgeeks.org/javascript-typeof-operator/" \t "_blank) operator is used ot check the type of an array. It returns “object” for arrays.

JavaScript

*// Creating an Array and Initializing with Values*

**let** courses = ["HTML", "CSS", "JavaScript", "React"];

*// Check type of array*

console.log(**typeof** courses);

**Output**

object

**Difference Between JavaScript Arrays and Objects**

|  |  |  |
| --- | --- | --- |
| **Feature** | **JavaScript Arrays** | **JavaScript Objects** |
| **Index Type** | Numeric indexes (0, 1, 2, …) | Named keys (strings or symbols) |
| **Order** | Ordered collection | Unordered collection |
| **Use Case** | Storing lists, sequences, ordered data | Storing data with key-value pairs, attributes |
| **Accessing Elements** | Accessed by index (e.g., arr[0]) | Accessed by key (e.g., obj[“key”]) |
| **Iteration** | Typically iterated using loops like for or forEach | Iterated using for…in, Object.keys(), or Object.entries() |
| **Size Flexibility** | Dynamic, can grow or shrink in size | Dynamic, can add or remove key-value pairs |

**When to use JavaScript Arrays and Objects?**

* Use arrays when you need numeric indexing and order matters.
* Use objects when you need named keys and the relationship between keys and values is important.

**Recognizing a JavaScript Array**

There are two methods by which we can recognize a JavaScript array:

* **By using [Array.isArray()](https://www.geeksforgeeks.org/javascript-array-isarray-method/" \t "_blank) method**
* **By using [instanceof](https://www.geeksforgeeks.org/instanceof-operator-in-javascript/" \t "_blank) method**

Below is an example showing both approaches:

JavaScript

**const** courses = ["HTML", "CSS", "Javascript"];

console.log("Using Array.isArray() method: ", Array.isArray(courses))

console.log("Using instanceof method: ", courses **instanceof** Array)

**Output**

Using Array.isArray() method: true

Using instanceof method: true

**Note:** A common error is faced while writing the arrays:

const numbers = [5]

// and

const numbers = new Array(5)

JavaScript

**const** numbers = [5]

console.log(numbers)

The above two statements are not the same.

**Output:** This statement creates an array with an element ” [5] “.

[5]

JavaScript

**const** numbers = **new** Array(5)

console.log(numbers)

**Output**

[ <5 empty items> ]

**JavaScript Arrays – FAQs**

**What is an array in JavaScript?**

*An array is a special type of object used to store multiple values in a single variable. Arrays can hold any combination of data types, including numbers, strings, objects, and even other arrays.*

**How do you create an array?**

*You can create an array using the array literal syntax or the Array constructor.*

**What is the array literal syntax?**

*The array literal syntax uses square brackets to enclose a comma-separated list of values.*

*Example: const fruits = [“apple”, “banana”, “cherry”];*

**What is the Array constructor?**

*The Array constructor creates an array by using the new Array() syntax.*

*Example: const fruits = new Array(“apple”, “banana”, “cherry”);*

**How do you access array elements?**

*You can access array elements using their index, which starts at 0 for the first element.*

*Example: const firstFruit = fruits[0];*

**How do you modify array elements?**

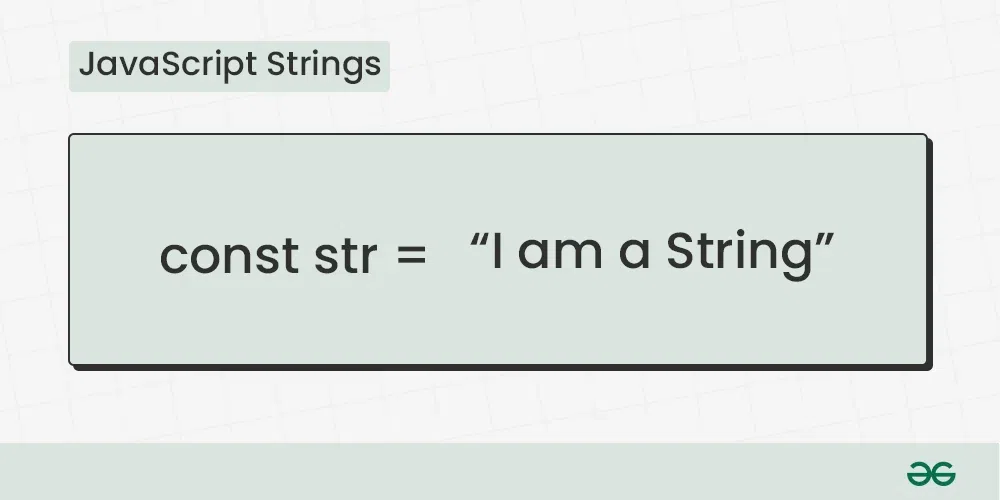
*You can modify array elements by assigning a new value to a specific index.*

*Example: fruits[0] = “orange”;*

* 1. [JS String](https://www.geeksforgeeks.org/javascript-string/)

**What is String in JavaScript?**

**JavaScript String** is a sequence of characters, typically used to represent text. It is enclosed in single or double quotes and supports various methods for text manipulation.



JavaScript strings can be created by enclosing text within either single or double quotes. You have options for creating strings using string literals or the String constructor. Strings offer flexibility for dynamic manipulation, allowing you to easily modify or extract elements as required.

**Table of Content**

* [What is String in JavaScript?](https://www.geeksforgeeks.org/javascript-strings/#what-is-string-in-javascript)
* [Basic Terminologies of JavaScript String](https://www.geeksforgeeks.org/javascript-strings/#basic-terminologies-of-javascript-string)
* [Declaration of a String](https://www.geeksforgeeks.org/javascript-strings/#declaration-of-a-string)
* [Basic Operations on JavaScript Strings](https://www.geeksforgeeks.org/javascript-strings/#basic-operations-on-javascript-strings)
* [Are the strings created by the new keyword is same as normal strings?](https://www.geeksforgeeks.org/javascript-strings/#are-the-strings-created-by-the-new-keyword-is-same-as-normal-strings)
* [JavaScript String Complete Reference](https://www.geeksforgeeks.org/javascript-strings/#javascript-string-complete-reference)

**Basic Terminologies of JavaScript String**

* **String:** A sequence of characters enclosed in single (‘ ‘) or double (” “) quotes.
* **Length:** The number of characters in a string, obtained using the length property.
* **Index:** The position of a character within a string, starting from 0.
* **Concatenation:** The process of combining two or more strings to create a new one.
* **Substring:** A portion of a string, obtained by extracting characters between specified indices.

**Declaration of a String**

**1. Using Single Quotes**

Single Quotes can be used to create a string in JavaScript. Simply enclose your text within single quotes to declare a string.

**Syntax:**

let str = 'String with single quote';

**Example:**

JavaScript

**let** str = 'Create String with Single Quote';

console.log(str);

**Output**

Create String with Single Quote

**2. Using Double Quotes**

Double Quotes can also be used to create a string in JavaScript. Simply enclose your text within double quotes to declare a string.

**Syntax:**

let str = “String with double quote”;

**Example:**

JavaScript

**let** str = "Create String with Double Quote";

console.log(str);

**Output**

Create String with Double Quote

**3. String Constructor**

You can create a string using the String Constructor. The String Constructor is less common for direct string creation, it provides additional methods for manipulating strings. Generally, using string literals is preferred for simplicity.

JavaScript

**let** str = **new** String('Create String with String Constructor');

console.log(str);

**Output**

[String: 'Create String with String Constructor']

**4. Using Template Literals (String Interpolation)**

You can create strings using Template Literals. Template literals allow you to embed expressions within backticks (`) for dynamic string creation, making it more readable and versatile.

**Syntax:**

let str = 'Template Litral String';  
let newStr = `String created using ${str}`;

**Example:**

JavaScript

**let** str = 'Template Litral String';

**let** newStr = `String created using **${**str**}**`;

console.log(newStr);

**Output**

String created using Template Litral String

**5. Empty String**

You can create an empty string by assigning either single or double quotes with no characters in between.

**Syntax:**

// Create Empty Strign with Single Quotes  
let str1 = '';  
// Create Empty Strign with Double Quotes  
let str2 = "";

**Example:**

JavaScript

**let** str1 = '';

**let** str2 = "";

console.log("Empty String with Single Quotes: " + str1);

console.log("Empty String with Double Quotes: " + str2);

**Output**

Empty String with Single Quotes:

Empty String with Double Quotes:

**6. Multiline Strings (ES6 and later)**

You can create a multiline string using backticks (“) with template literals. The backticks allows you to span the string across multiple lines, preserving the line breaks within the string.

**Syntax:**

let str = `  
 This is a  
 multiline  
 string`;

**Example:**

JavaScript

**let** str = `

This is a

multiline

string`;

console.log(str);

**Output**

This is a

multiline

string

**Basic Operations on JavaScript Strings**

**1. Finding the length of a String**

You can find the length of a string using the [length property](https://www.geeksforgeeks.org/javascript-string-length-property).

**Example:**Finding the length of a string.

JavaScript

**let** str = 'JavaScript';

**let** len = str.length;

console.log("String Length: " + len);

**Output**

String Length: 10

**2. String Concatenation**

You can combine two or more strings using [+ Operator](https://www.geeksforgeeks.org/addition-arithmetic-operator-in-javascript).

**Example:**

JavaScript

**let** str1 = 'Java';

**let** str2 = 'Script';

**let** result = str1 + str2;

console.log("Concatenated String: " + result);

**Output**

Concatenated String: JavaScript

**3. Escape Characters**

We can use escape characters in string to add single quotes, dual quotes, and backslash.

**Syntax:**

\' - Inserts a single quote  
\" - Inserts a double quote   
\\ - Inserts a backslash

**Example:**In this example we are using escape characters

JavaScript

**const** str1 = "\'GfG\' is a learning portal";

**const** str2 = "\"GfG\" is a learning portal";

**const** str3 = "\\GfG\\ is a learning portal";

console.log(str1);

console.log(str2);

console.log(str3);

**Output**

'GfG' is a learning portal

"GfG" is a learning portal

\GfG\ is a learning portal

**4. Breaking Long Strings**

We will use a backslash to break a long string in multiple lines of code.

JavaScript

**const** str = "'GeeksforGeeks' is \

a learning portal";

console.log(str);

**Output**

'GeeksforGeeks' is a learning portal

**Note:** This method might not be supported on all browsers.

**Example:** The better way to break a string is by using the string addition.

JavaScript

**const** str = "'GeeksforGeeks' is a"

+ " learning portal";

console.log(str);

**Output**

'GeeksforGeeks' is a learning portal

**5. Find Substring of a String**

We can extract a portion of a string using the [substring() method](https://www.geeksforgeeks.org/javascript-string-substring-method).

JavaScript

**let** str = 'JavaScript Tutorial';

**let** substr = str.substring(0, 10);

console.log(substr);

**Output**

JavaScript

**6. Convert String to Uppercase and Lowercase**

Convert a string to uppercase and lowercase using [toUpperCase()](https://www.geeksforgeeks.org/javascript-string-touppercase-method" \t "_blank) and [toLowerCase()](https://www.geeksforgeeks.org/javascript-string-tolowercase-method" \t "_blank) methods.

JavaScript

**let** str = 'JavaScript';

**let** upperCase = str.toUpperCase();

**let** lowerCase = str.toLowerCase();

console.log(upperCase);

console.log(lowerCase);

**Output**

JAVASCRIPT

javascript

**7. String Search in JavaScript**

Find the index of a substring within a string using [indexOf() method](https://www.geeksforgeeks.org/javascript-string-indexof-method" \t "_blank).

JavaScript

**let** str = 'Learn JavaScript at GfG';

**let** searchStr = str.indexOf('JavaScript');

console.log(searchStr);

**Output**

6

**8. String Replace in JavaScript**

Replace occurrences of a substring with another using [replace() method](https://www.geeksforgeeks.org/javascript-string-replace-method).

JavaScript

**let** str = 'Learn HTML at GfG';

**let** newStr = str.replace('HTML', 'JavaScript');

console.log(newStr);

**Output**

Learn JavaScript at GfG

**9. Trimming Whitespace from String**

Remove leading and trailing whitespaces using [trim() method](https://www.geeksforgeeks.org/javascript-string-trim-method).

JavaScript

**let** str = ' Learn JavaScript ';

**let** newStr = str.trim();

console.log(newStr);

**Output**

Learn JavaScript

**10. Access Characters from String**

Access individual characters in a string using bracket notation and [charAt() method](https://www.geeksforgeeks.org/javascript-string-charat-method" \t "_blank).

JavaScript

**let** str = 'Learn JavaScript';

**let** charAtIndex = str[6];

console.log(charAtIndex);

charAtIndex = str.charAt(6);

console.log(charAtIndex);

**Output**

J

J

**11. String Comparison in JavaScript**

There are some inbuilt methods that can be used to compare strings such as the equality operator and another like [localeCompare() method](https://www.geeksforgeeks.org/javascript-string-localecompare-method" \t "_blank).

JavaScript

**let** str1 = "John";

**let** str2 = **new** String("John");

console.log(str1 == str2);

console.log(str1.localeCompare(str2));

**Output**

true

0

**Note:**The Equality operator returns true, whereas the localeCompare method returns the difference of ASCII values.

**12. Passing JavaScript String as Objects**

We can create a JavaScript string using the new keyword.

JavaScript

**const** str = **new** String("GeeksforGeeks");

console.log(str);

**Output**

[String: 'GeeksforGeeks']

**Are the strings created by the new keyword is same as normal strings?**

**No**, the string created by the new keyword is an object and is not the same as normal strings.

JavaScript

**const** str1 = **new** String("GeeksforGeeks");

**const** str2 = "GeeksforGeeks";

console.log(str1 == str2);

console.log(str1 === str2);

**Output**

true

false

**JavaScript Strings – FAQs**

**What is a string in JavaScript?**

*A string is a sequence of characters used to represent text. Strings are one of the fundamental data types in JavaScript and are enclosed in single quotes (‘), double quotes (“), or backticks (`).*

**How do you create a string?**

*You can create a string by enclosing characters in single quotes, double quotes, or backticks.*

***Examples:***

* *‘Hello’*
* *“World”*
* *`Hello World`*

**What are template literals?**

*Template literals are strings enclosed in backticks (`) and allow for embedded expressions using ${expression}. They can span multiple lines and include interpolated variables and expressions.*

*Example: `Hello, ${name}!`*

**How do you access characters in a string?**

*You can access characters in a string using bracket notation and the index of the character. The index starts at 0 for the first character.*

*Example: str[0]*

**How do you find the length of a string?**

*You can find the length of a string using the length property.*

*Example: str.length*

**How do you concatenate strings?**

*You can concatenate strings using the + operator or the concat() method.*

*Example: str1 + str2 or str1.concat(str2)*

* 1. [JS Date](https://www.geeksforgeeks.org/javascript-date/)

The**JavaScript Date** object represents a single moment in time in a platform-independent format, encapsulating milliseconds since January 1, 1970, 00:00:00 UTC. It is fundamental for managing date and time in applications, providing methods for date arithmetic, formatting, and manipulation, essential for handling temporal data in web development.

**Understanding the Date Object**

The time value in a JavaScript Date object is measured in milliseconds since January 1, 1970, 00:00:00 UTC. The new Date() constructor initializes it, supporting parameters to specify year, month, day, hour, minute, second, and milliseconds.

**Table of Content**

* [Creating a Date Object](https://www.geeksforgeeks.org/javascript-date/#creating-a-date-object)
* [Getting Date Components](https://www.geeksforgeeks.org/javascript-date/#getting-date-components)
* [Formatting Dates](https://www.geeksforgeeks.org/javascript-date/#formatting-dates)
* [Manipulating Dates](https://www.geeksforgeeks.org/javascript-date/#manipulating-dates)

**Creating a Date Object**

Creating a Date object involves invoking the new Date() constructor, which initializes the object with the current date and time based on the system’s local time zone. The Date constructor supports various parameter options to specify a specific date and time, including year, month, day, hour, minute, second, and milliseconds.

You can create a Date object in several ways:

**Syntax**

new Date();  
new Date(value);  
new Date(dateString);  
new Date(year, month, day, hours, minutes, seconds, milliseconds);

**Parameters**

|  |  |
| --- | --- |
| **Field** | **Description** |
| value | The number of milliseconds since January 1, 1970, 00:00:00 UTC. |
| dateString | Represents a date format. |
| year | An integer representing the year, ranging from 1900 to 1999. |
| month | An integer representing the month, ranging from 0 for January to 11 for December. |
| day | An optional integer representing the day of the month. |
| hours | An optional integer representing the hour of the day. |
| minutes | An optional integer representing the minute of the time. |
| seconds | An optional integer representing the second of the time. |
| milliseconds | An optional integer representing the millisecond of the time. |

**Return Values**

It returns the present date and time if nothing as the parameter is given otherwise it returns the date format and time in which the parameter is given.

**Getting Date Components**

You can get various components of a date (such as year, month, day, hour, minute, second, etc.) using methods provided by the Date object:

**Example:** The code initializes a Date object representing the current date and time. It then retrieves various components such as year, month (zero-based), day of the month, hours, minutes, and seconds from this object. These components are stored in separate variables for further use or display.

JavaScript

**let** date = **new** Date();

**let** year = date.getFullYear();

**let** month = date.getMonth(); *// Note: Month is zero-based (0 for January, 11 for December)*

**let** day = date.getDate();

**let** hours = date.getHours();

**let** minutes = date.getMinutes();

**let** seconds = date.getSeconds();

**Formatting Dates**

Formatting dates in JavaScript can be done manually, or by using libraries like moment.js. However, with modern JavaScript, you can also achieve formatting using Intl.DateTimeFormat:

**Example:**The code initializes a `Date` object representing the current date. It then formats this date using the Intl.DateTimeFormat constructor with the locale set to ‘en-US’, displaying it in the format ‘month/day/year’.

JavaScript

**let** date = **new** Date();

**let** formattedDate = **new** Intl.DateTimeFormat('en-US').format(date);

console.log(formattedDate); *// Output: "2/23/2024" (assuming today's date is Feb 23, 2024)*

**Output**

7/8/2024

**Manipulating Dates**

You can manipulate dates using various methods provided by the Date object.

**Example:**The code initializes a Date object representing the current date. It then increments the date by 7 days using setDate(). Finally, it logs the modified date to the console.

JavaScript

**let** date = **new** Date();

date.setDate(date.getDate() + 7); *// Adds 7 days to the current date*

console.log(date);

**Output**

2024-07-15T10:25:17.602Z

**Example:** The code initializes a `Date` object with the provided parameters: year (1996), month (10 for November), day (13), hours (5), minutes (30), seconds (22), and milliseconds (0 by default). It then logs this date to the console.

javascript

*// When some numbers are taken as the parameter*

*// then they are considered as year, month, day,*

*// hours, minutes, seconds, milliseconds*

*// respectively.*

**let** A = **new** Date(1996, 10, 13, 5, 30, 22);

console.log(A);

**Output**

1996-11-13T05:30:22.000Z

Furthermore, the Date object provides a range of methods for retrieving and manipulating date and time components, such as **getFullYear(), getMonth(), getDate(), getHours(), getMinutes(), getSeconds(), and getMilliseconds()**.

The JavaScript Date object is essential for managing date and time in web applications, offering methods for arithmetic, formatting, and manipulation. It supports various parameters and provides extensive functionalities. For a complete reference, see the JavaScript Date Object Complete Reference article. Supported by all major browsers.

We have a complete list of Javascript Date object methods, to check those please go through this [JavaScript Date Object Complete Reference](https://www.geeksforgeeks.org/javascript-date-object-complete-reference/) article.

**Supported Browsers**

The browsers supported by JavaScript Date are listed below:

* [Google Chrome](https://www.geeksforgeeks.org/how-to-install-and-use-metamask-on-google-chrome/) 5.0
* [Edge](https://www.geeksforgeeks.org/microsoft-edge-browser/)12
* [Mozilla](https://www.geeksforgeeks.org/mozilla-firefox-browser/) 4.0
* [Safari](https://www.geeksforgeeks.org/apple-safari-browser/)5.0
* [Opera](https://www.geeksforgeeks.org/opera-browser/)11.1

**JavaScript Date – FAQs**

**What is the Date object in JavaScript?**

*The Date object in JavaScript is used to work with dates and times. It provides methods for creating, formatting, and manipulating dates and times.*

**How do you get the current date and time?**

*You can get the current date and time by creating a new Date object with no arguments: const now = new Date();*

**How do you get individual date and time components?**

*You can get individual date and time components using methods like getFullYear(), getMonth(), getDate(), getDay(), getHours(), getMinutes(), getSeconds(), and getMilliseconds().*

* 1. [JS Number](https://www.geeksforgeeks.org/javascript-numbers/)

JavaScript numbers are primitive data types and, unlike other programming languages, you don’t need to declare different numeric types like int, float, etc. JavaScript numbers are always stored in double-precision 64-bit binary format IEEE 754. This format stores numbers in 64 bits:

* 0-51 bits store the value (fraction)
* 52-62 bits store the exponent
* 63rd bit stores the sign

**Numeric Types in JavaScript**

In JavaScript, numbers play an important role, and understanding their behavior is essential for effective programming. Let’s explore the various aspects of numeric types in JavaScript.

**1. The Only Numeric Type**

As we know JavaScript has only one numeric type: the **double-precision 64-bit binary format IEEE 754**means that it doesn’t differentiate between integers and floating-point numbers explicitly. Instead, it uses a unified approach for all numeric values.

* Integers and floating-point numbers are both represented using this format.
* The numeric precision is **53 bits**, allowing for an accurate representation of integer values ranging from **-2^53 + 1** to **2^53 – 1**.

**2. Scientific Notation**

JavaScript allows writing extra-large or extra-small numbers using scientific (exponent) notation.

**Example:**

JavaScript

**let** a = 156e5;

**let** b = 156e-5;

console.log(a);

console.log(b);

**Output**

15600000

0.00156

**3. Integer Precision:**

Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

**Example:**

JavaScript

**let** a = 999999999999999;

**let** b = 9999999999999999;

console.log(a);

console.log(b);

**Output**

999999999999999

10000000000000000

**4. Floating Point Precision:**

Floating point arithmetic is **not always 100% accurate** due to binary representation limitations.

**Example:**

let x = 0.22 + 0.12; //x will be **0.33999999999999997**

**To solve this problem, multiply and divide:**

let x = (0.22 \* 10 + 0.12 \* 10) / 10; // x will be **0.34**

JavaScript

**let** x = 0.22 + 0.12;

**let** y = (0.22 \* 10 + 0.12 \* 10) / 10;

console.log(x);

console.log(y);

**Output**

0.33999999999999997

0.34

**5. Adding Numbers and Strings:**

* JavaScript uses the `**+**` operator for both addition and concatenation.
* Numbers are added, when strings are concatenated.

**Example:**

JavaScript

*// Adding two numbers*

**let** x = 10;

**let** y = 15;

**let** z = x + y;

console.log(z);

*// Concatenating two strings:*

**let** a = "10";

**let** b = "30";

**let** c = a + b;

console.log(c);

**Output**

25

1030

**6. Numeric Strings:**

JavaScript automatically converts the numeric strings to numbers in most operations like.

Example:

JavaScript

**let** x = "100" / "10";

**let** y = "100" \* "10";

**let** z = "100" - "10";

console.log(x);

console.log(y);

console.log(z);

**Output**

10

1000

90

**Number Literals:**

**The types of number literals You can use decimal, binary, octal, and hexadecimal.**

**1. Decimal Numbers:**

JavaScript Numbers does not have different types of numbers(ex: int, float, long, short) which other programming languages do. It has only one type of number and it can hold both with or without decimal values.

JavaScript

**let** a=33;

**let** b=3.3;

console.log(a);

console.log(b);

**Output**

33

3.3

**2. Octal Number:**

If the number starts with 0 and the following number is smaller than 8. It will be parsed as an Octal Number.

JavaScript

**let** x = 0562;

console.log(x);

**Output**

370

**3. Binary Numbers:**

They start with 0b or 0B followed by 0’s and 1’s.

JavaScript

**let** x = 0b11;

**let** y = 0B0111;

console.log(x);

console.log(y);

**Output**

3

7

**4. Hexadecimal Numbers:**

They start with 0x or 0X followed by any digit belonging (0123456789ABCDEF)

JavaScript

**let** x = 0xfff;

console.log(x);

**Output**

4095

**Number Coercion in JavaScript**

In JavaScript, **coercion** refers to the automatic or implicit conversion of values from one data type to another. When different types of operators are applied to values, JavaScript performs type coercion to ensure that the operation can proceed. Let’s explore some common examples of coercion:

**1. Undefined to NaN:**

When you perform an operation involving **undefined**, JavaScript returns **NaN**(Not-a-Number).

JavaScript

**const** result = **undefined** + 10;

console.log(result); *// NaN*

**Output**

NaN

**2. Null to 0:**

The value **null**is **coerced to 0**when used in arithmetic operations.

JavaScript

**const** total = **null** + 5;

console.log(total); *// 5*

**Output**

5

**3. Boolean to Number:**

Boolean values (true and false) are converted to numbers**: 1 for true** and**0 for false**.

JavaScript

**const** num1 = **true** + 10;

**const** num2 = **false** + 10;

console.log(num1);

console.log(num2);

**Output**

11

10

**4. String to Number**

When performing arithmetic operations, JavaScript converts strings to numbers. If the string cannot be parsed as a valid number, it returns **NaN**.

JavaScript

**const** str1 = '42';

**const** str2 = 'hello';

**const** numFromString1 = Number(str1);

**const** numFromString2 = Number(str2);

console.log(numFromString1);

console.log(numFromString2);

**Output**

42

NaN

**5. BigInts and Symbols**

Attempting to coerce **Symbol** values to numbers results in a TypeError.

JavaScript

**const** symbolValue = Symbol('mySymbol');

**const** numFromSymbol = Number(symbolValue); *// TypeError*

console.log(numFromSymbol);

**Output:**

TypeError: Cannot convert a Symbol value to a number

**Integer conversion**

Some operations such as those which work with an array, string indexes, or date/time expect integers. After performing the coercion if the number is greater than 0 it is returned as the same and if the number NaN or -0, it is returned as 0. The result is always an integer.

**Fixed-width number Conversion**

In Javascript, there are some functions that deal with the binary encoding of integers such as bitwise operators and typedArray objects. The bitwise operators always convert the operands to 32-bit integers.

**JavaScript Number Methods**

Now, we will use Number methods such as [toString()](https://www.geeksforgeeks.org/javascript-tostring-function/), [toExponential()](https://www.geeksforgeeks.org/javascript-toexponential-function/), [toPrecision()](https://www.geeksforgeeks.org/javascript-toprecision-function/), [isInteger()](https://www.geeksforgeeks.org/javascript-number-isinteger-function/), and [toLocaleString()](https://www.geeksforgeeks.org/javascript-number-tolocalestring-method/) method. Let’s see the examples of these Number methods.

JavaScript

**let** x = 21

console.log(x.toString());

console.log(x.toExponential());

console.log(x.toPrecision(4));

console.log(Number.isInteger(x));

console.log(x.toLocaleString("bn-BD"));

**Output:**

21  
2.1e+1  
21.00  
true  
২১

**Some Facts About Numbers in JavaScript**

* **String Concatenation with Numbers**: When you add a string and a number in JavaScript, the result will be a string concatenation.
* Javascript numbers which are primarily primitive values can also be defined as objects using a new keyword.
* Constants preceded by 0x are interpreted as hexadecimal in JavaScript.
* Javascript numbers are of base 10 by default, but we can use the toString() method to get output in the required base from base 2 to base 36.
* Apart from regular numbers, Javascript has BigInt numbers which are integers of arbitrary length.

*We have a complete list of Javascript Number Objects methods, to check those please go through this [Javascript Number Complete Reference](https://www.geeksforgeeks.org/javascript-number-complete-reference/) article.*

**JavaScript Numbers – FAQs**

**What are numbers in JavaScript?**

*Numbers in JavaScript are a data type used to represent both integer and floating-point values. JavaScript uses a 64-bit floating-point representation (IEEE 754) for all numeric values.*

**How do you create a number in JavaScript?**

*You can create a number by simply assigning a numeric value to a variable.*

*Example: let num = 42; or let pi = 3.14;*

**What is the difference between integers and floating-point numbers?**

* *Integers: Whole numbers without a decimal point, such as 1, 42, or -7.*
* *Floating-point numbers: Numbers with a decimal point, such as 3.14, -0.001, or 2.71828.*

**How do you round numbers?**

*You can round numbers using the methods provided by the Math object:*

* *Math.round(): Rounds to the nearest integer.*
* *Math.ceil(): Rounds up to the nearest integer.*
* *Math.floor(): Rounds down to the nearest integer.*
* *Math.trunc(): Truncates the decimal part and returns the integer part.*

**How do you generate random numbers?**

*You can generate random numbers using Math.random(), which returns a floating-point number between 0 (inclusive) and 1 (exclusive). To get a number in a specific range, you can scale and shift the value.*

* 1. [JS Math](https://www.geeksforgeeks.org/javascript-math-object/)

JavaScript **Math object** is used to perform mathematical operations on numbers. All the properties of Math are static and unlike other objects, it does not have a constructor.

We use Math only on [Number](https://www.geeksforgeeks.org/javascript-numbers/) data type and not on [BigInt](https://www.geeksforgeeks.org/bigint-in-javascript/)

**Example 1:** This example uses math object properties to return their values.

JavaScript

console.log("Math.LN10: " + Math.LN10);

console.log("Math.LOG2E: " + Math.LOG2E);

console.log("Math.Log10E: " + Math.LOG10E);

console.log("Math.SQRT2: " + Math.SQRT2);

console.log("Math.SQRT1\_2: " + Math.SQRT1\_2);

console.log("Math.LN2: " + Math.LN2);

console.log("Math.E: " + Math.E);

console.log("Math.PI: " + Math.PI);

**Output**

Math.LN10: 2.302585092994046

Math.LOG2E: 1.4426950408889634

Math.Log10E: 0.4342944819032518

Math.SQRT2: 1.4142135623730951

Math.SQRT1\_2: 0.7071067811865476

Math.LN2: 0.6931471805599453

Math.E: 2.71828...

**Example 2:** Math object methods are used in this example.

JavaScript

console.log("Math.abs(-4.7): " + Math.abs(-4.7));

console.log("Math.ceil(4.4): " + Math.ceil(4.4));

console.log("Math.floor(4.7): " + Math.floor(4.7));

console.log("Math.sin(90 \* Math.PI / 180): " +

Math.sin(90 \* Math.PI / 180));

console.log("Math.min(0, 150, 30, 20, -8, -200): " +

Math.min(0, 150, 30, 20, -8, -200));

console.log("Math.random(): " + Math.random());

**Output**

Math.abs(-4.7): 4.7

Math.ceil(4.4): 5

Math.floor(4.7): 4

Math.sin(90 \* Math.PI / 180): 1

Math.min(0, 150, 30, 20, -8, -200): -200

Math.random(): 0.7416861489868538

**Supported Browsers:**

* Chrome
* Edge
* Firefox
* Opera
* Safari

We have a complete list of JavaScript Math Object methods, to check those please go through the [JavaScript Math Complete Reference](https://www.geeksforgeeks.org/javascript-math-complete-reference/) article

**JavaScript Math Object – FAQs**

**What is the Math object in JavaScript?**

*The Math object is a built-in object that provides properties and methods for mathematical constants and functions. It is not a constructor, so all its properties and methods are static and can be called without creating a Math object instance.*

**How do you use the Math object?**

*You use the Math object by calling its properties and methods directly. For example, Math.PI for the value of π or Math.sqrt() for calculating the square root.*

**How do you generate random numbers using the Math object?**

*You can generate random numbers using Math.random(), which returns a floating-point number between 0 (inclusive) and 1 (exclusive). To generate a random number within a specific range, you can scale and shift the result.*

**How do you find the maximum or minimum of a set of numbers?**

*You can find the maximum or minimum of a set of numbers using Math.max() and Math.min() respectively. Both methods accept zero or more arguments.*

* 1. [JS Object](https://www.geeksforgeeks.org/javascript-objects/)

In our previous article on [Introduction to Object Oriented Programming in JavaScript](https://www.geeksforgeeks.org/introduction-object-oriented-programming-javascript/) we have seen all the common OOP terminology and got to know how they do or don’t exist in JavaScript. In this article, objects are discussed in detail.

**Creating Objects:**

In JavaScript, Objects can be created using two different methodologies namely Literal Form and Constructed Form.

* **Literal Form:** The literal form uses the construction of **object literals** that can be said as a collection of key-value pairs enclosed within a pair of curly braces. The syntaxial form is shown below.

let obj = {  
 key1: value1,  
 key2: value2,  
 ...  
};

* **Constructed Form:** The Constructed form uses either an object constructor function or the new keyword to create an empty object ad then adds properties to the object one by one. The syntaxial forms are shown below.
  + **Object Constructor Function:** In this methodology, the user creates an explicit function to take required values as parameters and assign them as the properties of the desired object.

function obj(value1, value2, ...) {  
 this.key1 = value1;  
 this.key2 = value2;  
 ...  
}

* **Using**[**New Keyword**](https://www.geeksforgeeks.org/javascript-new-keyword/)**:** This methodology uses the New keyword in front of any constructor method or any built-in constructor method ( such as Object, Date, String, etc) and creates a new instance of the following object by mounting it on memory.

let obj = new Object();  
obj.key1 = value1;  
obj.key2 = value2;  
...

**Differences between using Object Literals and the Constructed Form:** Both the constructed form and literal form result in creating exactly the same sort of object i.e. the end result is the same for both methodologies. The only difference between the both is that object literals can take care of several key-value pairs at once and thus is more convenient while on the other hand with the constructed-form objects, we must add the properties one-by-one in separate statements.

**Note:** It is highly uncommon to use the Constructed Form over the Object Literals for creating objects, hence for any further illustrations we will be using the object literals on most occasions.

**Built-In Objects:**

JavaScript consists of a bunch of Built-In Objects, the following list explores most of them. Although these built-ins have the appearance of being actual types or classes like in any other OOP, in JavaScript these are only functions that can be used as constructors to create objects of the particular sub-type.

* [String](https://www.geeksforgeeks.org/javascript-strings/)
* [Number](https://www.geeksforgeeks.org/javascript-numbers/)
* [Boolean](https://www.geeksforgeeks.org/javascript-boolean/)
* [Object](https://www.geeksforgeeks.org/objects-in-javascript/)
* [Function](https://www.geeksforgeeks.org/functions-in-javascript/)
* [Array](https://www.geeksforgeeks.org/arrays-in-javascript/)
* [Date](https://www.geeksforgeeks.org/javascript-date/)
* [RegExp](https://www.geeksforgeeks.org/javascript-regular-expressions/)
* Error

Now let us take an example to differentiate between Objects and Primitives.

javascript

*// Create string primitive.*

**let** strPrimitive = "GeeksforGeeks";

**typeof** strPrimitive; *// "string"*

strPrimitive **instanceof** String; *// false*

*// Use the Built-in String Function as Constructor.*

**let** strObject = **new** String( "GeeksforGeeks" );

**typeof** strObject; *// "object"*

strObject **instanceof** String; *// true*

*// inspect the object sub-type*

Object.prototype.toString.call( strObject ); *// [object String]*

In the above example, we saw that creating a string primitive didn’t create an object or an instance of a String. Primitives are literal and immutable values, to perform tasks like calculating the length or changing any character at any position we must use the Object of type String. But JavaScript is a dynamic language and luckily for the developers, JavaScript coerces a string primitive to a String class whenever any operation needs it to be. It is to be noted, that **due to internal coercion it is vastly preferred to use primitives as much as possible instead of objects**.

**Content of Objects:**

JavaScript objects consist of a set of key-value pairs, which are known as Properties. All Properties are named in JavaScript objects and the key part represents the Property name, while the value part represents the property Value. The Property Value can be of the primitive data type or an object or even a function. The property can also be globally accessible in spite of being owned by an object. The general syntax of defining an object property is as shown below,

objectName.objectProperty = propertyValue;

The following program will clear the concepts we discussed above,

javascript

**let** myObj = {

*// Integer Property.*

int\_prop: 5,

*// String Property.*

str\_prop: "GeeksforGeeks",

*// Object Property (Date).*

obj\_prop: **new** Date(),

*// Object Property.*

inner\_obj: {

int\_prop: 6

},

*// Function Property.*

func\_prop: **function**() {

console.log("Welcome to GeeksforGeeks!");

}

};

console.log(myObj.int\_prop);

console.log(myObj.str\_prop);

console.log(myObj.obj\_prop.toLocaleTimeString());

console.log(myObj.inner\_obj.int\_prop);

myObj.func\_prop();

**Output:**

5  
GeeksforGeeks  
5:47:55 PM  
6  
Welcome to GeeksforGeeks!

As per conventions, functions associated with an object are known as **methods**. This is considered to be a small difference between a function and a method. A function is an independent sequence of a bunch of statements whereas a method is associated with an object and is generally referenced by [**this keyword**](https://www.geeksforgeeks.org/javascript-this-identifier/).

**Defining Global Variables to be owned by Objects:**This is mostly done on methods, the process is fairly simple we will define our function as we are used to, and while defining the function to be a member of the object properties we will just give the name of the function as the value of one key. Let us see the example given below.

javascript

*// Define Function Explicitly.*

**function** toGreet() {

console.log("Hello There!");

}

**let** myObj = {

*// Mention Function-Name as Value.*

greet: toGreet,

*// Define Function implicitly.*

byWhom: **function**() {

console.log(" - GeeksforGeeks.org");

}

}

myObj.greet();

myObj.byWhom();

**Output:**

Hello There!  
 - GeeksforGeeks.org

**Note:**The **‘with’** keyword can be used to reference an object’s properties. The object specified as an argument to with becomes the default object for the duration of the block that follows. This is generally recommended not to be used by developers. The use of **with**is not allowed in **JavaScript strict mode.**

**Important Points:**

* Date values can only be created with their constructed object form, as they have no literal form.
* Objects, Arrays, Functions, and RegExps (regular expressions) are all objects regardless of their creation methodologies i.e. whether the literal or constructed form was used to create them.
* The constructed form may offer more customization while creating an object, this is the sole advantage over using the literal form.

With this, we can end this discussion about Objects in JavaScript and can start walking on the Path of defining and describing important topics related to objects.

**JavaScript Objects – FAQs**

**What is an object in JavaScript?**

*An object is a complex data structure that allows you to store collections of data. It is used to group related data and functionality together, consisting of properties (key-value pairs) and methods (functions).*

**How do you create an object in JavaScript?**

*You can create an object using object literals, the new Object() syntax, or by using constructor functions and classes.*

**What is an object literal?**

*An object literal is a comma-separated list of key-value pairs wrapped in curly braces. It is the most common way to create objects.*

**How do you access object properties?**

*You can access object properties using dot notation or bracket notation. Dot notation is typically used when you know the exact name of the property, while bracket notation is useful when the property name is dynamic or not a valid identifier.*

**How do you add or modify properties in an object?**

*You can add or modify properties using dot notation or bracket notation. Assign the new value to the property, whether it exists or not.*

**How do you delete properties from an object?**

*You can delete properties using the delete operator, which removes the property from the object.*

* 1. [JS Boolean](https://www.geeksforgeeks.org/javascript-boolean/)

**JavaScript Boolean** represents true or false values. It’s used for logical operations, condition testing, and variable assignments based on conditions. Values like 0, NaN, empty strings, undefined, and null are false; non-empty strings, numbers other than 0, objects, and arrays are true.

**Note:** A variable or object which has a value is treated as a **true** boolean value. ‘**0**‘, ‘NaN’, empty string, ‘undefined’, and ‘null’ is treated as **false** boolean values.

Here a1 and a2 store the boolean value i.e. true and false respectively.

let a1 = true;

let a2 = false;

**Note:** The below variables are initialized with strings, not boolean values.

let a1 ="true";

let a2 ="false";

**Boolean() function in JavaScript**

The Boolean() function in JavaScript converts any value to its corresponding Boolean representation: truthy values become true, and falsy values become false.

**Syntax:**

Boolean(variable/expression)

**Example 1:** The below program will give *true* values as output.

javascript

**function** gfg() {

console.log(Boolean(12));

}

gfg();

**Output**

true

**Example 2:** Below program will give *true* values as output.

JavaScript

console.log('Boolean(10) is ' + Boolean(10));

console.log('Boolean("GeeksforGeeks") is '+ Boolean("GeeksforGeeks"));

console.log('Boolean(2.74) is ' + Boolean(2.74));

console.log('Boolean(-1) is ' + Boolean(-1));

console.log("Boolean('true') is " + Boolean('true'));

console.log("Boolean('false') is " + Boolean('false'));

console.log('Boolean(3 \* 2 + 1.11) is '+ Boolean(3 \* 2 + 1.11));

console.log('Boolean(1<2) is ' + Boolean(1 < 2));

**Output**

Boolean(10) is true

Boolean("GeeksforGeeks") is true

Boolean(2.74) is true

Boolean(-1) is true

Boolean('true') is true

Boolean('false') is true

Boolean(3 \* 2 + 1.11) is true

Boolean(1<2) is true

**Example 3:** Below program will give *false* values as output.

javascript

**let** e; *//undefined*

console.log('Boolean(0) is ' + Boolean(0));

console.log('Boolean("") is ' + Boolean(""));

console.log('Boolean(e) undefined is '+ Boolean(e));

console.log('Boolean(-0) is ' + Boolean(-0));

console.log('Boolean(false) is ' + Boolean(**false**));

console.log('Boolean(NaN) is ' + Boolean(**NaN**));

console.log('Boolean(null) is ' + Boolean(**null**));

console.log('Boolean(1>2) is ' + Boolean(1 > 2));

**Output**

Boolean(0) is false

Boolean("") is false

Boolean(e) undefined is false

Boolean(-0) is false

Boolean(false) is false

Boolean(NaN) is false

Boolean(null) is false

Boolean(1>2) is false

**JavaScript Boolean object:**

The boolean object in javascript is an object wrapper for boolean values. Booleans in JavaScript can also be defined using the new keyword.

**Syntax:**

new Boolean(value)

Below are examples of the **JavaScript Boolean** method.

**Example 1:** Below program will give *false* values for the first 4 variables & *true* for last 2 values as output.

javascript

**let** v1 = **false**;

**let** v2 = **new** Boolean(**false**);

**let** v3 = **new** Boolean("");

**let** v4 = **new** Boolean(0);

**let** v5 = **new** Boolean(**true**);

**let** v6 = **new** Boolean("GeeksforGeeks");

console.log('v1 = ' + v1);

console.log('v2 = ' + v2);

console.log('v3 = ' + v3);

console.log('v4 = ' + v4);

console.log('v5 = ' + v5);

console.log('v6 = ' + v6);

**Output**

v1 = false

v2 = false

v3 = false

v4 = false

v5 = true

v6 = true

**Example 2:** Below program will give *true* for the first value & *false* for the second value as output.

javascript

**let** v1 = **true**;

**let** v2 = **new** Boolean(**true**);

console.log('v1 = = v2 is ' + (v1 == v2));

console.log('v1 = = = v2 is ' + (v1 === v2));

**Output**

v1 = = v2 is true

v1 = = = v2 is false

**Note:** *v1 = = = v2* is not true as the type of v1 and v2(object) is not the same.

**Supported Browsers**

* [Google Chrome](https://www.geeksforgeeks.org/how-to-install-and-use-metamask-on-google-chrome/) 5.0
* [Edge](https://www.geeksforgeeks.org/microsoft-edge-browser/)12
* [Mozilla](https://www.geeksforgeeks.org/mozilla-firefox-browser/) 4.0
* [Safari](https://www.geeksforgeeks.org/apple-safari-browser/)5.0
* [Opera](https://www.geeksforgeeks.org/opera-browser/)11.1

*We have a Cheat Sheet on Javascript where we covered all the important topics of Javascript to check those please go through [Javascript Cheat Sheet-A Basic guide to JavaScript](https://www.geeksforgeeks.org/javascript-cheat-sheet-a-basic-guide-to-javascript/).*

**JavaScript Boolean – FAQs**

**What is a Boolean in JavaScript?**

*A Boolean is a primitive data type in JavaScript that can have one of two values: true or false. It is used to represent logical values and control the flow of the program.*

**What values are considered truthy or falsy in JavaScript?**

* *Falsy values: false, 0, -0, 0n, “” (empty string), null, undefined, NaN.*
* *Truthy values: All values that are not falsy, including objects, non-zero numbers, non-empty strings, and arrays.*

**How do you use Booleans in conditional statements?**

*Booleans are commonly used in conditional statements like if, else, while, and for loops to control the flow of the program.*

**How do you compare Boolean values?**

*You can compare Boolean values using standard comparison operators (==, !=, ===, !==). The strict equality operators (===, !==) are recommended to avoid type coercion.*

**How do Boolean objects differ from Boolean primitives?**

*Boolean objects are created using the Boolean constructor and are objects, while Boolean primitives are simply true or false. Boolean objects are always truthy, even if they represent false.*

* 1. [JS JSON](https://www.geeksforgeeks.org/javascript-json/)

JSON, short for***JavaScript Object Notation***, is a way to organize data. It’s similar to XML in that it structures information, but it’s more lightweight and easier for humans to read and write. Web applications commonly use JSON to exchange data between each other.

**What is JSON?**

JSON *(JavaScript Object Notation*) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is built on two structures:

1. A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
2. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

**Why JSON?**

The fact that whenever we declare a variable and assign a value to it, it’s not the variable that holds the value but rather the variable just holds an address in the memory where the initialized value is stored. Further explaining, take for example:

let age=21;

when we use age, it gets replaced with 21, but that does not mean that age contains 21, rather what it means is that the variable age contains the address of the memory location where 21 is stored.

**How is JSON helpful?**

Well, yes, you are right! it is fine here till now but imagine you have to transfer the data and use it somewhere else (like an API maybe), so how will we share this?

One way could be to send your computer’s entire memory along with the address of the locations that are required, as you might have understood now this is not at all a good way to do it, it is risky to send your entire computer memory.

Here comes JSON to the rescue, JSON serializes the data and converts it into a human-readable and understandable format, which also makes it transferal and to be able to communicate.

**Characteristics of JSON**

* **Human-readable and writable**: JSON is easy to read and write.
* **Lightweight text-based data interchange format**: JSON is simpler to read and write when compared to XML.
* **Widely used**: JSON is a common format for data storage and communication on the web.
* **Language-independent**: Although derived from JavaScript, JSON can be used with many programming languages.

**JSON Syntax Rules**

JSON syntax is derived from JavaScript object notation syntax:

* Data is in name/value pairs Example:

{ "name":"Thanos" }

***Types of Values****:*

***Array:****An associative array of values.*

***Boolean:****True or false.*

***Number:****An integer.*

***Object:****An associative array of key/value pairs.*

***String:****Several plain text characters which usually form a word.*

Data is separated by commas Example:

{ "name":"Thanos", "Occupation":"Destroying half of humanity" }

* Curly braces hold objects Example:

let person={ "name":"Thanos", "Occupation":"Destroying half of humanity" }

* Square brackets hold arrays Example:

let person={ "name":"Thanos", "Occupation":"Destroying half of humanity", "powers": ["Can destroy anything with snap of his fingers", "Damage resistance", "Superhuman reflexes"] }

**JSON Objects**

A JSON object is a collection of key/value pairs. The keys are strings, and the values can be strings, numbers, objects, arrays, true, false, or null.

**JSON Arrays**

A JSON array is an ordered collection of values. The values can be strings, numbers, objects, arrays, true, false, or null.

**Example:**This example shows the JSON text.

javascript

{

"Avengers": [

{

"Name": "Tony stark",

"also known as": "Iron man",

"Abilities": [

"Genius",

"Billionaire",

"Playboy",

"Philanthropist"

]

},

{

"Name": "Peter parker",

"also known as": "Spider man",

"Abilities": [

"Spider web",

"Spidy sense"

]

}

]

}

**Convert a JSON Text to a JavaScript Object**

We will see how to convert a JSON text into a JavaScript Object.

**Example:** We will be using the JSON.parse() method to convert the JSON text to a JavaScript Object.

JavaScript

**let** text = '{"model":[' +

'{"carName":"Baleno","brandName":"Maruti" },' +

'{"carName":"Aura","brandName":"Hyndai" },' +

'{"carName":"Nexon","brandName":"Tata" }]}';

**const** cars = JSON.parse(text);

console.log("The car name is: " + cars.model[2].carName +

" of brand: " + cars.model[2].brandName);

**Output**

The car name is: Nexon of brand: Tata

**JSON to JavaScript Object**

To convert JSON text into a JavaScript object, you can use the JSON.parse() method as shown in the example above. This method parses the JSON string and constructs the JavaScript value or object described by the string.

**JavaScript JSON – FAQs**

**What is JSON?**

*JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is often used for transmitting data in web applications.*

**How do you create a JSON object in JavaScript?**

*A JSON object is created using JavaScript object notation. It is a text format that represents structured data.*

**How do you convert a JavaScript object to a JSON string?**

*You use the JSON.stringify() method to convert a JavaScript object to a JSON string.*

**How do you parse a JSON string to a JavaScript object?**

*You use the JSON.parse() method to parse a JSON string and convert it to a JavaScript object.*

* 1. [JS Map](https://www.geeksforgeeks.org/javascript-map/)

The**JavaScript Map** object holds key-value pairs and preserves the original insertion order. It supports any value, including objects and primitives, as keys or values. This feature allows for efficient data retrieval and manipulation, making Map a versatile tool for managing collections.

On iterating a map object returns the key, and value pair in the same order as inserted. [Map() constructor](https://www.geeksforgeeks.org/javascript-map-constructor/) is used to create Map in JavaScript.

JavaScript **Map** has a property that represents the size of the map.

**Example:**

**Input:**  
let map1 = new Map([  
 [1 , 10], [2 , 20] ,  
 [3, 30],[4, 40]  
 ]);  
   
console.log("Map1: ");  
console.log(map1);  
**Output:**  
// Map1:   
// Map(4) { 1 => 10, 2 => 20, 3 => 30, 4 => 40 }

**Steps to Create a Map**

* Passing an Array to new Map()
* Create a Map and use Map.set()

**Examples of JavaScript Map**

**new Map()**

In this we use new Map() constructor,

**Example:** In this example, a Map named prices is created to associate product names with their respective prices, allowing for efficient retrieval and management of price information.

// Creating a Map for product prices  
const prices = new Map([   
 ["Laptop", 1000],  
 ["Smartphone", 800],  
 ["Tablet", 400]   
]);

**Map.set()**

You can add elements to a Map with the set() method.

**Example:**In this example, the **Map.set()** method is employed to add product prices to the Map named prices.

// Creating a Map for product prices  
 const prices = new Map();   
// Using Map.set() to add product prices   
 prices.set('Laptop', 1000);   
 prices.set('Smartphone', 800);   
// The Map now contains { 'Laptop' => 1000, 'Smartphone' => 800 }

**Example 1:**In this example, we will create a basic map object

JavaScript

**let** map1 = **new** Map([

[1, 2],

[2, 3],

[4, 5]

]);

console.log("Map1");

console.log(map1);

**let** map2 = **new** Map([

["firstname", "sumit"],

["lastname", "ghosh"],

["website", "geeksforgeeks"]

]);

console.log("Map2");

console.log(map2);

**Output**

Map1

Map(3) { 1 => 2, 2 => 3, 4 => 5 }

Map2

Map(3) {

'firstname' => 'sumit',

'lastname' => 'ghosh',

'website' => 'geeksforgeeks'

}

**Example 2:**This example adds elements to the map using [set()](https://www.geeksforgeeks.org/javascript-map-set-method/) method.

JavaScript

**let** map1 = **new** Map();

map1.set("FirstName", "Shobhit");

map1.set("LastName", "Sharma");

map1.set("website", "GeeksforGeeks");

console.log(map1);

**Output**

Map(3) {

'FirstName' => 'Shobhit',

'LastName' => 'Sharma',

'website' => 'GeeksforGeeks'

}

**Methods of JavaScript Map**

* **set(key, value):** Adds or updates an element with a specified key and value.
* **get(key):**Returns the value associated with the specified key.
* **has(key):** Returns a boolean indicating whether an element with the specified key exists.
* **delete(key):** Removes the element with the specified key.
* **clear():** Removes all elements from the Map.
* **size:** Returns the number of key-value pairs in the Map.

This example explains the use of Map methods like [has()](https://www.geeksforgeeks.org/javascript-map-has-method/), [get()](https://www.geeksforgeeks.org/map-get-javascript/), [delete()](https://www.geeksforgeeks.org/map-delete-javascript/), and [clear()](https://www.geeksforgeeks.org/map-clear-javascript/).

JavaScript

**let** map1 = **new** Map();

map1.set("first name", "sumit");

map1.set("last name", "ghosh");

map1.set("website", "geeksforgeeks")

.set("friend 1","gourav")

.set("friend 2","sourav");

console.log(map1);

console.log("map1 has website ? "+

map1.has("website"));

console.log("map1 has friend 3 ? " +

map1.has("friend 3"));

console.log("get value for key website "+

map1.get("website"));

console.log("get value for key friend 3 "+

map1.get("friend 3"));

console.log("delete element with key website "

+ map1.**delete**("website"));

console.log("map1 has website ? "+

map1.has("website"));

console.log("delete element with key website " +

map1.**delete**("friend 3"));

map1.clear();

console.log(map1);

**Output**

Map(5) {

'first name' => 'sumit',

'last name' => 'ghosh',

'website' => 'geeksforgeeks',

'friend 1' => 'gourav',

'friend 2' => 'sourav'

}

map1 has website ? true

map1 has friend 3 ? false

get...

**Advantages of Map**

**Map** object provided by [**ES6**](https://www.geeksforgeeks.org/introduction-to-es6/). A key of a Map may occur once, which will be unique in the map’s collection. There are slight advantages to using a map rather than an object.

* **Unique Keys:** A key can occur only once, ensuring uniqueness within the collection.
* **Security:** No default keys are stored; only what is explicitly added, making it safer.
* **Flexible Key Types:**Any value (object, function, etc.) can be used as a key.
* **Order:** Maintains the order of entry insertion.
* **Size Property:** The size property makes it easy to retrieve the number of elements.
* **Performance:** Operations on Maps can be performed efficiently.
* **Serialization and Parsing:** Custom serialization and parsing support using[JSON.stringify()](https://www.geeksforgeeks.org/javascript-json-stringify-method/) and [JSON.parse()](https://www.geeksforgeeks.org/javascript-json-parse-method/" \t "_blank) methods.

JavaScript Maps provide a robust mechanism for handling key-value pairs, offering unique advantages over plain objects. With their secure, flexible, and efficient operations, Maps are an essential tool for modern web development. Coupled with the map() method for arrays, JavaScript offers versatile ways to manipulate and iterate over data collections effectively.

**Supported Browsers**

* [Google Chrome](https://www.geeksforgeeks.org/how-to-install-and-use-metamask-on-google-chrome/) 5.0
* [Edge](https://www.geeksforgeeks.org/microsoft-edge-browser/)12
* [Mozilla](https://www.geeksforgeeks.org/mozilla-firefox-browser/) 4.0
* [Safari](https://www.geeksforgeeks.org/apple-safari-browser/)5.0
* [Opera](https://www.geeksforgeeks.org/opera-browser/)11.1

**JavaScript Map – FAQs**

**What is a Map in JavaScript?**

*A Map is a built-in object that allows you to store key-value pairs. Unlike regular objects, which only allow string or symbol keys, a Map can have keys of any type, including objects, functions, and primitives.*

**How do you create a Map?**

*You can create a Map using the Map constructor.*

*Example: const myMap = new Map();*

**How do you add key-value pairs to a Map?**

*You can add key-value pairs to a Map using the set() method. This method takes two arguments: the key and the value.*

*Example: myMap.set(‘key’, ‘value’);*

**How do you get a value from a Map?**

*You can retrieve a value from a Map using the get() method. This method takes one argument, the key, and returns the associated value.*

*Example: myMap.get(‘key’);*

**How do you check if a key exists in a Map?**

*You can check if a key exists in a Map using the has() method. This method returns true if the key exists, and false otherwise.*

*Example: myMap.has(‘key’);*

**How do you remove a key-value pair from a Map?**

*You can remove a key-value pair from a Map using the delete() method. This method takes one argument, the key, and removes the associated key-value pair.*

*Example: myMap.delete(‘key’);*

* 1. [JS Set](https://www.geeksforgeeks.org/sets-in-javascript/)

**Sets in JavaScript** are collections of unique values, meaning no duplicates are allowed. They provide efficient ways to store and manage distinct elements. Sets support operations like adding, deleting, and checking the presence of items, enhancing performance for tasks requiring uniqueness.

**Syntax:**

new Set([it]);

**Parameter:**

* **it**: It is an iterable object whose all elements are added to the new set created, If the parameter is not specified or null is passed then a new set created is empty.

**Return Value:**

A new set object.

**Example:** This example shows the implementation of a JavaScript set.

JavaScript

*// ["sumit","amit","anil","anish"]*

**let** set1 = **new** Set(["sumit","sumit","amit","anil","anish"]);

*// it contains 'f', 'o', 'd'*

**let** set2 = **new** Set("fooooooood");

*// it contains [10, 20, 30, 40]*

**let** set3 = **new** Set([10, 20, 30, 30, 40, 40]);

*// it is an empty set*

**let** set4 = **new** Set();

**Table of Content**

* [Properties of Set in JavaScript:](https://www.geeksforgeeks.org/sets-in-javascript/#properties-of-set-in-javascript)
* [Methods of Set in JavaScript:](https://www.geeksforgeeks.org/sets-in-javascript/#methods-of-set-in-javascript)
  + [1. Set.add()](https://www.geeksforgeeks.org/sets-in-javascript/#1-setadd)
  + [2. Set.delete()](https://www.geeksforgeeks.org/sets-in-javascript/#2-setdelete)
  + [3. Set.clear()](https://www.geeksforgeeks.org/sets-in-javascript/#3-setclear)
  + [4. Set.entries()](https://www.geeksforgeeks.org/sets-in-javascript/#4-setentries)
  + [5. Set.has()](https://www.geeksforgeeks.org/sets-in-javascript/#5-sethas)
  + [6. Set.values()](https://www.geeksforgeeks.org/sets-in-javascript/#6-setvalues)
  + [7. Set.keys()](https://www.geeksforgeeks.org/sets-in-javascript/#7-setkeys)
  + [8. Set.forEach()](https://www.geeksforgeeks.org/sets-in-javascript/#8-setforeach)
  + [9. Set.prototype[@@iterator]()](https://www.geeksforgeeks.org/sets-in-javascript/#9-setprototypeiterator)
* [Set Operations in JavaScript](https://www.geeksforgeeks.org/sets-in-javascript/#set-operations-in-javascript)
  + [JavaScript subSet() Method:](https://www.geeksforgeeks.org/sets-in-javascript/#javascript-subset-method)
  + [JavaScript union() Method:](https://www.geeksforgeeks.org/sets-in-javascript/#javascript-union-method)
  + [JavaScript intersection() Method:](https://www.geeksforgeeks.org/sets-in-javascript/#javascript-intersection-method)
  + [JavaScript difference() Method:](https://www.geeksforgeeks.org/sets-in-javascript/#javascript-difference-method)

**Properties of Set in JavaScript:**

[**Set.size**](https://www.geeksforgeeks.org/javascript-set-size-property)– It returns the number of elements in the Set.

**Methods of Set in JavaScript:**

**1. Set.add()**

[**Set.add()**](https://www.geeksforgeeks.org/javascript-set-add-method) adds the new element with a specified *value*at the end of the Set object.

**Syntax:**

set1.add(val);

**Parameter:**

* **val:**It is a value to be added to the set.

**Return value:**The set object

**Example:** In this example, we are adding values into the set by using add() method.

JavaScript

**let** set1 = **new** Set();

set1.add(10);

set1.add(20);

*// As this method returns*

*// the set object hence chaining*

*// of add method can be done.*

set1.add(30).add(40).add(50);

console.log(set1);

**Output:**

Set(5) {10, 20, 30, 40, 50}

**2. Set.delete()**

[**Set.delete()**](https://www.geeksforgeeks.org/javascript-set-delete-method) deletes an element with the specified *value* from the Set object.

**Syntax:**

set1.delete(val);

**Parameter:**

* **val:**It is a value to be deleted from the set.

**Return value:**true if the value is successfully deleted from the set else returns false.

**Example:**In this example, we are deleting the values into the set by using delete() method.

JavaScript

**let** set1 = **new** Set("foooodiiiieee");

*// deleting e from the set*

*// it prints true*

console.log(set1.**delete**('e'));

console.log(set1);

*// deleting an element which is*

*// not in the set*

*// prints false*

console.log(set1.**delete**('g'));

**Output:**

true  
Set(4) {'f', 'o', 'd', 'i'}  
false

**3. Set.clear()**

[**Set.clear()**](https://www.geeksforgeeks.org/javascript-set-clear-method) removes all the element from the set.

**Syntax:**

set1.clear();

**Parameter:**

This method does not take any parameter

**Return value:**Undefined

**Example:** In this example, we are clearing the values into the set by using clear() method.

JavaScript

**let** set2 = **new** Set([10, 20, 30, 40, 50]);

console.log(set2);

set2.clear()

console.log(set2);

**Output:**

Set(5) {10, 20, 30, 40, 50}  
Set(0) {size: 0}

**4. Set.entries()**

[**Set.entries()**](https://www.geeksforgeeks.org/javascript-set-entries-method) returns an iterator object which contains an array having the entries of the set, in the insertion order.

**Syntax:**

set1.entries();

**Parameter:**

This method does not take any parameter

**Return value:**It returns an iterator object that contains an array of [value, value] for every element of the set, in the insertion order.

**Example:**In this example, we are using enteries() method.

JavaScript

**let** set1 = **new** Set();

set1.add(50);

set1.add(30);

set1.add(40);

set1.add(20);

set1.add(10);

*// using entries to get iterator*

**let** getEntriesArry = set1.entries();

*// each iterator is array of [value, value]*

console.log(getEntriesArry.next().value);

console.log(getEntriesArry.next().value);

console.log(getEntriesArry.next().value);

**Output:**

(2) [50, 50]  
(2) [30, 30]  
(2) [40, 40]

**5. Set.has()**

[**Set.has()**](https://www.geeksforgeeks.org/javascript-set-has-method) returns true if the specified *value* is present in the Set object.

**Syntax:**

set1.has(val);

**Parameter:**

* **val:**The value to be searched in the Set

**Return value:**True if the value is present else it returns false.

**Example:** In this example, we are checking whether the value is present in the set by using has() method.

JavaScript

**let** set1 = **new** Set();

*// adding element to the set*

set1.add(50);

set1.add(30);

console.log(set1.has(50));

console.log(set1.has(10));

**Output:**

true  
false

**6. Set.values()**

[**Set.values()**](https://www.geeksforgeeks.org/javascript-set-values-method) returns all the values from the Set in the same insertion order.

**Syntax:**

set1.values();

**Parameter:**

This method does not take any parameter

**Return value:**An iterator object that contains all the values of the set in the same order as they are inserted.

**7. Set.keys()**

[**Set.keys()**](https://www.geeksforgeeks.org/javascript-set-keys-method) also returns all the values from the Set in the insertion order.

**Note:** It is similar to thevalues() in the case of Sets

**Syntax:**

set1.keys();

**Parameter:**

This method does not take any parameter

**Return Value:** An iterator object that contains all the values of the set in the same order as they are inserted.

**Example:** In this example, we are printing all the values of the set by using keys() method.

JavaScript

**let** set1 = **new** Set();

*// adding element to the set*

set1.add(50);

set1.add(30);

set1.add(40);

set1.add("Geeks");

set1.add("GFG");

*// getting all the values*

**let** getValues = set1.values();

console.log(getValues);

**let** getKeys = set1.keys();

console.log(getKeys);

**Output:**

SetIterator {50, 30, 40, 'Geeks', 'GFG'}  
SetIterator {50, 30, 40, 'Geeks', 'GFG'}

**8. Set.forEach()**

[**Set.forEach()**](https://www.geeksforgeeks.org/javascript-set-foreach-method) executes the given *function* once for every element in the Set, in the insertion order.

**Syntax:**

set1.forEach(callback[,thisargument]);

**Parameter:**

* **callback**– It is a function that is to be executed for each element of the Set.
  + The callback function is provided with three parameters as follows:
    - the *element key*
    - the *element value*
    - the *Set object* to be traversed
* **thisargument**– Value to be used as this when executing the callback.

**Return value:**Undefined

**9. Set.prototype[@@iterator]()**

[**Set.prototype[@@iterator]()**](https://www.geeksforgeeks.org/sets-in-javascript) returns a Set iterator function which is *values()* function by default.

**Syntax:**

set1[Symbol.iterator]();

**Parameter:**

This method does not take any parameter

**Return value:** A Set iterator function and it is values() by default.

**Example:** In this example, we are iterating the values from the set.

JavaScript

**let** set1 = **new** Set(["sumit","sumit","amit","anish"]);

**let** getit = set1[Symbol.iterator]();

console.log(getit.next());

console.log(getit.next());

console.log(getit.next());

console.log(getit.next());

**Output:**

{value: 'sumit', done: false}  
{value: 'amit', done: false}  
{value: 'anish', done: false}  
{value: undefined, done: true}

**Set Operations in JavaScript**

**JavaScript subSet() Method:**

It returns true if *Set A* is a subset of *Set B*.  A *Set A* is said to be a subset of *Set B*, if all the elements of *Set A*is also present in *Set B*. Now lets implement and use the subset function.

**Example:** In this example, we are checking whether the given subset is present in the given set or not and returning the result according to it.

JavaScript

Set.prototype.subSet = **function**(otherSet)

{

*// if size of this set is greater*

*// than otherSet then it can't be*

*// a subset*

**if**(**this**.size > otherSet.size)

**return** **false**;

**else**

{

**for**(**let** elem **of** **this**)

{

*// if any of the element of*

*// this is not present in the*

*// otherset then return false*

**if**(!otherSet.has(elem))

**return** **false**;

}

**return** **true**;

}

}

*// using the subSet function*

*// Declaring different sets*

**let** setA = **new** Set([10, 20, 30]);

**let** setB = **new** Set([50, 60, 10, 20, 30, 40]);

**let** setC = **new** Set([10, 30, 40, 50]);

*// prints true*

console.log(setA.subSet(setB));

*// prints false*

console.log(setA.subSet(setC));

*// prints true*

console.log(setC.subSet(setB));

**Output:**

true  
false  
true

**JavaScript union() Method:**

It returns a Set which consists of the union of *Set A* and *Set B.*A Set is said to be a union of two sets, if it contains all elements of *Set A* as well as all elements of *Set B*, but it doesn’t contain duplicate elements.

If an element is present in both *Set A* and*Set B* then the union of Set A and B will contain a single copy of the element. Let’s implement and use the union function

**Example:** In this example, we are merging the two sets.

JavaScript

Set.prototype.union = **function**(otherSet)

{

*// creating new set to store union*

**let** unionSet = **new** Set();

*// iterate over the values and add*

*// it to unionSet*

**for** (**let** elem **of** **this**)

{

unionSet.add(elem);

}

*// iterate over the values and add it to*

*// the unionSet*

**for**(**let** elem **of** otherSet)

unionSet.add(elem);

*// return the values of unionSet*

**return** unionSet;

}

*// using the union function*

*// Declaring values for set1 and set2*

**let** set1 = **new** Set([10, 20, 30, 40, 50]);

**let** set2 = **new** Set([40, 50, 60, 70, 80]);

*// performing union operation*

*// and storing the resultant set in*

*// unionSet*

**let** unionSet = set1.union(set2);

console.log(unionSet.values());

**Output:**

SetIterator {10, 20, 30, 40, 50, …}

**JavaScript intersection() Method:**

It returns the intersection of *Set A* and *Set B.* A Set is said to be the intersection of*Set A and B* if contains an element which is present both in*Set A* and *Set B*. Let’s implement and use the intersection function

**Example:**In this example, we are finding the insersection of two sets.

JavaScript

Set.prototype.intersection = **function**(otherSet)

{

*// creating new set to store intersection*

**let** intersectionSet = **new** Set();

*// Iterate over the values*

**for**(**let** elem **of** otherSet)

{

*// if the other set contains a*

*// similar value as of value[i]*

*// then add it to intersectionSet*

**if**(**this**.has(elem))

intersectionSet.add(elem);

}

*// return values of intersectionSet*

**return** intersectionSet;

}

*// using intersection function*

*// Declaring values for set1 and set2*

**let** set1 = **new** Set([10, 20, 30, 40, 50]);

**let** set2 = **new** Set([40, 50, 60, 70, 80]);

*// performing union operation*

*// and storing the resultant set in*

*// intersectionset*

**let** intersectionSet = set1.intersection(set2);

console.log(intersectionSet.values());

**Output:**

SetIterator {40, 50}

**JavaScript difference() Method:**

It returns the Set which contains the difference between *Set A* and *Set B*. A Set is said to be a difference between *Set A and B* if it contains set of elements e which are present in *Set A* but not in *Set B*. Let’s implement and use the difference function

**Example:**In this example, we are finding the difference of two sets.

JavaScript

Set.prototype.difference = **function**(otherSet)

{

*// creating new set to store difference*

**let** differenceSet = **new** Set();

*// iterate over the values*

**for**(**let** elem **of** **this**)

{

*// if the value[i] is not present*

*// in otherSet add to the differenceSet*

**if**(!otherSet.has(elem))

differenceSet.add(elem);

}

*// returns values of differenceSet*

**return** differenceSet;

}

*// using difference function*

*// Declaring values for set1 and set2*

**let** set1 = **new** Set([10, 20, 30, 40, 50]);

**let** set2 = **new** Set([40, 50, 60, 70, 80]);

*// performing union operation*

*// and storing the resultant set in*

*// intersectionset*

**let** differenceSet = set1.difference(set2);

console.log(differenceSet);

**Output:**

Set(3) {10, 20, 30}

JavaScript is best known for web page development but it is also used in a variety of non-browser environments. You can learn JavaScript from the ground up by following this [JavaScript Tutorial](https://www.geeksforgeeks.org/javascript-tutorial) and [JavaScript Examples](https://www.geeksforgeeks.org/javascript-examples).

**Sets in JavaScript – FAQs**

**What is a Set in JavaScript?**

*A Set is a built-in object that allows you to store unique values of any type, whether primitive values or object references. Unlike arrays, Sets automatically ensure that no duplicate values are present.*

**How do you create a Set?**

*You can create a Set using the Set constructor: const mySet = new Set();*

**How do you add values to a Set?**

*You can add values to a Set using the add() method, which takes one argument, the value to be added: mySet.add(1);*

**How do you check if a value exists in a Set?**

*You can check if a value exists in a Set using the has() method, which returns true if the value is present, and false otherwise: mySet.has(1);*

**How do you remove a value from a Set?**

*You can remove a value from a Set using the delete() method, which takes one argument, the value to be removed: mySet.delete(1);*

**How do you clear all values from a Set?**

*You can remove all values from a Set using the clear() method: mySet.clear();*

**How do you get the size of a Set?**

*You can get the number of values in a Set using the size property: mySet.size;*

* 1. [JS Atomics](https://www.geeksforgeeks.org/atomics-in-javascript/)

**Atomics:**Atomics is a JavaScript object which gives atomic tasks to proceed as static strategies. Much the same as the strategies for Math object, the techniques, and properties of Atomics are additionally static. Atomics are utilized with SharedArrayBuffer objects. The Atomic activities are introduced on an Atomics module. In contrast to other worldwide articles, Atomics isn’t a constructor. Atomics can’t be utilized with another administrator or can be summoned as a capacity.

**Atomic Operations:** Atomic operations are not continuous. Multiple threads can read and write data in the memory when memory is shared. There is a loss of data if any data has changed Atomic operations ensure the data is written and accurately read by the predicted values. There is no way to change existing information until the current operation is completed and atomic operations will start.

**Methods:**

* [**Atomics.add():**](https://www.geeksforgeeks.org/atomics-add-javascript/)Adds the value provided to the current value in the array index specified. Returns the old index value.
* [**Atomics.and():**](https://www.geeksforgeeks.org/atomics-and-in-javascript/)The value AND is computed bitwise on the index of the array specified with the value provided. Returns that index’s old value.
* [**Atomics.exchange():**](https://www.geeksforgeeks.org/atomics-exchange-javascript/)Specifies a value at the array index specified. The old value is returned.
* [**Atomics.compareExchange():**](https://www.geeksforgeeks.org/atomics-compareexchange-javascript/)Specifies the value in the specified array index if the value is the same. Old value returns.
* [**Atomics.isLockFree(size):**](https://www.geeksforgeeks.org/atomics-islockfree-javascript/)Primitive optimization to determine whether locks or atomic operations are to be used. Returns true if a hardware atomic operation is carried out in the arrays of the given element size (as opposed to a lock).
* [**Atomics.load():**](https://www.geeksforgeeks.org/atomics-load-javascript/)The value returns to the array index specified.
* [Atomics.or()](https://www.geeksforgeeks.org/atomics-or-in-javascript/)**:**Bitwise OR computes the value with the given value at the specified array index. Returns the old index value.
* [**Atomics.notify():**](https://www.geeksforgeeks.org/javascript-atomics-notify-method/)Notify agents waiting for the specified array index. Returns the notified number of agents.
* [**Atomics.sub():**](https://www.geeksforgeeks.org/atomics-sub-javascript/)Deletes a value at the array index specified. Returns the old index value.
* [**Atomics.store():**](https://www.geeksforgeeks.org/atomics-store-javascript/)Save a value on the array index specified. Returns value.
* [**Atomics.wait():**](https://www.geeksforgeeks.org/javascript-atomics-wait-method/)Verifies that the specified array index still has a value and waiting or waiting times are sleeping. Returns “ok,” “not the same,” or “time-out.” If the calling agent is unable to wait, it throws an exception to an error.
* [**Atomics.xor():**](https://www.geeksforgeeks.org/atomics-xor-javascript/)Compute a bitwise XOR with the given value on the given array index. Returns the old index value.

**Example 1:**

* Javascript

**var** buffer = **new**

    // create a SharedArrayBuffer

    SharedArrayBuffer(50);

**var** a = **new** Uint8Array(buffer);

    // Initialising element at zeroth position of array with 9

    a[0] = 9;

    console.log(Atomics.load(a, 0));

    // Displaying the return value of the Atomics.store() method

    console.log(Atomics.store(a, 0, 3));

    // Displaying the updated SharedArrayBuffer

    console.log(Atomics.load(a, 0));

**Output:**

933

**Example 2:**

* Javascript

const buffer = **new** SharedArrayBuffer(2048);

    const ta = **new** Uint8Array(buffer);

    ta[0]; // 0

    ta[0] = 5; // 5

    Atomics.add(ta, 0, 12);   // 5

    Atomics.load(ta, 0);      // 17

    Atomics.and(ta, 0, 1); // 17

    Atomics.load(ta, 0); // 1

    Atomics.exchange(ta, 0, 12); // 1

    Atomics.load(ta, 0); // 12

    Atomics.compareExchange(ta, 0, 5, 12); // 1

    Atomics.load(ta, 0); // 1

    Atomics.isLockFree(1); // true

    Atomics.isLockFree(2); // true

    Atomics.or(ta, 0, 1); // 12

    Atomics.load(ta, 0);  // 13

    Atomics.store(ta, 0, 12); // 12

    Atomics.sub(ta, 0, 2); // 12

    Atomics.load(ta, 0); // 10

    Atomics.xor(ta, 0, 1); // 10

    Atomics.load(ta, 0); // 11

**Output:**

5

17

17

1

1

12

1

1

True

True

13

13

12

12

10

10

* 1. [JS BigInt](https://www.geeksforgeeks.org/javascript-bigint/)

**JavaScript BigInt** is a built-in object that represents whole numbers larger than \(2^{53} – 1\). A BigInt value, also known as a bigint primitive, is created by appending n to an integer literal or by calling the BigInt() function with an integer or string value. It allows precise arithmetic with integers beyond the safe integer limit of regular numbers.

**Syntax**

BigInt( number )

or

Appending n to end of an integer literal

**Parameters**

It accepts a single integer literal as a string that needs to be represented as BigInt.

**Return Type**

This method returns the given value as BigInt data type.

**1. Creating BigInt using BigInt() Function**

**Example:** In this example we creates BigInt numbers in decimal, hexadecimal, and binary formats, then prints each using console.log. It demonstrates different ways to represent large integers.

JavaScript

*// Parameter in decimal format*

**let** bigNum = BigInt(

"123422222222222222222222222222222222222");

console.log(bigNum);

*// Parameter in hexadecimal format*

**let** bigHex = BigInt("0x1ffffffeeeeeeeeef");

console.log(bigHex);

*// Parameter in binary format*

**let** bigBin = BigInt(

"0b1010101001010101001111111111111111");

console.log(bigBin);

**Output**

123422222222222222222222222222222222222n

36893488074118328047n

11430854655n

**2. Creating BigInt by appending n**

**Example:** In this example we creates BigInt numbers directly in decimal, hexadecimal, and binary formats, then prints each using console.log. It demonstrates various BigInt literals.

JavaScript

*// Decimal format*

**let** bigNum = 123422222222222222222222222222222222222n

console.log(bigNum)

*// Hexadecimal format*

**let** bigHex = 0x1ffffffeeeeeeeeefn

console.log(bigHex)

*// Binary format*

**let** bigBin = 0b1010101001010101001111111111111111n

console.log(bigBin)

**Output**

123422222222222222222222222222222222222n

36893488074118328047n

11430854655n

**3. Comparing BigInt other types**

A BigInt is similar to a Number in some ways, however, it cannot be used with methods of the builtin Math object and cannot be mixed with instances of Number in operations.

**Example:** Comparing BigInt with a Number.

typeof 100n === 100 // Returns false

typeof 100n == 100 // Returns true due to coercion

typeof 100n === 'bigint' // Returns true

100n < 101 // Returns true due to coercion

**Sorting**

An array can hold both primitive data types and BigInts. This allows the **sort()** method to work when both normal Number and BigInt values are present in the array.

**Example:**In this example we creates an array with both Number and BigInt types, sorts it using arr.sort(), and prints the sorted array, which will be [2, 2n, 4, 5n].

JavaScript

*// Array consisting of both*

*// Number and BigInt*

**let** arr = [4, 2, 5n, 2n]

*// Sorting the array*

arr.sort()

console.log(arr) *// [2, 2n, 4, 5n]*  
**Output**

[ 2, 2n, 4, 5n ]

**Usage Recommendation**

The following applications are not recommended to be used with BigInt due to its implementation:

* **Coercion:** Coercing between Number and BigInt can lead to loss of precision, it is recommended to only use BigInt when values greater than 253 are reasonably expected and not to coerce between the two types.
* **Cryptography:** The operations supported on BigInt are not constant time. BigInt is therefore unsuitable for use in cryptography.

**Limitations and Considerations**

* Some operators don’t support mixed types (both operands must be BigInt or neither).
* Be cautious when coercing between BigInt and regular numbers (precision may be lost).
* Unsigned right shift (**>>>**) is not supported for BigInt.

**Supported Browsers**

The browsers supporting BigInt method are listed below:

* [Google Chrome](https://www.geeksforgeeks.org/how-to-install-and-use-metamask-on-google-chrome/) 5.0
* [Edge](https://www.geeksforgeeks.org/microsoft-edge-browser/)12
* [Mozilla](https://www.geeksforgeeks.org/mozilla-firefox-browser/) 4.0
* [Safari](https://www.geeksforgeeks.org/apple-safari-browser/)5.0
* [Opera](https://www.geeksforgeeks.org/opera-browser/)11.1

**JavaScript BigInt – FAQs**

**What is BigInt in JavaScript?**

*BigInt is a built-in object that provides a way to represent whole numbers larger than the largest number JavaScript can reliably represent with the Number primitive. This is useful for applications requiring high-precision arithmetic.*

**How do you create a BigInt?**

*You can create a BigInt by appending n to the end of an integer literal or by using the BigInt() constructor.*

**How do you perform arithmetic operations with BigInt?**

*You can perform arithmetic operations using standard operators such as +, -, \*, /, and %. Both operands must be BigInt for these operations.*

**Can you mix BigInt and Number in operations?**

*No, you cannot directly mix BigInt and Number in arithmetic operations. You need to convert one type to the other explicitly.*

* 1. [JS Promise](https://www.geeksforgeeks.org/javascript-promise/)

JavaScript promises might sound a bit complicated at first, but once you get a clear understanding of them, they make working with code that takes time to complete, like fetching data from a website or waiting for a timer, much easier to manage. Let’s break down what promises are and how you can use them.

**What is a Promise?**

A promise in JavaScript is like a **container** for a **future value**. It is a way of saying, “**I don’t have this value right now**, **but I will have it later**.” Imagine you order a book online. You don’t get the book right away, but the store promises to send it to you. While you wait, you can do other things, and when the book arrives, you can read it.

In the same way, **a promise** lets you keep working with your code while waiting for something else to finish, like **loading**data from a server. When the data is ready, the promise will deliver it.

**How Does a Promise Work?**

A promise can be in one of three states:

* **Pending:**The promise is waiting for something to finish. For example, waiting for data to load from a website.
* **Fulfilled:**The promise has been completed successfully. The data you were waiting for is now available.
* **Rejected:** The promise has failed. Maybe there was a problem, like the server not responding.

When you create a promise, you write some code that will eventually tell the promise whether it was successful (fulfilled) or not (rejected).

**Syntax**

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

//do something

});

**Parameters**

* The promise constructor takes only one argument which is a callback function
* The callback function takes two arguments, *resolve* and *reject*
  + Perform operations inside the callback function and if everything went well then call resolve.
  + If desired operations do not go well then call reject.

**Creating a Promise**

Let’s see how to create the promise in JavaScript:

Here we have created a new promise using the **Promise constructor**. Inside the promise, there are two functions: **resolve** and **reject**. If everything goes well, we call resolve and pass the result. If something goes wrong, we call **reject**and **pass an error message**.

JavaScript

**let** myPromise = **new** Promise(**function**(resolve, reject) {

*// some code that takes time, like loading data*

**let** success = **true**; *// change this to false to check error*

**if** (success) {

resolve("The data has loaded successfully!");

} **else** {

reject("There was an error loading the data.");

}

});

**Using a Promise**

Once you have a promise, you can use it to do something when it’s fulfilled or rejected. You can do this using two methods: **then**and **catch**.

JavaScript

myPromise.then(**function**(message) {

*// This runs if the promise is fulfilled*

console.log(message);

}).**catch**(**function**(error) {

*// This runs if the promise is rejected*

console.log(error);

});

**Here’s what’s happening:**

* The **then**method is called when the promise is **fulfilled**. It takes a function as an **argument**, which will run when the promise is successful.
* The **catch**method is called when the promise is **rejected**. It also takes a **function**, which will run if there’s an error.

So, if the promise is successful, you will see “**The data has loaded successfully!**” in the console. If there’s an error, you will see “**There was an error loading the data.**”

**Example of Using Promise**

We will create a promise comparing two strings. If they match, resolve; otherwise, reject. Then, log success or error accordingly. Simplifies asynchronous handling in JavaScript.

JavaScript

**let** promise = **new** Promise(**function** (resolve, reject) {

**const** x = "geeksforgeeks";

**const** y = "geeksforgeeks"

**if** (x === y) {

resolve();

} **else** {

reject();

}

});

promise.

then(**function** () {

console.log('Success, You are a GEEK');

}).

**catch**(**function** () {

console.log('Some error has occurred');

});

**Output**

Success, You are a GEEK

Now, that we have learned about how we can create promise let’s see promise consumers that how we can consume them.

**Promise Consumers**

Promises can be consumed by registering functions using ***.then*** and ***.catch***methods.

**1. Promise then() Method**

[Promise method](https://www.geeksforgeeks.org/javascript-promise-then-method) is invoked when a promise is either resolved or rejected. It may also be defined as a carrier that takes data from promise and further executes it successfully.

**Parameters:** It takes two functions as parameters.

* The first function is executed if the promise is resolved and a result is received.
* The second function is executed if the promise is rejected and an error is received. (It is optional and there is a better way to handle error using *.catch() method*

**Syntax:**

.then(function(result){

//handle success

}, function(error){

//handle error

})

**Example 1:**This example shows how the then method handles when a promise is resolved

JavaScript

**let** promise = **new** Promise(**function** (resolve, reject) {

resolve('Geeks For Geeks');

})

promise

.then(**function** (successMessage) {

*//success handler function is invoked*

console.log(successMessage);

}, **function** (errorMessage) {

console.log(errorMessage);

});

**Output**

Geeks For Geeks

**Example 2:**This example shows the condition when a rejected promise is handled by second function of then method

JavaScript

**let** promise = **new** Promise(**function** (resolve, reject) {

reject('Promise Rejected')

})

promise

.then(**function** (successMessage) {

console.log(successMessage);

}, **function** (errorMessage) {

*//error handler function is invoked*

console.log(errorMessage);

});

**Output**

Promise Rejected

**2. Promise catch() Method**

[**Promise catch() Method**](https://www.geeksforgeeks.org/javascript-promise-catch-method)is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.

**Parameters:** It takes one function as a parameter.

* Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler), i.e. .catch() is just a shorthand for .then(null, errorHandler) )

**Syntax:**

.catch(function(error){

//handle error

})

**Examples 1:**This example shows the catch method handling the reject function of promise.

JavaScript

**let** promise = **new** Promise(**function** (resolve, reject) {

reject('Promise Rejected')

})

promise

.then(**function** (successMessage) {

console.log(successMessage);

})

.**catch**(**function** (errorMessage) {

*//error handler function is invoked*

console.log(errorMessage);

});

**Output**

Promise Rejected

**Why Use Promises?**

Before promises, handling code that took time to complete, like loading data, was more difficult. You had to use something called **callbacks**, which could get messy and hard to follow, especially when you had to do several things in a row. Promises make this easier by providing a clear way to work with asynchronous code (code that doesn’t run right away). They help you write code that is easier to read and maintain.

**Chaining Promises**

Sometimes, you need to do several things one after another, like**load some data**, **process**it, and then display it. With promises, you can do this by chaining then methods:

JavaScript

fetchData().then(**function**(data) {

console.log("Data received:", data);

*// Suppose this is another function that returns a promise*

**return** processData(data);

}).then(**function**(processedData) {

console.log("Processed data:", processedData);

}).**catch**(**function**(error) {

console.log("Error:", error);

});

**In this example:**

* The first then gets the data and passes it to processData.
* processData returns another promise.
* The second then handles the result of processData.
* If anything goes wrong along the way, the catch handles the error.

**Supported Browsers:**

* [Google Chrome](https://www.geeksforgeeks.org/how-to-install-and-use-metamask-on-google-chrome/) 5.0
* [Edge](https://www.geeksforgeeks.org/microsoft-edge-browser/)12
* [Mozilla](https://www.geeksforgeeks.org/mozilla-firefox-browser/) 4.0
* [Safari](https://www.geeksforgeeks.org/apple-safari-browser/)5.0
* [Opera](https://www.geeksforgeeks.org/opera-browser/)11.1

**FAQs – JavaScript Promise**

**How do Promises work in JavaScript?**

*Promises use then() and catch() methods to handle asynchronous results, allowing chaining of operations.*

**What are the states of a Promise?**

*Promises have three states: pending (initial state), fulfilled (successful completion), and rejected (failure).*

**How do you create a Promise in JavaScript?**

*Promises are created using the new Promise() constructor, which takes an executor function with resolve and reject parameters*

**What is Promise chaining?**

*Promise chaining is the practice of sequentially executing asynchronous operations using multiple then() calls on a Promise.*

**Can Promises be canceled in JavaScript?**

*Promises cannot be canceled natively, but techniques like using an external flag or a custom implementation can simulate cancellation.*

* 1. [JS Proxy](https://www.geeksforgeeks.org/javascript-proxy-handler/)

[**JavaScript**](https://www.geeksforgeeks.org/introduction-to-javascript/)**Proxy**is an object which intercepts another object and resists the fundamental operations on it. This object is mostly used when we want to hide information about one object from unauthorized access. A Proxy consists of two parts which are its target and handler. A target is a JavaScript object on which the proxy is applied and the handler object contains the function to intercept any other operation on it.

**Syntax:**

const prox = new Proxy(tar, handle)

**Parameters:**This object accepts two parameters.

* **tar:**It is the object on which the Proxy is to be applied
* **handle:**It is the object in which the intercept condition is defined

**Returns:**A proxy object.

**Example 1:**Here the method applies a Proxy object with an empty handler.

JavaScript

**let** details = {

name: "Raj",

Course: "DSA",

}

**const** prox = **new** Proxy(details, {})

console.log(prox.name);

console.log(prox.Course);

**Output:**

Raj

DSA

**Example 2:**Here the method applies a handler function to intercept calls on the target object.

JavaScript

**let** details = {

name: "Raj",

Course: "DSA",

}

**const** prox = **new** Proxy(details, {

get: **function**(){

**return** "unauthorized"

}

})

console.log(prox.name);

console.log(prox.Course);

**Output:**

unauthorized

unauthorized

**Example 3:** Here the method traps calls on the target object based on the condition.

JavaScript

**let** details = {

name: "Raj",

Course: "DSA",

}

**const** proxy = **new** Proxy(details, {

get: **function**(tar, prop){

**if**(prop == "Course"){

**return** **undefined**;

}

**return** tar[prop];

}

});

console.log(proxy.name);

console.log(proxy.Course);

**Output:**

Raj

undefined

**Example 4:**This example uses Proxy methods to delete properties.

JavaScript

**const** courseDetail = {

name: "DSA",

time: "6 months",

status: "Ongoing",

}

**const** handler = {

deleteProperty(target, prop) {

**if** (prop **in** target) {

**delete** target[prop];

console.log(`Removed: **${**prop**}**`);

}

}

};

**const** pro = **new** Proxy(courseDetail, handler);

console.log(pro.name);

**delete** pro.name

console.log(pro.name);

**Output:**

DSA

Removed: name

undefined

**Supported Browsers:**

* Chrome
* Edge
* Firefox
* Opera
* Safari

We have a complete list of JavaScript Proxy methods, to check please go through [JavaScript Proxy/handler Reference](https://www.geeksforgeeks.org/javascript-proxy-handler-reference/) article.

**JavaScript Proxy/Handler – FAQs**

**What is a Proxy in JavaScript?**

*A Proxy is a built-in object that allows you to create a custom behavior for fundamental operations on another object (called the target object). Proxies enable you to intercept and define custom behavior for operations such as property lookup, assignment, enumeration, function invocation, etc.*

**What is a Handler in JavaScript?**

*A Handler is an object that contains traps. Traps are methods that provide property access. These traps are methods that define the behavior of the proxy when an operation is performed on it.*

**How do you create a Proxy?**

*You can create a Proxy by using the new Proxy() constructor, which takes two arguments: the target object and the handler object.*

**How does the get trap work?**

*The get trap intercepts property access on the target object. It takes three arguments: the target object, the property name, and the receiver (typically the proxy itself).*

**How does the set trap work?**

*The set trap intercepts property assignment on the target object. It takes four arguments: the target object, the property name, the value to be assigned, and the receiver (typically the proxy itself).*

* 1. [JS Reflect](https://www.geeksforgeeks.org/javascript-reflect/)

JavaScript Reflect is a built-in object that gives access to other elements for interceptable operations. This object can be used to check whether an object has a particular property or to change or add new properties to an existing object. This object cannot be explicitly created using the new keyword and we cannot call it a function. Reflect has a set of static functions to perform operations.

**Syntax:**

Reflect.staticFunc()

**Parameters:**This object does not have fix parameters, it depends upon the static function being used with it

**Return Type:**The return values depend on the function being used.

**Example 1:**This example uses Reflect method to check and add properties in a particular object.

* Javascript

**var** details = {

    name: "Raj",

    course: "DSA",

    website: "geeksforgeeks.org",

}

console.log(Reflect.has(details, "course"))

Reflect.set(details, "Rating", "5");

console.log(details)

**Output:**

true

{name: 'Raj', course: 'DSA', website: 'geeksforgeeks.org', Rating: '5'}

**Example 2:**This example uses Reflect functions to construct a new object.

* Javascript

class Details {

    constructor(name, course) {

**this**.name = name;

**this**.course = course;

    }

    get fullDetails() {

**return** `${**this**.name} ${**this**.course}`;

    }

}

**var** person = ["Shobhit", "DSA"]

**var** enroll = Reflect.construct(

    Details,

    person

);

console.log(enroll **instanceof** Details);

console.log(enroll);

**Output:**

true

Details {name: 'Shobhit', course: 'DSA'}

**Example 3:**This example uses Reflect methods to freeze an array so that new elements cannot be added to it.

* Javascript

**var** arr = [];

Reflect.set(arr, 0, "Hello");

Reflect.set(arr, 1, "Welcome");

Reflect.set(arr, 2, "to");

Reflect.set(arr, 3, "GeeksforGeeks");

console.log(arr);

console.log(Reflect.isExtensible(arr))

Reflect.preventExtensions(arr);

Reflect.set(arr, 4, "DSA");

console.log(arr)

**Output:**Using the preventExtensions method new properties cannot be added to the array. This helps to freeze the array

(4) ['Hello', 'Welcome', 'to', 'GeeksforGeeks']

true

(4) ['Hello', 'Welcome', 'to', 'GeeksforGeeks']

* 1. [JS WeakMap](https://www.geeksforgeeks.org/javascript-weakmap/)

A **WeakMap in JavaScript** is a collection where keys can only be objects or non-registered symbols. It allows values of any type and doesn’t prevent the keys from being garbage collected, making its values eligible for garbage collection when their keys are collected.

**Syntax**

new WeakMap()  
new WeakMap(iter)

**Parameter:**It has only one optional parameter.

* **iter:**It is an iterable JavaScript object that implements the @@iterator method. It contains two elements where the first is key and the second is value.

**Example 1:**In this example The myGeeks function creates a WeakMap looseMap, sets objects as keys with names, assigns values, and checks if it has a specific key. Outputs the map and checks for presence of Ram.

JavaScript

**function** myGeeks() {

**let** looseMap = **new** WeakMap();

**let** Ram = {name};

**let** Raj = {name};

**let** Rahul = {name};

looseMap.set(Ram, "Ram");

looseMap.set(Raj, "Raj");

looseMap.set(Rahul, "Rahul");

console.log(looseMap);

console.log(looseMap.has(Ram))

}

myGeeks();

**Output:**

WeakMap {{…} => 'Raj', {…} => 'Rahul', {…} => 'Ram'}  
true

**Example 2:**In this example, we creates a WeakMap looseMap, sets an object Ram as a key with a value, nullifies Ram, and logs looseMap at different intervals.

JavaScript

**let** looseMap = **new** WeakMap();

**let** Ram = { name };

looseMap.set(Ram, "Ram");

console.log(looseMap);

Ram = **null**;

console.log(looseMap)

setTimeout(**function** () {

console.log(looseMap);

}, 300)

**Output:**As the reference is removed from the memory so the value in looseMap are garbage collected

WeakMap {{…} => 'Ram'}  
WeakMap {{…} => 'Ram'}  
WeakMap {}

**Supported Browsers:**

* [Google Chrome](https://www.geeksforgeeks.org/how-to-install-and-use-metamask-on-google-chrome/) 5.0
* [Edge](https://www.geeksforgeeks.org/microsoft-edge-browser/)12
* [Mozilla](https://www.geeksforgeeks.org/mozilla-firefox-browser/) 4.0
* [Safari](https://www.geeksforgeeks.org/apple-safari-browser/)5.0
* [Opera](https://www.geeksforgeeks.org/opera-browser/)11.1

We have a complete list of Javascript WeakMap methods, to check those please go through this [JavaScript WeakMap Complete Reference](https://www.geeksforgeeks.org/javascript-weakmap-complete-reference/) article.

**JavaScript WeakMap – FAQs**

**What is a WeakMap in JavaScript?**

*A WeakMap is a collection of key/value pairs where the keys are objects and the values can be arbitrary values. The primary feature of a WeakMap is that it holds “weak” references to the keys, meaning that if there are no other references to the key object, it can be garbage collected.*

**How do you create a WeakMap?**

*You can create a WeakMap using the WeakMap constructor.*

**What are the main differences between Map and WeakMap?**

* *Key Type: Map keys can be of any type, while WeakMap keys must be objects.*
* *Garbage Collection: WeakMap holds weak references to keys, allowing them to be garbage collected, whereas Map holds strong references.*
* *Iterability: Map is iterable, meaning you can loop through its entries. WeakMap is not iterable, and you cannot get a list of its keys or values.*

**How do you set and get values in a WeakMap?**

*To set a value in a WeakMap, you use the set method, and to get a value, you use the get method.*

**Can you use primitive values as keys in a WeakMap?**

*No, WeakMap keys must be objects. Using a primitive value (like a string, number, or boolean) will result in a TypeError.*

**How do you check if a WeakMap contains a specific key?**

*You can use the has method to check if a WeakMap contains a specific key.*

* 1. [JS WeakSet](https://www.geeksforgeeks.org/javascript-weakset/)

[**JavaScript**](https://www.geeksforgeeks.org/introduction-to-javascript/)**WeakSet** is used to store a collection of objects. It adapts the same properties of that of a set i.e. does not store duplicates. The major difference of a WeakSet with a set is that a WeakSet is a collection of objects and not values of some particular type.

**Syntax:**

new WeakSet(object)

**Parameters:** Here parameter “object” is an iterable object. All the elements of the iterable object are added to the WeakSet.

**Return type:**It returns a weakset object.

**Example 1:**In this example, we will create a weakSet object and add an element to it, then we will check if the element exists in the weakSet. We will use [has() method](https://www.geeksforgeeks.org/javascript-weakset-has-method/)and [add() method](https://www.geeksforgeeks.org/javascript-weakset-add-with-examples/)

javascript

**function** gfg() {

**let** weakSetObject = **new** WeakSet();

**let** objectOne = {};

*// add(value)*

weakSetObject.add(objectOne);

console.log("objectOne added");

*// has(value)*

console.log("WeakSet has objectOne : " +

weakSetObject.has(objectOne));

}

gfg();

**Output:**

objectOne added  
true

**Example 2:**In this example, we will see the working of weakSet functions also we will delete data using the [delete() method.](https://www.geeksforgeeks.org/javascript-weakset-delete-with-example/)

javascript

**let** weakSetObject = **new** WeakSet();

**let** objectOne = {};

**let** objectTwo = {};

*// add(value)*

weakSetObject.add(objectOne);

console.log("objectOne added");

weakSetObject.add(objectTwo);

console.log("objectTwo added");

*// has(value)*

console.log("WeakSet has objectTwo : " +

weakSetObject.has(objectTwo));

*// delete(value)*

weakSetObject.**delete**(objectTwo);

console.log("objectTwo deleted");

console.log("WeakSet has objectTwo : " +

weakSetObject.has(objectTwo));

**Output:**

objectOne added   
objectTwo added   
WeakSet has objectTwo : true  
objectTwo deleted  
WeakSet has objectTwo : false

**Supported Browsers:**

* Google Chrome
* Internet Explorer
* Firefox
* Apple Safari
* Opera

We have a complete list of Javascript weakSet methods, to check those please go through this [JavaScript WeakSet Reference](https://www.geeksforgeeks.org/javascript-weakset-complete-reference/) article.

**JavaScript WeakSet – FAQs**

**What is a WeakSet in JavaScript?**

*A WeakSet is a collection of objects, where each object can only appear once. It holds “weak” references to the objects, meaning if there are no other references to an object, it can be garbage collected.*

**How do you create a WeakSet?**

*You create a WeakSet using the WeakSet constructor.*

**What are the main differences between Set and WeakSet?**

* *Element Type: Set can contain any type of values (objects, primitives), whereas WeakSet can only contain objects.*
* *Garbage Collection: WeakSet holds weak references to its objects, allowing them to be garbage collected, while Set holds strong references.*
* *Iterability: Set is iterable, so you can loop through its elements. WeakSet is not iterable and does not have methods to retrieve its elements.*

**How do you add and check for objects in a WeakSet?**

*To add an object to a WeakSet, use the add method. To check if an object is in a WeakSet, use the has method.*

**Can you use primitive values in a WeakSet?**

*No, WeakSet can only contain objects. Adding a primitive value will result in a TypeError.*

**How do you remove an object from a WeakSet?**

*You can use the delete method to remove an object from a WeakSet.*

1. [JavaScript Functions](https://www.geeksforgeeks.org/javascript/#javascript-functions) : Functions in JavaScript are reusable blocks of code that perform a specific task.
   1. [JS Functions](https://www.geeksforgeeks.org/functions-in-javascript/)

A function in JavaScript is a reusable block of code that performs a specific task. You define it once, and then you can run (or “call”) it whenever you need that task done in your program.

*A JavaScript function runs when it is “called” by some part of your code.*

**Syntax**: The basic syntax to create a function in JavaScript is shown below.

function functionName(Parameter1, Parameter2, ...)

{

// Function body

}

To create a function in JavaScript, we have to first use the **keyword *function***, separated by the name of the function and parameters within parenthesis. The part of the function inside the curly braces**{}** is the body of the function.

*In javascript, functions can be used in the same way as variables for assignments, or calculations.*

**Why Functions?**

* Functions can be used multiple times, reducing redundancy.
* Break down complex problems into manageable pieces.
* Manage complexity by hiding implementation details.
* Can call themselves to solve problems recursively.

**Function Invocation**

The function code you have written will be executed whenever it is called.

* Triggered by an event (e.g., a button click by a user).
* When explicitly called from JavaScript code.
* Automatically executed, such as in self-invoking functions.

**Function Definition**

Before, using a user-defined function in JavaScript we have to create one. We can use the above syntax to create a function in JavaScript. A function definition is sometimes also termed a function declaration or function statement. Below are the rules for creating a function in JavaScript:

* Every function should begin with the keyword *function* followed by,
* A user-defined function name that should be unique,
* A list of parameters enclosed within parentheses and separated by commas,
* A list of statements composing the body of the function enclosed within curly braces {}.

**Example:**This example shows a basic declaration of a function in javascript.

JavaScript

**function** calcAddition(number1, number2) {

**return** number1 + number2;

}

console.log(calcAddition(6,9));

**Output**

15

In the above example, we have created a function named **calcAddition**,

* This function accepts two numbers as parameters and returns the addition of these two numbers.
* Accessing the function with just the function name without () will return the function object instead of the function result.

There are three ways of writing a function in JavaScript:

**Function Declaration:** It declares a function with a function keyword. The function declaration must have a function name.

**Syntax:**

function geeksforGeeks(paramA, paramB) {

// Set of statements

}

**Function Expression**

It is similar to a function declaration without the function name.[Function expressions](https://www.geeksforgeeks.org/javascript-function-expression) can be stored in a variable assignment.

**Syntax:**

let geeksforGeeks= function(paramA, paramB) {

// Set of statements

}

**Example:**This example explains the usage of the Function expression.

JavaScript

**const** square = **function** (number) {

**return** number \* number;

};

**const** x = square(4); *// x gets the value 16*

console.log(x);

**Output**

16

**Functions as Variable Values**

Functions can be used the same way as you use variables.

**Example:**

// Function to convert Fahrenheit to Celsius

function toCelsius(fahrenheit) {

return (fahrenheit - 32) \* 5/9;

}

// Using the function to convert temperature

let temperatureInFahrenheit = 77;

let temperatureInCelsius = toCelsius(temperatureInFahrenheit);

let text = "The temperature is " + temperatureInCelsius + " Celsius";

**Arrow Function:**

[**Arrow Function**](https://www.geeksforgeeks.org/es6-arrow-function) is one of the most used and efficient methods to create a function in JavaScript because of its comparatively easy implementation. It is a simplified as well as a more compact version of a regular or normal function expression or syntax.

**Syntax:**

let function\_name = (argument1, argument2 ,..) => expression

**Example:**This example describes the usage of the Arrow function.

JavaScript

**const** a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

**const** a2 = a.map(**function** (s) {

**return** s.length;

});

console.log("Normal way ", a2); *// [8, 6, 7, 9]*

**const** a3 = a.map((s) => s.length);

console.log("Using Arrow Function ", a3); *// [8, 6, 7, 9]*

**Output**

Normal way [ 8, 6, 7, 9 ]

Using Arrow Function [ 8, 6, 7, 9 ]

**Function Parameters**

Till now, we have heard a lot about function parameters but haven’t discussed them in detail. Parameters are additional information passed to a function. For example, in the above example, the task of the function *calcAddition* is to calculate the addition of two numbers. These two numbers on which we want to perform the addition operation are passed to this function as parameters. The parameters are passed to the function within parentheses after the function name and separated by commas. A function in JavaScript can have any number of parameters and also at the same time, a function in JavaScript cannot have a single parameter.

**Example:**In this example, we pass the argument to the function.

JavaScript

**function** multiply(a, b) {

b = **typeof** b !== "undefined" ? b : 1;

**return** a \* b;

}

console.log(multiply(69));

**Output**

69

**Calling Functions**

After defining a function, the next step is to call them to make use of the function. We can call a function by using the function name separated by the value of parameters enclosed between the parenthesis and a semicolon at the end. The below syntax shows how to call functions in JavaScript:

**Syntax:**

functionName( Value1, Value2, ..);

**Example:**Below is a sample program that illustrates the working of functions in JavaScript:

JavaScript

**function** welcomeMsg(name) {

**return** ("Hello " + name + " welcome to GeeksforGeeks");

}

*// creating a variable*

**let** nameVal = "Admin";

*// calling the function*

console.log(welcomeMsg(nameVal));

**Output**

Hello Admin welcome to GeeksforGeeks

**Return Statement**

There are some situations when we want to return some values from a function after performing some operations. In such cases, we can make use of the return statement in JavaScript. This is an optional statement and most of the time the last statement in a JavaScript function. Look at our first example with the function named as *calcAddition*. This function is calculating two numbers and then returns the result.

**Syntax:**The most basic syntax for using the return statement is:

return value;

The return statement begins with the keyword *return* separated by the value which we want to return from it. We can use an expression also instead of directly returning the value.

**Functions:**

* [Javascript | Arrow functions](https://www.geeksforgeeks.org/es6-arrow-function)
* [JavaScript | escape()](https://www.geeksforgeeks.org/javascript-escape)
* [JavaScript | unescape()](https://www.geeksforgeeks.org/javascript-unescape)
* [JavaScript | Window print()](https://www.geeksforgeeks.org/javascript-window-print-method)
* [Javascript | Window Blur() and Window Focus() Method](https://www.geeksforgeeks.org/javascript-window-blur-and-window-focus-method)
* [JavaScript | console.log()](https://www.geeksforgeeks.org/javascript-console-log-with-examples)
* [JavaScript | parseFloat()](https://www.geeksforgeeks.org/javascript-parsefloat-with-examples)
* [JavaScript | uneval()](https://www.geeksforgeeks.org/javascript-uneval-with-examples)
* [JavaScript | parseInt()](https://www.geeksforgeeks.org/javascript-parseint-with-examples)
* [JavaScript | match()](https://www.geeksforgeeks.org/javascript-match)
* [JavaScript | Date.parse()](https://www.geeksforgeeks.org/javascript-date-parse)
* [JavaScript | Replace() Method](https://www.geeksforgeeks.org/javascript-replace-method)
* [JavaScript | Map.get( )](https://www.geeksforgeeks.org/map-get-javascript)
* [JavaScript | Map.entries( )](https://www.geeksforgeeks.org/map-entries-javascript)
* [JavaScript | Map.clear( )](https://www.geeksforgeeks.org/map-clear-javascript)
* [JavaScript | Map.delete()](https://www.geeksforgeeks.org/map-delete-javascript)
* [JavaScript | Map.has( )](https://www.geeksforgeeks.org/map-has-in-javascript)

We have a Cheat Sheet on Javascript where we covered all the important topics of Javascript to check those please go through [Javascript Cheat Sheet-A Basic guide to JavaScript](https://www.geeksforgeeks.org/javascript-cheat-sheet-a-basic-guide-to-javascript).

**Functions in JavaScript – FAQs**

**What is a function in JavaScript?**

*A function is a reusable block of code designed to perform a particular task. Functions can take inputs, process them, and return a result.*

**How do you define a function?**

*Functions can be defined using function declarations or function expressions.*

**What is a function declaration?**

*A function declaration defines a function with the specified parameters and code block. The function can be called before it is defined due to hoisting.*

**What is a function expression?**

*A function expression defines a function inside an expression. The function can be anonymous and is not hoisted, so it cannot be called before it is defined.*

**What are arrow functions?**

*Arrow functions provide a shorter syntax for writing functions. They do not have their own this context and are not hoisted.*

*Example: const name = (parameters) => { // code to be executed };*

**How do you call a function?**

*You call a function by using its name followed by parentheses, which may include arguments.*

* 1. [JS Function Definitions](https://www.geeksforgeeks.org/javascript-function-definitions/)

JavaScript functions are declared using the**function** keyword, either as a declaration or expression. Declarations define named functions, while expressions assign functions to variables. Both enable code reuse and modularity.

**Syntax:**

* **Function Declarations:**

function functionName( parameters ) {

// Statements

};

* **Function Expressions:**

let variableName = function( parameter ) {

// Statements

};

* **Function Constructor:**

let FunctionName = new Function("parameter", "return parameter");

let variableName = FunctionName(values);

**Parameter:** It contains single parameter **functionName** which is mandatory and used to specify the name of function.

**Examples of JavaScript Function Definitions**

**Example:** This example we demonstrates a function declaration named GFG, which multiplies two numbers. The result is displayed in the paragraph element.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Declarations</**title**>

    </**head**>

    <**body** style="text-align: center">

        <**h2**>GeeksForGeeks</**h2**>

        <**p** id="geeks"></**p**>

        <**script**>

            let var1 = GFG(40, 3);

            document.getElementById(

                "geeks"

            ).innerHTML = var1;

            function GFG(num1, num2) {

                return num1 \* num2;

            }

        </**script**>

    </**body**>

</**html**>

**Output:**



**Example 2:** This example describes a function expression assigned to var1, multiplying two numbers. The result is displayed using innerHTML.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Expressions</**title**>

    </**head**>

    <**body**>

        <**h2**>GeeksForGeeks</**h2**>

        <**p** id="geeks"></**p**>

        <**script**>

            let var1 = function (num1, num2) {

                return num1 \* num2;

            };

            document.getElementById(

                "geeks"

            ).innerHTML = var1(20, 30);

        </**script**>

    </**body**>

</**html**>

**Output:**



**Example 3:** This example describes a function expression created with the Function constructor, multiplying two numbers and displaying the result in a paragraph element.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Expressions</**title**>

    </**head**>

    <**body**>

        <**h2**>GeeksForGeeks</**h2**>

        <**p** id="geeks"></**p**>

        <**script**>

            let GFG = new Function(

                "num1",

                "num2",

                "return num1 \* num2"

            );

            document.getElementById(

                "geeks"

            ).innerHTML = GFG(25, 4);

        </**script**>

    </**body**>

</**html**>

**Output:**



**Function Hoisting**

[Function hoisting](https://www.geeksforgeeks.org/javascript-hoisting/) moves function declarations to the top of their scope, allowing them to be used before declaration. Function expressions are not hoisted.

**Example:**In this example we define function hoisting by invoking a function before its declaration, displaying a welcome message from GeeksForGeeks.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Hoisting</**title**>

    </**head**>

    <**body** style="text-align: center">

        <**h1**>GeeksForGeeks</**h1**>

        <**script**>

            GeeksForGeeks();

            function GeeksForGeeks() {

                document.write(

                    "Welcome to GeeksForGeeks"

                );

            }

        </**script**>

    </**body**>

</**html**>

**Output:**



**Self-Invoking Functions**

Self-invoking functions execute automatically upon creation, without a name. Function expressions followed by () execute immediately, while function declarations cannot be invoked directly.

**Example:**In this example we define a self-invoking function that sets content in a paragraph element, showcasing its execution upon creation.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Hoisting</**title**>

    </**head**>

    <**body** style="text-align: center">

        <**h1**>GeeksForGeeks</**h1**>

        <**p** id="geeks"></**p**>

        <**script**>

            (function () {

                document.getElementById(

                    "geeks"

                ).innerHTML =

                    "GeeksForGeeks is the best way to learn";

            })();

        </**script**>

    </**body**>

</**html**>

**Output:**



**Functions are Objects**

It can describe functions as objects and have both properties and methods.

* When define function as property of an object then it is known as method to the object.
* When design a function to create new objects then it is known as object constructor.

**Example:**In this example we demonstrates the use of the arguments object to count the number of arguments passed to a function.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Hoisting</**title**>

    </**head**>

    <**body** style="text-align: center">

        <**h1**>GeeksForGeeks</**h1**>

        <**p**>Number of arguments :</**p**>

        <**p** id="geeks"></**p**>

        <**script**>

            function GeeksForGeeks(num1, num2) {

                return arguments.length;

            }

            document.getElementById(

                "geeks"

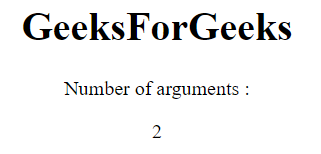
            ).innerHTML = GeeksForGeeks(4, 3);

        </**script**>

    </**body**>

</**html**>

**Output:**



**Arrow Functions**

[Arrow functions](https://www.geeksforgeeks.org/arrow-functions-in-javascript/) simplify writing function expressions by providing a concise syntax without the need for the function keyword, return keyword, or curly brackets.

**Example:**In This example we defines an arrow function to multiply two numbers and displays the result using JavaScript.

* html

<!DOCTYPE html>

<**html**>

    <**head**>

        <**title**>Function Hoisting</**title**>

    </**head**>

    <**body** style="text-align: center">

        <**h1**>GeeksForGeeks</**h1**>

        <**p** id="geeks"></**p**>

        <**script**>

            const var1 = (num1, num2) =>

                num1 \* num2;

            document.getElementById(

                "geeks"

            ).innerHTML = var1(5, 5);

        </**script**>

    </**body**>

</**html**>

**Output:**



* 1. [JS Function Call](https://www.geeksforgeeks.org/javascript-function-call/)

The**call() method** is a predefined JavaScript method. It can be used to invoke (call) a method with an owner object as an argument (parameter). This allows borrowing methods from other objects, executing them within a different context, overriding the default value, and passing arguments.

**Syntax:**

call()

**Return Value:** It calls and returns a method with the owner object being the argument.

**JavaScript Function Call Examples**

**Example 1:**In this example, we defines a product() function that returns the product of two numbers. It then calls product() using call() with `this` as the context (which is typically the global object), passing 20 and 5 as arguments. It logs the result, which is 100

JavaScript

*// function that returns product of two numbers*

**function** product(a, b) {

**return** a \* b;

}

*// Calling product() function*

**let** result = product.call(**this**, 20, 5);

console.log(result);

**Output**

100

**Example 2:**This example we defines an object “employee” with a method “details” to retrieve employee details. Using call(), it invokes “details” with “emp2” as its context, passing arguments “Manager” and “4 years”, outputting the result.

JavaScript

**let** employee = {

details: **function** (designation, experience) {

**return** **this**.name

+ " "

+ **this**.id

+ designation

+ experience;

}

}

*// Objects declaration*

**let** emp1 = {

name: "A",

id: "123",

}

**let** emp2 = {

name: "B",

id: "456",

}

**let** x = employee.details.call(emp2, " Manager ", "4 years");

console.log(x);

**Output**

B 456 Manager 4 years

* 1. [JS Function Expression](https://www.geeksforgeeks.org/javascript-function-expression/)

The Javascript **Function Expression**is used to define a function inside any expression. The Function Expressionallows us to create an anonymous function that doesn’t have any function name which is the main difference between Function Expression and Function Declaration. A function expression can be used as an [IIFE (Immediately Invoked Function Expression)](https://www.geeksforgeeks.org/javascript-immediately-invoked-function-expressions-iife/)which runs as soon as it is defined. A function expression has to be stored in a variable and can be accessed using *variableName.* With the ES6 features introducing [Arrow Function](https://www.geeksforgeeks.org/arrow-functions-in-javascript/), it becomes more easier to declare function expression.

**Syntax for Function Declaration:**

function *functionName(x, y)* { *statements...* return *(z)* };

**Syntax for Function Expression (anonymous):**

let *variableName* = function*(x, y)* { *statements...* return *(z)* };

**Syntax for Function Expression (named):**

let *variableName* = function *functionName(x, y)*

{ *statements...* return *(z)* };

**Syntax for Arrow Function:**

let *variableName* = *(x, y)* => { *statements...* return *(z)* };

**Note:**

* A function expression has to be defined first before calling it or using it as a parameter.
* An arrow function must have a return statement.

The below examples illustrate the function expression in JavaScript:

**Example 1:**Code for Function Declaration.

* Javascript

**function** callAdd(x, y) {

    let z = x + y;

**return** z;

}

console.log("Addition : " + callAdd(7, 4));

**Output:**

Addition : 11

**Example 2:**CodeforFunctionExpression**(**anonymous**)**

* Javascript

let calSub = **function** (x, y) {

    let z = x - y;

**return** z;

}

console.log("Subtraction : " + calSub(7, 4));

**Output:**

Subtraction : 3

**Example 3:** CodeforFunctionExpression**(**named**)**

* Javascript

let calMul = **function** Mul(x, y) {

    let z = x \* y;

**return** z;

}

console.log("Multiplication : " + calMul(7, 4));

**Output:**

Multiplication : 28

**Example 4:** CodeforArrowFunction

* Javascript

let calDiv = (x, y) => {

    let z = x / y;

**return** z;

}

console.log("Division : " + calDiv(24, 4));

**Output:**

Division : 6

* 1. [JS Pure Functions](https://www.geeksforgeeks.org/pure-functions-in-javascript/)

A**Pure Function** is a function (a block of code) that **always returns the same result if the same arguments are passed**. It does not depend on any state or data change during a program’s execution. Rather, it only depends on its input arguments. Also, a pure function does not produce any observable side effects such as network requests or data mutation, etc.

Let’s see the below JavaScript Function:

* Javascript

**function** calculateGST(productPrice) {

**return** productPrice \* 0.05;

}

console.log(calculateGST(15))

The above function will always return the same result if we pass the same product price. In other words, its output doesn’t get affected by any other values/state changes. So we can call the “calculate GST” function a Pure Function.

**Output:**

0.75

Now, let’s see one more function below:

* Javascript

let tax = 20;

**function** calculateGST(productPrice) {

**return** productPrice \* (tax / 100) + productPrice;

}

console.log(calculateGST(15))

Pause a second and can you guess whether the above function is Pure or not?

If you guessed that it isn’t, you are right! It is not a pure function as the output is dependent on an external variable “tax”. So if the tax value is updated somehow, then we will get a different output though we pass the same productPrice as a parameter to the function.

**Output:**

18

But here we need to make an important note:

**Note:** If a pure function calls a pure function, this isn’t a side effect, and the calling function is still considered pure. (Example: using [Math.max()](https://www.geeksforgeeks.org/javascript-math-max-method/)inside a function)

Below are some side effects (but not limited to) that a function should not produce in order to be considered a pure function –

* Making an HTTP request
* Mutating data
* Printing to a screen or console
* DOM Query/Manipulation
* [Math.random()](https://www.geeksforgeeks.org/javascript-math-random-method/)
* Getting the current time
  1. [JS Function Parameters](https://www.geeksforgeeks.org/javascript-function-parameters/)

Function parameters in JavaScript act as placeholders for values that the function can accept when it’s called.

**Syntax:**

function Name(paramet1, paramet2, paramet3,...) {

// Statements

}

**Rules:**

* There is no need to specify the data type for parameters in [JavaScript function](https://www.geeksforgeeks.org/functions-in-javascript) definitions.
* It does not perform type-checking based on the passed-in [JavaScript functions](https://www.geeksforgeeks.org/functions-in-javascript).
* It does not check the number of received arguments.

**Parameters:**

* **Name:** It is used to specify the name of the function.
* **Arguments:** It is provided in the argument field of the function.

These are the types of parameters that can be used in JavaScript

* **Defaults Parameter**
* **Function Rest Parameter**
* **Arguments Object**
* **Arguments Pass by Value**
* **Objects passed by Reference**

**JavaScript Function Parameters Examples**

**1. Defaults Parameter**

Default parameters in JavaScript are utilised to set initial values for named parameters in case no value or undefined is passed when the function is called.

**Syntax:**

function Name(paramet1 = value1, paramet2 = value2 .. .) {

// statements

}

**Example:** This example uses default parameters and performs the multiplication of numbers.

JavaScript

**function** GFG(num1, num2 = 2) {

**return** num1 \* num2;

}

console.log(GFG(4));

**Output**

8

**2. Function Rest Parameter**

In JavaScript, the rest parameter syntax enables a function to accept an unlimited number of arguments, which are then gathered into an array.

**Example:** here’s an example of using the rest parameter syntax in a function

JavaScript

**function** sum(...numbers) {

**return** numbers.reduce((acc, num) => acc + num, 0);

}

console.log(sum(1, 2, 3)); *// Output: 6*

console.log(sum(1, 2, 3, 4, 5)); *// Output: 15*

console.log(sum(10)); *// Output: 10*

console.log(sum()); *// Output: 0*

**Output**

6

15

10

0

**Explanation:**The sum function accepts any number of arguments and calculates their sum using the rest parameter ...numbers

**3. Arguments Object**

The arguments object is an inherent feature in JavaScript functions. It serves as a local variable in all non-arrow functions. You can analyze the arguments passed to a function using its arguments object.

**Example:** This example uses argument objects as parameters and finds the largest of numbers.

JavaScript

**function** GFG() {

**let** i;

**let** maxnum = -**Infinity**;

**for** (i = 0; i < arguments.length; i++) {

**if** (arguments[i] > maxnum) {

maxnum = arguments[i];

}

}

**return** maxnum;

}

console.log(GFG(10, 12, 500, 5, 440, 45));

**Output**

500

**4. Arguments Pass by Value**

In a function call, the parameters are called as arguments. The pass-by value sends the value of the variable to the function. It does not send the address of the variable. If the function changes the value of arguments then it does not affect the original value.

**Example:**This example demonstrates the above-used approach.

JavaScript

*/\* Function definition \*/*

**function** GeeksForGeeks(var1, var2) {

console.log("Inside the GeeksForGeeks function");

var1 = 100;

var2 = 200;

*/\* Display the value of variable inside function \*/*

console.log("var1 =" + var1 + " var2 =" + var2);

}

var1 = 10;

var2 = 20;

*/\* The value of variable before Function call \*/*

console.log("Before function calling");

console.log("var1 =" + var1 + " var2 =" + var2);

*/\* Function call \*/*

GeeksForGeeks(var1, var2);

*/\* The value of variable after Function call \*/*

console.log("After function calling");

console.log("var1 =" + var1 + " var2 =" + var2);

**Output**

Before function calling

var1 =10 var2 =20

Inside the GeeksForGeeks function

var1 =100 var2 =200

After function calling

var1 =10 var2 =20

**Explanation:**

* Initially, var1 is assigned the value 10 and var2 is assigned the value 20.
* When the GeeksForGeeks function is called with var1 and var2 as arguments, it modifies the values of var1 and var2 to 100 and 200 respectively within its scope.
* However, outside the function, the values of var1 and var2 remain unchanged, demonstrating that JavaScript passes arguments by value, not by reference.

**5. Objects passed by Reference**

In Pass by Reference for objects, the function receives the address of the variable rather than the value itself as the argument. If we alter the value of the variable inside the function, it affects the variables outside the function as well.

**Example:** This example demonstrates the above-used approach.

JavaScript

**function** GeeksForGeeks(varObj) {

console.log("Inside GeeksForGeeks function");

varObj.a = 100;

console.log(varObj.a);

}

*// Create object*

varObj = { a: 1 };

*/\* Display value of object before function call \*/*

console.log("Before function calling");

console.log(varObj.a);

*/\* Function calling \*/*

GeeksForGeeks(varObj)

*/\* Display value of object after function call \*/*

console.log("After function calling");

console.log(varObj.a);

**Output**

Before function calling

1

Inside GeeksForGeeks function

100

After function calling

100

**Explanation:**

* Initially, an object varObj with property a set to 1 is created.
* When the GeeksForGeeks function is called with varObj as an argument, it modifies the property a of varObj to 100 within its scope.
* After the function call, the property a of varObj remains modified outside the function, demonstrating that objects are passed by reference in JavaScript.
  1. [JS Function Invocation](https://www.geeksforgeeks.org/javascript-function-invocation/)

The [JavaScript](https://www.geeksforgeeks.org/introduction-to-javascript/) **Function Invocation** is used to execute the function code and it is common to use the term “call a function” instead of “invoke a function”. The code inside a function is executed when the function is invoked.

**Syntax:**

* **Invoking a Function as a Function:**

function myFunction( var ) {

return var;

}

myFunction( value );

* **Invoking a Function as a Method:**

let myObject = {

let : value,

functionName: function () {

return this.let;

}

}

myObject.functionName();

**Parameters:** It contains two parameters as mentioned above and described below:

* **functionName:** The functionName method is a function and this function belongs to the object and myObject is the owner of the function.
* **this:** The parameter this is the object that owns the JavaScript code and in this case the value of this is myObject.

**Example 1:** This example uses function invocation to add two numbers.

* html

<!DOCTYPE html>

<**html** lang="en">

<**head**>

    <**title**>JavaScript Function Invocation</**title**>

</**head**>

<**body** style="text-align:center;">

    <**h2** style="color:green">GeeksForGeeks</**h2**>

    <**h4**>JavaScript Function Invocation</**h4**>

    <**p**>

        Function returns the addition

        of 50 and 60

    </**p**>

    <**p** id="geeks"></**p**>

    <!-- Script to add two numbers -->

    <**script**>

        function myFunction(a, b) {

            return a + b;

        }

        document.getElementById("geeks").innerHTML

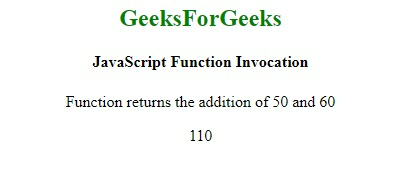
            = window.myFunction(50, 60);

    </**script**>

</**body**>

</**html**>

**Output:**



**Example 2:** This example uses function invocation to concatenate strings.

* Javascript

let myObject = {

    firstName: "Geeks",

    middleName: "for",

    lastName: "Geeks",

    fullName: **function** () {

**return** **this**.firstName + **this**.middleName

            + **this**.lastName;

    }

}

console.log(myObject.fullName());

**Output**

GeeksforGeeks

* 1. [JS Anonymous Functions](https://www.geeksforgeeks.org/javascript-anonymous-functions/)

**What are Anonymous Functions?**

An anonymous function is simply a function that does **not have a name**. Unlike named functions, which are declared with a name for easy reference, anonymous functions are usually created for specific tasks and are often assigned to variables or used as arguments for other functions.

In JavaScript, you normally use the **function keyword followed by a name** to declare a function. However, in an anonymous function, the name is **omitted**. These functions are often used in situations where you don’t need to reuse the function outside its immediate context.

**Syntax**

The below-enlightened syntax illustrates the declaration of an anonymous function using the normal declaration:

function() {

// Function Body

}

We may also declare an anonymous function using the arrow function technique which is shown below:

( () => {

// Function Body...

} )();

The below examples demonstrate anonymous functions.

**Example 1:**In this example, we define an anonymous function that prints a message to the console. The function is then stored in the *greet*variable. We can call the function by invoking *greet().*

JavaScript

<script>

**var** greet = **function** () {

console.log("Welcome to GeeksforGeeks!");

};

greet();

</script>

**Output:**

Welcome to GeeksforGeeks!

**Example 2:**In this example, we pass arguments to the anonymous function.

JavaScript

<script>

**var** greet = **function** (platform) {

console.log("Welcome to ", platform);

};

greet("GeeksforGeeks!");

</script>

**Output:**

Welcome to GeeksforGeeks!

As JavaScript supports Higher-Order Functions, we can also pass anonymous functions as parameters into another function.

**Example 3:**In this example, we pass an anonymous function as a callback function to the *[setTimeout()](https://www.geeksforgeeks.org/java-script-settimeout-setinterval-method/)*method. This executes this anonymous function 2000ms later.

JavaScript

<script>

setTimeout(**function** () {

console.log("Welcome to GeeksforGeeks!");

}, 2000);

</script>

**Output:**

Welcome to GeeksforGeeks!

**Self-Executing Anonymous Functions**

Another common use of anonymous functions is to create self-executing functions (also known as IIFE – Immediately Invoked Function Expressions). These functions run immediately after they are defined.

**Example 4:**In this example, we have created a self-executing function.

JavaScript

<script>

(**function** () {

console.log("Welcome to GeeksforGeeks!");

})();

</script>

**Output:**

Welcome to GeeksforGeeks!

**Arrow functions**

ES6 introduced a new and shorter way of declaring an anonymous function, which is known as [**Arrow Functions.**](https://www.geeksforgeeks.org/arrow-functions-in-javascript/)In an Arrow function, everything remains the same, except here we don’t need the *function*keyword also. Here, we define the function by a single parenthesis and then ‘=>’ followed by the function body.

**Example 5:**In this example, we will see the use of arrow function.

JavaScript

<script>

**var** greet = () =>

{

console.log("Welcome to GeeksforGeeks!");

}

greet();

</script>

**Output:**

Welcome to GeeksforGeeks!

If we have only a single statement in the function body, we can even remove the curly braces.

**Example 6:**In this example, we create a self-executing function.

JavaScript

<script>

**let** greet = () => console.log("Welcome to GeeksforGeeks!");

greet();

</script>

**Output:**

Welcome to Geeksforgeeks!

**Example 7:**In this example, we will declare a self-executing anonymous function (without the name itself) and will see how we may declare it as well as how we may call it in order to print the resultant value.

JavaScript

<script>

(() => {

console.log("GeeksforGeeks");

})();

</script>

**Output:**

GeeksforGeeks

* 1. [JS Arrow functions](https://www.geeksforgeeks.org/arrow-functions-in-javascript/)

ES6 introduced the Arrow functions in JavaScript which offer a more concise and readable way to write function expressions. They use the ***=>*** (arrow) syntax, which not only reduces boilerplate but also binds this lexically, making them particularly useful in certain scenarios like handling **callbacks**or working within **objects**.

**What is Arrow Function?**

An **arrow function** is essentially an anonymous function with a shorter syntax. They are often assigned to variables, making them reusable. Arrow functions are also known as **lambda functions** in some other programming languages.

**Syntax**

const gfg = () => {

console.log( "Hi Geek!" );

}

**The below examples show the working of the Arrow functions in JavaScript.**

**1. Arrow Function without Parameters**

An arrow function without parameters is defined using empty parentheses (). This is useful when you need a function that doesn’t require any arguments.

**Example:** In this example we Defines an arrow function gfg without parameters that logs “Hi from GeekforGeeks!” when called.

JavaScript

**const** gfg = () => {

console.log( "Hi from GeekforGeeks!" );

}

gfg();

**Output**

Hi from GeekforGeeks!

**2. Arrow Function with Single Parameters**

If your arrow function has a single parameter, you can omit the parentheses around it.

**Example:** In this example we defines an arrow function square with a single parameter x, returning the square of x.

JavaScript

**const** square = x => x\*x;

console.log(square(4));

*// output: 16*

**Output**

16

**3. Arrow Function with Multiple Parameters**

Arrow functions with multiple parameters, like **(param1, param2) => { }**, simplify writing concise function expressions in JavaScript, useful for functions requiring more than one argument.

**Example :**In this example we defines an arrow function gfg with parameters x, y, z, logging their sum.

JavaScript

**const** gfg = ( x, y, z ) => {

console.log( x + y + z )

}

gfg( 10, 20, 30 );

**Output**

60

**4. Arrow Function with Default Parameters**

Arrow functions support default parameters, allowing predefined values if no argument is passed, making JavaScript function definitions more flexible and concise.

**Example :** In this example we defines an arrow function gfg with parameters x, y, and a default parameter z = 30.

JavaScript

**const** gfg = ( x, y, z = 30 ) => {

console.log( x + " " + y + " " + z);

}

gfg( 10, 20 );

**Output**

10 20 30

**5. Return Object Literals**

In JavaScript, returning object literals within functions is concise: **() => ({ key: value })** returns an object { key: value }, useful for immediate object creation and returning.

**Example :**In this example we defines an arrow function makePerson with parameters firstName, lastName, returning an object.

JavaScript

**const** makePerson = (firstName, lastName) =>

({first: firstName, last: lastName});

console.log(makePerson("Pankaj", "Bind"));

**Output**

{ first: 'Pankaj', last: 'Bind' }

**Async Arrow Functions**

Arrow functions can be made asynchronous by adding the async keyword before the parameter list.

const fetchData = async () => {  
 const data = await fetch('https://api.example.com/data');  
 return data.json();  
};

**Advantages of Arrow Functions**

* **Concise Syntax:** Arrow functions reduce the amount of code needed for function expressions.
* **Lexical *this*Binding:** Arrow functions automatically bind this to the surrounding context, eliminating common issues when dealing with callbacks.
* **Improved Readability:** For shorter functions, arrow syntax can make your code more readable.

**Limitations of Arrow Functions**

* **No prototype Property:**Arrow functions do not have the prototype property, so they cannot be used as constructors.
* **Cannot be Used with *new*:** Since they lack a prototype, they cannot be used with the new keyword to create instances.
* **Cannot be Generators:**Arrow functions cannot be used as generator functions (function\*) because they do not support the yield keyword.
* **Anonymous Nature:** Debugging can be harder because arrow functions are anonymous by default.
* **No Own this, arguments, super, or new.target:** Arrow functions do not have their own bindings for these properties, which can limit their use in some cases.

**FAQs- Arrow functions in JavaScript**

**How do Arrow Functions differ from regular functions?**

*Arrow Functions have a shorter syntax, lexically bind this, and do not have their own this, arguments, super, or new.target.*

**When should I use Arrow Functions?**

*Use Arrow Functions for concise anonymous functions, especially for short callbacks or when this should lexically bind to the surrounding scope.*

**Can Arrow Functions have default parameters?**

*Yes, Arrow Functions support default parameters: (param = defaultValue) => { }.*

**Do Arrow Functions support rest parameters?**

*Yes, Arrow Functions can use rest parameters (…rest) to represent an indefinite number of arguments as an array.*

**Can Arrow Functions be used as constructors?**

*No, Arrow Functions cannot be used as constructors because they do not have their own this context.*

**How do Arrow Functions handle this?**

*Arrow Functions inherit this from the surrounding lexical context, making them useful for methods inside objects or for maintaining the context of callbacks.*

* 1. [JS Nested functions](https://www.geeksforgeeks.org/javascript-nested-functions/)

In JavaScript, Functions within another function are called “Nested function.” These nested functions have access to the variables and parameters of the outer (enclosing) function, creating a scope hierarchy. A function can have one or more inner functions.

**Syntax:**

// Outer function

function outerFunction() {

// Nested function

function nestedFunction() {

// Function logic here

}

// Call the nested function

nestedFunction();

// Rest of the outer function logic

}

// Call the outer function

outerFunction();

**Approach:**

* Write one function inside another function.
* Make a call to the inner function in the return statement of the outer function.
* Call it **fun(a)(b)** where a is a parameter to the outer and b is to the inner function.
* Finally, return the combined output from the nested function.

**Example 1:**This example uses the approach discussed above.

**Javascript**

**function** fun1(a) {

**function** fun2(b) {

**return** a + b;

    }

**return** fun2;

}

**function** GFG\_Fun() {

    console.log(fun1("A Online Computer Science Portal")

        (" GeeksforGeeks"));

}

GFG\_Fun()

**Output**

A Online Computer Science Portal GeeksforGeeks

**Example 2:** This example uses the approach discussed above, but here the nested function is created differently than the previous one.

**Javascript**

**function** fun1(a) {

    fun = **function** fun2(b) {

**return** a + b;

    }

**return** fun;

}

**function** GFG\_Fun() {

    console.log(fun1("This is ")("GeeksforGeeks"));

}

GFG\_Fun()

**Output**

This is GeeksforGeeks

* 1. [JS Function Generator](https://www.geeksforgeeks.org/javascript-function-generator/)

A generator functionuses the yield keyword to generate values, pausing execution and sending values to the caller. It retains the state to resume execution after yield, continuing immediately after the last yield run.

**Syntax :**

// An example of generator function  
function\* gen(){  
 yield 1;  
 yield 2;  
 ...  
 ...  
}

**Generator-Object :**Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for of” loop (as shown in the above program)   
The Generator object is returned by a generating function and it conforms to both the iterable protocol and the iterator protocol.

**Example 1:** In this example, we will see the creation of basic generator object.

javascript

*// Generate Function generates three*

*// different numbers in three calls*

**function**\* fun() {

**yield** 10;

**yield** 20;

**yield** 30;

}

*// Calling the Generate Function*

**let** gen = fun();

console.log(gen.next().value);

console.log(gen.next().value);

console.log(gen.next().value);

**Output:**

10  
20  
30

**Example 2:** This example code prints infinite series of natural numbers using a simple generator.

javascript

*// Generate Function generates an*

*// infinite series of Natural Numbers*

**function**\* nextNatural() {

**let** naturalNumber = 1;

*// Infinite Generation*

**while** (**true**) {

**yield** naturalNumber++;

}

}

*// Calling the Generate Function*

**let** gen = nextNatural();

*// Loop to print the first*

*// 10 Generated number*

**for** (**let** i = 0; i < 10; i++) {

*// Generating Next Number*

console.log(gen.next().value);

}

**Output:**

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

**Example 3:**This example of how to manually return from a generator.

javascript

**let** array = ['a', 'b', 'c'];

**function**\* generator(arr) {

**let** i = 0;

**while** (i < arr.length) {

**yield** arr[i++]

}

}

**const** it = generator(array);

*// We can do it.return() to finish the generator*

**Encountering yield and yield\***

* **yield:** pauses the generator execution and returns the value of the expression which is being written after the yield keyword.
* **yield\*:** it iterates over the operand and returns each value until done is true.

**Example 4:**

javascript

**const** arr = ['a', 'b', 'c'];

**function**\* generator() {

**yield** 1;

**yield**\* arr;

**yield** 2;

}

**for** (**let** value **of** generator()) {

console.log(value);

}

**Output:**

1  
a  
b  
c  
2

**Example 5:**Another way to create iterable.

javascript

**let** createOwnIterable = {

\*[Symbol.iterator]() {

**yield** 'a';

**yield** 'b';

**yield** 'c';

}

}

**for** (**let** value **of** createOwnIterable) {

console.log(value);

}

**Output:**

a  
b  
c

**Example 6:** Return from a generator function.

javascript

**function**\* generator() {

**yield** 'a';

**return** 'result';

**yield** 'b';

}

**let** it = generator();

console.log(JSON.stringify(it.next()));

*// {value: "a", done: false}*

console.log(JSON.stringify(it.next()));

*// {value: "result", done: true}*

**Output**

{"value":"a","done":false}

{"value":"result","done":true}

**Example 7:** How to throw an exception from the generator.

javascript

**function**\* generator() {

**throw** **new** **Error**('Error Occurred');

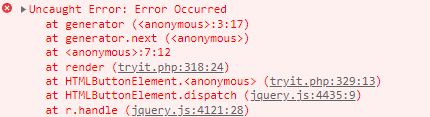
}

**const** it = generator();

it.next();

*// Uncaught Error: Error Occurred*

**Output:**



**Example 8:** Calling a generator from another generator.

javascript

**function**\* firstGenerator() {

**yield** 2;

**yield** 3;

}

**function**\* secondGenerator() {

**yield** 1;

**yield**\* firstGenerator();

**yield** 4;

}

**for** (**let** value **of** secondGenerator()) {

console.log(value)

}

**Output:**

1  
2  
3  
4

**Limitation of Generators:** You can’t yield inside a callback in a generator.

**Example 9:** In this example, we will try to give yield inside a generator function.

javascript

**function**\* generator() {

['a', 'b', 'c'].forEach(value => **yield** value)

*// This will give syntax error*

}

**Output:**

SyntaxError: missing ) after argument list

**Example 10:** Using async generators (for api call).

javascript

**const** firstPromise = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => resolve(1), 5000)

})

}

**const** secondPromise = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => resolve(2), 3000)

})

}

**async** **function**\* generator() {

**const** firstPromiseResult = **await** firstPromise();

**yield** firstPromiseResult;

**const** secondPromiseResult = **await** secondPromise();

**yield** secondPromiseResult;

}

**let** it = generator();

**for** **await** (**let** value **of** it) {

console.log(value);

}

**Output:**

(after 5 seconds)  
1   
(after 3 seconds)  
2

**Advantages of generators:** They are memory efficient as lazy evaluation takes place, i.e, delays the evaluation of an expression until its value is needed.**JavaScript use-case (generators)**

* Writing generators in redux-saga
* JavaScript async-await (Implemented with promise and generators)

**JavaScript Function Generator – FAQs**

**What is a Generator Function in JavaScript?**

*A generator function is a special type of function that can pause its execution and resume later. They are defined using the function\* syntax and use the yield keyword to yield values.*

**How do you create a Generator Function?**

*You create a generator function by adding an asterisk (\*) after the function keyword and using the yield keyword to yield values.*

**How do Generator Functions work?**

*Generator functions return a generator object when called. This generator object has methods like next(), return(), and throw(). The next() method is used to resume the generator function’s execution and retrieve the next value.*

**How do you use the yield keyword?**

*The yield keyword is used to pause the execution of a generator function and return a value. When the generator’s next() method is called, the function resumes execution from where it left off, just after the yield statement.*

**How do you iterate over values from a Generator?**

*You can iterate over values from a generator using a for…of loop or the generator’s next() method.*

* 1. [JS Function binding](https://www.geeksforgeeks.org/javascript-function-binding/)

In [JavaScript](https://www.geeksforgeeks.org/introduction-to-javascript/) function binding happens using the [**Bind()** method](https://www.geeksforgeeks.org/explain-call-apply-and-bind-methods-in-javascript/). With this method, we can bind an object to a common function, so that the function gives different results when needed. otherwise, it gives the same result or gives an error while the code is executing. We use the [**Bind()** method](https://www.geeksforgeeks.org/explain-call-apply-and-bind-methods-in-javascript/) to call a function with ‘**this’** value.

**What is ‘this’?**

‘this’ refers to the object it belongs to. The exact value of ‘this’ depends on how a function is called. In a method, ‘this’ represents the object that the method was called on, while in a regular function, it typically refers to the global object (in the case of browser environments, it’s the ‘window’ object).

**Example 1:** In this example, *this* keyword binds the *name* variable to the function. It is known as default binding. *this* keyword refers to **geeks** object.

* javascript

let geeks = {

    name: "ABC",

    printFunc: **function** () {

        console.log(**this**.name);

    }

}

geeks.printFunc();

**Output**

ABC

**Example 2:**In this example, the binding of *this* is lost, so no output is produced.

* javascript

let geeks = {

    name: "ABC",

    printFunc: **function** () {

        console.log(**this**.name);

    }

}

let printFunc2 = geeks.printFunc;

printFunc2();

**Output**

undefined

**Example 3:** In this example, we are using the **bind()** method in the previous example. The **bind()** method creates a new function where **this** keyword refers to the parameter in the parenthesis. This way the **bind()** method enables calling a function with a specified **this** value.

* javascript

let geeks = {

    name: "ABC",

    printFunc: **function** () {

        console.log(**this**.name);

    }

}

let printFunc2 = geeks.printFunc.bind(geeks);

//using bind()

// bind() takes the object "geeks" as parameter//

printFunc2();

**Output**

ABC

**Example 4:** In this example, there are 3 objects, and each time we call each object by using the **bind()**method.

* javascript

//object geeks1

let geeks1 = {

    name: "ABC",

    article: "C++"

}

//object geeks2

let geeks2 = {

    name: "CDE",

    article: "JAVA"

}

//object geeks3

let geeks3 = {

    name: "IJK",

    article: "C#"

}

**function** printVal() {

    console.log(**this**.name + " contributes about " +

**this**.article + "<br>");

}

let printFunc2 = printVal.bind(geeks1);

//using bind()

// bind() takes the object "geeks1" as parameter//

printFunc2();

let printFunc3 = printVal.bind(geeks2);

printFunc3();

let printFunc4 = printVal.bind(geeks3);

printFunc4();

//uniquely defines each objects

**Output**

ABC contributes about C++<br>

CDE contributes about JAVA<br>

IJK contributes about C#<br>

* 1. [JS Async/Await Function](https://www.geeksforgeeks.org/async-await-function-in-javascript/)

Async and Await in JavaScript is used to simplify handling asynchronous operations using promises. By enabling asynchronous code to appear synchronous, they enhance code readability and make it easier to manage complex asynchronous flows.

**Async Function**

The async function allows us to write promise-based code as if it were synchronous. This ensures that the execution thread is not blocked.

* **Promise Handling**: Async functions always return a promise. If a value is returned that is not a promise, JavaScript automatically wraps it in a resolved promise.

**Async Syntax**

async function myFunction() {

  return "Hello";

}

**Example:** Here, we will see the basic use of async in JavaScript.

javascript

**const** getData = **async** () => {

**let** data = "Hello World";

**return** data;

}

getData().then(data => console.log(data));

**Output**

Hello World

**Await Keyword**

The await keyword is used to wait for a promise to resolve. It can only be used within an async block.

* **Execution Pause**: Await makes the code wait until the promise returns a result, allowing for cleaner and more manageable asynchronous code.

**Syntax**

let value = await promise;

**Example** : This example shows the basic use of the await keyword in JavaScript.

javascript

**const** getData = **async** () => {

**let** y = **await** "Hello World";

console.log(y);

}

console.log(1);

getData();

console.log(2);

**Output**

1

2

Hello World

*The****async****keyword transforms a regular JavaScript function into an asynchronous function, causing it to return a Promise.*

*The****await****keyword is used inside an async function to pause its execution and wait for a Promise to resolve before continuing.*

**Async/Await Example**

Here, we will be implementing several promises in a method, and then that method we will use for displaying our result. You can check the JS **async/await syntax** in the example.

JavaScript

**function** asynchronous\_operational\_method() {

**let** first\_promise =

**new** Promise((resolve, reject) => resolve("Hello"));

**let** second\_promise =

**new** Promise((resolve, reject) => {

setTimeout(() => {

resolve(" GeeksforGeeks..");

}, 1000);

});

**let** combined\_promise =

Promise.all([first\_promise, second\_promise]);

**return** combined\_promise;

}

**async** **function** display() {

**let** data = **await** asynchronous\_operational\_method();

console.log(data);

}

display();

**Output:**

[ 'Hello', ' GeeksforGeeks..' ]

**Explanation:**

1. **Promise Creation**:
   * Two promises are created: one resolve immediately with “Hello”, and the other resolves after 1 second with ” GeeksforGeeks..”.
2. **Combining Promises**:
   * The Promise.all() method combines both promises into a single promise, combined\_promise.
3. **Asynchronous Function**:
   * The display() function is declared as async, indicating it contains asynchronous operations.
4. **Awaiting Promise Resolution**:
   * The await keyword pauses execution until combined\_promise is resolved.
5. **Logging Result**:
   * The resolved array from combined\_promise is logged to the console.

***Note***

*To****resolve****and****reject****are predefined arguments by JavaScript.*

* *resolve function is used when an asynchronous task is completed and returns the result.*
* *reject function is used when an asynchronous task fails and returns reasons for failure.*

**Error Handling in Async/Await**

JavaScript provides predefined arguments for handling promises: resolve and reject.

* **resolve:** Used when an asynchronous task is completed successfully.
* **reject:** Used when an asynchronous task fails, providing the reason for failure.

**Example:**

JavaScript

**async** **function** fetchData() {

**try** {

**let** response = **await** fetch('https://api.example.com/data');

**let** data = **await** response.json();

console.log(data);

} **catch** (error) {

console.error('Error fetching data:', error);

}

}

**Advantages of Async and Await**

1. **Improved Readability:**Async and Await allow asynchronous code to be written in a synchronous style, making it easier to read and understand.
2. **Error Handling:**Using try/catch blocks with async/await simplifies error handling.
3. **Avoids Callback Hell:** Async and Await prevent nested callbacks and complex promise chains, making the code more linear and readable.
4. **Better Debugging:** Debugging async/await code is more intuitive since it behaves similarly to synchronous code.

**Conclusion**

Async and Await in JavaScript have revolutionized asynchronous programming by making code more readable and maintainable. By allowing asynchronous code to be written in a synchronous style, they reduce the complexity associated with callbacks and promise chaining. Understanding and using async and await effectively can significantly enhance your JavaScript programming skills, making it easier to handle asynchronous operations in your projects.

* 1. [Hoisting in JavaScript](https://www.geeksforgeeks.org/javascript-hoisting/)

**JavaScript Hoisting** is the behavior where the interpreter moves function and variable declarations to the top of their respective scope before executing the code. This allows variables to be accessed before declaration, aiding in more flexible coding practices and avoiding “undefined” errors during execution.

**What is Hoisting in JavaScript?**

Hoisting is the default behavior in JavaScript where variable and function declarations are moved to the top of their respective scopes during the compilation phase. This guarantees that regardless of where these declarations appear within a scope, they can be accessed throughout that scope.

**Features of Hoisting**

* Declarations are hoisted, not initializations.
* Allows calling functions before their declarations.
* All variable and function declarations are processed before any code execution.
* Undeclared variables are implicitly created as global variables when assigned a value.

***Note:****JavaScript only hoists declarations, not initializations.*

JavaScript allocates memory for all variables and functions defined in the program before execution.

**Sequence of variable declaration**

The following is the sequence in which variable declaration and initialization occur.

***Declaration –> Initialisation/Assignment –> Usage***

**Variable lifecycle**

let a; // Declaration  
a = 100; // Assignment  
console.log(a); // Usage

However, since JavaScript allows us to both declare and initialize our variables simultaneously, so we can declare and initialize at the same time.

let a = 100;

**Note:** Always remember that in the background the Javascript is first declaring the variable and then initializing them. It is also good to know that variable declarations are processed before any code is executed.

However, in javascript, undeclared variables do not exist until the code assigning them is executed. Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed. This means that all undeclared variables are global variables.

**Different Examples of JavaScript Hoisting**

**1. Global Scope**

JavaScript

*// Hoisting*

**function** codeHoist() {

a = 10;

**let** b = 50;

}

codeHoist();

console.log(a); *// 10*

console.log(b); *// ReferenceError : b is not defined*

**Output:**

10  
ReferenceError: b is not defined

**Explanation:** In the above example, hoisting allows variables declared with var to be accessed before declaration, but not those declared with let or const. Thus, a is accessible, but b throws a ReferenceError

**Note:** There’s a difference between ReferenceError and undefined errors. An undefined error occurs when we have a variable that is either not defined or explicitly defined as type undefined. ReferenceError is thrown when trying to access a previously undeclared variable.

**2. JavaScript var hoisting**

When we talk about ES5, the variable that comes into our minds is var. Hoisting with var is somewhat different. When it is compared to let/const. Let’s make use of var and see how hoisting works.

**Example:**

JavaScript

*// var code (global)*

console.log(name); *// undefined*

**var** name = 'Mukul Latiyan';

**Output:**

ReferenceError: Cannot access 'name' before initialization

**Explanation:** In the above code example variables declared with var are hoisted but not initialized, resulting in undefined when accessed before declaration. Variables declared with let or const do not exhibit this behavior.

But the interpreter sees this differently, the above code is seen like this:

JavaScript

*// how interpreter sees the above code*

**let** name;

console.log(name); *// undefined*

name = 'Mukul Latiyan';

**Output**

undefined

**3. Function scoped variable**

Let’s look at how function-scoped variables are hoisted.

**Example:**

JavaScript

*// Function scoped*

**function** fun() {

console.log(name);

**let** name = 'Mukul Latiyan';

}

fun(); *// Undefined*

**Output:**

undefined

There is no difference here as when compared to the code where we declared the variable globally.

**Example:**We get undefined as the code seen by the interpreter.

JavaScript

**var** name;

**function** fun() {

console.log(name);

name = 'Mukul Latiyan';

}

fun(); *// undefined*

**Output**

undefined

In order to avoid this pitfall, we can make sure to **declare and assign the variable at the same time,** **before using it.**

**Example:**

JavaScript

**function** fun() {

**let** name = 'Mukul Latiyan';

console.log(name); *// Mukul Latiyan*

}

fun();

**Output**

Mukul Latiyan

**4. JavaScript hoisting with Let**

We know that variables declared with let keywords are block scoped not function scoped and hence there is no problem when it comes to hoisting.

**Example:**

JavaScript

*//let example(global)*

console.log(name);

**let** name = 'Mukul Latiyan'; *// ReferenceError: name is not defined*

**Output:**

ReferenceError: name is not defined

**Explanation:**Like before, for the var keyword, we expect the output of the log to be undefined. However, since the es6 let doesn’t take kindly on us using undeclared variables, the interpreter explicitly spits out a Reference error. This ensures that we always **declare**our variable first.

**5. JavaScript hoisting with const**

It behaves similarly to let when it comes to hoisting. A **function**as a whole can also be hoisted and we can call it before the declaration.

**Example:**

JavaScript

fun(); *// Calling before declaration*

**function** fun() { *// Declaring*

console.log("Function is hoisted");

}

**Output**

Function is hoisted

Also, if a function is used as an **expression**and we try to access it before the assignment an error will occur as only declarations are hoisted.

**Example:**

JavaScript

fun() *// Calling the expression*

**let** fun = () =>{ *// Declaring*

**let** name = 'Mukul Latiyan';

console.log(name);

}

**Output:**

ReferenceError: Cannot access 'fun' before initialization

However, if var is used in the expression instead of let we will get the following Type Error as follows.

**6. Hoisting with Functions**

**Example:**

JavaScript

fun() *// Calling the expression*

**var** fun = () =>{ *// Declaring*

**let** name = 'Mukul Latiyan';

console.log(name);

}

**Output:**

TypeError: fun is not a function

1. [JavaScript Regular Expression](https://www.geeksforgeeks.org/javascript/#javascript-regular-expression)
   1. [JS Regular expressions](https://www.geeksforgeeks.org/javascript-regexpregular-expression/)
   2. [JS RegExp [abc] Expression](https://www.geeksforgeeks.org/javascript-regexp-abc-expression/)
   3. [JS RegExp S Metacharacter](https://www.geeksforgeeks.org/javascript-regexp-s-metacharacter-2/)
   4. [JS RegExp m Modifier](https://www.geeksforgeeks.org/javascript-regexp-m-modifier/)
   5. [JS RegExp ?! Quantifier](https://www.geeksforgeeks.org/javascript-regexp-quantifier-4/)
   6. [JS RegExp {X,Y} Quantifier](https://www.geeksforgeeks.org/javascript-regexp-xy-quantifier/)
   7. [JS RegExp test() Method](https://www.geeksforgeeks.org/javascript-regexp-test-method/)
   8. [JS RegExp [^0-9] Expression](https://www.geeksforgeeks.org/javascript-regexp-0-9-expression-2/)
2. [JavaScript Events](https://www.geeksforgeeks.org/javascript/#javascript-events)
   1. [JS Events](https://www.geeksforgeeks.org/javascript-events/)
   2. [JS onclick Event](https://www.geeksforgeeks.org/html-dom-onclick-event/)
   3. [JS dblclick Event](https://www.geeksforgeeks.org/html-dom-ondblclick-event/)
   4. [JS onload Event](https://www.geeksforgeeks.org/html-dom-onload-event/)
   5. [JS onresize Event](https://www.geeksforgeeks.org/html-dom-onresize-event/)
   6. [JS onblur Event](https://www.geeksforgeeks.org/html-dom-onblur-event/)
   7. [JS onchange Event](https://www.geeksforgeeks.org/html-dom-onchange-event/)
   8. [JS Focus Event](https://www.geeksforgeeks.org/html-dom-focusevent/)
   9. [JS Clipboard Event](https://www.geeksforgeeks.org/html-dom-clipboardevent/)
   10. [JS onscroll Event](https://www.geeksforgeeks.org/html-dom-onscroll-event/)
3. [JavaScript OOPs](https://www.geeksforgeeks.org/javascript/#javascript-oops)
   1. [JS Classes In JavaScript](https://www.geeksforgeeks.org/javascript-classes/)
   2. [JS class expression](https://www.geeksforgeeks.org/javascript-class-expression/)
   3. [JS Object Constructors](https://www.geeksforgeeks.org/javascript-object-constructors/)
   4. [JS Static Methods](https://www.geeksforgeeks.org/static-methods-in-javascript/)
   5. [JS Prototype](https://www.geeksforgeeks.org/prototype-in-javascript/)
   6. [JS Constructor Method](https://www.geeksforgeeks.org/javascript-object-prototype-constructor-property/)
   7. [JS Encapsulation](https://www.geeksforgeeks.org/introduction-object-oriented-programming-javascript/#Encapsulation)
   8. [JS Inheritance](https://www.geeksforgeeks.org/introduction-object-oriented-programming-javascript/#Inheritance)
   9. [JS Polymorphism](https://www.geeksforgeeks.org/introduction-object-oriented-programming-javascript/#Polymorphism)
   10. [JS Abstraction](https://www.geeksforgeeks.org/introduction-object-oriented-programming-javascript/#Abstraction)
4. [JavaScript Inheritance and Prototype Chain](https://www.geeksforgeeks.org/javascript/#javascript-inheritance-and-prototype-chain)
   1. [Prototype Inheritance in JavaScript](https://www.geeksforgeeks.org/explain-prototype-inheritance-in-javascript/)
   2. [Prototype Chain in JavaScript](https://www.geeksforgeeks.org/understanding-the-prototype-chain-in-javascript/)
   3. [JS Object Constructors](https://www.geeksforgeeks.org/javascript-object-constructors/)
   4. [JS Prototype in JavaScript](https://www.geeksforgeeks.org/prototype-in-javascript/)
5. [JavaScript Memory Management](https://www.geeksforgeeks.org/javascript/#javascript-memory-management)
   1. [JS Memory Management](https://www.geeksforgeeks.org/memory-management-in-javascript/)
   2. [JS Garbage Collection](https://www.geeksforgeeks.org/garbage-collection-in-javascript/)
6. [JavaScript Promises](https://www.geeksforgeeks.org/javascript/#javascript-promises)
   1. [JS Promise](https://www.geeksforgeeks.org/javascript-promise/)
   2. [JS Promise Chaining](https://www.geeksforgeeks.org/javascript-promise-chaining/)
   3. [JS Errors Throw and Try to Catch](https://www.geeksforgeeks.org/javascript-errors-throw-and-try-to-catch/)
   4. [JS Class compositions in JavaScript](https://www.geeksforgeeks.org/what-are-the-class-compositions-in-javascript/)
7. [JavaScript Iterators and generators](https://www.geeksforgeeks.org/javascript/#javascript-iterators-and-generators)
   1. [JS Iterator](https://www.geeksforgeeks.org/javascript-iterator/)
   2. [JS Function Generator](https://www.geeksforgeeks.org/javascript-function-generator/)
8. [JavaScript Validations](https://www.geeksforgeeks.org/javascript/#javascript-validations)
   1. [JS Form Validation](https://www.geeksforgeeks.org/form-validation-using-html-javascript/)
   2. [JS Email Validation](https://www.geeksforgeeks.org/how-to-validate-email-address-using-regexp-in-javascript/)
9. [JavaScript Exception Handling](https://www.geeksforgeeks.org/javascript/#javascript-exception-handling)
   1. [JS Exception Handling](https://www.geeksforgeeks.org/javascript-error-and-exceptional-handling-with-examples/)
   2. [JS try-catch Statement](https://www.geeksforgeeks.org/javascript-errors-throw-and-try-to-catch/)
   3. [JS Promises](https://www.geeksforgeeks.org/javascript-promises/)
   4. [JS async/await](https://www.geeksforgeeks.org/async-await-function-in-javascript/)
10. [JavaScript Global Objects](https://www.geeksforgeeks.org/javascript/#javascript-global-objects)
    1. [JS encodeURI(), encodeURIComponent() and decodeURIComponent() Method](https://www.geeksforgeeks.org/javascript-encodeuri-decodeuri-and-its-components-functions/)
    2. [JS eval() Method](https://www.geeksforgeeks.org/javascript-eval-function/)
    3. [JS globalThis Property](https://www.geeksforgeeks.org/javascript-this-keyword/)
    4. [JS Global property](https://www.geeksforgeeks.org/javascript-global-variables/)
    5. [JS Infinity](https://www.geeksforgeeks.org/javascript-infinity-property/)
    6. [JS Undefined vs Undeclared](https://www.geeksforgeeks.org/differences-between-undeclared-and-undefined-variables-in-javascript/)
11. [JavaScript Miscellaneous](https://www.geeksforgeeks.org/javascript/#javascript-miscellaneous)
    1. [JS this Keyword](https://www.geeksforgeeks.org/this-in-javascript/)
    2. [JS Strict Mode](https://www.geeksforgeeks.org/strict-mode-javascript/)
    3. [JS setTimeOut() and setInterval() Method](https://www.geeksforgeeks.org/java-script-settimeout-setinterval-method/)
    4. [JS typeOf Operator](https://www.geeksforgeeks.org/javascript-typeof-operator/)
    5. [JS Debugging](https://www.geeksforgeeks.org/debugging-in-javascript/)
    6. [JS Local Storage](https://www.geeksforgeeks.org/javascript-localstorage/)
    7. [JS Callback](https://www.geeksforgeeks.org/javascript-callbacks/)
    8. [JS Closures](https://www.geeksforgeeks.org/closure-in-javascript/)
    9. [JS defer](https://www.geeksforgeeks.org/html-defer-attribute/)
    10. [JS scope](https://www.geeksforgeeks.org/understanding-variable-scopes-in-javascript/)
    11. [JS Void](https://www.geeksforgeeks.org/what-is-void-and-when-to-use-void-type-in-javascript/)
12. [JavaScript Practice Quiz](https://www.geeksforgeeks.org/javascript/#javascript-practice-quiz)
13. [JavaScript Interview Questions](https://www.geeksforgeeks.org/javascript/#javascript-interview-questions)