**JavaScript**

JavaScript is a high-level object oriented, multi paradigm programming language. High-level languages typically have features like automatic memory management and a rich set of built-in libraries, and JavaScript being a high-level programming language simplify coding tasks. And as it being an object oriented programming language mostly everything revolbes around objects which are instances of classes. JavaScript being a multi paradigm programming language we can style the same piece of code in various ways. A **paradigm** refers to a style or way of thinking about and structuring code so in javaScript we can we can implement our code base in declarative style/ imperative style or even in functional style of programming.

**Role in Web Development**

JavaScript is most commonly used in web development for creating interactive and dynamic web pages. It runs on the client side in the user's browser, which allows it to respond to user actions instantly without needing to communicate with the server for every interaction. However, JavaScript is not limited to client-side scripting. With the advent of environments like Node.js, JavaScript can also be used for server-side programming.

How we can write the JavaScript code:

Basically the code can be written in many ways like,

1. **Inline Script :**

Here we write the script inside HTML element attributes. Here wedirectly write the script within an HTML element using the **onclick**, **onload**, or other event attributes.

|  |
| --- |
| <body>  <h1>Inline JavaScript Example</h1>  **<button onclick="alert('Button clicked!')">Click me</button>**  </body> |

1. **Internal Script:**

In this approach we add a **<script>** tag within the HTML file (mostly within the <head> tag ) and we can write the script within this tag.

|  |
| --- |
| <html>  <head>  <title>Internal JavaScript Example</title>  <**script**>  function showMessage() {  alert('Button clicked!');  }  **</script>**  </head>  <body>  <h1>Internal JavaScript Example</h1>  <button onclick="showMessage()">Click me</button>  </body>  </html> |

This approach is pretty bad because the coupling between the javascript content and the website content. So we should look forward in separating the script and the website content which means the UI part and the background logic should be separated from each other.

1. **External Script**

**(How to link a JavaScript file with the Html file !!! )**

JavaScript code is written in a separate file with a **.js** extension, and this file is linked to the HTML file using a **<script>** tag with the **src** attribute.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>External JavaScript Example</title>  </head>  <body>  <h1>External JavaScript Example</h1>  <button id="myButton">Click me</button>  **<script src="script.js"></script>**  </body>  </html>  Script.js  ----------  function showMessage() {  alert('Button clicked!');  }  document.getElementById('myButton').addEventListener('click', showMessage); |

Basically we use the <script> tag at the end of the <body> tag and we link the javascript file with an extension of .js so that the Browser knows at this point that the script is in a separate file and it has to load this content and evaluate it as well. But how does the browser knows which file has to be loaded and where the file is!! So we give the source location of the file with the src (Source) attribute inside the <script> tag which holds the information about the file that has to be loaded.

**Variable and Values in JavaScript [ var, let, const ]**

In JavaScript, you can declare variables in several ways using **var**, **let**, and **const**. Each has its own scope rules and characteristics. Here are the ways you can declare a variable **name = 'Sam';**:

**1. Using var**

The **var** keyword declares a variable that is function-scoped or globally-scoped if declared outside a function. Variables declared with **var** can be re-declared and updated.

var name = 'Sam';

**2. Using let**

The **let** keyword declares a variable that is block-scoped, meaning it is limited to the block, statement, or expression in which it is used. Variables declared with **let** can be updated but not re-declared within the same scope.

let name = 'Sam';

**3. Using const**

The **const** keyword declares a variable that is block-scoped and cannot be re-assigned. However, if the **const** variable holds an object, the properties of the object can still be modified.

const name = 'Sam';

* Generally while declaring a varibale name we give an appropriate meaningful name for them which should be self-explanatory and we should follow **CammelCase** pattern to write the variable name. And should not start a variable name with Upper case.
* While naming a variable there are certain restrictions that we need to follow like we can only use letters, numbers, underscore(\_) and $ dollar sign. Any special symbol used in naming a variable will lead to syntax error.

**Allowed**: Letters (A-Z, a-z), numbers (0-9, but not as the first character), underscores (\_), and dollar signs ($). But the name should not start with a number though.

**Not Allowed**: Spaces, special characters (other than $ and \_), starting with numbers, and reserved keywords.

* We can’t use reserved keywords. In JavaScript, reserved keywords have special meanings and are part of the language syntax. Using them as variable names will result in syntax errors. For example some reserved keywords are class/break/case/if/throw/do/else/finally/if/new/while etc. .

**Differences between var, let, and const**

| **var** | **let** | **const** |
| --- | --- | --- |
| The scope of a [*var*](https://www.geeksforgeeks.org/javascript-var/)variable is functional or global scope. | The scope of a[*let*](https://www.geeksforgeeks.org/javascript-let/) variable is block scope. | The scope of a [*const*](https://www.geeksforgeeks.org/javascript-const/) variable is block scope. |
| It can be updated and re-declared in the same scope. | It can be updated but cannot be re-declared in the same scope. | It can neither be updated or re-declared in any scope. |
| It can be declared without initialization. | It can be declared without initialization. | It cannot be declared without initialization. |
| It can be accessed without initialization as its default value is “undefined”. | It cannot be accessed without initialization otherwise it will give ‘referenceError’. | It cannot be accessed without initialization, as it cannot be declared without initialization. |
| These variables are hoisted. | These variables are hoisted but stay in the temporal dead zone untill the initialization. | These variables are hoisted but stays in the temporal dead zone until the initialization. |

Hoisting means, we can access/use those variables even before they are declared. But if we are accessing the hoisted variable then we will get its value as “undefined” as the variable is hoisted with a default value of **undefined.**

**How to hoist variables declared with var**

Variables declared with var are hoisted to the top of their global or local scope, which makes them accessible before the line they are declared. Here's an example:

console.log(number) // undefined

var number = 50

console.log(number) // 50

The number variable here has a global scope. Since it is declared with var, the variable is hoisted. This means that we can access the variable before the line where it was declared without errors.

But the variable is hoisted with a default value of **undefined**. So that's the value returned from the variable (until the line where the variable is declared with an initial value gets executed).

**Temporal Dead Zone !!**

When variables are declared with **let** or **const**, they are indeed hoisted, but they enter a "temporal dead zone" (TDZ) from the start of the block until the declaration is encountered. During this period, accessing the variable results in a **ReferenceError**.

**What is the Temporal Dead Zone?**

The Temporal Dead Zone refers to the time between the entering of a block scope and the point where the variable is declared and initialized. Although the variable is hoisted to the top of the block, it is not initialized, and any attempt to access it during the TDZ results in an error.

**Example to Illustrate TDZ** Consider the following example:

console.log(myVar); // undefined

myVar = 'var variable'; console.log(myLet); // ReferenceError: Cannot access 'myLet' before initialization let myLet = 'let variable';

**Explanation**

**var Hoisting:**

* The declaration of **var myVar** is hoisted to the top of the scope and initialized to **undefined**.
* Therefore, **console.log(myVar);** outputs **undefined** because the assignment happens later.

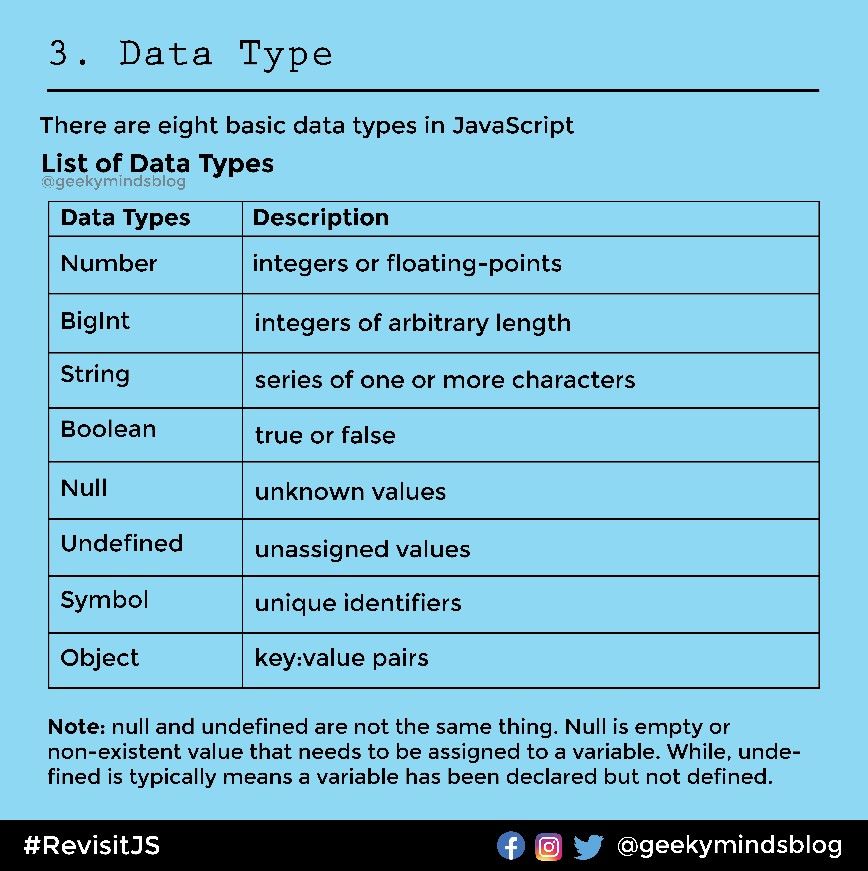
**let Hoisting:**

* The declaration of **let myLet** is hoisted to the top of the scope, but it is not initialized.
* **myLet** is in the TDZ from the start of the block until the declaration and initialization (**let myLet = 'let variable';**) are executed.
* Accessing **myLet** before its declaration results in a **ReferenceError**.

**Var / let / const when to use what :**

* Use **const** by default for all variables that do not need to be reassigned. This prevents accidental reassignments and helps make your code more predictable. To declare CONSTANT we use const which doesn’t change over the time (immutable variables). As they are immutable they should be initialized during declaration else we’ll get syntax error.
* Use **let** when you know the value of the variable needs to change at some later point of time(e.g., in loops or conditional blocks).
* Avoid using **var** unless you need to support older codebases or understand its quirks and scope behavior for specific use cases. (Old way)

**Data Types**



JavaScript is a **dynamically typed** (also called loosely typed) scripting language. In JavaScript, variables can receive different data types over time.   
JavaScript is considered a dynamically typed language, meaning that **types are associated with values rather than variables.** This has several implications for how you write and interact with code in JavaScript.

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

JavaScript is a **dynamic type language**, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine. You need to use **var** here to specify the data type. It can hold any type of values such as numbers, strings etc. For example:

1. var a=40;//holding number
2. var b="sambit";//holding string

Due to JS being dynamically typed we get more flexibility as we can re-assign different types of values to the same variable over the periode of time. And another big thing is that you don’t need to declare th type of the variable explacitely.

Dynamic typing simply means we can change the type of the variable whenever we want even after the variable is initialized already.

**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.

* [**Number:**](https://www.geeksforgeeks.org/javascript-numbers/) JavaScript numbers are always stored in double-precision 64-bit binary format IEEE 754. Unlike other programming languages, you don’t need int, float, etc to declare different numeric values.
* [**String:**](https://www.geeksforgeeks.org/javascript-strings/) JavaScript Strings are similar to sentences. They are made up of a list of characters, which is essentially just an “array of characters, like “Hello GeeksforGeeks” etc.
* [**Boolean:**](https://www.geeksforgeeks.org/javascript-boolean/) Represent a logical entity and can have two values: true or false.
* [**Null:**](https://www.geeksforgeeks.org/null-in-javascript/) This type has only one value that is *null.*
* [**Undefined:**](https://www.geeksforgeeks.org/undefined-in-javascript/) A variable that has not been assigned a value is *undefined.*
* [**Symbol:**](https://www.geeksforgeeks.org/javascript-symbol-method/) 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.
* [**BigInt:**](https://www.geeksforgeeks.org/bigint-in-javascript/) BigInt is a built-in object in JavaScript that provides a way to represent whole numbers larger than 253-1.

**Non-Primitive Data Types:**

non-primitive data types are essentially objects.It is also known as derived data types or reference data types.

For example Object, Map, RegExp, Array, Function, Date, WeakMap, WeakSet etc.

Why in JavaScript typeOf(null) is Object !! Does it make sense !!

The behavior where **typeof null** returns **"object"** is a well-known quirk in JavaScript, and it stems from a historical bug in the language's implementation. And later on it wasn’t even changed because of legacy reasons.

Then how null checks are done!

When you need to check for **null** values, use strict equality (**===**) instead of **typeof**.

The quirk where **typeof null** returns **"object"** is a legacy issue rooted in the early design of JavaScript. While it can't be changed without breaking existing code, understanding this behavior helps avoid pitfalls and allows you to write more reliable type-checking code. For better type safety, especially in larger projects, consider using TypeScript, which provides static typing and catches such issues at compile time.

**Javascript Operators**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Operator** | **Description** | **Example** | **Result** |
| **Arithmetic** | **+** | Addition | **5 + 3** | **8** |
|  | **-** | Subtraction | **5 - 3** | **2** |
|  | **\*** | Multiplication | **5 \* 3** | **15** |
|  | **/** | Division | **10 / 2** | **5** |
|  | **%** | Modulus | **10 % 3** | **1** |
|  | **\*\*** | Exponentiation | **2 \*\* 3** | **8** |
| **Assignment** | **=** | Assignment | **let x = 10** | **x** is **10** |
|  | **+=** | Addition assignment | **x += 5** | **x** is **15** |
|  | **-=** | Subtraction assignment | **x -= 3** | **x** is **7** |
|  | **\*=** | Multiplication assignment | **x \*= 2** | **x** is **20** |
|  | **/=** | Division assignment | **x /= 2** | **x** is **5** |
|  | **%=** | Modulus assignment | **x %= 3** | **x** is **1** |
| **Comparison** | **==** | Equal | **5 == '5'** | **true** |
|  | **!=** | Not equal | **5 != '5'** | **false** |
|  | **===** | Strict equal | **5 === 5** | **true** |
|  | **!==** | Strict not equal | **5 !== '5'** | **true** |
|  | **>** | Greater than | **10 > 5** | **true** |
|  | **>=** | Greater than or equal | **10 >= 10** | **true** |
|  | **<** | Less than | **5 < 10** | **true** |
|  | **<=** | Less than or equal | **5 <= 5** | **True** |
| **Logical** | **&&** | Logical AND | **true && true** | **true** |
|  | ` |  | ` | Logical OR |
|  | **!** | Logical NOT | **!true** | **false** |
| **Bitwise** | **&** | AND | **5 & 1** | **1** |
|  | ` | ` | OR | `5 |
|  | **~** | NOT | **~5** | **-6** |
|  | **^** | XOR | **5 ^ 1** | **4** |
|  | **<<** | Left shift | **5 << 1** | **10** |
|  | **>>** | Right shift | **5 >> 1** | **2** |
|  | **>>>** | Unsigned right shift | **5 >>> 1** | **2** |
| **String** | **+** | Concatenation | **'Hello' + ' ' + 'World'** | **'Hello World'** |
|  | **+=** | Concatenation assignment | **let greeting = 'Hello'; greeting += ' World';** | **greeting** is **'Hello World'** |
| **Other** | **condition ? expr1 : expr2** | Conditional (ternary) operator | **(5 > 3) ? 'Yes' : 'No'** | **'Yes'** |
|  | **,** | Comma operator | **let a = (1, 2);** | **a** is **2** |
|  | **typeof** | Type of | **typeof 42** | **'number'** |
|  | **instanceof** | Instance of | **let date = new Date(); date instanceof Date** | **true** |

**Operator Precedence in JavaScript**

Operator precedence determines the order in which operations are evaluated in expressions. Operators with higher precedence are evaluated before those with lower precedence. When operators have the same precedence, associativity determines the order of evaluation.

**Examples of Operator Precedence**

Consider the expression:

let result = 3 + 4 \* 2 / (1 - 5) \*\* 2;

To understand how JavaScript evaluates this expression, let's break it down according to operator precedence:

1. **Parentheses ()**: Expressions inside parentheses are evaluated first.(1 - 5) => -4
2. **Exponentiation \*\***: Exponentiation has higher precedence than multiplication, division, addition, and subtraction.(-4) \*\* 2 => 16
3. **Multiplication \* and Division /**: These operators have the same precedence and are evaluated from left to right (left-associative).4 \* 2 => 8 8 / 16 => 0.5
4. **Addition +**: This operator has lower precedence than multiplication, division, and exponentiation.3 + 0.5 => 3.5

So, the final result of the expression **3 + 4 \* 2 / (1 - 5) \*\* 2** is **3.5**.

**Examples to Illustrate Precedence and Associativity**

1. **Combining Multiplication and Addition:**let result = 2 + 3 \* 4;  
    // 3 \* 4 is evaluated first, result = 2 + 12, so result = 14
2. **Using Parentheses to Change Precedence:**let result = (2 + 3) \* 4;  
    // 2 + 3 is evaluated first, result = 5 \* 4, so result = 20
3. **Combining Multiple Operators:**let result = 10 - 2 \*\* 2 + 3 \* 2;  
    // 2 \*\* 2 is evaluated first, then 10 - 4, then 3 \* 2, then 6 + 6 // result = 10 - 4 + 6,   
   so result = 12

**Operator Precedence Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Precedence** | **Operator Type** | **Operators** | **Associativity** | **Description** |
| 1 | Grouping | **()** | n/a | Parentheses |
| 2 | Member Access | **[]**, **.** | left-to-right | Property access |
| 3 | Call/Creation | **new**, **()** | left-to-right | Function call, object creation |
| 4 | Unary | **!**, **typeof**, **+**, **-**, **++**, **--**, **delete**, **void** | right-to-left | Unary operations |
| 5 | Exponentiation | **\*\*** | right-to-left | Exponentiation |
| 6 | Multiplicative | **\***, **/**, **%** | left-to-right | Multiplication, division, modulus |
| 7 | Additive | **+**, **-** | left-to-right | Addition, subtraction |
| 8 | Bitwise Shift | **<<**, **>>**, **>>>** | left-to-right | Bitwise shift operations |
| 9 | Relational | **<**, **<=**, **>**, **>=**, **instanceof**, **in** | left-to-right | Relational comparisons |
| 10 | Equality | **==**, **!=**, **===**, **!==** | left-to-right | Equality comparisons |
| 11 | Bitwise AND | **&** | left-to-right | Bitwise AND |
| 12 | Bitwise XOR | **^** | left-to-right | Bitwise XOR |
| 13 | Bitwise OR | ` | ` | left-to-right |
| 14 | Logical AND | **&&** | left-to-right | Logical AND |
| 15 | Logical OR | ` |  | ` |
| 16 | Conditional (ternary) | **?:** | right-to-left | Ternary operator |
| 17 | Assignment | **=**, **+=**, **-=**, **\*=**, **/=**, **%=** | right-to-left | Assignment operations |
| 18 | Comma | **,** | left-to-right | Comma operator |

**Template Literals:**

Template literals in JavaScript are a powerful way to work with strings, allowing you to include variables, expressions, and multi-line strings more easily than traditional string concatenation methods. Introduced in ECMAScript 6 (ES6), template literals are enclosed by backticks (**`**) instead of single or double quotes.

**Features of Template Literals**

1. **Variable and Expression Interpolation**:
   * Embed variables and expressions directly within a string.
   * Use the **${}** syntax to insert the value of a variable or the result of an expression.
2. **Multi-line Strings**:
   * Create strings that span multiple lines without the need for concatenation or escape characters for new lines.
3. **Tagged Template Literals**:
   * Process template literals with a function to perform more advanced operations, such as escaping HTML or internationalization.

**Examples**

**1. Variable and Expression Interpolation**

With traditional string concatenation:

let name = "Alice"; let age = 25; let message = "Hello, my name is " + name + " and I am " + age + " years old."; console.log(message); // Output: Hello, my name is Alice and I am 25 years old.

With template literals:

let name = "Alice"; let age = 25; let message = `Hello, my name is ${name} and I am ${age} years old.`; console.log(message); // Output: Hello, my name is Alice and I am 25 years old.

**2. Multi-line Strings**

Without template literals:

let poem = "Roses are red,\n" + "Violets are blue,\n" + "Sugar is sweet,\n" + "And so are you."; console.log(poem);

With template literals:

let poem = `Roses are red,

Violets are blue,

Sugar is sweet,

And so are you.`;

console.log(poem);

**3. Embedding Expressions**

let a = 5; let b = 10;

console.log(`The sum of ${a} and ${b} is ${a + b}.`);

// Output: The sum of 5 and 10 is 15.

**4. Tagged Template Literals**

A tag function can process a template literal. This is useful for escaping, localization, or custom formatting.

function highlight(strings, ...values) { return strings.reduce((result, str, i) => `${result}<b>${str}</b>${values[i] || ''}`, ''); } let name = "Alice"; let age = 25; let message = highlight`Name: ${name}, Age: ${age}`; console.log(message);

// Output: <b>Name: </b>Alice<b>, Age: </b>25

The problems with traditional string concatenation,

* Code becomes verbose and harder to read with multiple **+** operators.
* Embedding complex expressions requires additional parentheses, making the code less readable.
* Creating multi-line strings involves cumbersome concatenation with newline characters (**\n**).
* Increased risk of syntax errors, such as missing **+** operators or mismatched quotes.

**Control Structure:**

Decision making with if/else Control structure

In JavaScript, decision-making is often implemented using **if/else** statements. These statements allow you to execute different blocks of code based on certain conditions. Here’s a basic overview of how to use **if**, **else if**, and **else** in JavaScript.

Syntax (put in a table left right)

if (condition) {

// code to be executed if the condition is true

}

if (condition) {

// code to be executed if the condition is true

} else {

// code to be executed if the condition is false

}

**Basic if Statement**

An **if** statement evaluates a condition (expression). If the condition is true, the block of code inside the **if** statement executes. If the condition is false, the code inside the **if** block does not execute.

let age = 18; if (age >= 18) { console.log("You are an adult."); }

**if/else Statement**

An **if/else** statement allows you to execute one block of code if the condition is true, and another block if the condition is false.

let age = 16;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

**if/else if/else Statement**

An **if/else if/else** statement can be used to test multiple conditions. The first block of code that meets a true condition will execute.

let score = 85;

if (score >= 90) {

console.log("You got an A.");

} else if (score >= 80) {

console.log("You got a B.");

} else if (score >= 70) {

console.log("You got a C.");

} else if (score >= 60) {

console.log("You got a D.");

} else {

console.log("You got an F."); }

**Nested if Statements**

You can also nest **if** statements within each other to handle more complex conditions.

let age = 20;

let hasPermission = true;

if (age >= 18) {

if (hasPermission) {

console.log("You can enter the club.");

} else {

console.log("You need permission to enter the club.");

}

} else {

console.log("You are not allowed to enter the club.");

}

**Type Conversion and Type Coercion**