**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.

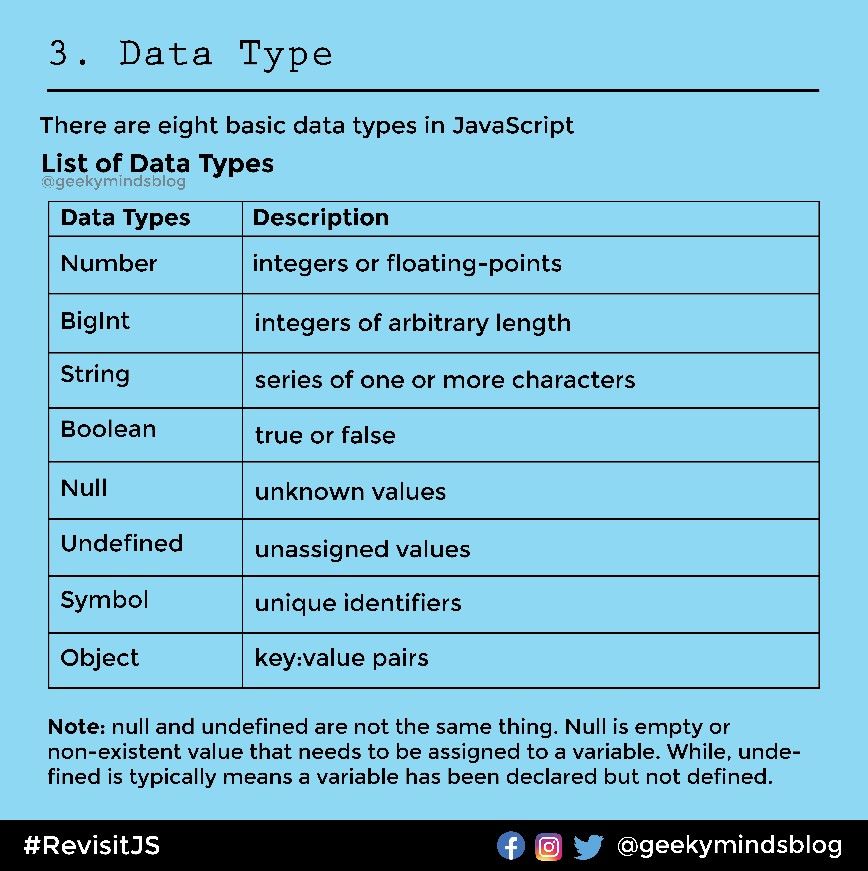
**let Hoisting:**

* 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**

As the name implies, type conversion is the process of converting a value from one type to another.

Values in JavaScript can be of different types. You could have a number, string, object, boolean – you name it. Sometimes, you may want to convert data from one type to another to fit a certain operation.

Type conversion can either be implicit (automatically done during code execution) or explicit (done by you the developer).

Implicit Type Conversion is also known (and more commonly referred to) as **Coercion** while Explicit Type Conversion is also known as **Type Casting**.

**What is Implicit Type Conversion (Coercion)?**

There are some operations that you might try to execute in JavaScript which are literally not possible. For example, look at the following code:

const sum = 35 + "hello"

Here, you're trying to add a number and a string. This is, practically speaking, not possible. You can only add numbers (**sum**) together or add strings (**concatenate**) together.

So what happens here if you try to run the code?

Well, JavaScript is a weakly typed language. Instead of JavaScript throwing an error, it coerces the type of one value to fit the type of the other value so that the operation can be carried out.

In this case, using the **+** sign with a number and a string, the number is coerced to a string, then the **+** sign is used for a concatenation operation.

const sum = 35 + "hello"

console.log(sum)

// 35hello

console.log(typeof sum)

// string

This is an example of coercion where the type of one value is coerced to fit the other so that the operation can continue.

With the plus sign, it is more ideal for the number to be converted to a string (instead of the string converted to a number). This is because a number equivalent to a string is NaN but a string equivalent for a number, say 15, is "15" – so it makes more sense to **concatenate** two strings than to **sum** a number and NaN.

Look at another example below:

const times = 35 \* "hello"

console.log(times)

// NaN

Here, we use times **\*** for a number and a string. There's no operation with strings that involves multiplication, so here, the ideal coercion is from string to number (as numbers have compatible operations with multiplication).

But since a string (in this case, "hello") is converted to a number (which is NaN) and that number is multiplied by 35, the final result is NaN.

Coercion is usually caused by different operators used between different data types:

const string = ""

const number = 40

const boolean = true

console.log(!string)

// true - string is coerced to boolean `false`, then the NOT operator negates it

console.log(boolean + string)

// "true" - boolean is coerced to string "true", and concatenated with the empty string

console.log(40 + true)

// 41 - boolean is coerced to number 1, and summed with 40

## Double Equality and Coercion (==)

In JavaScript, there's both the double equality operator (**==** which is called the **loose equality operator**) and the triple equality operator (**===** which is called the **strict equality operator**). You use both operators to compare values' equality.

How the Loose Equality Operator Works

The **loose equality operator** does a loose check. It checks if values are equal. The types are not a focus for this operator – only the values are the major factor.

What I mean here is **20**, a value of a number type, and **"20"**, a value of the string type, are equal when you use double equality:

const variable1 = 20

const variable2 = "20"

console.log(variable1 == variable2)

// true

Though the types are not equal, the operator returns true because the values are equal. What happens here is **coercion**.

When you use the **loose equality operator** with values of different types, what happens first is coercion. Again, this is where one value is converted to the type that fits the other, before the comparison occurs.

In this case, the **string "20"** is converted to a number type (which is 20) and then compared with the other value, and they are both equal.

Another example:

const variable1 = false

const variable2 = ""

console.log(variable1 == variable2)

// true

Here, variable1 is the value **false** (boolean type) and variable2 is the value **""** (an empty string, of the string type). Comparing both variables with the double equality returns true. That's because the empty string is coerced to a boolean type (which is **false**).

### **How the Strict Equality Operator Works (===)**

This operator does a strict check – that is, it strictly checks the values compared, as well as the types. Type coercion does not occur here, so there are no unexpected answers. Here are the examples from above:

const variable1 = 20

const variable2 = "20"

console.log(variable1 === variable2)

// false

const variable3 = false

const variable4 = ""

console.log(variable3 === variable4)

// false

In the case of variable1 and variable2, they have the same values, but the types are not the same. So the triple equality returns false.

In the case of variable3 and variable4, they have the same values (if one is converted to the type of the other) but the types are not the same, so the triple equality returns false this time, too.

## What is Explicit Type Conversion (Type Casting)?

Here, you explicitly convert a value from one type to another. This can also be for you to execute a certain operation successfully.

To explicitly convert types, you use the type Constructors. For example, to convert a number to a string:

const number = 30

const numberConvert = String(number)

console.log(numberConvert)

// "30"

console.log(typeof numberConvert)// string

Another example is to convert a number to a boolean:

const number = 30

const numberConvert = Boolean(number)

console.log(numberConvert)

// true

console.log(typeof numberConvert)

// boolean

And one more example, to convert a boolean to a string:

const boolean = false

const booleanConvert = String(boolean)

console.log(booleanConvert)

// "false"

console.log(typeof booleanConvert)

// string

In these examples, we explicitly convert a value from one type to another. What are cases where you need to do this?

This is useful when you don't know what type you're expecting for a value. For example, data coming from an API. Let's say an API is configured to return a string, maybe "50" and you want to compare it to a number using strict equality like this:

const apiData = {

rate: "50"

}

console.log(apiData.rate === 50)

// false

In such a case, you want to first ensure that the value is a number type explicitly (instead of relying on the double equality to trigger coercion) before doing the check:

const apiData = {

rate: "50"

}

const rate = Number(apiData.rate)

console.log(rate === 50)

// true

**Differences between ( == loose equality ) & ( === Strict equality )**

|  |
| --- |
|  |
| | **Feature** | **== (Equality Operator)** | **=== (Strict Equality Operator)** | | --- | --- | --- | | **Type Comparison** | Compares values with type conversion if necessary | Compares values without type conversion | | **Type Strictness** | Less strict (types can differ) | More strict (types must be the same) | | **Performance** | Slower due to possible type conversion | Faster due to no type conversion | | **Use Case** | Used when type conversion is acceptable | Used when exact type and value match is required | | **Type Handling** | Converts types to match before comparison | Does not convert types, must match exactly | | **Common Examples** | 1 == "1" (true), 0 == false (true) | 1 === "1" (false), 0 === false (false) | | **Null and Undefined** | null == undefined (true) | null === undefined (false) | | **Object Comparison** | Compares object references | Compares object references | | **NaN Handling** | NaN == NaN (false) | NaN === NaN (false) | | **Recommendation** | Use cautiously, potential for unexpected results | Use for most comparisons to ensure precision | |

**Key Differences**

* **Strictness**: === is strict and requires both value and type to be the same, while == is loose and performs type conversion if necessary.
* **Performance**: === tends to be faster as it doesn't involve type conversion.
* **Common Use**: === is recommended for most cases to avoid unexpected results due to type conversion. Use == when you specifically want type conversion.

**Truthy , Falsy**

In JavaScript, the concepts of "truthy" and "falsy" values extend the language's flexibility and provide more nuanced control over conditional logic. While true and false are explicit Boolean values, "truthy" and "falsy" values refer to non-Boolean values that, when evaluated in a Boolean context (such as an if statement or a logical operation), are implicitly converted to true or false.

**What Are Truthy and Falsy Values?**

* **Truthy Values**: Values that evaluate to true when used in a Boolean context.
* **Falsy Values**: Values that evaluate to false when used in a Boolean context.

**Falsy Values**

Falsy values are the values that will become false when converted to boolean

In JavaScript, the following values are considered falsy:

1. false
2. 0 (zero)
3. -0 (negative zero)
4. 0n (BigInt zero)
5. "" (empty string)
6. null
7. undefined
8. NaN (Not-a-Number)

So if we try to convert any of these values to boolean then we will get a false result.

if (0) {

console.log("This won't run because 0 is falsy.");

}

if ("") {

console.log("This won't run because an empty string is falsy.");

}

**Truthy Values**

Any value that will become true when converted to boolean is Truthy value .\

This includes:

1. true
2. Any non-zero number (positive or negative)
3. Any non-empty string
4. Objects
5. Arrays (even empty ones)
6. Functions
7. Non-zero BigInt values

Example:

if (1) {

console.log("This will run because 1 is truthy.");

}

if ("hello") {

console.log("This will run because a non-empty string is truthy.");

}

if ({}) {

console.log("This will run because an object is truthy.");

}

**Why Use Truthy and Falsy Values?**

1. **Simplified Conditionals**:
   * Truthy and falsy values simplify conditional statements, allowing for concise and readable code.

let name = ""; // falsy

if (!name) {

console.log("Name is required");

}

let count = 42; // truthy

if (count) {

console.log("Count is valid");

}

1. **Default Values**:
   * They enable setting default values using logical OR (||) or nullish coalescing (??).

javascript

Copy code

let username = user.name || "Guest"; // if user.name is falsy, use "Guest"

1. **Error Handling**:
   * Simplify error handling by checking for null, undefined, or empty values in one go.

function greet(name) {

if (!name) {

throw new Error("Name is required");

}

console.log(`Hello, ${name}`);

}

**Examples in Context**

* **Checking Object Properties**:

let user = {

name: "Alice",

age: 0

};

if (user.name) {

console.log("User has a name"); // runs

}

if (!user.age) {

console.log("User age is falsy"); // runs because 0 is falsy

}

* **Loop Control**:

let items = [];

while (items.length) {

console.log("This won't run because items is an empty array (falsy).");

}

While true and false are explicit Boolean values, truthy and falsy values in JavaScript allow for more flexible and concise conditional checks. They provide a way to handle various data types in a Boolean context without explicit conversion, making the code more expressive and often easier to read. This is particularly useful in JavaScript's loosely-typed environment, where variables can hold different types of values.

**Switch statement syntax:**

|  |
| --- |
| **switch** (expression) {  case value1:  // Code to be executed if expression === value1  break;  case value2:  // Code to be executed if expression === value2  break;  // ...  default:  // Code to be executed if expression does not match any case  }  Example:  let fruit = 'apple';  switch (fruit) {  case 'banana':  console.log('Bananas are yellow.');  break;  case 'apple':  console.log('Apples are red or green.');  break;  case 'orange':  console.log('Oranges are orange.');  break;  default:  console.log('Unknown fruit.');  } |

**Strict mode:**

Strict mode is a powerful feature in JavaScript that enhances code reliability, security, and performance by enforcing stricter parsing and error handling. By default, strict mode is not enabled, allowing developers to opt-in as needed. This approach maintains backward compatibility with legacy code while providing the benefits of strict mode for new codebases. Enabling strict mode is highly recommended for writing clean, maintainable, and secure JavaScript code.

Generally without strict mode turned on if we use a variable that isn’t even declared then we will not get any syntax error but if we turn on the strict mode then we get an error message for the syntax error. These things makes easier to find the problems while debugging or creating bigger applications.

We can enable strict mode just by adding a String (‘use strit’ ) to the top of the script.   
Basically we can enable the strict mode for either the whole script by adding the String ‘use Strict’ on top of the .js file (as the first line) or if we want to enable the strict mode for a specific function then we can use the String as the first line of any function.

Enabling strict mode:

“use strict” at top of the js file to enable it for whole script

At block level we can enable it like:

function myFunction() {

**'use strict';**

var y = 3.14;

console.log(y); // This code runs in strict mode

}

<https://www.freecodecamp.org/news/how-to-use-strict-mode-in-javascript/>

[**Changes in strict mode**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#changes_in_strict_mode)

Strict mode changes both syntax and runtime behavior. Changes generally fall into these categories:

* changes converting mistakes into errors (as syntax errors or at runtime)
* changes simplifying how variable references are resolved
* changes simplifying eval and arguments
* changes making it easier to write "secure" JavaScript
* changes anticipating future ECMAScript evolution.

[**Converting mistakes into errors**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#converting_mistakes_into_errors)

Strict mode changes some previously-accepted mistakes into errors. JavaScript was designed to be easy for novice developers, and sometimes it gives operations which should be errors non-error semantics. Sometimes this fixes the immediate problem, but sometimes this creates worse problems in the future. Strict mode treats these mistakes as errors so that they're discovered and promptly fixed.

**Assigning to undeclared variables**

Strict mode makes it impossible to accidentally create global variables. In sloppy mode, mistyping a variable in an assignment creates a new property on the global object and continues to "work". Assignments which would accidentally create global variables throw an error in strict mode:

"use strict";

let mistypeVariable;

// Assuming no global variable mistypeVarible exists

// this line throws a ReferenceError due to the

// misspelling of "mistypeVariable" (lack of an "a")

mistypeVarible = 17;

**Failing to assign to object properties**

Strict mode makes assignments which would otherwise silently fail to throw an exception. There are three ways to fail a property assignment:

* assignment to a non-writable data property
* assignment to a getter-only accessor property
* assignment to a new property on a [non-extensible](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible) object

For example, [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) is a non-writable global variable. In sloppy mode, assigning to NaN does nothing; the developer receives no failure feedback. In strict mode, assigning to NaN throws an exception.

"use strict";

// Assignment to a non-writable global

undefined = 5; // TypeError

Infinity = 5; // TypeError

// Assignment to a non-writable property

const obj1 = {};

Object.defineProperty(obj1, "x", { value: 42, writable: false });

obj1.x = 9; // TypeError

// Assignment to a getter-only property

const obj2 = {

get x() {

return 17;

},

};

obj2.x = 5; // TypeError

// Assignment to a new property on a non-extensible object

const fixed = {};

Object.preventExtensions(fixed);

fixed.newProp = "ohai"; // TypeError

**Failing to delete object properties**

Attempts to [delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) a non-configurable or otherwise undeletable (e.g. it's intercepted by a proxy's [deleteProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/Proxy/deleteProperty) handler which returns false) property throw in strict mode (where before the attempt would have no effect):

JSCopy to Clipboard

"use strict";

delete Object.prototype; // TypeError

delete [].length; // TypeError

Strict mode also forbids deleting plain names. delete name in strict mode is a syntax error:

JSCopy to Clipboard

"use strict";

var x;

delete x; // syntax error

If the name is a configurable global property, prefix it with [globalThis](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis) to delete it.

JSCopy to Clipboard

"use strict";

delete globalThis.x;

**Duplicate parameter names**

Strict mode requires that function parameter names be unique. In sloppy mode, the last duplicated argument hides previous identically-named arguments. Those previous arguments remain available through [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), so they're not completely inaccessible. Still, this hiding makes little sense and is probably undesirable (it might hide a typo, for example), so in strict mode, duplicate argument names are a syntax error:

JSCopy to Clipboard

function sum(a, a, c) {

// syntax error

"use strict";

return a + a + c; // wrong if this code ran

}

**Functions in JavaScript:**

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output. To use a function, you must define it somewhere in the scope from which you wish to call it.

[**Function declarations**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#function_declarations) (Hoisted)

A **function definition** (also called a **function declaration**, or **function statement**) consists of the [function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function) keyword, followed by:

* The name of the function.
* A list of parameters to the function, enclosed in parentheses and separated by commas.
* The JavaScript statements that define the function, enclosed in curly braces, { /\* … \*/ }.

For example, the following code defines a simple function named square:

function square(number) {

return number \* number;

}

This is how we can declare a function and we can then reuse he function whenever we want. But in JS this is not the only way we can declare a function rather there are various ways. And one is where the function would be anonomous whtiout having a name. We can declare the function as an expression and we can assign that expression to a variable and invoke the anonomous function using that variable.

**Function Expression:** (Not - Hoisted)

It is similar to function declarations, its just that we don’t need to name this function anymore. This will be a function without any name. So its just kind of a expression and we can assign this expression to some variable as well to reuse / invoke this function.

let funcVar = function(paramA, paramB) {  
 // Set of statements  
}

 ex:

const squire = function (num1,num2) {

    return num1\*num2;

 }

 console.log(squire(12,12));

Function expression works exactly the same as function declarations its just that its just declared without any name and can be used as a statement and can hold it with some variable.

But there’s on **Difference between function declaration and function Expression**:

Apart from the difference that the expressions are without names so they are anonomous there’s another big difference that is the **Hoisting**. The function declarations are Hoisted that means we can call/invoke these functions even before they are declared. But when it comes to function expressions they are not Hoisted, so we can’t invoke these functions before they are declared else we will get an syntax error (ReferenceError : can’t access ‘funcName’ before initialization)

 console.log(squire(12,12));

 function squire (number1,number2){

    return number1\*number2;

 }

 // This is correct because function declarations are Hoisted

 console.log(squire(12,12));

 const squire = function (num1,num2) {

    return num1\*num2;

 }

 // This is not correct because function expressions are not Hoisted

script.js:20 Uncaught ReferenceError: Cannot access 'squire' before initialization

at script.js:20:14

**Arrow Functions:**

Another

[**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.

Arrow functions are also anonomous functions without any name and special thing is that they are not bound by any identifier that means we don’t need to hold them with a variable. The arrow functions are declared without the “**function keyword**”.

**Arrow functions are not hoisted. They must be defined before they are used.**

**Syntax:**

let function\_name = (argument1, argument2 ,..) => expression

//const varName = (args) => expression ;

const squire = (num1,num2) => num1 \* num2 ;

console.log(squire(12,12));

Arrow functions are a concise way to write function expressions in JavaScript, introduced in ES6 (ECMAScript 2015). They offer a shorter syntax and come with some specific behavior differences compared to traditional function expressions.

//with multiple lines expression and a single argument

const calcYear = birthYear => {

    const currentYear = new Date().getFullYear();

    const age = currentYear - birthYear ;

    return age

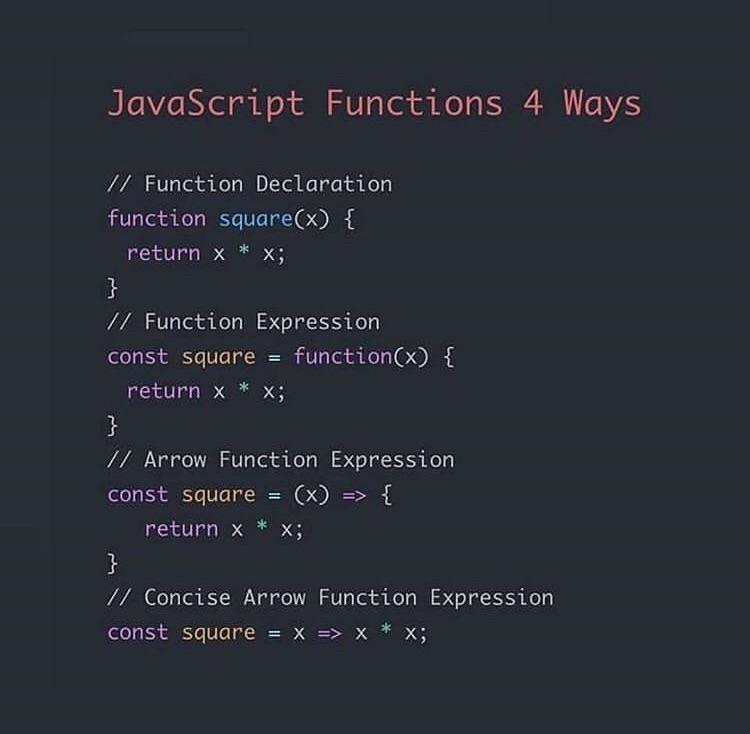
}

console.log(calcYear(1998));

When we use a single argument the we don’t even need to use the braces, we can directly have tem without any braces ( ).

The this keyword behaves differently in arrow functions compared to regular functions.

1. **Lexical scoping:** Arrow functions do not have their own this context. Instead, they inherit the this value from the enclosing scope. This is known as lexical scoping.
2. **No binding of this:** Unlike regular functions, arrow functions do not bind their own this. In regular functions, this is determined by how the function is called, leading to potential issues in certain situations. Arrow functions eliminate this ambiguity.
3. **Use of surrounding this:** The value of this in an arrow function is the same as the value of this in the surrounding (enclosing) scope. This can be beneficial when working with callback functions or methods within objects.
4. **Commonly used in callbacks:** Arrow functions are often preferred in callback functions where the lexical scoping of this can be advantageous, avoiding the need for workarounds like using .bind() or creating a closure.



**Differences between Traditional and Arrow Functions:**

| **Feature** | **Traditional Function** | **Arrow Function** |
| --- | --- | --- |
| **Usage as a Method** | Suitable, has its own this binding. | Not suitable, lacks its own this binding. |
| **Use of yield within its body** | Can use yield in a generator function. | Unable to use yield within its body. |
| **Presence of Return Statements** | Can be used with return statements. | Should not be used if return statements exist. |
| **Keyword Targeted (let, var, const)** | No specific restriction on variable keywords. | No specific restriction on variable keywords. |
| **Methods (call, apply, bind)** | Suited for methods, allows setting a scope. | Not suitable for methods that require scope setting (call, apply, bind). |
| **Object Creation (new keyword)** | Can be used as a constructor function. | Cannot be used with the new keyword to create a new object. |
| **Presence of arguments object** | Has arguments object available. | Lacks the arguments object. |
| **Presence of prototype property** | Has a prototype property. | Lacks the prototype property. |

**JavaScript Objects**

A javaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc. These objects will be having their own properties (data) and behaviour (functions), so we can illustrate real world entities using these objects.

JavaScript is an object-based language. Everything is an object in JavaScript. JavaScript is template based not class based. Here, **we don't create class to get the object. But, we direct create objects**.

There are several ways of creating an Object in javascript.

**1. Object Literal Notation:**

const person = {

    firstName: "John",

    lastName: "Doe",

    age: 30,

    friends: ["sam", "arya", "Lice"]

};

**2. Using the new Object syntax:**

const person = new Object();

person.firstName = "John";

person.lastName = "Doe";

person.age = 30;

person.friends =  ["sam", "arya", "Lice"] ;

**Accessing Object properties :**

We can access the properties of an object in 2 different ways where we can use the dot(.) notation to access a property or we can use the bracket notation to access the property.

person.firstName

// john

person['firstName']

// john

The same way we can access the properties of the Object we can even create/delete/modify the properties of the Object.

const person = new Object();

person.firstName = "John";

person.lastName = "Doe";

person.age = 30;

person.friends =  ["sam", "arya", "Lice"] ;

person.firstName = "sam" //we changed it from "john" to "sam"

person.middleName = "Kumar"; // added another property

delete person.lastName; // deleted lastName property

console.log(person)

//{firstName: 'sam', age: 30, friends: Array(3), middleName: 'Kumar'}

**Functions in JavaScript:**

As an object is a collection of properties i.e, data and behaviours that are functions, so we can have a function inside an Object as well. Where using the Object reference we can invoke the function and get the behaviour.

We can directly define an function inside the Objcet literal.

const person = {

    firstName: "John",

    lastName: "Doe",

    age: 30,

    friends: ["sam", "arya", "Lice"],

    // Method definition

    fullName: function() {

        return this.firstName + " " + this.lastName;

    },

    greet() {

        console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);

    }

};

console.log(person.fullName());

console.log (person[“fullName”]();

person.greet()

    fullName: function() {

        return this.firstName + " " + this.lastName;

    },

We can consider this as an property holding a function value, just as a property holding an array as value here a property is holding a function.

And as its just acts as a property holding a function so as we can access a property with dot ( . ) nptatopm or bracket notation here we can access this function same way as well.