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**

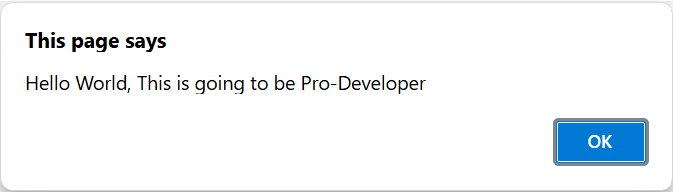
The [alert() method](https://www.geeksforgeeks.org/html-dom-window-alert-method/) displays a popup alert box with the message. It’s useful for simple notifications or warnings but should be used sparingly as it disrupts user interaction.

**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;

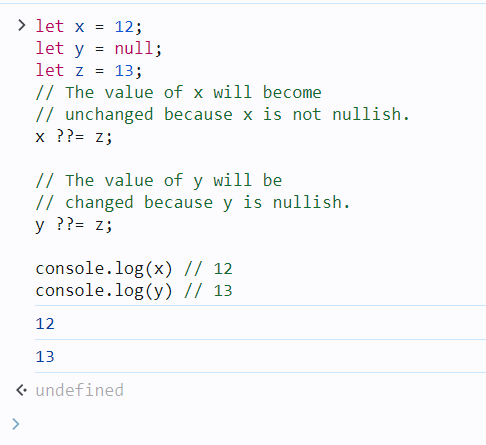
// The value of y will be

// 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

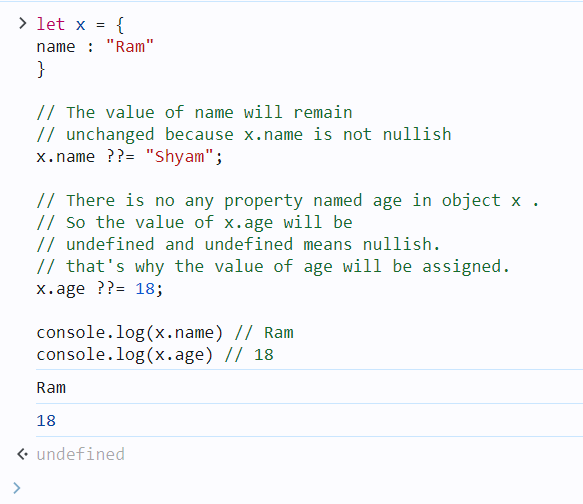
// undefined and undefined means nullish.

// 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() {

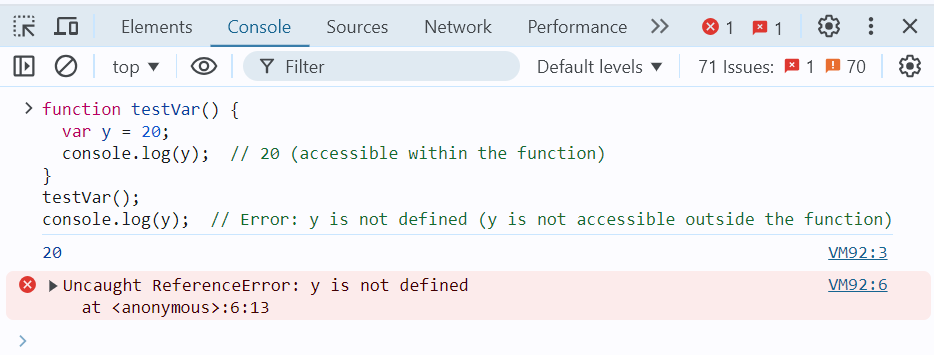
var y = 20;

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

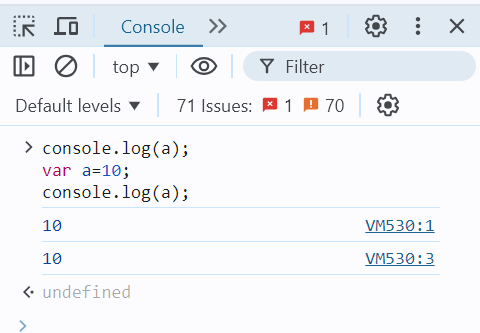
This code behaves as though it was written like this:

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)

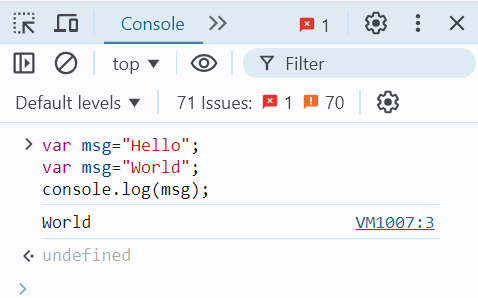


**5. Redeclaration:**

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:**

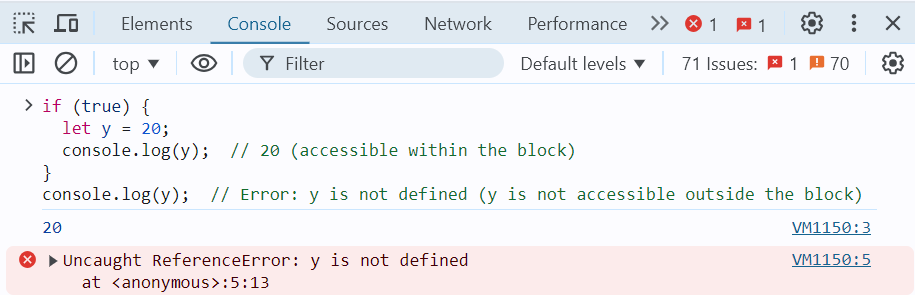
if (true) {

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)



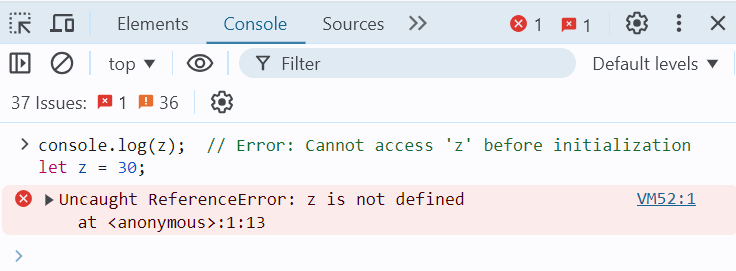
**3. No Hoisting with Initialization:**

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.



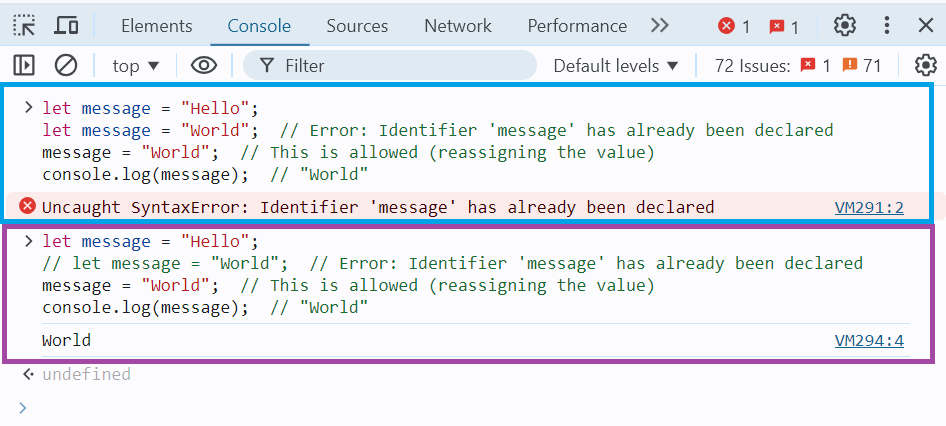
**4. No Redeclaration:**

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:**

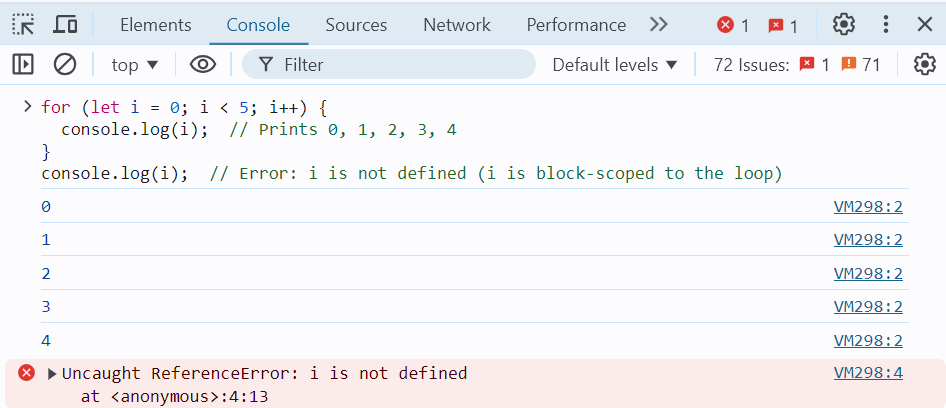
let is commonly used in loops, especially in scenarios where each iteration needs its own scope.

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;

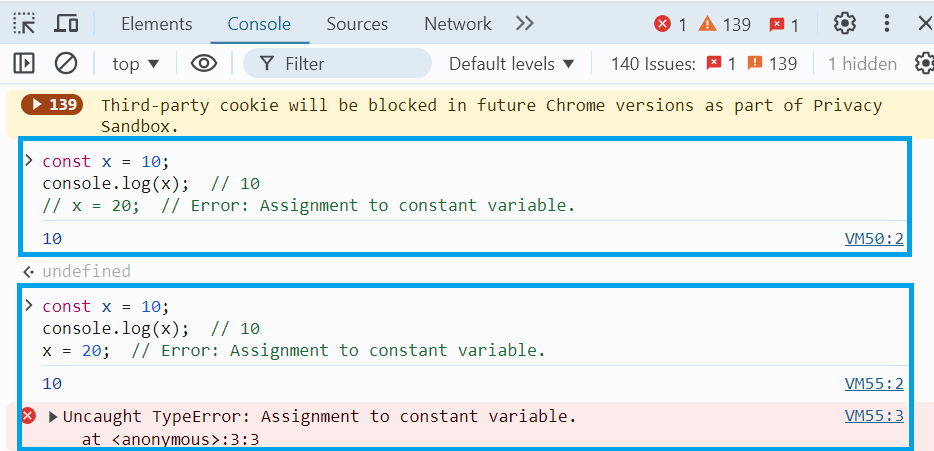
**Examples:**

**1. Basic Declaration and Assignment:**

const x = 10;

console.log(x); // 10

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



**2. Block Scope:**

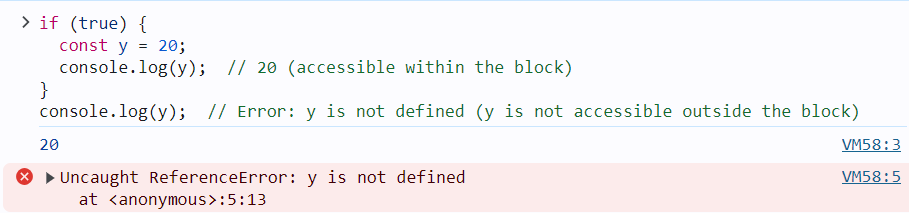
if (true) {

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:

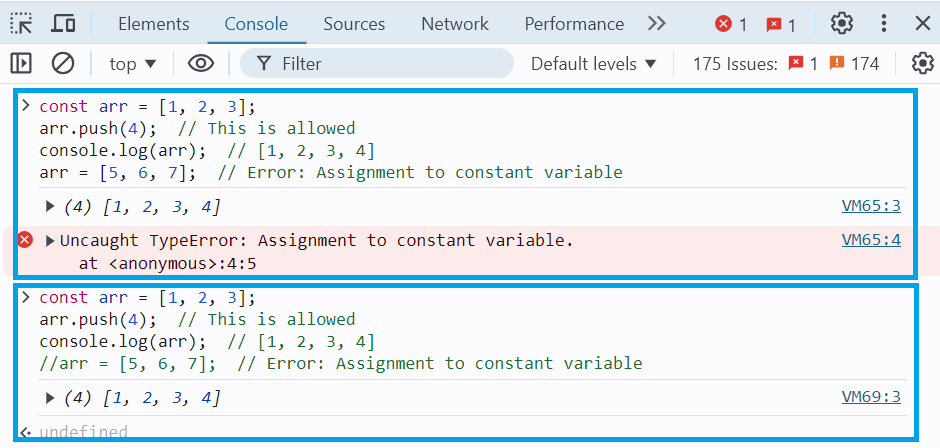
* **Array Example**:

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



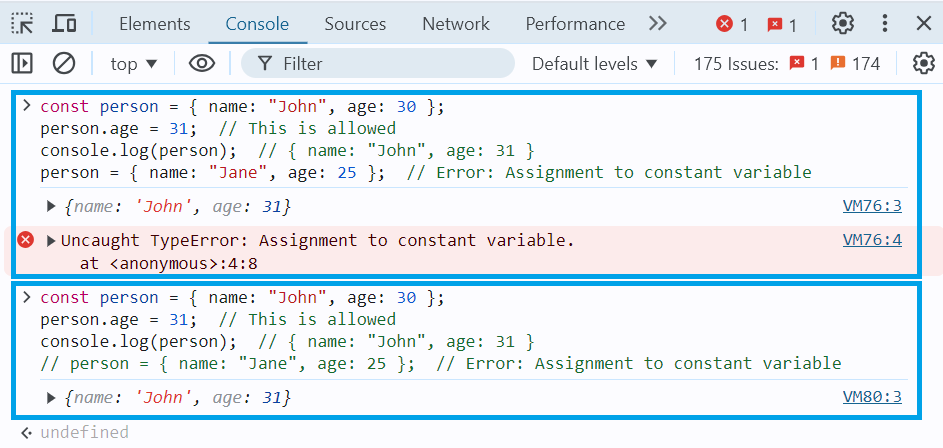
* **Object Example**:

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) {

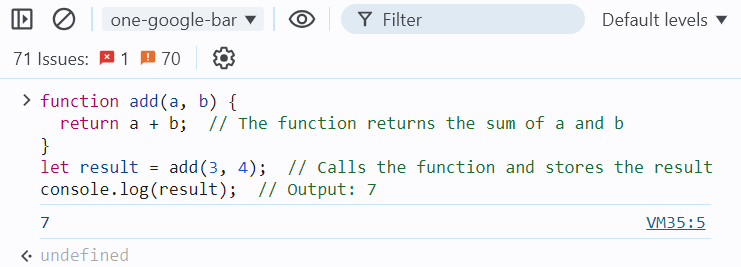
return a + b; // The function returns the sum of a and b

}

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)

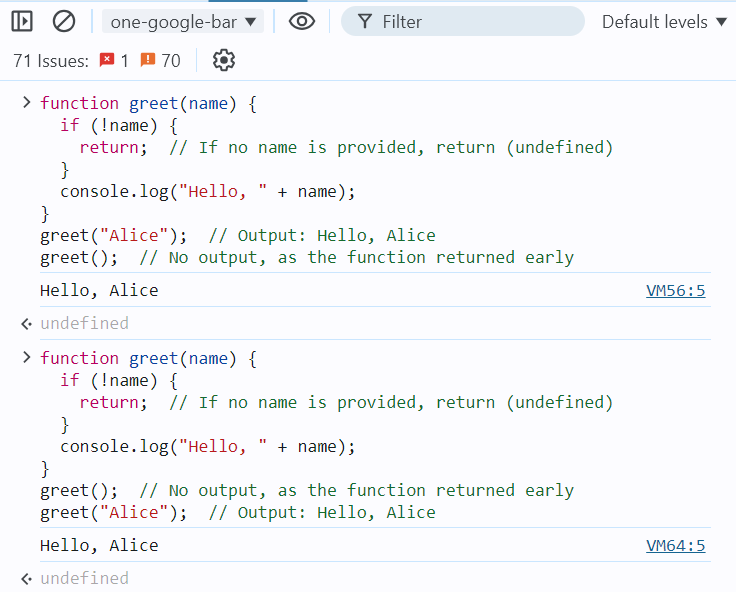
}

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,

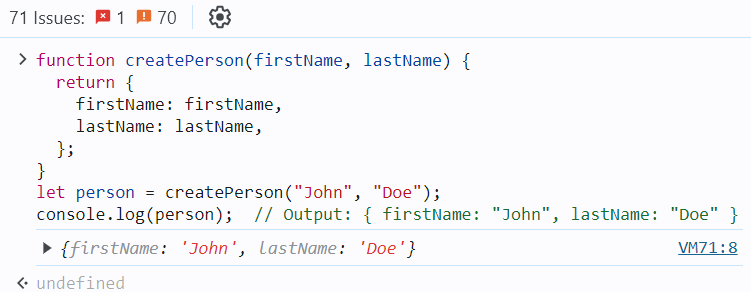
lastName: lastName,

};

}

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++) {

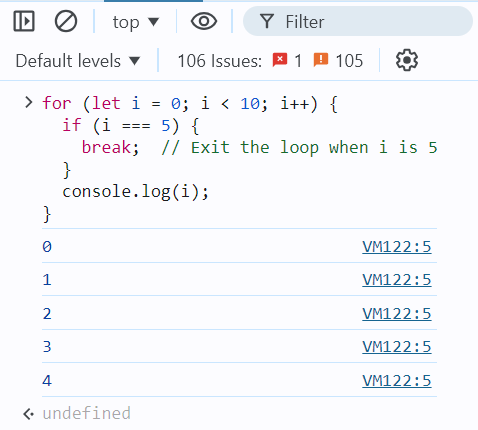
if (i === 5) {

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);

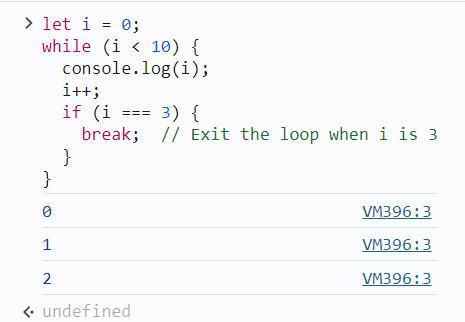
i++;

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":

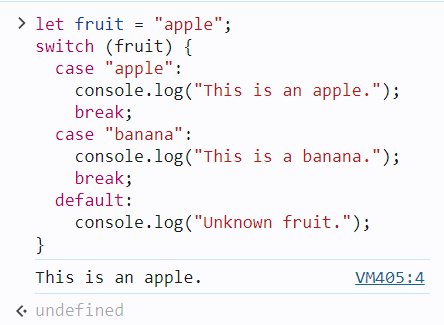
console.log("This is a banana.");

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++) {

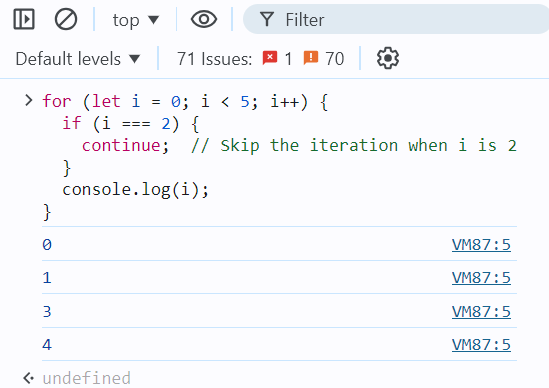
if (i === 2) {

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++;

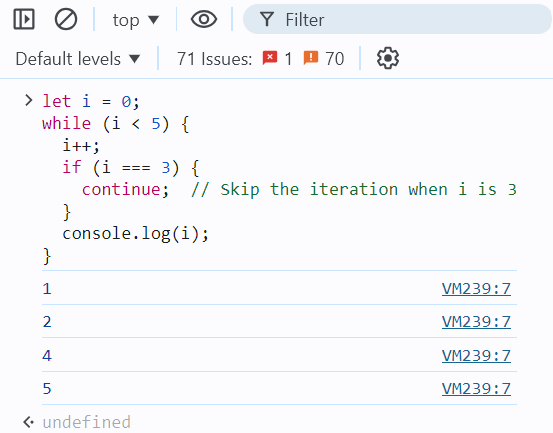
if (i === 3) {

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) {

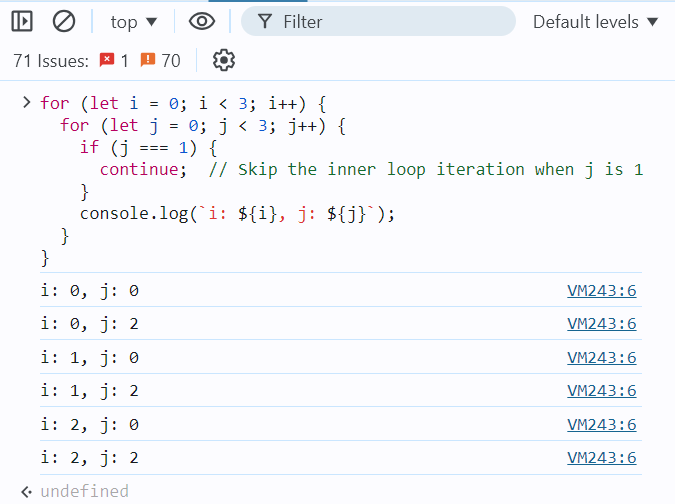
continue; // Skip the inner loop iteration when j is 1

}

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.");

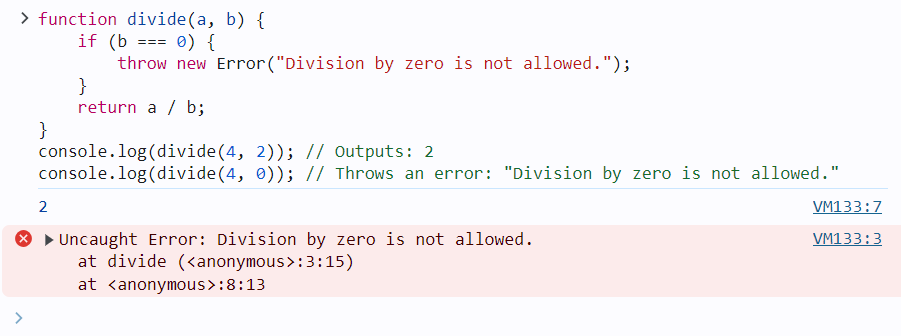
}

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 {

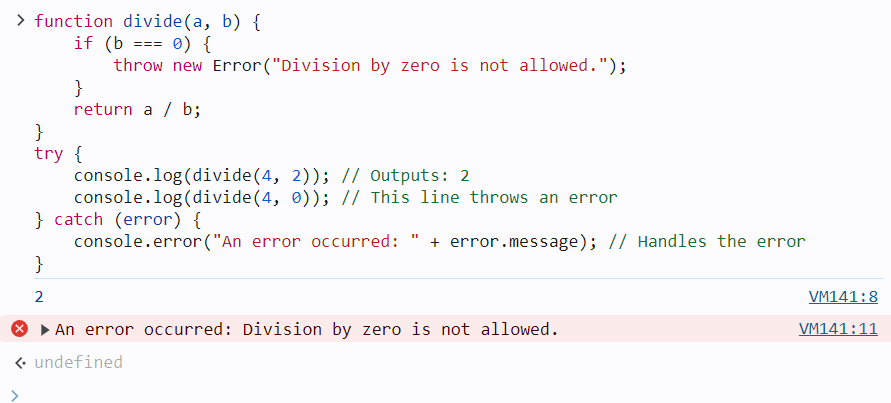
console.log(divide(4, 2)); // Outputs: 2

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

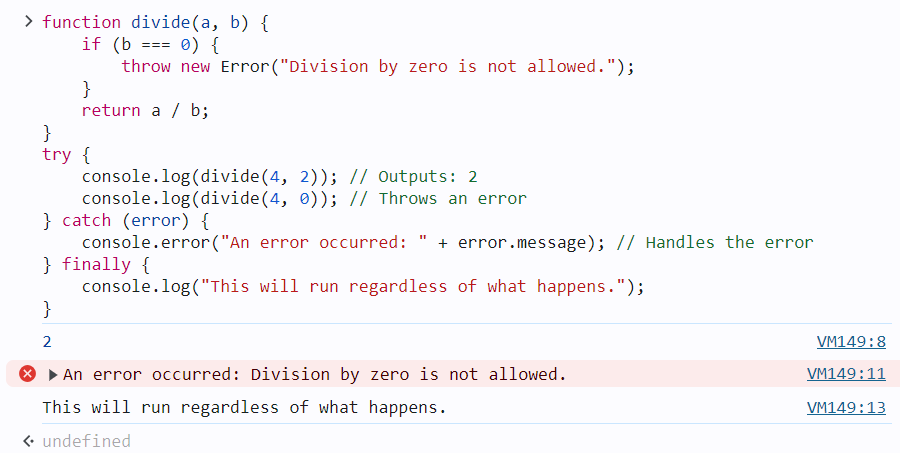
} catch (error) {

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) {

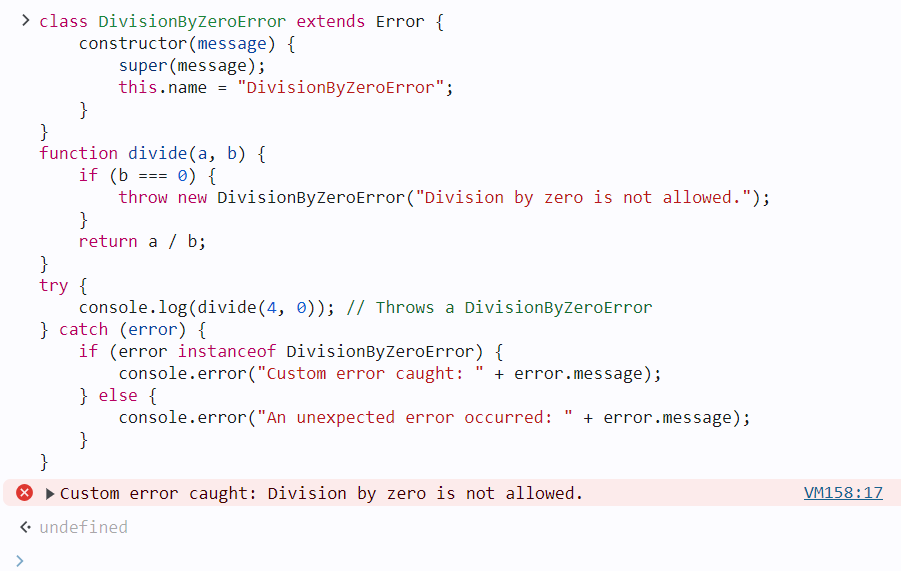
console.error("Custom error caught: " + error.message);

} 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) {

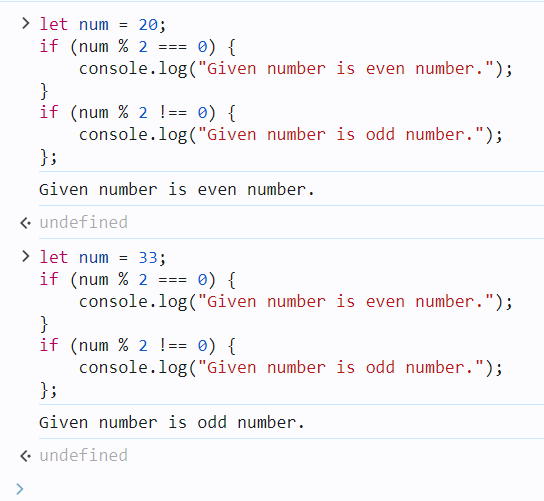
console.log("Given number is even number.");

}

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;

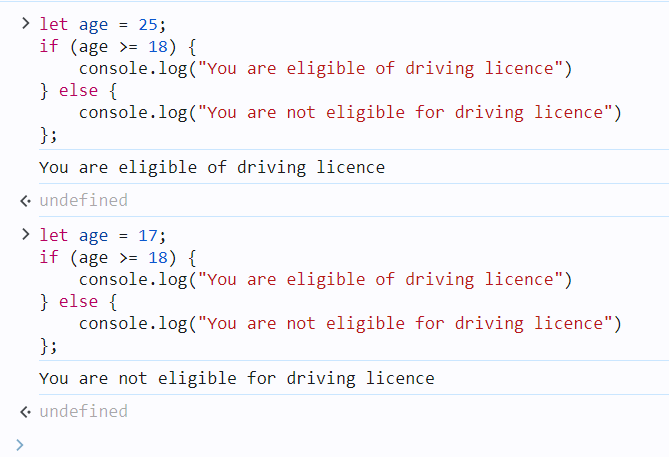
if (age >= 18) {

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.");

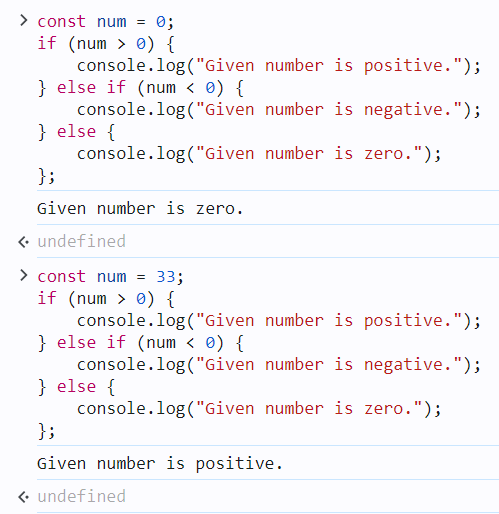
} else if (num < 0) {

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**;

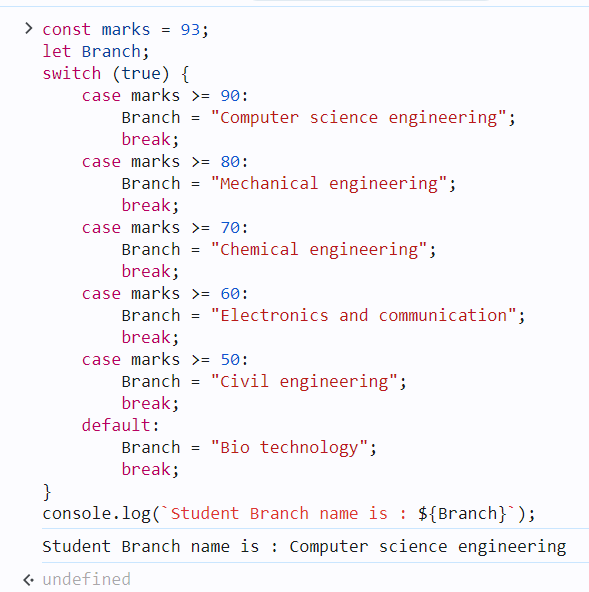
**default**:

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

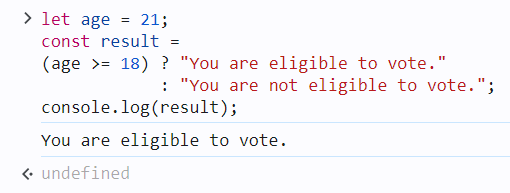
**Example:**This JavaScript code checks if the variable `age` is greater than or equal to 18. If true, it assigns the string “You are eligible to vote.” to the variable `result`. Otherwise, it assigns “You are not eligible to vote.” The value of `result` is then logged to the console.

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.");

}

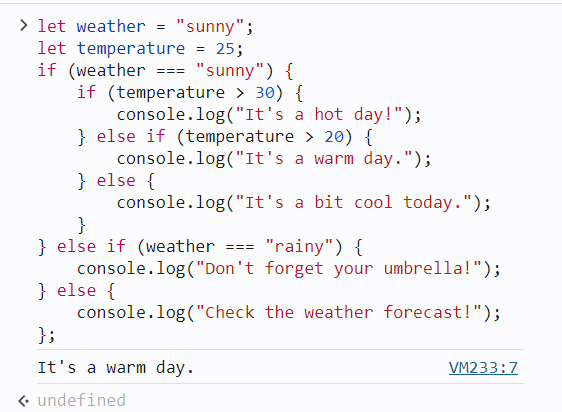
} **else** **if** (weather === "rainy") {

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";

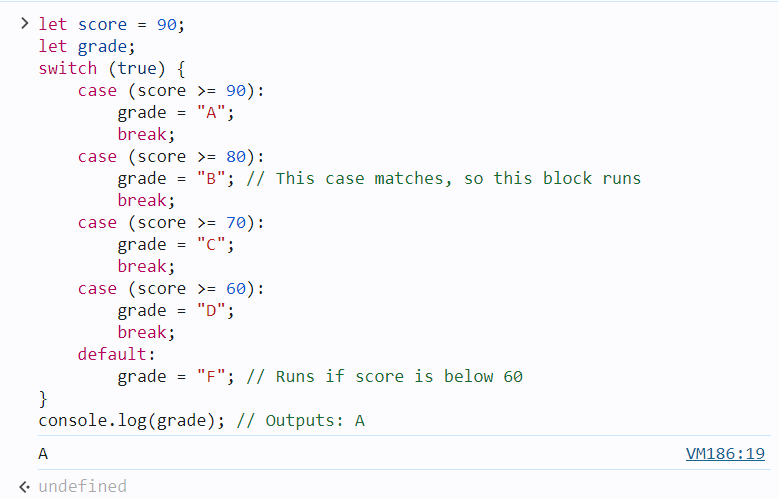
break;

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).

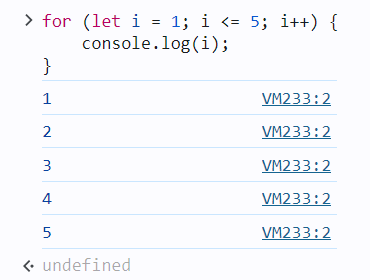
**Example: Basic for Loop**

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**

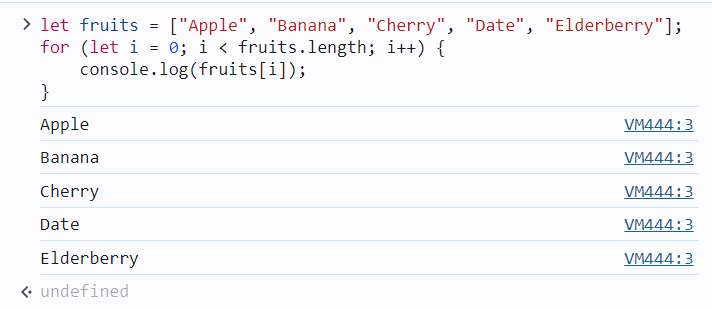
You can also use a for loop to iterate over the elements of an array:

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:

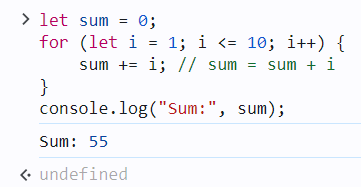
let sum = 0;

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.

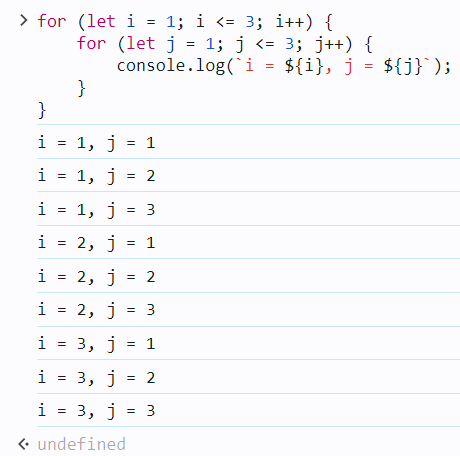
for (let i = 1; i <= 3; i++) {

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;

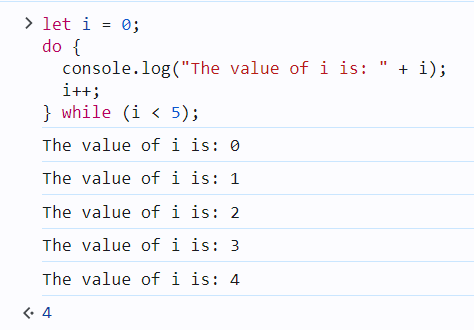
do {

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**

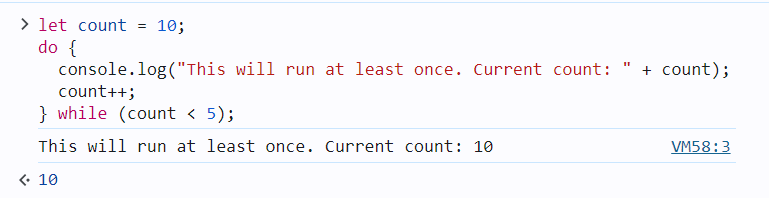
let count = 10;

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**

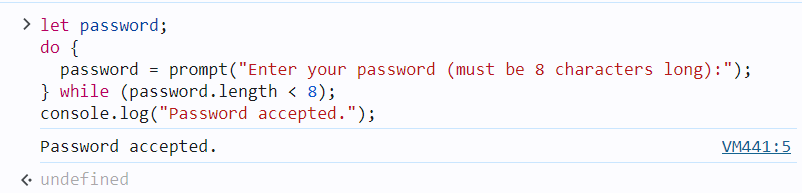
let password;

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.

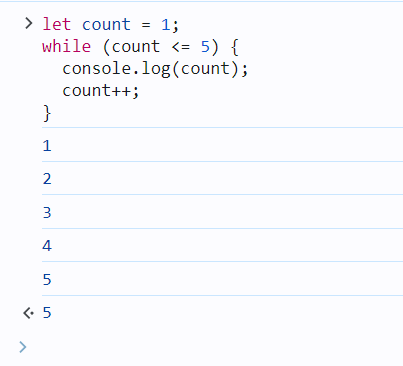
**let** count = 1;

**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.

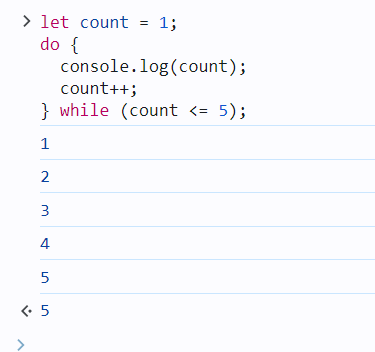
**let** count = 1;

**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/)
2. [JS for…of Loop](https://www.geeksforgeeks.org/javascript-for-of-loop/)
3. [JS labeled Statement](https://www.geeksforgeeks.org/javascript-label-statement/)
4. [JS break Statement](https://www.geeksforgeeks.org/javascript-break-statement/)
5. [JS continue Statement](https://www.geeksforgeeks.org/javascript-continue-statement/)