**Logical Operators**

Logical operators are used to determine the logic between variables or values. They return a [Boolean](https://www.w3docs.com/learn-javascript/javascript-data-types.html)value of true or false depending on the evaluation. There are three **logical operators** in JavaScript:

**||(OR),**

**&&(AND),**

**! (NOT).**

They are called **“logical”**, but can be applied to values of any type, their result can also be of any type.

**|| (OR) operator**

The OR operator is performed with two vertical lines and can manipulate boolean values only. The logical || (OR) can manipulate boolean values only. In case **any** of its arguments are **true**, it returns **true**, if not, it returns **false**. With this operator, at least one of the statements has to evaluate to true, otherwise it returns false.

console.log( true || true );   // true

console.log( false || true );  // true

console.log( true || false );  // true

console.log( false || false ); // false

the result is always true except for the case when both operands are false. If an operand isn’t a boolean, then it’s converted to a boolean for the rating.

if (1 || 0) { //works the same way( true || false )

    console.log('true!');

  }  // Becomes true

Sometimes **||(OR)** is used in an if statement to test if any of the given conditions is true.

let age = 10;

if (age > 0 || age < 18) {

  console.log('You are too young');

}

The **||(OR)** operator executes the following things:

* Evaluates operands from left to right.
* For each operand, converts it to Boolean.
* In case the result is true, it stops and returns the original value of that operand.
* If all operands are false, it returns the last operand.

**AND (&&) operator**

The logical AND (&&) operator (logical conjunction) for a set of boolean operands will be true if and only if all the operands are true. Otherwise it will be false.

console.log( true && true );   // true

console.log( false && true );  // false

console.log( true && false );  // false

console.log( false && false ); // false

Example:

let hour = 11;

if (hour > 10 && hour < 18) {

  console.log('We are open!');

}

else{

    console.log("closed");

}

The **AND (&&)** operator makes the following:

* Asses operands from left to right.
* Converts it to a boolean for each operand. In case the result is false, it stops and returns the original value of that operand.
* If all operands were truthy, it returns the last operand.
* Shortly, **AND** returns the first falsy value or the last value if none were found.

**!(NOT) Operator**

The logical NOT operator is represented with an exclamation sign ! (NOT) . It reverses or negates the value of a Boolean.

Syntax:

Result = ! value

The operator accepts a single argument and does present below functions:

* Transforms the operand to **boolean type: true/false**.
* Returns the opposite**value.**

// !(NOT) Operator //

console.log( !false ); // true

console.log( !1 ); // false

**Sometimes we use a double !! (NOT) to convert a value to boolean type:**

console.log( !!"not-empty string" ); // true

console.log( !!null ); // false

**The first NOT transforms the value to boolean and returns the opposite, the second NOT inverses it again. After all, we have a plain value-to-boolean conversion.**

**The Ternary Operator**

A ternary operator evaluates a condition and executes a block of code based on the condition.

Syntax:

The**conditional** (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is false.

Condition ? expression1 : expression2

let marks = 50;

// check the condition

let result = (marks >= 40) ? 'pass' : 'fail';

console.log(`You ${result} the exam.`);

**Typeof Operator**

The typeof operator is used to check the type of a value. It will often evaluate to either primitive type, object or function. The value produced by the typeof operator is always string format:

console.log(typeof 124) // number

console.log(typeof 124n) //Bigint

console.log(typeof "Hint") // string

console.log(typeof true) // Boolean

console.log(typeof []) // object

console.log(typeof {}) // object

console.log(typeof null) // object

null vs undefined

undefined means a variable has been declared but has not yet been assigned a value. undefined is a type by itself (undefined). Unassigned variables are initialized by JavaScript with a default value of undefined.

var n;

console.log(n); // Undefined

Here as the variable is declared but not assigned to any value, the variable by default is assigned a value of undefined.

**null** is an object. It can be assigned to a variable as a representation of no value.

**Scope**

A scope can be defined as the region of the execution, a region where the expressions and values can be referenced.

There are 3 unique scope types:

The global scope, block scope and function scope. Each expects different things and has unique rules when it comes to variable definitions. Event callback functions follow the same rules as function scope, they are just used in a slightly different context. Loops can also have their own block-scope.

**Variable Definitions**

Variables can be defined using var, let or const keywords.

Of course, if you tried to refer to a variable that wasn’t defined anywhere, you would generate a ReferenceError error ”variable name is not defined”

console.log(apple); // undefined

var apple = 1

console.log(apple) // 1

Here apple is defined in global scope. But it can also be accessed from an inner block-scope. Anything (even a function definition) defined in global scope becomes available anywhere in your program. The value propagates into all inner scopes. When a variable is defined in global scope using var keyword, it also automatically becomes available as a property on window object.

Hoisting

If apple was defined using var keyword inside a block-scope, it would be hoisted back to global scope! Hoisting simply means ”raised” or ”placed on top of”. Hoisting is limited to variables defined using var keyword and function name defined using function keyword. Variables defined using let and const are not hoisted and their use remains limited only to the scope in which they were defined. As an exception, variables defined var keyword inside function-level scope are not hoisted. Commonly, when we talk about hoisting block-scope is implied. We will talk more about hoisting in just a moment!

Likewise, variables defined in global scope will propagate to pretty much every other scope defined in global context, including block-level scope, for-loop scope, function-level scope, and event callback functions created using setTimeout, setInterval or addEventListener functions.

console.log(orange) //undefined

{

    var orange = 2

}

Variable orange is hoisted to global scope. But the value of the hoisted variable is now undefined – not 1. Only its name definition was hoisted. Hoisting is like a safety feature. You should not rely on it when writing code. You may not retain the value of a hoisted variable in global scope, but you will still save your program from generating an error and halting execution flow. Thankfully, hoisting in JavaScript is automatic. When writing your program more than half of the time, you won’t even need to think about it.

**Function Name Hoisting**

Hoisting also applies to function names. But variable hoisting always takes precedence. We’ll see how that works in this section. You can call a function in your code, as long as it is defined at some point later:

fun(); // Hello from fun() function

function fun(){

    console.log("Hello from fun() function")

}

It goes without saying if the function was already defined prior to being called, there’d be no hoisting but everything would still work as planned. Statements inside a function’s body are executed when the function is called by its name. Nameless functions can still be assigned as values themselves.

**Defining Variables Inside Function**

Scope At this point you might want to know that variables defined inside a function will be limited only to the scope of that function. Trying to access them outside of the function will result in a reference error:

function hello(){

    var invite = "Welcome";

}

console.log(invite)  // Reference Error // Not Definded

var is defined in Global Scope, but its value propagates into the block scope as well. What actually happens is, when block scope 1 cannot find var definition in within its own brackets, it looks for it in the parent scope. If it finds it there, it inherits its value.

var a1 = "Prakash"

function mName(){

    console.log(a1) // Here the variable value takes from globle scope

}

mName() // Prakash

**Variable Types**

JavaScript is a dynamically-typed language. The type of the variable (defined using var or let keyword) can be assigned and changed at any time during the run-time of your application, after it was already compiled by browser’s JavaScript engine. The keywords var, let and const do not determine the variable’s type. Instead, they determine how the variable can be used: can it be used outside of the scope in which defined? Can it be re-assigned to another value during run-time? For example, var and let can, but const can’t.

**var**

The var keyword is still with us from original specification. You should probably start using let and const instead. For the most part it is still available but only to support legacy code.

**Let**

The let defines a variable but limits its use to the scope in which it was defined.

**const**

const is the same as let but you can’t re-assign it to a new value once defined.

**Scope Visibility Differences**

No Difference In Global Scope When variables are defined in global scope there is no differences between var, let and const in terms of scope visibility.

Variables defined using let and const are not hoisted. Only var is supporting to hoisting.

**In Function Scope**

However, when it comes to functions, all variable types, including var remain limited to their scope.

You cannot access variables outside of the function scope in which they were defined regardless of which keyword was used

**In Block-level Local Scope**

The let and const keywords conceal variable visibility to scope in which they were defined and its inner scopes. Scope visibility differences surface when you start defining variables inside local block-level scope or function-level scope.