

Javascript

What is Javascript?

JavaScript is a **high-level**, **dynamic**, and **multi-paradigm** programming language primarily used for creating interactive and dynamic content on webpages. It supports **object-oriented**, **functional**, and **event-driven** programming styles, making it versatile for various tasks. As one of the core technologies of the web, alongside **HTML** and **CSS**, **JavaScript** plays a vital role in building modern, user-friendly websites and web applications.

History of JavaScript: -

- JavaScript, one of the three core web development languages alongside **HTML** and **CSS**, was created to add interactivity to static web pages. In 1995, Brendan Eich, a programmer at Netscape, developed **JavaScript** in just 10 days. Initially named **Mocha**, it was later renamed **LiveScript** and finally **JavaScript** to align with the popular Java language at the time.
- The language combines syntax from Java, first-class functions from Scheme, and prototype-based inheritance from Self. Its adoption by all major browsers made it an essential tool for modern web development.

JavaScript vs. Java: -

- The name “**JavaScript**” often causes confusion, leading some to think it is directly related to **Java**. However, apart from some syntactic similarities, the two are entirely different programming languages.
- When JavaScript was introduced in **1995**, Java was heavily marketed and widely popular. Netscape decided to name their new scripting language “**JavaScript**” as a marketing strategy to leverage Java’s success and help the new language gain quick acceptance.

What Is ECMAScript?

- When JavaScript was first introduced by **Netscape**, different browser vendors, including Microsoft, created their own versions of JavaScript with different names and syntax. This caused issues for developers, as code that worked in one browser often didn’t work in another. To resolve this, all the vendors agreed to standardize on the same language, JavaScript.
- **Netscape** then submitted JavaScript to the “**European Computer Manufacturers Association (ECMA)**” for standardization, resulting in the language being officially named **ECMAScript**.
- **ECMA** is an international standards organization that creates and maintains standards for information and communication technologies. It ensures consistency, compatibility, and innovation across different technologies.
- Though closely related, **JavaScript** and **ECMAScript** are not the same. JavaScript is ECMAScript with additional features provided by host environments, such as **web browsers** and **Node.js**, which add their own APIs. In simple terms, ECMAScript is the core language of JavaScript without these additional features.

Evolution of ECMAScript Versions: -

The ECMAScript programming language has undergone a series of advancements and updates since its inception. Here's a brief overview of its evolution:

1. **ECMAScript 1 (1997):** The journey began in 1997 with the release of **ECMAScript 1**. This inaugural version laid the foundation for the language and introduced essential features that formed the basis for future developments. It standardized the **core syntax** and **structure** of the language.
2. **ECMAScript 2 (1998):** A year later, **ECMAScript 2** was introduced. This version was mainly a revision of **ECMAScript 1** to align with ISO/IEC standards, and it did not introduce significant new features but helped to formalize the language's specifications.
3. **ECMAScript 3 (1999):** Released in 1999, **ECMAScript 3** was a major update that refined and expanded upon the features of **ECMAScript 2**. This version standardized important features such as **regular expressions**, better **string handling**, and improved **error handling**, setting the foundation for more advanced versions of JavaScript.
4. **ECMAScript 4 (Abandoned):** Although planned, **ECMAScript 4** never saw the light of day. This version was abandoned during development and never finalized due to disagreements and challenges among stakeholders. The development efforts were redirected towards other iterations of the language.
5. **ECMAScript 5 (2009):** **ECMAScript 5**, released in 2009, was a major update that modernized the language. It introduced features like **strict mode** (a way to opt-in to a more secure and less error-prone subset of JavaScript), **getters** and **setters**, **JSON support**, and improved **array handling**.
6. **ECMAScript 6 (2015):** **ECMAScript 6**, also known as **ES6** or **ECMAScript 2015**, marked a significant milestone in the language's evolution. released in 2015. Key additions included **arrow functions**, **classes**, **promises**, **modules**, **template literals**, **destructuring**, and more. ES6 is often regarded as a complete reworking of JavaScript, making it much more aligned with modern programming paradigms.
7. **Subsequent Versions (Starting from 2016):** Starting in 2016, **ECMAScript** adopted an annual release cycle, delivering incremental updates each year rather than waiting for large versions. These versions continue to refine and introduce new features. Notable updates included new syntax and features such as **async/await** (ES2017), **shared memory and atomics** (ES2017), and **native modules** (ES2020). This ongoing update pattern has made ECMAScript increasingly adaptive to developer needs and the changing landscape of web development.

The evolution of ECMAScript showcases its adaptability and responsiveness to the demands of the programming community. With each version, the language evolves, enabling developers to write more efficient, expressive, and feature-rich code.

What are variables?

A variable is like a **container** where we can store **information (values)** that we want to use later in our code. Imagine it as a **labeled box** where we can put something (like a **number**, **word**, or **data**), and whenever we need it, we can refer to that box by its name to get the value inside it.

How to Create a Variable

In JavaScript, we can **create (declare)** a **variable** using these three keywords: **var**, **let**, and **const**.

Steps:

1. **Declare the variable:** Tell JavaScript we want to create a variable.
 - Use **let**, **const**, or **var** keyword followed by the variable name.

2. **Assign a value:** we put a value (like a **number**, **word**, or **data**) inside the **variable**.

- Use '=' to assign the value.

```
let name = "Ajay"; // 'name' is the variable, and "Ajay" is the value stored in it
let age = 25;      // Variable 'age' stores the number 25
const city = "Paris"; // Variable 'city' stores the string "Paris"
```

Now, we can use the variable name anywhere in our code, and it will have the values.

Why Use Variables?

1. **Reusability:** we don't have to repeat the same value multiple times.
2. **Dynamic:** we can change the value stored in a **variable** (with **let** or **var**).
3. **Readability:** Variables make our code easier to understand, especially when we give them meaningful names.

Naming Convention and general rules for Variables:


General Rules:

1. Must start with a **letter**, **underscore _**, or dollar sign **\$**. Variables cannot start with a **number**.
 - let name = "John";
 - let _age = 25;
 - let \$price = 10;
 - let 1stPlace = "Winner"; (This will cause an error because it starts with a number.)
2. Can contain **letters**, **numbers**, **underscores (_)**, and **dollar signs (\$)** after the first character.
 - let user123 = "Alice";
 - let _data_value\$ = 45;
3. Cannot use JavaScript **reserved words** (like **let**, **const**, **function**, **class**, etc.) as variable names.
 - let function = 10; (This will cause an error because function is a reserved word.)
4. Case-sensitive (**myVariable** and **myvariable** are different)

Best Practices:

1. **CamelCase:** Start the variable name with a **lowercase letter** and **capitalize** the **first letter** of each **subsequent word**.

Example: firstName, totalAmount, isLoggedIn.



```
let userName = "Ajay";
let userAge = 22;
let isLoggedIn = true;
```

2. **Meaningful Names:** Use **descriptive** and **meaningful** names that explain what the variable is used for.

Example: Instead of **x**, use **totalPrice**.

```
let totalPrice = 100;  
let isLoggedIn = true;
```

3. **Constants in UPPERCASE (For const variables):** Use **UPPERCASE** letters with underscores for **constant values** that shouldn't change.

Example: `const MAX_USERS = 100;` , `const API_KEY = "1234abcd";`

```
const MAX_ATTEMPTS = 5;  
const API_URL = "https://api.example.com";
```

4. **Avoid Starting with Underscore (_) or Dollar Sign (\$):** Although allowed, these are typically used for specific purposes, like **private** properties in some coding styles.

Example: `_privateVar` (for **internal/private** use), `$element` (for jQuery elements).

```
let _internalData = "secret";  
let $element = document.querySelector("#myElement");
```

Examples of Properly Named Variables:

```
let firstName = "Alice"; // camelCase for regular variables  
const PI = 3.14159;      // UPPERCASE for constants  
let isLoggedIn = true;   // Descriptive and meaningful name  
let userAge = 25;        // Use of words for better understanding
```

While these conventions aren't enforced by **JavaScript**, following them improves **code readability** and **maintainability**.

What are Data Types: -

Data types in JavaScript **describe the various kinds of data we can work with and store in variables**.

JavaScript data types are divided into two types.

- **Primitive Data Types**
- **Non-Primitive Data Types**

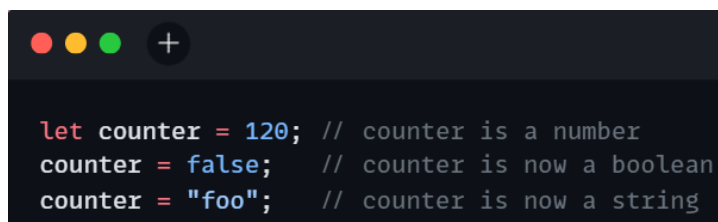
Primitive Data Types: -

Primitive data types are **immutable** (**Immutable** means something that cannot be changed after it has been created. In programming, an immutable value is one that, once set, cannot be modified. For example, in JavaScript, **strings** are **immutable**—if we try to change a string, a **new string** is created instead of modifying the original.) and represent simple values.

There are seven primitive data types in JavaScript:

1. **Number**
2. **string**
3. **Boolean**
4. **Undefined**
5. **Null**
6. **symbol**
7. **bigint**

JavaScript is a **dynamically typed language**, meaning we don't need to declare the **data type** of a variable when we define it. In other words, a **variable** can hold a value of different types. For example:



```
let counter = 120; // counter is a number
counter = false;  // counter is now a boolean
counter = "foo";  // counter is now a string
```

Number:-

In JavaScript, the number type represents **numeric values** (both **integers** and **floating-point** numbers).

- **Integers** - Numeric values without any decimal parts. Ex: 3, -74, etc.
- **Floating-Point** - Numeric values with decimal parts. Ex: 3.15, -1.3, etc.



```
let age = 25;      // Integer
let price = 99.99; // Float
```

String: -

In JavaScript, a **string** is a sequence of zero or more characters enclosed in single (' '), double (" "), or backticks (` `).



```
let name = "Alice";    // Double quotes
let greeting = 'Hello'; // Single quotes
let message = `Hi, ${name}`; // Template literal
```

Boolean: -

A Boolean data can only have one of two values: **true** or **false**. Useful in conditional logic.

```
let isReady = true;
let hasPermission = false;
```

Undefined: -

In JavaScript, **undefined** represents the absence of a **value**. If a variable is declared but the value is not assigned, then the value of that variable will be **undefined**. For example,

```
let name;
let value; // undefined
console.log(name); // undefined
```

Null: -

In JavaScript, **null** represents an **explicitly empty** or **non-existent value**.

```
let number = null;
console.log(number); // null
```

Symbol: -

A Symbol is a **unique** and **primitive value**. This data type was introduced in **ES6**. Represents a **unique identifier**. Useful for creating **unique keys** in objects.

```
let uniqueID = Symbol("id");
```

BigInt: -

BigInt is a type of number used to represent **large integers** that are larger than the **Number** type can safely represent. A BigInt number is created by appending **n** to the end of an integer.

Note: The regular number data type can handle values less than $(2^{53} - 1)$ and greater than $-(2^{53} - 1)$.

The result of 2^{53} is 9,007,199,254,740,992.

```
let bigNumber = 9007199254740991n;
let largeNumber = 123456789012345678901234567890n;
```

Non-primitive Data Types: -

Non-primitive data types in JavaScript are complex data types that can hold **multiple values** and more **complex entities**. Non-primitive data types, also known as **reference types**. Unlike primitive data types, **non-primitive** types are **mutable** and are **passed by reference**. The two key non-primitive data types in JavaScript are:

Object: -

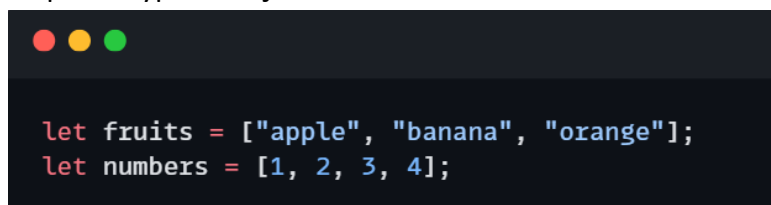
An **object** is a **collection of key-value pairs (properties and methods)**. **Keys** are usually **strings** (or **Symbol**), and **values** can be any type. Everything is an object in Javascript.



```
let person = {  
  name: "Alice",  
  age: 30,  
  isStudent: false  
};
```

Array: -

A special type of **object** used to store **ordered lists of data** that can be of **any data type**.



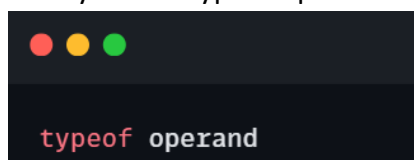
```
let fruits = ["apple", "banana", "orange"];  
let numbers = [1, 2, 3, 4];
```

typeof Operator: -

The **typeof operator** in JavaScript is used to determine the **type of a value** or **expression**. It returns a **string** indicating the type of the operand.

Syntax of typeof Operator:


The syntax of typeof operator is:



```
typeof operand
```

Here, operand is a variable name or a value.

Example:



```
let age = 25;
console.log(typeof age); // "number"

let name = "Alice";
console.log(typeof name); // "string"

let person = { name: "John", age: 30 };
console.log(typeof person); // "object"


let greet = function() { return "Hello"; };
console.log(typeof greet); // "function"
```

var, let and const keywords: -

In JavaScript, **const**, **let**, and **var** are **keywords** used to declare variables.

Var: -


The **var** is the oldest keyword to declare a variable in JavaScript.



```
var x = 10;
var firstName = "Ajay";
```

Const: -


Used for variables that won't be **reassigned**. Must be **initialized when declared**.



```
var x = 10;
var firstName = "Ajay";
```

Let: -

Used for variables that may be **reassigned**. It can be declared without **initialization**.



```
let count = 1;
count = 2; // This is allowed
```


JavaScript Operators: -

What are Operators?

Operators in JavaScript are **symbols** or **keywords** used to perform operations on values (operands). They allow us to manipulate data, perform calculations, compare values, and make decisions based on conditions.

Arithmetic Operators: -

JavaScript **Arithmetic Operators** are used to perform mathematical operations on **numeric values**.

1. Addition (+):

- Adds two numbers.
- Can also be used for **string concatenation**.



```
let sum = 5 + 3; // 8
let concat = "Hello" + " World"; // "Hello World"
```

2. Subtraction (-):

- Subtract the **right operand** from the **left operand**.



```
let difference = 10 - 4; // 6
```

3. Multiplication (*):

- Multiplies two numbers.



```
let product = 6 * 7; // 42
```

4. Division (/):

- Divide the left operand by the right operand.



```
let quotient = 15 / 3; // 5
```

5. Modulus (%):

- Returns the **remainder** of a **division** operation.

```
let remainder = 17 % 5; // 2
let remainder = 10 % 3; // 1 (because 10 divided by 3 leaves a remainder of 1)
```

6. Exponentiation (**):

- Raises the **left operand** to the power of the **right operand**.

```
let power = 3 ** 3; // 27
let power = 2 ** 3; // 8 (2 raised to the power of 3)
```

Assignment Operators:-

JavaScript Assignment Operators are used to **assign values to variables**. They combine an **arithmetic operation** with **assignment**. Here's a comprehensive overview:

- Basic Assignment (=):** Assigns the value on the **right** to the variable on the **left**.

```
let x = 5; // x is now 5
let x = 10;
```

- Addition Assignment (+=):** Adds the **right operand** to the **left operand** and **assigns the result** to the **left operand**.

```
let a = 10;
a += 5; // Equivalent to a = a + 5; a is now 15

let x = 5;
x += 3; // x is now 8 (equivalent to x = x + 3)
```

- Subtraction Assignment (-=):** Subtracts the **right operand** from the **left operand** and then assigns the result to the **left operand**.

```
let y = 10;
y -= 4; // y is now 6 (equivalent to y = y - 4)

let b = 20;
b -= 7; // Equivalent to b = b - 7; b is now 13
```

4. **Multiplication Assignment (*=):** Multiplies the **left operand** by the **right operand** and assigns the result.

```
let c = 6;
c *= 3; // Equivalent to c = c * 3; c is now 18

let z = 7;
z *= 2; // z is now 14 (equivalent to z = z * 2)
```

5. **Division Assignment (/=):** Divides the **left operand** by the **right operand** and then assigns the result to the **left operand**.

```
let a = 20;
a /= 5; // a is now 4 (equivalent to a = a / 5)

let d = 24;
d /= 4; // Equivalent to d = d / 4; d is now 6
```

6. **Modulus Assignment (%=):** Calculates the **remainder** of the **division** of the **left operand** by the **right operand** and assigns the result to the **left operand**.

```
let b = 10;
b %= 3; // b is now 1 (equivalent to b = b % 3)

let e = 17;
e %= 5; // Equivalent to e = e % 5; e is now 2
```

7. **Exponentiation Assignment (**=):** Raises the **left operand** to the **power** of the **right operand** and assigns the result.

```
let f = 2;
f **= 3; // Equivalent to f = f ** 3; f is now 8

let c = 4;
c **= 3; // c is now 43 (equivalent to c = c ** 3)
```

Comparison Operators: -

In JavaScript, **comparison operators** are used to **compare values**. The result of a comparison is always a **Boolean** value: **true** or **false**.

1. **Equal (==):** Compares two values for equality, but without checking their data types.

```
console.log(5 == '5'); // true (value is equal, type is not considered)
console.log(0 == false); // true
```

2. **Not Equal (!=)**: Compares two values for **inequality**, without considering their **data types**.

```
console.log(5 != "6"); // true
console.log(1 != true); // false
console.log(5 != '5'); // false (values are considered equal, types ignored)
```

3. **Greater Than (>)**: Checks if the **left value** is greater than the **right value**.

```
console.log(6 > 5); // true
console.log(10 > 5); // true
console.log(5 > 10); // false
```

4. **Less Than (<)**: Checks if the **left value** is less than the **right**.

```
console.log(5 < 10); // true
console.log(10 < 5); // false
```

5. **Greater Than or Equal To (>=)**: Checks if the **left value** is greater than or **equal** to the **right**.

```
console.log(10 >= 10); // true
console.log(11 >= 10); // true
console.log(9 >= 10); // false
```

6. **Less Than or Equal To (<=)**: Checks if the **left value** is less than or **equal** to the **right**.

```
console.log(10 <= 10); // true
console.log(9 <= 10); // true
console.log(11 <= 10); // false
```

Logical Operators:

In JavaScript, **logical operators** are used to combine **multiple conditions** or **expressions** and return a **Boolean value** (true or false).

1. **Logical AND (&&)**: Returns **true** only if **both operands are true**. If the first operand is **false**, it **short-circuits** and returns that value without evaluating the **second operand**.

```
console.log(true && true); // true
console.log(true && false); // false
console.log(false && true); // false
console.log(false && false); // false
console.log(5 > 3 && 10 < 20); // true (both comparisons are true)
```

2. **Logical OR (||)**: Returns **true** if at least **one operand is true**. If the **first operand is true**, it **short-circuits** and returns that value without evaluating the **second operand**. It only returns **false** if both operands are false.

```
console.log(true || true); // true
console.log(true || false); // true
console.log(false || true); // true
console.log(false || false); // false
console.log(true || false); // true
console.log(5 > 3 || 10 > 20); // true (first condition is true)
```

3. **Logical NOT (!)**: Returns the **inverse of the Boolean value of the operand**. If the value is **true**, ! Turns it into **false** and vice versa.

```
console.log(!true); // false
console.log(!false); // true
console.log(!""); // true (empty string is falsy)
console.log(!!""); // false (double negation)
console.log(!(5 > 3)); // false (since 5 > 3 is true, `!` turns it into false)
```

Template Literals:

Template literals are a feature in JavaScript that provide an improved way to work with **strings**. They are enclosed by **backticks (`)** instead of **single (')** or **double (")** quotes. They offer more flexibility and readability compared to traditional **string concatenation**. This lets us embed **variables** and **expressions** within our **strings**.

→ **Syntax**: Template literals are enclosed by **backticks (`)** instead of **single** or **double quotes**.

```
let greeting = `Hello, world!`;
```

→ **String Interpolation**: we can embed **expressions** or **variables** directly within a string using **\${}** syntax. The values inside **\${}** are evaluated and then inserted into the **string**.

```
const name = "Ajay";
const age = 25;
const greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting); // "Hello, my name is Ajay and I am 25 years old."
```

→ **Expression evaluation**: The expressions inside **\${}** are evaluated and their results are converted to strings.

```
const x = 10;
const y = 20;
const result = `The sum of ${x} and ${y} is ${x + y}.`;
console.log(result); // "The sum of 10 and 20 is 30."
```

Template Literals vs. Regular Strings:

String Concatenation:

Before **template literals** were introduced, string concatenation required using the plus (+) symbol.

```
const userName = 'Ajay'
const balance = 10

// Using regular string
const str1 = 'Hi ' + userName + ', ' + ' your balance is ' + balance + '.'
console.log("Regular string: ", str1)

// Using template literal
const str2 = `Hi ${userName}, your balance is ${balance}.`
console.log("Template literal: ", str2)
```

Regular string: Hi Ajay, your balance is 10.

Template literal: Hi Ajay, your balance is 10.

The regular **string** and **template literal** produce the same output for the example.

Note how using regular strings involves adding many **plus signs**. And also accounting for **whitespace** at the right places. This can get difficult to deal with when working with **lengthy strings**.

With the **template literal** example, there was no need to add any **plus signs**. we write everything together as a single string. The variables are directly embedded using the **\${}** syntax.

Multi-line Strings:

Another way **template literals** make it easier to work with strings is when dealing with **multi line strings**. For regular strings, we have to use a combination of the **plus + sign** and **\n** to denote a new line. But **template literals** don't require any of that.

Multi line string examples for regular string and template literal.

```
const regularString =  
'Hello there! \n' +  
'Welcome. \n' +  
'How can we help you today?'
```

```
const templateLiteral =  
`Hello there!  
Welcome.  
How can we help you today?`
```

```
console.log(regularString)  
console.log(templateLiteral)
```

```
Hello there!  
Welcome.  
How can we help you today?
```

```
Hello there!  
Welcome.  
How can we help you today?
```

Both the **regularString** and **templateLiteral** variables give the same output. The **template string** recognises **whitespaces** and **linebreaks** automatically.

Readability and Maintainability:

Template literals also make our code more **readable** and **easier** to maintain. As we've seen already, they don't require any **concatenation** with the **plus + sign**. And we also don't need to worry about **escaping quotations marks**.

Here's an example:

```
const city = "India"  
const str1 = 'They said, "we love ' + city + ', it\'s a beautiful place."'`  
const str2 = `They said, "we love ${city}, it's a beautiful place`  
  
console.log(regularString)  
console.log(templateLiteral)
```

Results of regular string and template literal examples:

```
They said, "we love India, it's a beautiful place."  
They said, "we love India, it's a beautiful place
```

Unlike the template literal, the regular string requires the following;

- The use of the **plus +** for concatenation.
- Careful use of single and double quotes.
- The need to escape the single quote in string with ****.

Decision Making Statements:

Decision making statements in JavaScript are used to execute different **blocks of code** based on **specified conditions**. Here are the main types of decision making statements in JavaScript:

if statement:

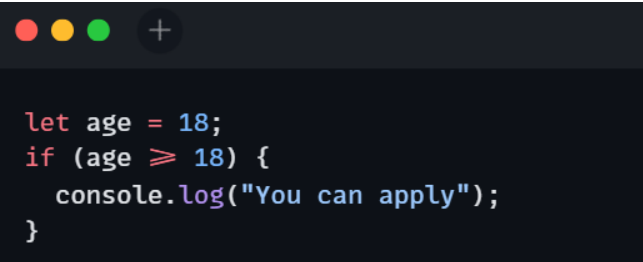
if statement is the most simple decision making statement. The if statement executes a **block of code** only if a specified condition is **true** otherwise not.

Syntax:



```
if ( condition )
{
    // block of code to be executed
}
```

Example:



```
let age = 18;
if (age ≥ 18) {
    console.log("You can apply");
}
```

We can omit the **braces** if there is only **single statement** in **block**.

Example:

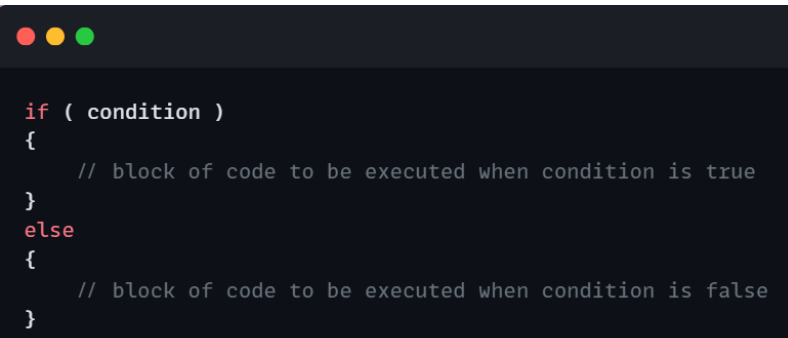


```
let word = 'hello';
if (word === 'hello') console.log('Hello World');
```

If-else statement:

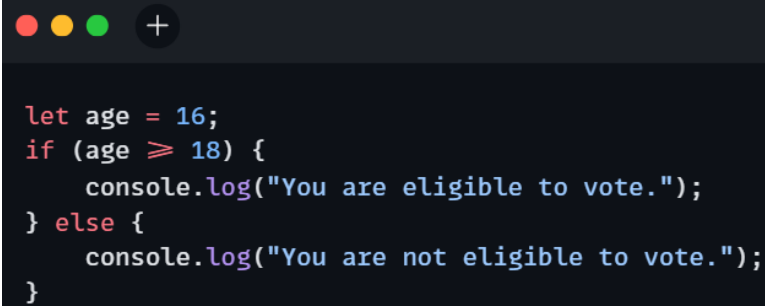
The **if-else** statement has two parts if **block** and else **block**. If the condition is **true** then **if block** (true block) will get executed and if the condition is **false** then **else block** (false block) will get executed.

Syntax:



```
if ( condition )
{
    // block of code to be executed when condition is true
}
else
{
    // block of code to be executed when condition is false
}
```

Example:



```


let age = 16;
if (age ≥ 18) {
    console.log("You are eligible to vote.");
} else {
    console.log("You are not eligible to vote.");
}

```

If-else-if statement:

The **if-else-if statement** is an advanced form of **if...else** that allows JavaScript to make a correct decision out of several conditions. All the **if** conditions will be checked one by one. If any condition is **true** out of given then that block will get executed and other blocks are skipped.

Syntax:

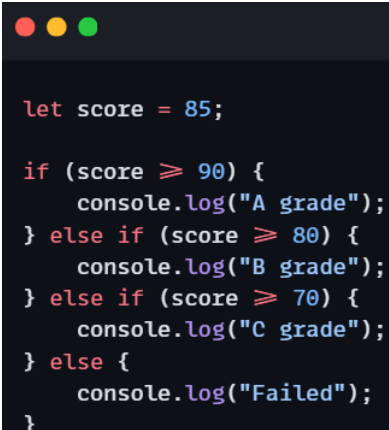


```

if ( condition 1 )
{
    // block of code to be executed when condition 1 is true
}
else if ( condition 2 )
{
    // block of code to be executed when condition 2 is true
}
else if ( condition 3 )
{
    // block of code to be executed when condition 2 is true
}
else
{
    // block of code to be executed if no condition matches
}

```

Example:



```

let score = 85;

if (score ≥ 90) {
    console.log("A grade");
} else if (score ≥ 80) {
    console.log("B grade");
} else if (score ≥ 70) {
    console.log("C grade");
} else {
    console.log("Failed");
}

```

Switch statement:

The **switch statement evaluates** an **expression** and matches its value against **multiple case labels**. If a match is found, the corresponding block of code is executed. It's useful when we want to compare a **value** against many possible options.

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 doesn't match any cases  
}
```

Example:

```
let fruit = 'apple';  
  
switch (fruit) {  
  case 'banana':  
    console.log('Bananas are ₹50 per kg.');    break;  
  
  case 'apple':  
    console.log('Apples are ₹20 per kg.');    break;  
  
  case 'cherry':  
    console.log('Cherries are ₹38 per kg.');    break;  
  
  case 'mango':  
  case 'papaya':  
    console.log('Mangoes and papayas are ₹55 per kg.');    break;  
  
  default:  
    console.log('Sorry, we are out of ' + fruit + '.');}
```

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there's a match, the associated block of code is executed.
- The **break** keyword breaks out of the switch block. This stops the execution from falling through to the next case.
- The **default** keyword specifies the code to run if there's no case match. It's optional, but recommended.
- We can have multiple cases for the same code block (like "**mango**" and "**papaya**" in the example).
- If we omit the **break statement**, execution will continue to the next case even if the case matches. This is called "**fall-through**" and can be useful in some scenarios, but is often unintended and can lead to bugs.

Nested if Statements:

Nested if statements are a way to create more complex decision-making structures by placing one or more if statements inside another if statement. This allows for more granular control flow and decision-making in our code.

Syntax:

```
if (condition1) {  
  if (condition2) {  
    // code to execute if both conditions are true  
  } else {  
    // code to execute if condition1 is true and condition2 is false  
  }  
} else {  
  // code to execute if condition1 is false  
}
```

Example:

```
let temperature = 22;  
let isRaining = false;  
let isWeekend = true;  
  
if (isWeekend) {  
  if (temperature > 25) {  
    if (!isRaining) {  
      console.log("Great day for a picnic!");  
    } else {  
      console.log("Too bad it's raining, otherwise it'd be perfect for a picnic.");  
    }  
  } else {  
    console.log("It's a bit cool for a picnic.");  
  }  
} else {  
  console.log("It's a workday, no time for a picnic.");  
}
```

Type Conversion:

In programming, **type conversion** is the process of **converting data of one type to another**. For example, converting **string data** to **number**.

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

There are two types of type conversion in JavaScript:

- **Implicit Conversion (Coercion)** - automatically done during code execution.
- **Explicit Conversion (Type Casting)** - Manual type conversion (done by us the developer).

Implicit Conversion:

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



```
const sum = 35 + "hello"
```


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

So what happens here if we 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.

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




```
const sum = 35 + "hello"

console.log(sum)
// 35hello

console.log(typeof sum)
// string
```

When using the **plus + sign**, it's better for the **number** to be changed into a **string** rather than trying to turn the **string** into a **number**. This is because converting a **string** to a **number** can result in **'NaN'**, but converting a number like **15** to a string gives **'15'**. So, it's more logical to combine two strings than to try adding a number and **'NaN'**.

Another example:



```
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 subtraction, so here, the ideal **coercion** is from **string** to **number** (as numbers have compatible operations with subtraction).

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
```

One very common operator that causes **coercion** is the **loose equality operator** (**==**, or **double equals**).

Double Equality and Coercion:

In JavaScript, there's both the **double equality operator**. We can use both operators to compare values' equality.

- “==” which is called the **loose equality operator**.
- “===” which is called the **strict equality operator**.

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.

Here is **20**, a value of a **number type**, and “20”, a value of the **string type**, are equal when we 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 we 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 performs a **strict comparison**, meaning it checks both the **values** and their **types**. It doesn't do any type **conversion**, so we won't get any unexpected results.



```
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.

Explicit Conversion:

In explicit type conversion, we manually convert **one type of data** into **another** using **built-in functions**. 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
```

Another example is 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.

Truthy & Falsy Values:

In JavaScript, the concepts of **truthy** and **falsy** values refer to how different values are evaluated in a **boolean** context, such as in conditions for **if statements** or **loops**.

Falsy Values:

Falsy values are values that evaluate to **false** when coerced into a **Boolean** context. Falsy values in JavaScript are unique because there are only **six** of them. Apart from these six, all other values are **truthy** values. Here are the six falsy values in JavaScript:

1. **False**: The Boolean value false.
2. **0**: The number zero.
3. **""** or **"** or **``**: An empty string.
4. **Null**: The null keyword, representing the absence of any object value.
5. **Undefined**: The undefined keyword, representing an **uninitialized** value.
6. **NaN**: Stands for **"Not a Number"**. It represents a special value returned from an operation that should return a numeric value but doesn't.

Example 1 – An empty string:

```
let input = ""; // User input can be an empty string

if (input) {
  console.log("Input is provided.");
} else {
  console.log("Input is empty."); // This will be logged because empty string is falsy value
}
```

Example 2 – The Boolean value false.

```
let checkValue = false;

if (checkValue) {
  console.log('Truthy');
} else {
  console.log('Falsy'); // It return Falsy result because checkValue variable value is false
}
```

Example 3 – The number zero.

```
const num = 0;

if (num) {
  console.log("Number is not Zero");
} else {
  console.log("Number is Zero");
}
```

Example 4 – null:


```

let user = null;

if (user && user.name) {
  console.log("Welcome, " + user.name + "!");
} else {
  console.log("Please log in to access the website."); //It will return this line because user has null value
}

```

Example 5 – undefined:

```

let age;
if (age === undefined) {
  console.log("The age is undefined.");
}

```

Example 6 – NaN:

```

let value1 = "Ten";
let value2 = 10;

let result = value1 / value2;

if (result) {
  console.log("The result is a number.");
} else {
  console.log("The result is not a number(NaN)."); //It will Executes This
  // line because when value1 is divide by value 2 it will return the NaN result.
}

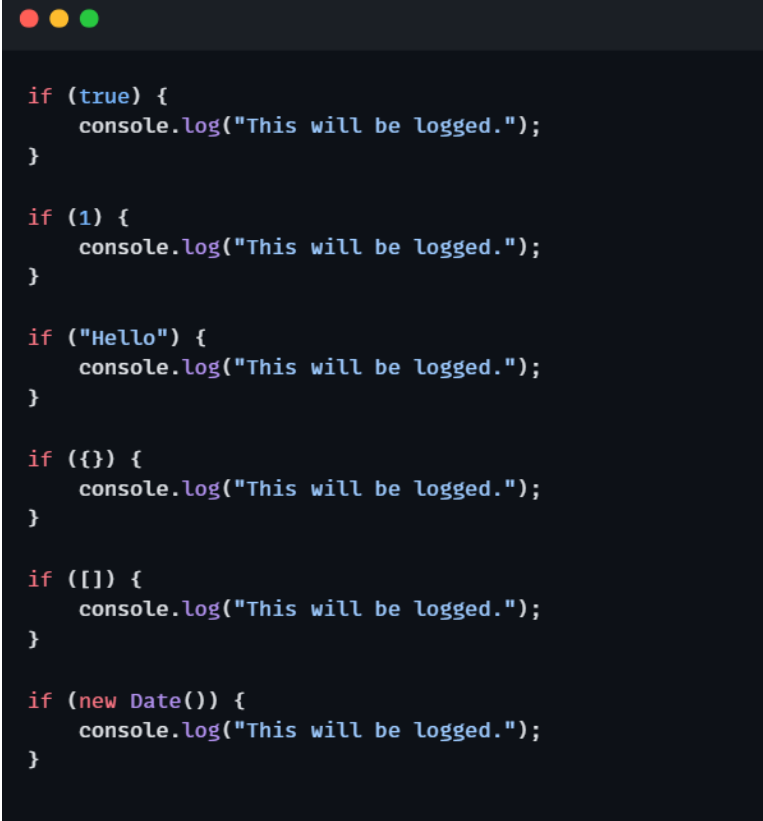
```

Truthy Values

Truthy values are values that evaluate to **true** in a boolean context. In JavaScript, any value that is not **falsy** is considered **truthy**. This includes:

- **True** - The boolean true itself.
- **Non-zero numbers** - Any number other than **0** (e.g., 1, -1, 3.14).
- **Non-empty strings** - Any string that is not empty (e.g., "hello").
- **Objects** - All objects are considered truthy, including **arrays** and **functions**.
- **Dates** - Instances of Date are also truthy.

Example:



```

if (true) {
    console.log("This will be logged.");
}

if (1) {
    console.log("This will be logged.");
}

if ("Hello") {
    console.log("This will be logged.");
}

if ({}) {
    console.log("This will be logged.");
}

if ([]) {
    console.log("This will be logged.");
}

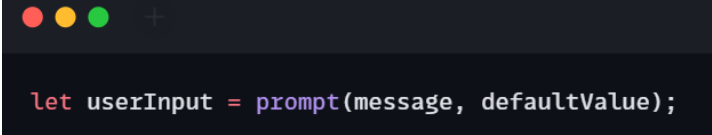
if (new Date()) {
    console.log("This will be logged.");
}

```

Window prompt () Method:

The **prompt ()** function in JavaScript is used to display a **dialog box** that prompts the **user** to input some text. This function **pauses** the **execution of the script** and waits for the user to submit a response by either entering text or pressing "OK" or "Cancel."

Syntax:



```

let userInput = prompt(message, defaultValue);

```

- **Message** is a string of text to display to the **user**. It can be omitted if there is nothing to show in the prompt window i.e. **it is optional**.
- **Default** is a string containing the default value displayed in the text input field. **It is also optional**.

Return Value

- If the user enters a value and presses **OK**, the **prompt ()** function returns the **string** entered by the user.
- If the user presses **Cancel** or **closes the dialog**, the function returns **null**.

Example:

```

//Simple prompt
let name = prompt("What is your name?");
console.log(name); // Logs the user input to the console

// Prompt with a default value
let age = prompt("How old are you?", "25");
console.log(age); // Logs the user input, or "25" if the user does not change it

// Checking for null (Cancel button)
let response = prompt("Enter something:");
if (response === null) {
    console.log("User cancelled the prompt.");
} else {
    console.log("User input:", response);
}

```

Ternary Operator:

The ternary operator is a concise way to execute one of **two expressions** based on a condition. It is often used as a shorter alternative to an **if-else statement**. The ternary operator is written with the following syntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

- condition: A condition to evaluate.
- expressionIfTrue: This expression is executed if the condition is **true**.
- expressionIfFalse: This expression is executed if the condition is **false**.

Here's a simple example:

```

let age = 20;
let canVote = age ≥ 18 ? "Yes" : "No";
console.log(canVote); // Outputs: "Yes"

```

How to Use the Ternary Operator in Place of if Statements:

The ternary operator can be a good replacement for **if statements** in some cases. It allows us to write **concise, cleaner, and easier-to-read** lines of code if used well.

Let's see an example:

```

const score = 80
let scoreRating

if (score > 70) {
  scoreRating = "Excellent"
} else {
  scoreRating = "Do better"
}

console.log(scoreRating)
// "Excellent"

```

In this example, we have a **score variable** with value of **80** and a **scoreRating** variable. Then we have an **if statement** that checks **score > 70**. If this condition evaluates to **true**, the **scoreRating** variable is assigned **"Excellent"**, else, it is assigned **"Do better"**.

We can improve this code with the ternary operator. Here's how.

```

const score = 80

const scoreRating =
  score > 70 ? "Excellent" : "Do better"

console.log(scoreRating)
// Excellent

```

This is how we use the **ternary operator**. The **true** and **false** expressions here are strings that will be returned to the **scoreRating variable** depending on our condition **score > 70**.

The **true** and **false** expressions can be any kind of expression from **function executions** to **arithmetic operations** and so on. Here's an example with a function execution:

```

function printPoor() {
  console.log("Poor result")
  return "poor"
}

function printSuccess() {
  console.log("Nice result")
  return "success"
}

const pass = false;

const result = pass ? printSuccess() : printPoor()
// Poor result (console.log executed)

console.log(result)
// poor

```

Here, we see that as the condition returns **false**, the **false expression**, **printPoor()** is executed which prints **"Poor result"** to the console. And as the false expression returns **"poor"**, we can see that value assigned to the result variable.

How to Use Nested Ternary Operators:

What if we wanted to achieve an **if...else if...else statement** with a **ternary operator**? Then we can use **nested ternary operators**. We should be careful how we use this, however, as it can make our code harder to read.

Let's see an example:

```
const score = 60
let scoreRating

if (score > 70) {
  scoreRating = "Excellent"
} else if (score > 50) {
  scoreRating = "Average"
} else {
  scoreRating = "Do better"
}

console.log(scoreRating)
// "Average"
```

We have an **if-else-if-else statement** here where we first check if **score > 70**. If this returns **true**, we assign **"Excellent"** to the **scoreRating** variable. If this returns **false**, we check if **score > 50**. If this second condition returns **true**, we assign **"Average"** to the variable but if this also returns **false**, we finally (else) assign **"Do better"** to the variable.

Let's see how to do this with the ternary operator:

```
const score = 60

const scoreRating =
  score > 70
    ? "Excellent"
    : score > 50
      ? "Average"
      : "Do better"

console.log(scoreRating)
// "Average"
```

Here, we have two **question marks** and **colons**. In the first ternary operator, we have the conditional expression **score > 70**. After the first question mark, we have the **true expression** which is **"Excellent"**. After the first colon, the next expression is supposed to be the false expression. For the false expression, we declare another **conditional expression** using the ternary operator.

The second condition here is **score > 50**. After the second question mark, we have the true expression which is **"Average"**. After the second colon, we now have another false expression, which is **"Do better"**.

With this, if the first condition is true, **"Excellent"** is returned to **scoreRating**. If the first condition is false, then we have another condition check. If this second condition is true, **"Average"** is returned to the **variable**. If this second condition is also false, then we have the final false expression, **"Do better"**, which will be assigned to the variable.

Expression vs. statement:

JavaScript Expression:


- An expression is a piece of code that evaluates to a **single value**.
- Expressions can be used in any place where a value is expected.
- Examples of expressions include **arithmetic operations**, **function calls**, and **variable assignments**.



```
// JS Expressions
5 + 7
myFunction()
x = 100
```

JavaScript Statement:

- A statement is a piece of code that **performs an action**.
- Statements are executed by the **JavaScript engine** and **do not produce a value**.
- Examples of statements include **loops**, **conditionals**, and **declarations**.



```
// JS Statements
if (x > 10) {
  console.log('x is greater than 10');
}

for (let i = 0; i < 5; i++) {
  console.log(i);
}

function myFunction() {
  return 'Hello, Readers!';
}
```