Javascript

What is Javascript?

- JavaScript is a programming language used to make websites interactive and dynamic. It works alongside **HTML** (which structures the page) and **CSS** (which styles the page) to create rich user experiences.
- It supports multiple programming styles. JavaScript can be written in an **object-oriented** way using objects and classes, in a **functional** style using functions as first-class citizens, and in an **event-driven** manner by responding to user actions like clicks or key presses.
- JavaScript is one of the core technologies of the web and is essential for modern web development.

History of JavaScript: -

- JavaScript is one of the three core languages of web development, along with HTML and CSS. It was created in 1995 by Brendan Eich, a programmer at Netscape, to make websites more interactive. Amazingly, he built the first version of JavaScript in just 10 days.
- At first, it was called **Mocha**, then **LiveScript**, and finally renamed to **JavaScript**—partly to ride the popularity wave of Java, even though the two are very different languages.
- JavaScript's design takes inspiration from several other languages: its syntax is similar to Java, it uses first-class functions like Scheme, and it supports prototype-based inheritance like the language Self.
- Because all major browsers eventually supported JavaScript, it became a key technology for building dynamic, interactive websites.

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.

- JavaScript and ECMAScript are closely related but not the same. **ECMAScript** is the basic language, while JavaScript is ECMAScript plus extra features. These extra features come from the environments where JavaScript runs, like **web browsers** and **Node.js**, which add their own tools and APIs.

Evolution of ECMAScript Versions: -

The ECMAScript language (the foundation of JavaScript) has gone through many updates since its creation. Here's a simplified timeline of its key versions:

1. **ECMAScript 1 (1997)**
   The first official version. It standardized the basic syntax and structure of JavaScript.
2. **ECMAScript 2 (1998)**
   A minor update focused on aligning with ISO/IEC standards. No major new features.
3. **ECMAScript 3 (1999)**
   A big step forward! Added regular expressions, better string handling, and improved error handling.
4. **ECMAScript 4 (Abandoned)**
   Planned as a huge update but was never released due to disagreements. Development shifted toward more manageable updates.
5. **ECMAScript 5 (2009)**
   Introduced strict mode, getters/setters, JSON support, and better array methods. A major modernizing release.
6. **ECMAScript 6 / ES6 (2015)**
   A massive upgrade. Added arrow functions, classes, promises, modules, template literals, destructuring, and more. Often considered a reboot of JavaScript for modern development.
7. **Annual Updates (ES2016 and beyond)**
   From 2016 onward, ECMAScript adopted a yearly release cycle, adding small but useful features every year:
   - ES2017: async/await, shared memory
   - ES2018+: spread operators in objects, async iteration
   - ES2020: optional chaining (?.), nullish coalescing (??), and native modules

What are variables?

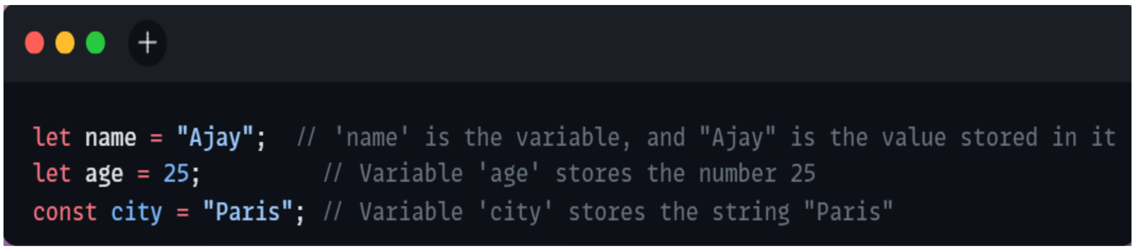
A **variable** is like a **labeled container** that holds information (called a **value**) which we can use later in our code. We can think of it as a **box with a name** written on it. Inside the box, we can store things like **numbers**, **text**, or other types of data. We use the variable's **name** to **access**, **change**, or **use** the value stored 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.

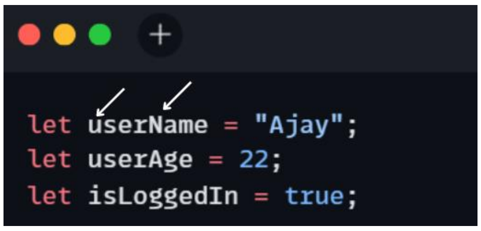
Naming Convention and general rules for Variables:

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:

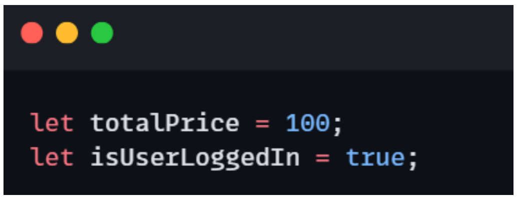
1. **CamelCase:** Start the variable name with a **lowercase letter** and **capitalize** the **first letter** of each subsequent word.
   Example: **firstName**, **totalAmount**, **isLoggedIn**.



```
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.

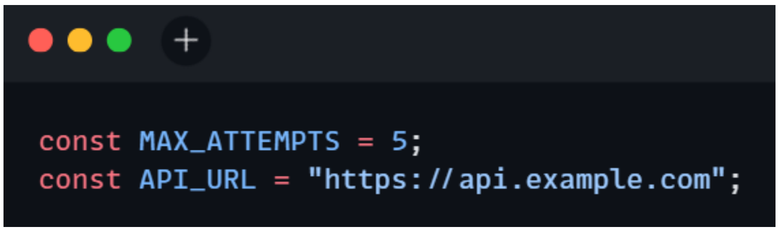
   Example: Instead of **x**, use **totalPrice**.



```
let totalPrice = 100;
let isUserLoggedIn = true;
```

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

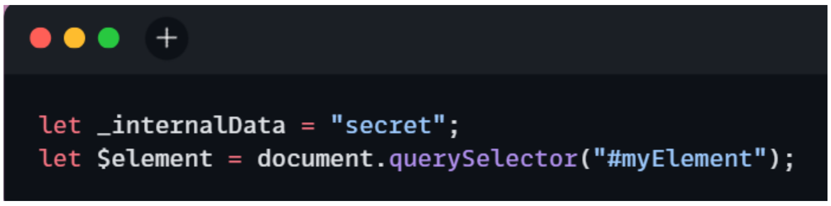
   Example: const MAX_USERS = 100, const API_KEY = "1234abcd";



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

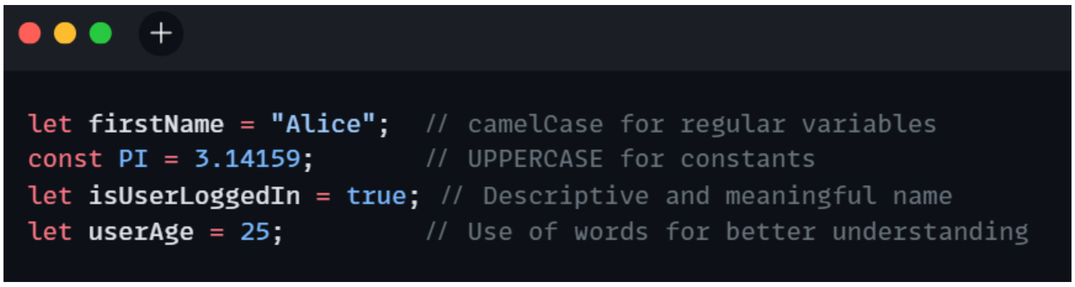
4. **Avoid Starting with Underscore (_) or Dollar Sign ($):** Although allowed, these are typically used for specific purposes, like **private** properties in some coding styles.

   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 isUserLoggedIn = 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 describe the different kinds of **values** that can be stored in variables They help JavaScript understand **how to use** the data. whether it's a number, text, or something more complex.

JavaScript data types are divided into two types.

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

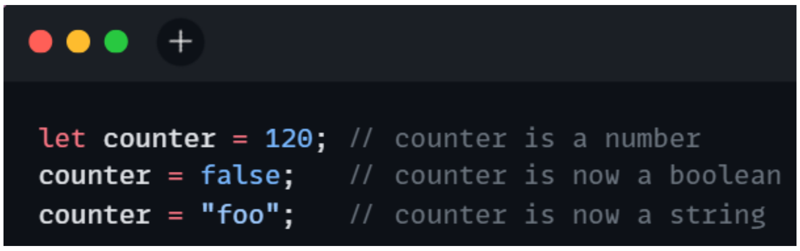
Primitive Data Types: -

Primitive data types in JavaScript represent the most **basic and simple values**. Primitive data types are **immutable (**Immutable means something that cannot be changed after it has been created. Strings are immutable. If we try to modify a string, JavaScript creates a **new string** instead of changing the original one. **These types are stored directly in memory, and each variable holds its own copy of the value.**

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:

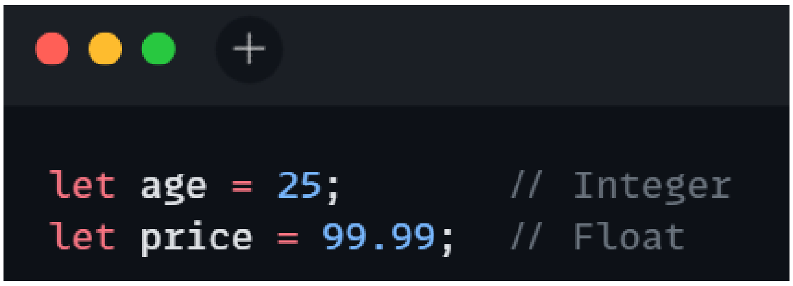


```
let counter = 120; // counter is a number
counter = false;   // counter is now a boolean
counter = "foo";   // counter is now a string
```

Number: -

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

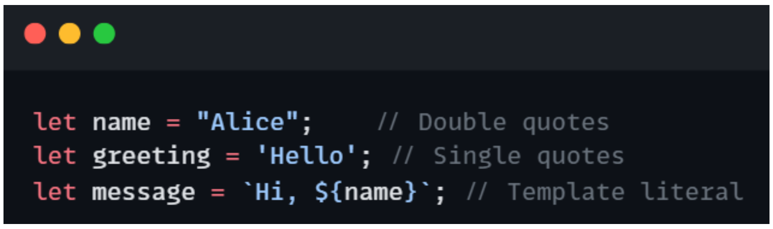
- **Integers** - Numeric values without any decimal parts. Ex: 3, -74, etc.
- **Floating-Point** - Numeric values with decimal parts. Ex: 3.15, -1.3, etc.



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

String: -

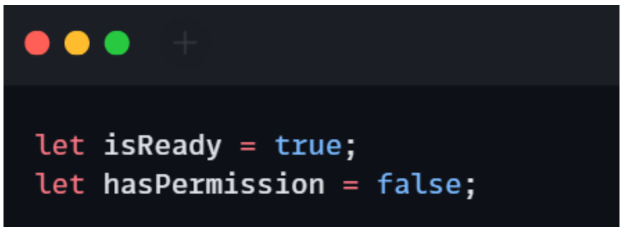
In JavaScript, **a string is a sequence of zero or more characters enclosed in single (' '), double (" "), or backticks (` `).**



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

Boolean: -

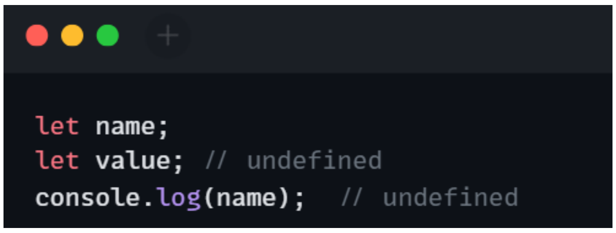
A Boolean data can only have one of two values: **true** or **false**. Useful in conditional logic.



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

Undefined: -

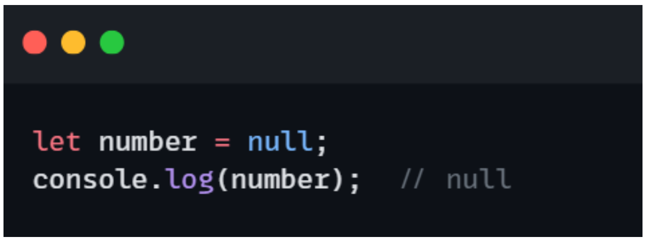
In JavaScript, **undefined** represents the absence of a **value**. If a variable is declared but the value is not assigned, then the value of that variable will be **undefined**. For example,



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

Null: -

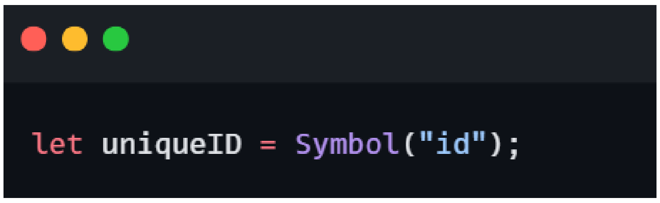
In JavaScript, **null** represents an **explicitly empty** or **non-existent value**.



```
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.



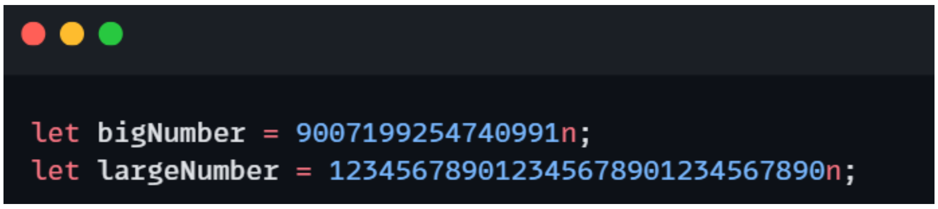
```
let uniqueID = Symbol("id");
```

BigInt: -

BigInt is a special number type in JavaScript designed to represent very large integers that are too big for the regular Number type to handle safely**.** 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 = 1234567890123456789012345678901234567890n;
```

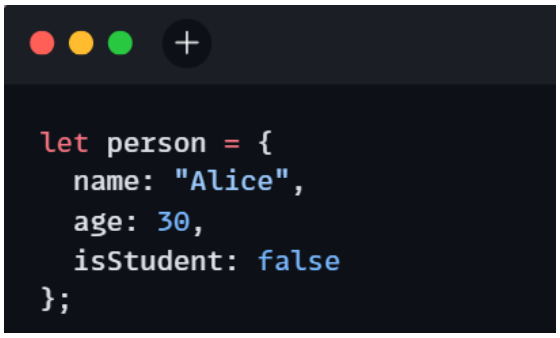
Non-primitive Data Types: -

Non-primitive data types in JavaScript are complex and can store **multiple values** or more **complicated data**. They are also called **reference types** because:

- They are **mutable** — their content can be changed after creation.
- When passed to functions or assigned to variables, you pass a **reference** to the data, not a copy of the actual value. The two key non-primitive data types in JavaScript are:

Object: -

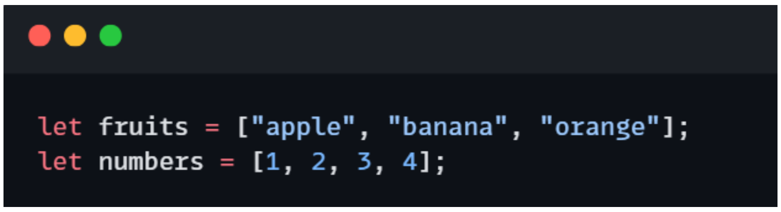
An **object** is a collection of **key-value pairs**, where keys (usually **strings** or **Symbols**) are used to identify values. These values can be of any data type, including other **objects** or **functions**. Objects are fundamental in JavaScript, as almost everything in the language is an object.



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

Array: -

An array is a special type of object in JavaScript used to store **ordered collections of data**. Each item in the array can be of any data type like numbers, strings, objects, or even other arrays., and the items are accessed using their **index**, starting from **"0"**.

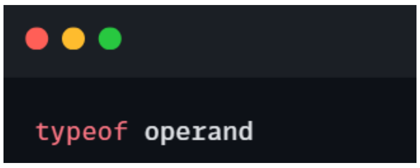


```
let fruits = ["apple", "banana", "orange"];
let numbers = [1, 2, 3, 4];
```

typeof Operator: -

In JavaScript, the typeof operator is used to **check the type** of a value or expression. It returns a **string** that tells you what kind of data the operand is such as "**string**", "**number**", "**boolean**", "**object**", or "**undefined**".

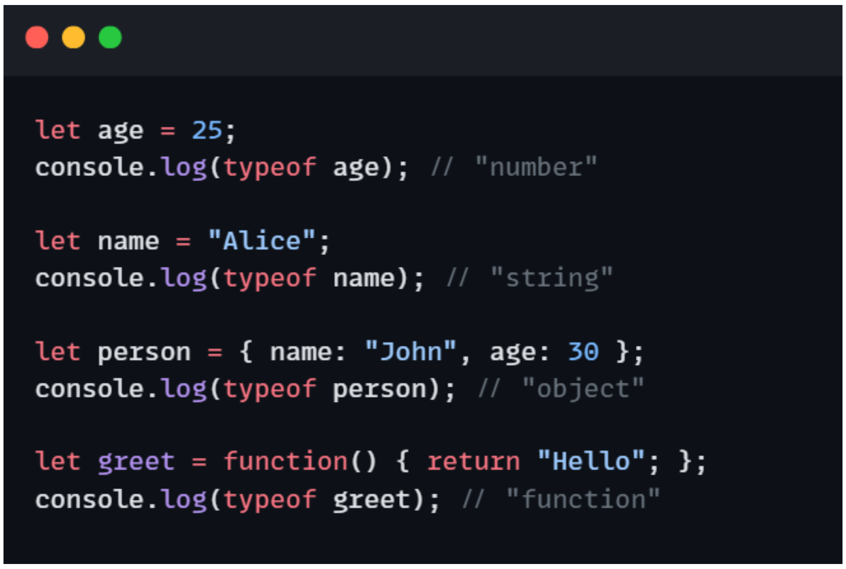
Syntax of typeof Operator:



```
typeof operand
```

Here, **operand** is a variable name or a value.

Example:



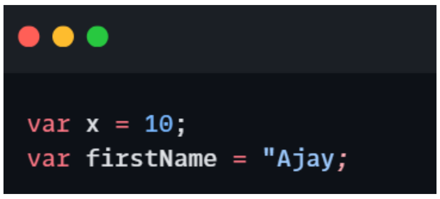
```
let age = 25;
console.log(typeof age); // "number"

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: -

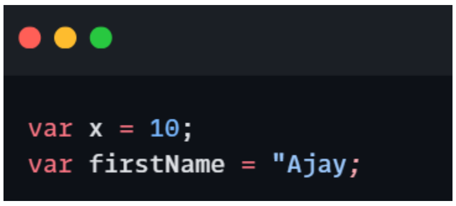
The **var** is the oldest keyword to declare a variable in JavaScript. It's less commonly used in modern JavaScript due to some limitations, like hoisting issues.



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

Const: -

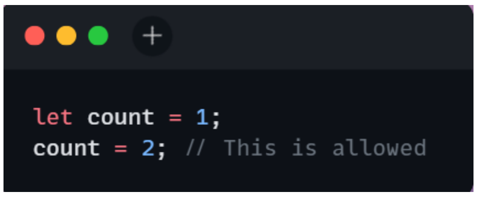
Used to declare variables that will not be **reassigned**. A const variable must be initialized when it is declared and cannot be updated afterward.



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

Let: -

Used to declare variables that can be reassigned later.



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

JavaScript Operators: -

What are Operators?

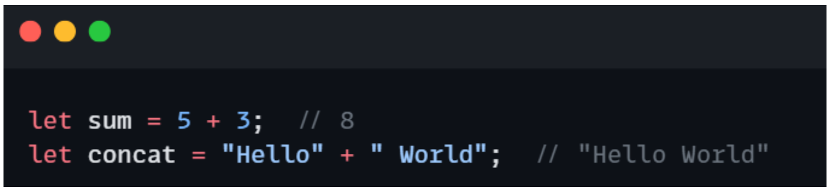
Operators in JavaScript are **symbols** or **keywords** used to perform operations on values (operands). They allow us to manipulate data, perform calculations, compare values, and make decisions based on conditions.

Arithmetic Operators: -

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

1. **Addition (+):**

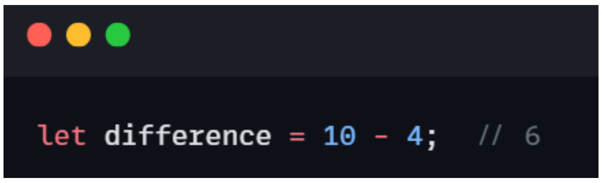
   Adds two numbers. Can also be used for **string concatenation**.



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

2. **Subtraction (-):**

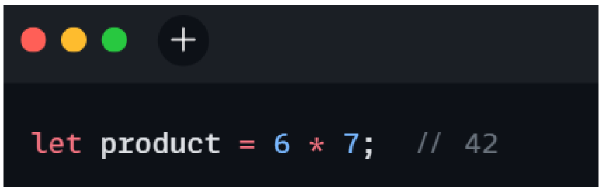
   Subtract the **right operand** from the **left operand**.



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

3. **Multiplication (*):**

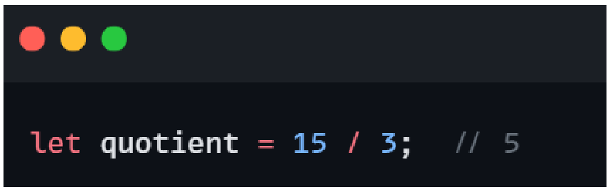
   Multiplies two numbers.



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

4. **Division (/):**

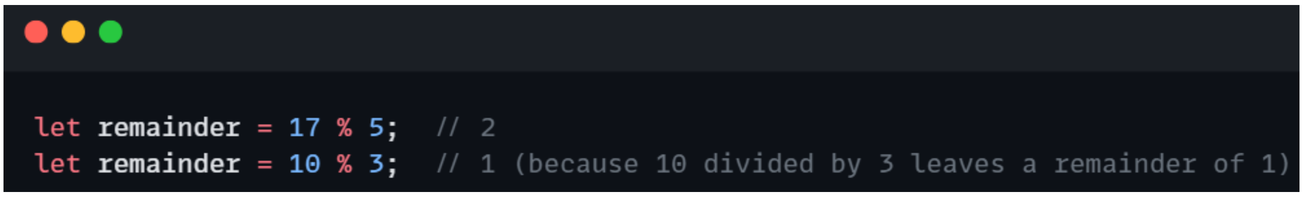
   Divide the left operand by the right operand.



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

5. **Modulus (%):**

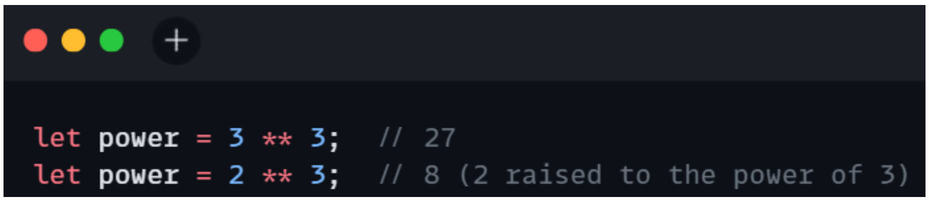
Returns the **remainder** of a **division** operation.



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

6. **Exponentiation (**):**

The exponentiation operator raises the value of the **left operand** to the power of the right **operand**. For example, **a ** b** means "**a** raised to the power of **b**."

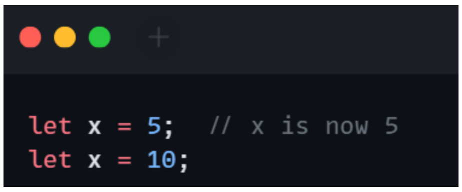


```
let power = 3 ** 3;   // 27
let power = 2 ** 3;   // 8 (2 raised to the power of 3)
```

Assignment Operators: -

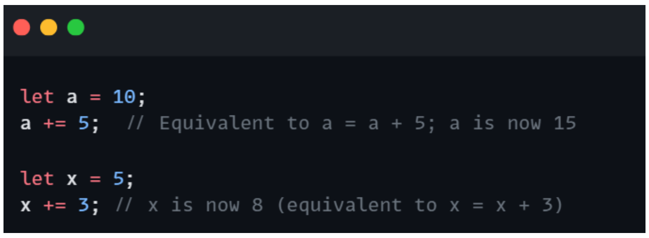
In JavaScript, assignment operators are used to **assign values to variables**. Many assignment operators combine an **arithmetic operation** with the **assignment**, making them concise and efficient. Here's a comprehensive overview:

1. **Basic Assignment (=):** The basic assignment operator **assigns the value** on the right-hand side to the variable on the left-hand side.



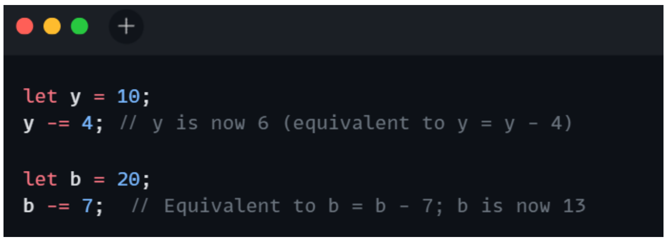
```
let x = 5;   // x is now 5
let x = 10;
```

2. **Addition Assignment:** Adds the value of the **right operand** to the **left operand** and assigns the result to the **left operand**.



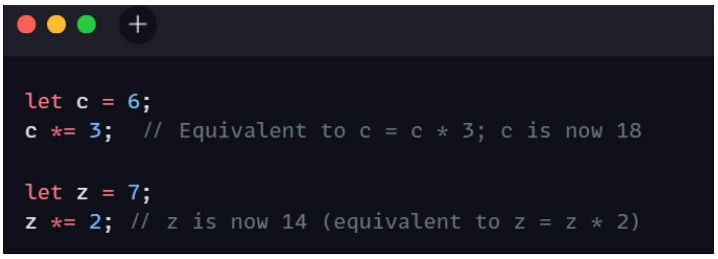
```
let a = 10;
a += 5;   // Equivalent to a = a + 5; a is now 15

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

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



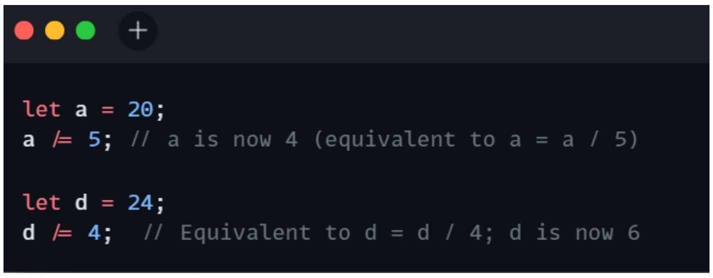
```
let y = 10;
y -= 4;  // y is now 6 (equivalent to y = y - 4)

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**.



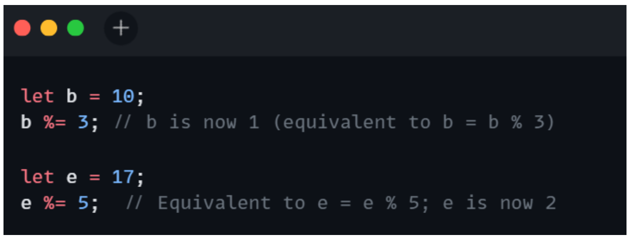
```
let c = 6;
c *= 3;  // Equivalent to c = c * 3; c is now 18

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**.



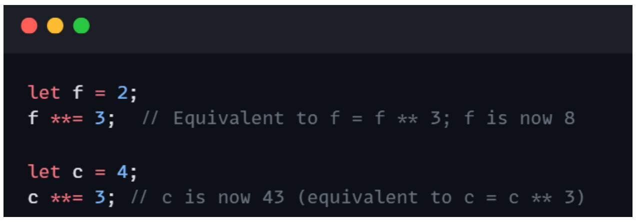
```
let a = 20;
a /= 5; // a is now 4 (equivalent to a = a / 5)

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.

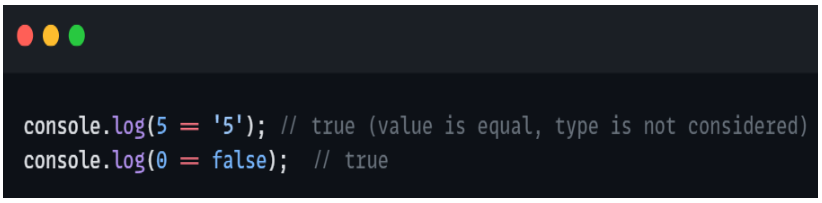


```
let f = 2;
f **= 3;  // Equivalent to f = f ** 3; f is now 8

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

Comparison Operators: -

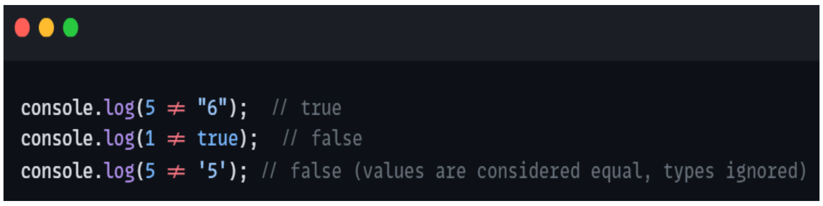
In JavaScript, comparison operators are used to **compare values**. The result of a comparison is always a **Boolean** value: **true** or **false**.

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



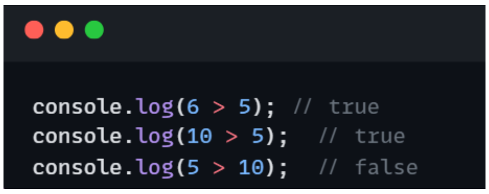
```
console.log(5 == '5'); // true (value is equal, type is not considered)
console.log(0 == false);  // true
```

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



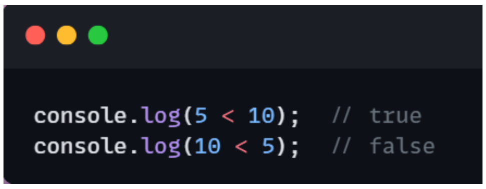
```
console.log(5 ≠ "6");  // true
console.log(1 ≠ true);  // false
console.log(5 ≠ '5'); // false (values are considered equal, types ignored)
```

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



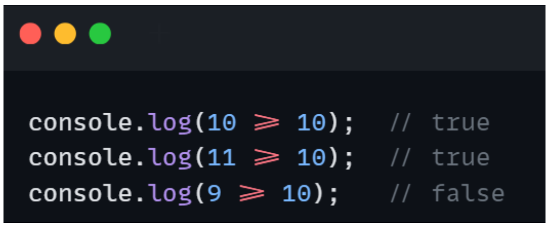
```
console.log(6 > 5); // true
console.log(10 > 5);  // true
console.log(5 > 10);  // false
```

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



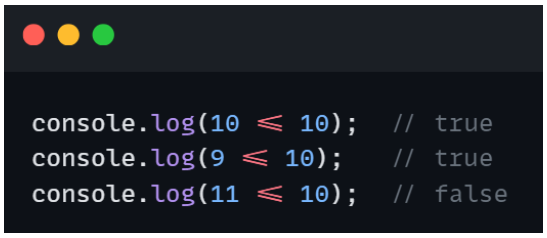
```
console.log(5 < 10);  // true
console.log(10 < 5);  // false
```

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.

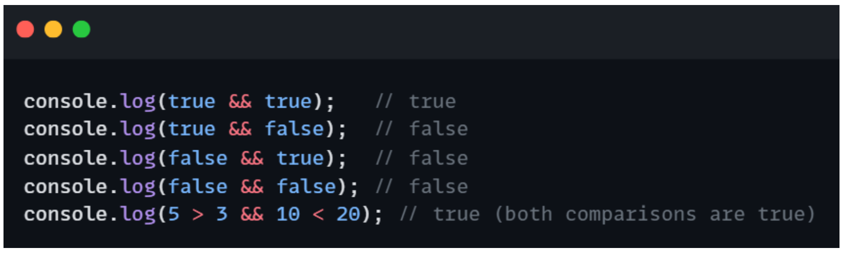


```
console.log(10 ⩽ 10);  // true
console.log(9 ⩽ 10);   // true
console.log(11 ⩽ 10);  // false
```

Logical Operators:

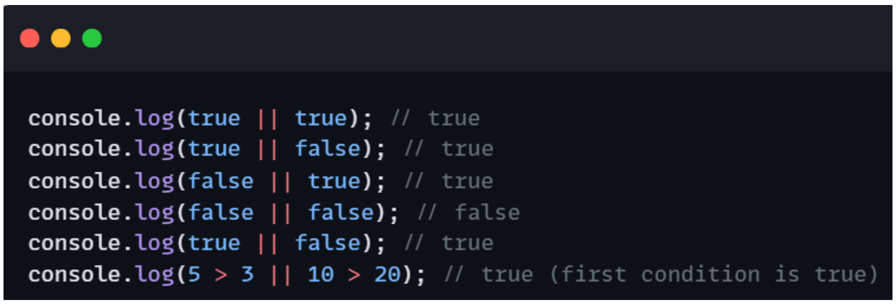
In JavaScript, **logical operators** are used to combine **multiple conditions** or **expressions** and return a **Boolean value** (**true** or **false**).

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**.



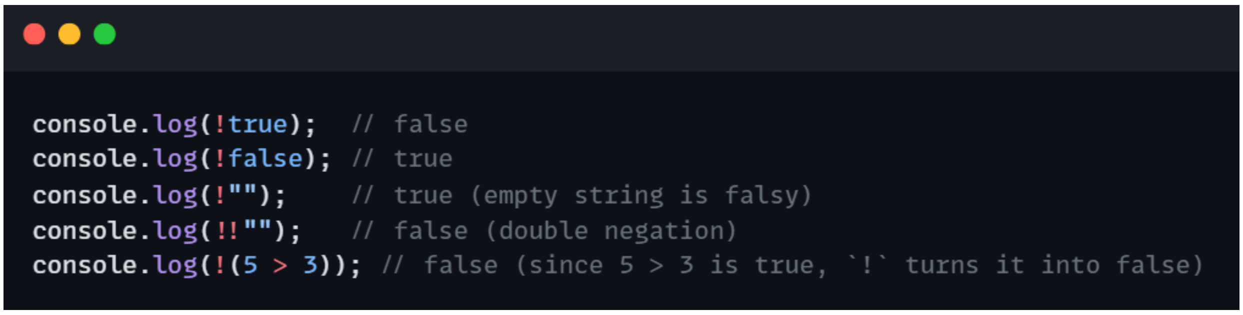
```
console.log(true && true);   // true
console.log(true && false);  // false
console.log(false && true);  // false
console.log(false && false); // false
console.log(5 > 3 && 10 < 20); // true (both comparisons are true)
```

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.

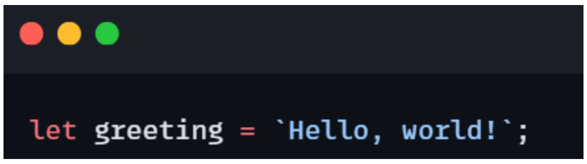


```
console.log(!true);  // false
console.log(!false); // true
console.log(!"");    // true (empty string is falsy)
console.log(!!"");   // false (double negation)
console.log(!(5 > 3)); // false (since 5 > 3 is true, `!` turns it into false)
```

Template Literals:

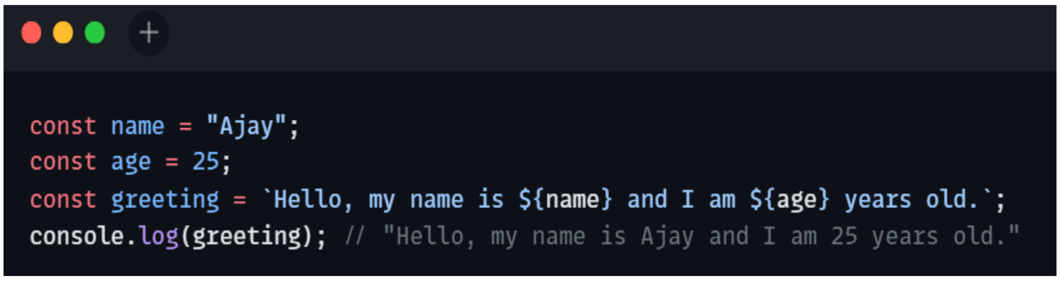
Template literals are a modern feature in JavaScript that make working with strings easier and more readable. Unlike traditional strings that use single (') or double (") quotes, template literals are enclosed in backticks (`). They allow us to insert variables and expressions directly into a string using the **${...}** syntax, which makes the code cleaner and more expressive.

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



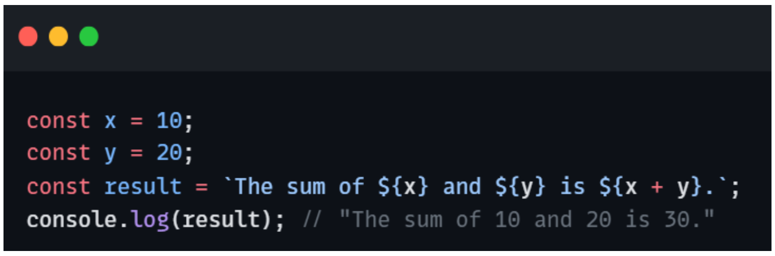
```
let greeting = `Hello, world!`;
```

→ **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.

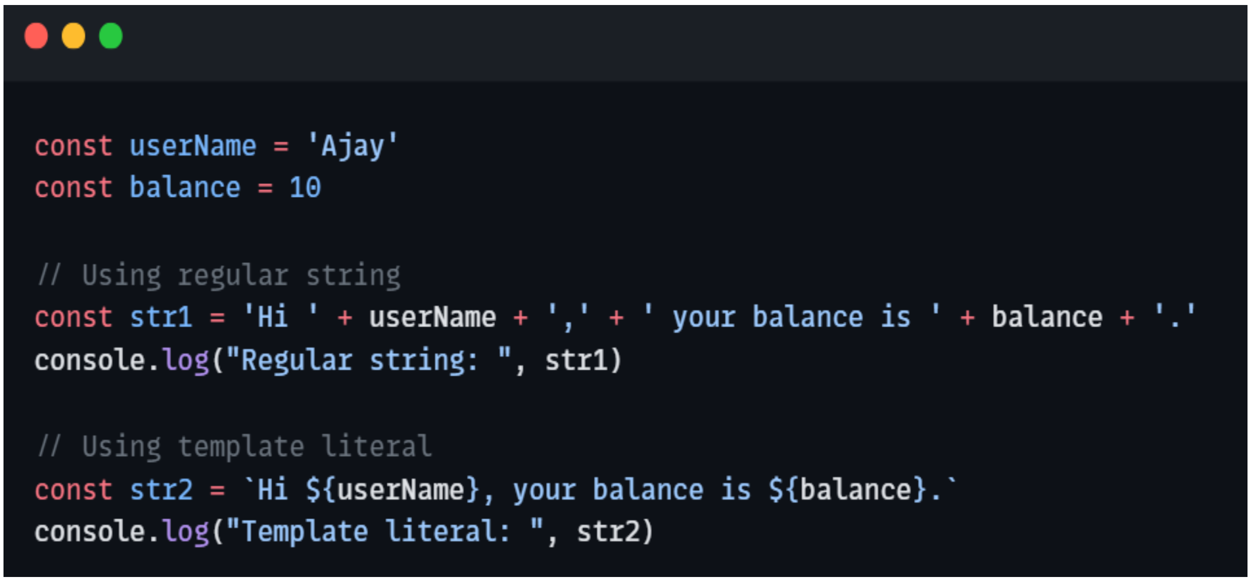


```
const x = 10;
const y = 20;
const result = `The sum of ${x} and ${y} is ${x + y}.`;
console.log(result); // "The sum of 10 and 20 is 30."
```

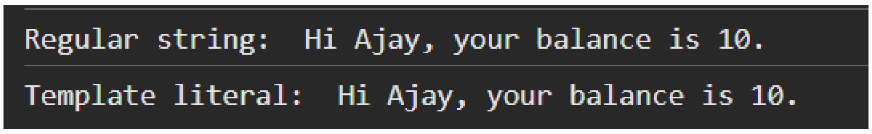
Template Literals vs. Regular Strings: -

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 example.

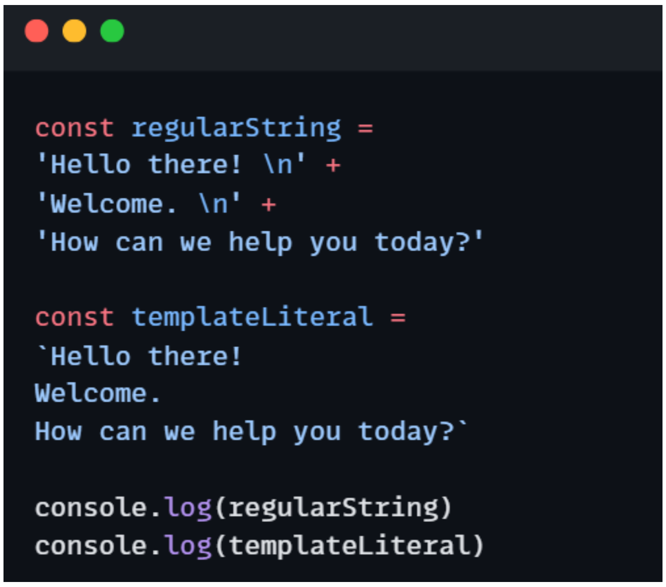
Note how using regular strings involves adding many **plus signs**. And also accounting for whitespace at the right places. This can get difficult to deal with when working with **lengthy strings**.

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.

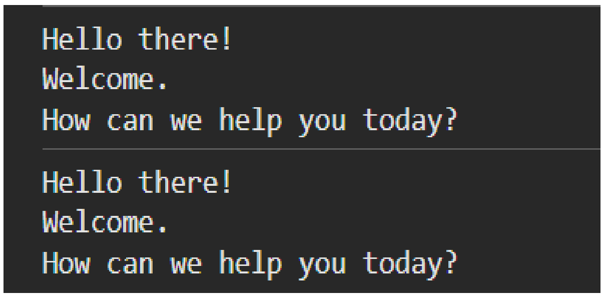
Multi-line Strings:

Another way **template literals** makes 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)
```



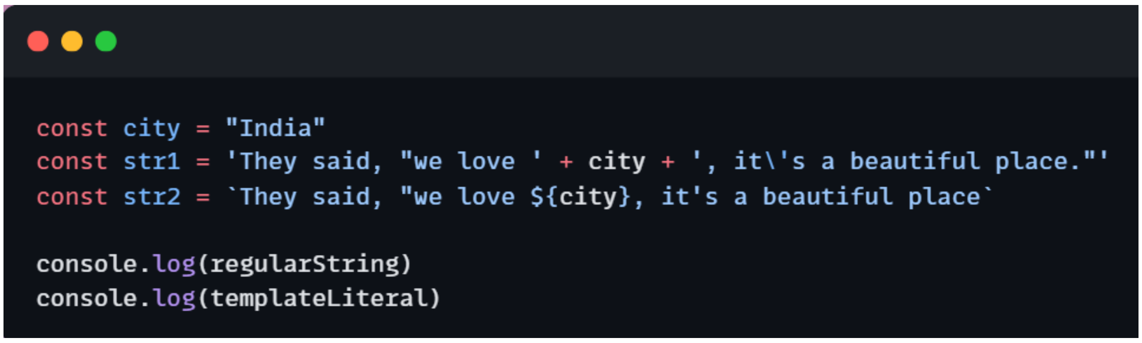
```
Hello there!
Welcome.
How can we help you today?
Hello there!
Welcome.
How can we help you today?
```

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

Readability and Maintainability:

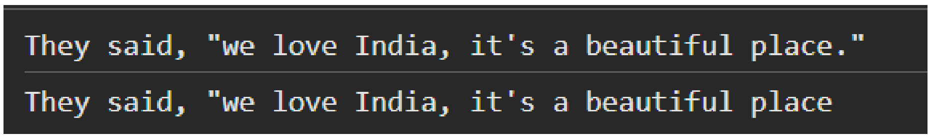
Template literals also make our code more **readable** and **easier** to maintain. As we've seen already, they don't require any **concatenation** with the **plus + sign**. And we also don't need to worry about **escaping quotations marks**.

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 \.

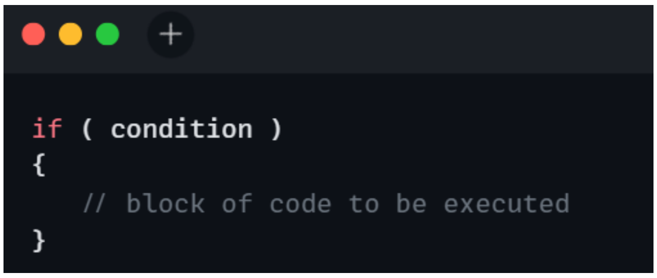
Decision Making Statements:

Decision-making statements in JavaScript are used to control the flow of a program by executing different blocks of code depending on certain conditions. These statements help programs make choices and respond to different inputs or situations. Here are the main types of decision-making statements in JavaScript:

if statement:

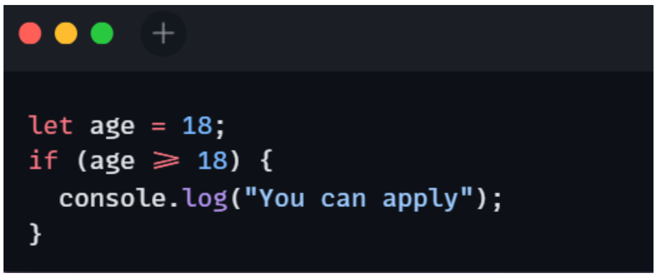
The if statement is the simplest form of decision-making in JavaScript. It allows us to run a block of code **only when a specific condition is true**. If the condition is false, the code inside the if block is skipped.

Syntax:



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

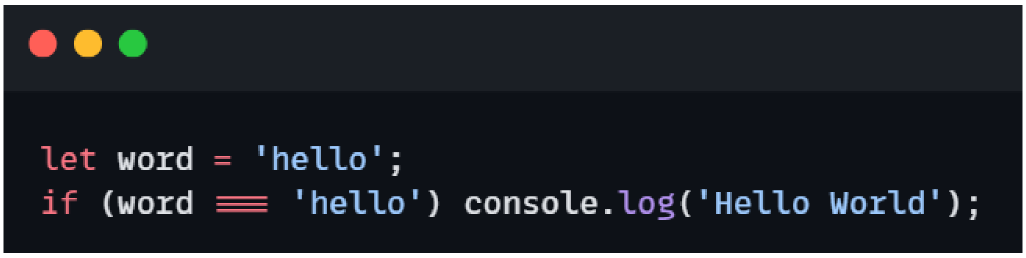
Example:



```
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:

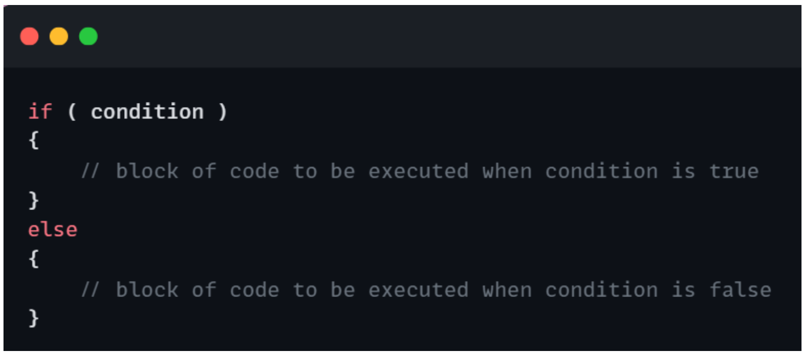


```
let word = 'hello';
if (word === 'hello') console.log('Hello World');
```

If-else statement:

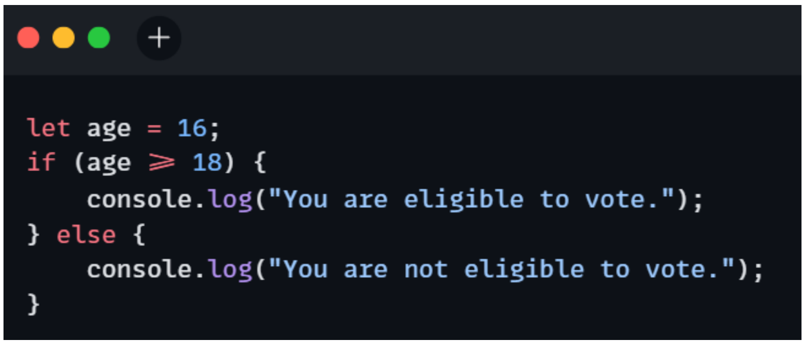
The **if-else** statement has two parts if block and else block. If the condition is true**,** then **if block** (true block) will get executed and if the condition is false then else block (false block) will get executed.

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:

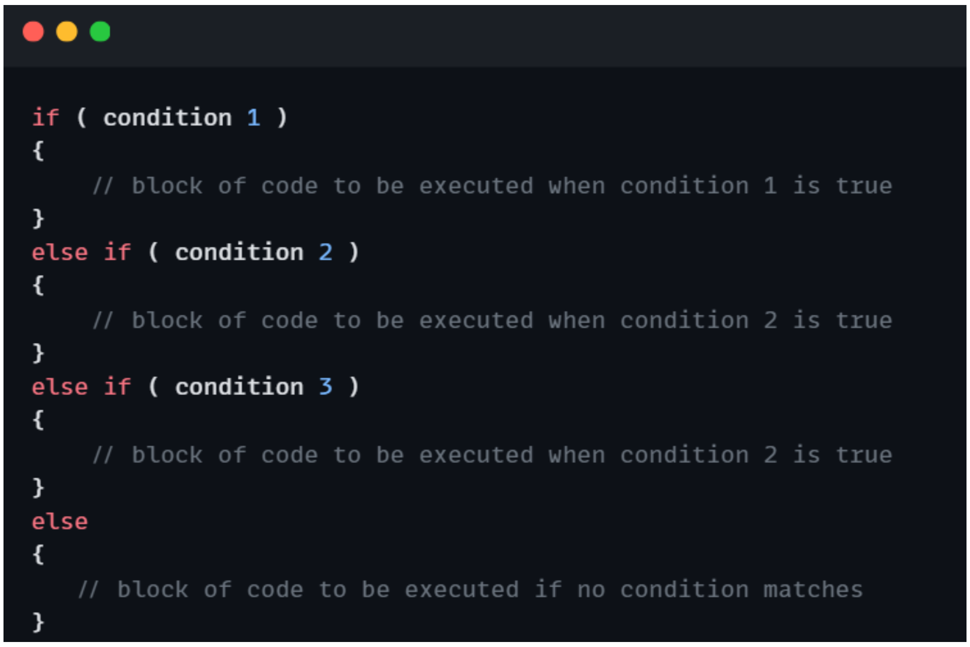


```javascript
let age = 16;
if (age >= 18) {
    console.log("You are eligible to vote.");
} else {
    console.log("You are not eligible to vote.");
}
```

If-else-if statement:

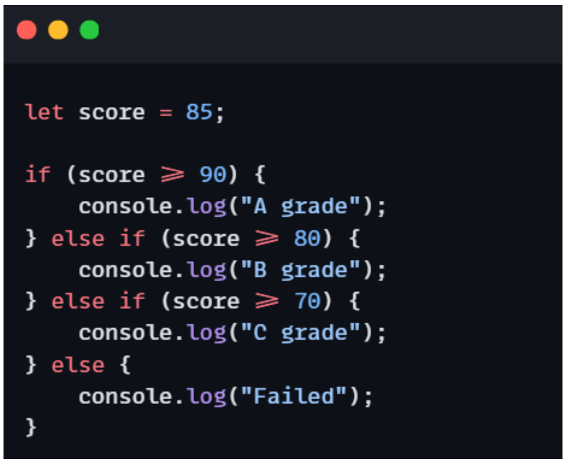
The if-else-if statement is an advanced form of if…else. that allows JavaScript to make a correct decision out of several conditions. All the if conditions will be checked one by one. If any condition is true out of given, then that block will get executed and other blocks are skipped.

Syntax:



```javascript
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:

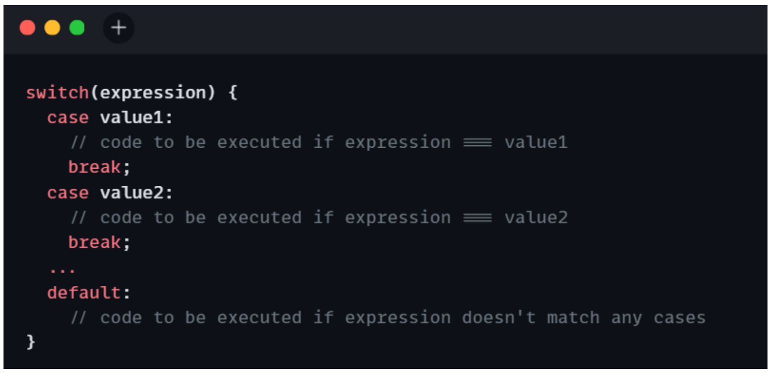


```javascript
let score = 85;

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:

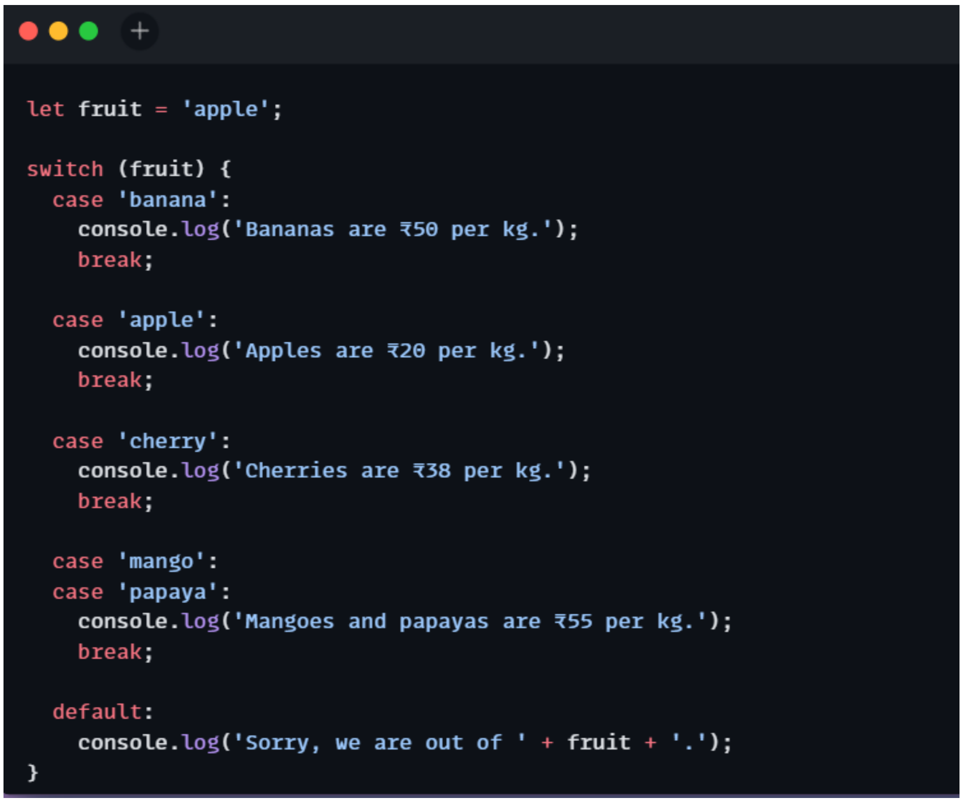
The switch statement evaluates an expression and matches its value against multiple case labels. If a match is found, the corresponding block of code is executed. It’s useful when we want to compare a value against many possible options.

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:



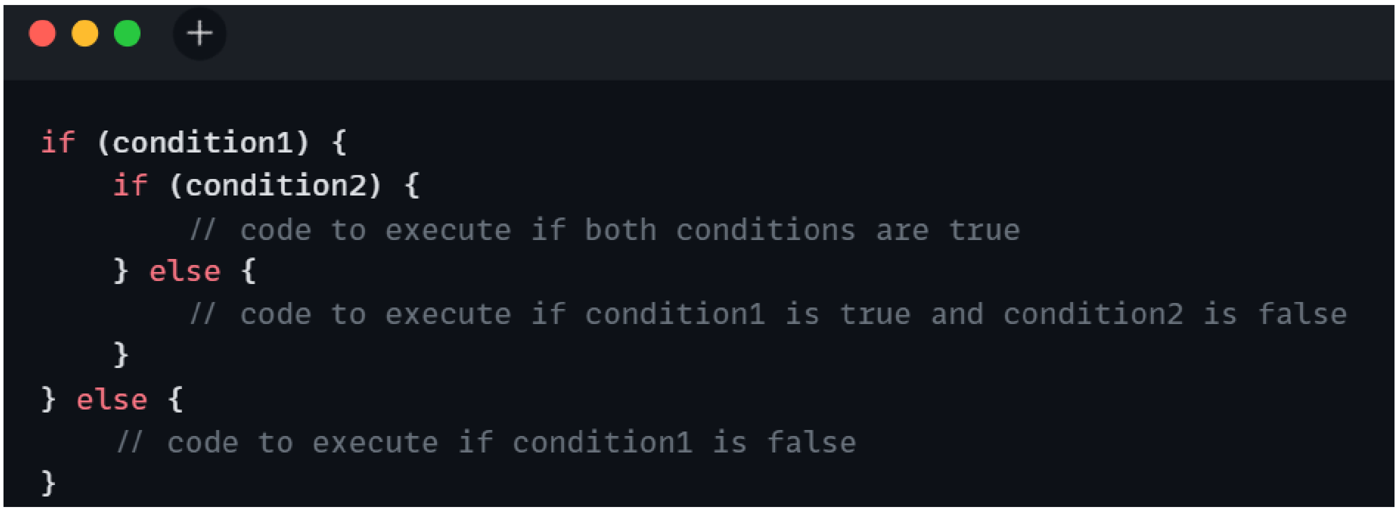
```
let fruit = 'apple';

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:

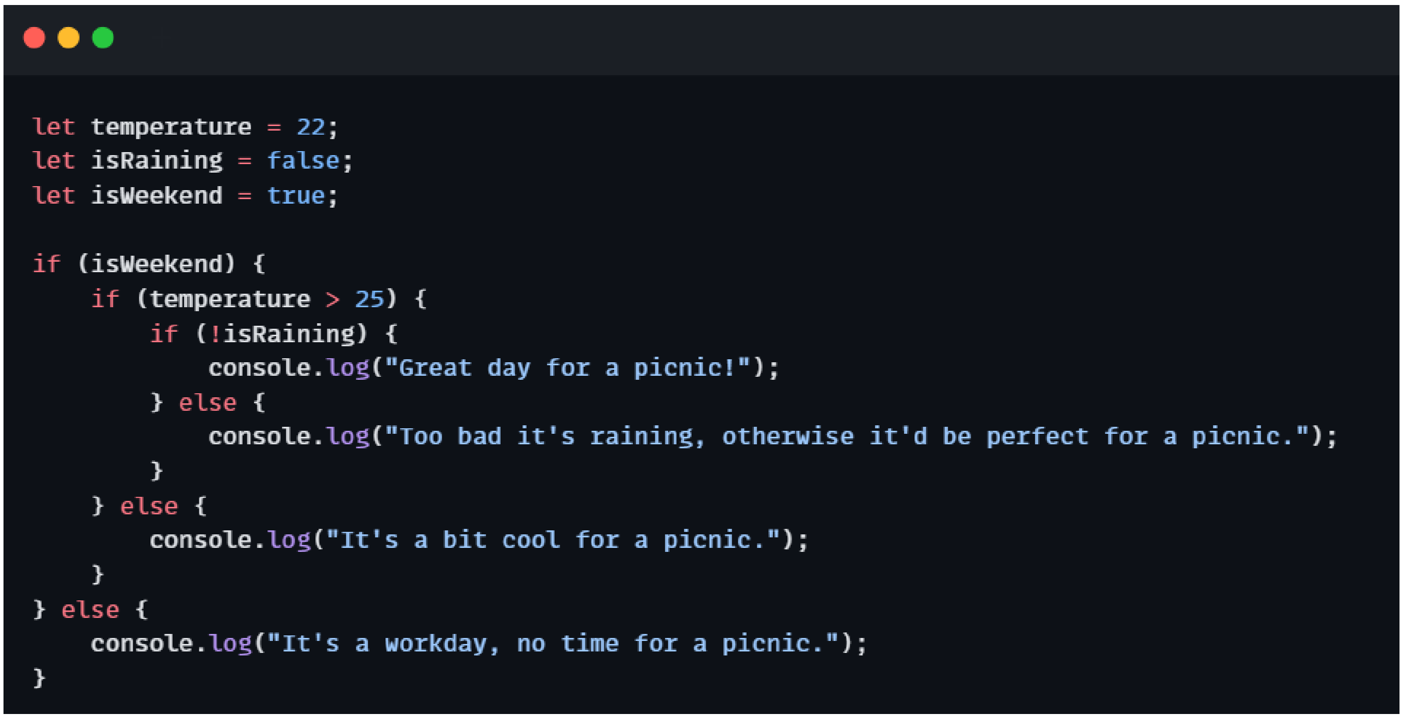
Nested if statements allow you to place one if statement inside another. This is useful when you want to check **multiple related conditions** in a more detailed way. It gives your code more precise control over decision-making based on multiple layers of logic.

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:

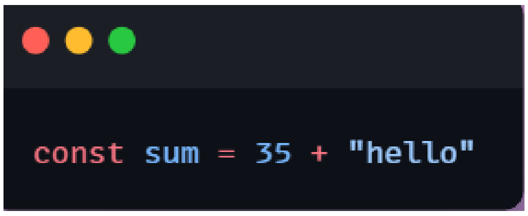
Type conversion is the process of converting a value from one data type to another. This is common in programming when we need to perform operations that require specific types. For example, converting a **string** to a **number**.

Values in JavaScript can be of different types, such as **numbers**, **strings**, **objects**, and **booleans**. Sometimes, we may need to convert data from one type to another to perform 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 (Coercion):

Sometimes, in JavaScript, we try to perform operations that don't logically make sense between different data types. For example, adding a number and a string directly.

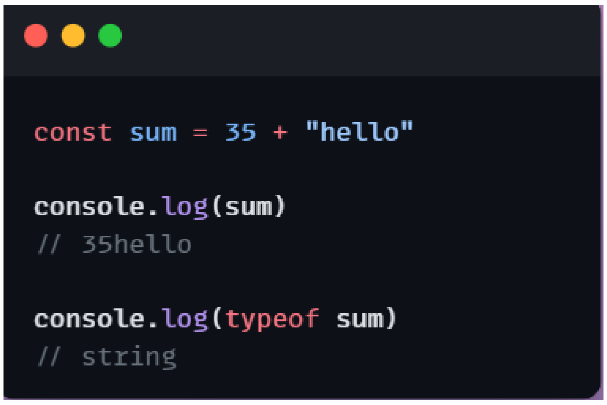


```javascript
const sum = 35 + "hello"
```

In most programming languages, this operation might cause an error because we cannot add a number and a string directly. we can either **add numbers** (to get a sum) or **add strings** (to concatenate). 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(**converts**) the type of one value to fit the type of the other value so that the operation can be carried out. In the example above, JavaScript converts the number 35 into the string "35" and then performs string concatenation with "hello", resulting in the string "35hello".

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

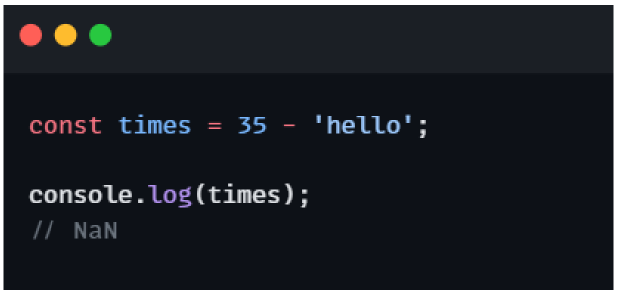


```javascript
const sum = 35 + "hello"

console.log(sum)
// 35hello

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

When we use the **plus (+)** sign with a number and a string, JavaScript usually turns the **number into a string** instead of trying to change the string into a number. This is because:

- Turning a number like 15 into a string "15" works perfectly.
- But trying to change a string that isn't a number (like "hello") into a number can cause problems and result in **NaN** (which means "Not a Number").

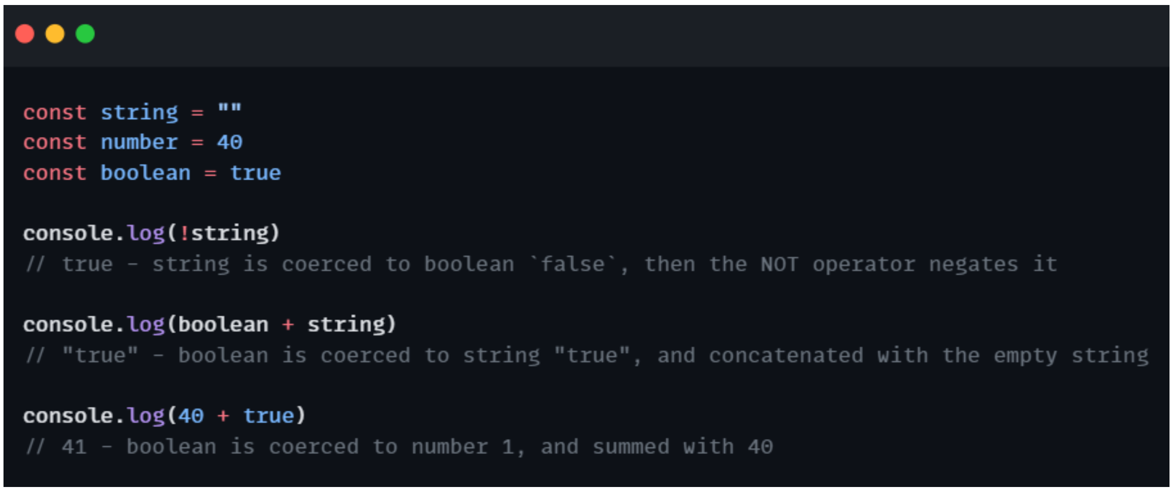
Another example:



```
const times = 35 - 'hello';

console.log(times);
// NaN
```

In this example, we're using the **minus (-)** sign with a **number** and a **string**. JavaScript doesn't know how to subtract a string from a number, so it tries to **convert the string into a number**. But if the string isn't a valid number — like "hello" — the result becomes NaN (which means "Not a Number"). Then JavaScript tries to do the subtraction, but since one of the values is NaN, the final result is also 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**).

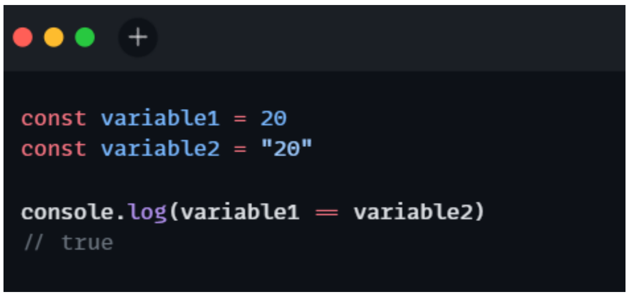
Double Equality and Coercion:

In JavaScript there are two equality operators: 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 (==) compares **two values**, but it **does not check their data types**. Before comparing, JavaScript tries to **convert the values to the same type**. This is called **type coercion**. So even if the types are different, it might still return true if the values are the same after conversion.

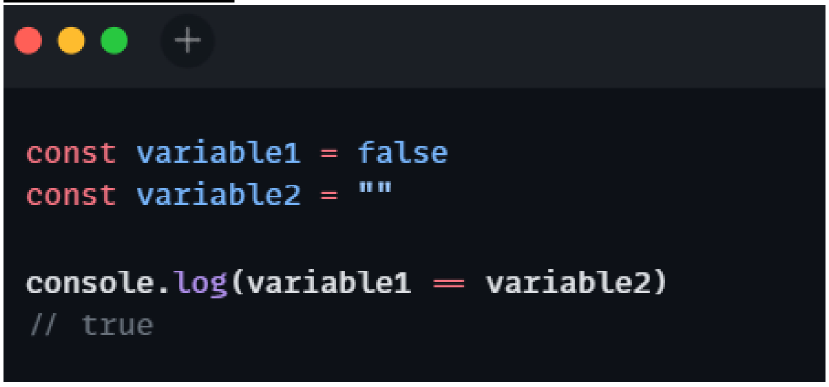


```
const variable1 = 20
const variable2 = "20"

console.log(variable1 == variable2)
// true
```

The number 20 and the string "20" are considered equal when using the **double equality (==)** operator, because JavaScript converts the string to a number before comparing.

What's Happening Behind the Scenes?
When we use "==" with values of **different types**, JavaScript first performs **coercion**—that means it **converts one value** to match the type of the other.
In the above example, the string "20" is converted into the number 20 before the comparison takes place. Since both values are now the same (20), the result is true.

Another example:

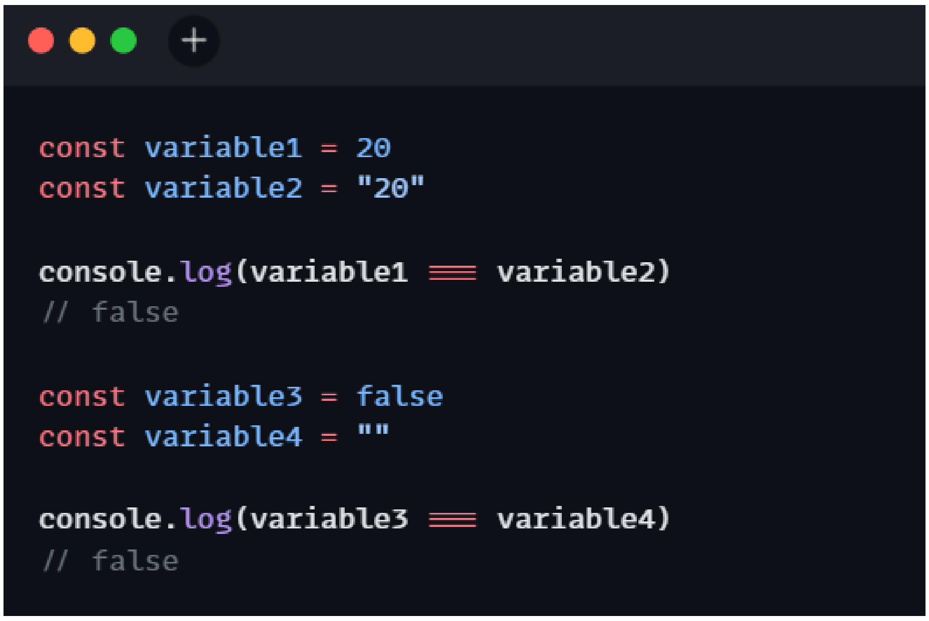


```
const variable1 = false
const variable2 = ""

console.log(variable1 == variable2)
// true
```

Here, variable1 has the value false (a Boolean), and variable2 is an empty string "" (a string). When we compare them using the **double equality (==)** operator, the result is true, This happens because JavaScript **coerces** the empty string "" to a Boolean. An empty string is considered **falsy**, just like false. Since both are treated as false, the comparison returns true.

How the Strict Equality Operator (===) Works:

The **strict equality** operator (===) compares **both the value and the data type**. Unlike the loose equality (==), it **does not perform type coercion**. That means JavaScript does **not** try to convert one value to match the other — both must be exactly the same in **type** and **value**.



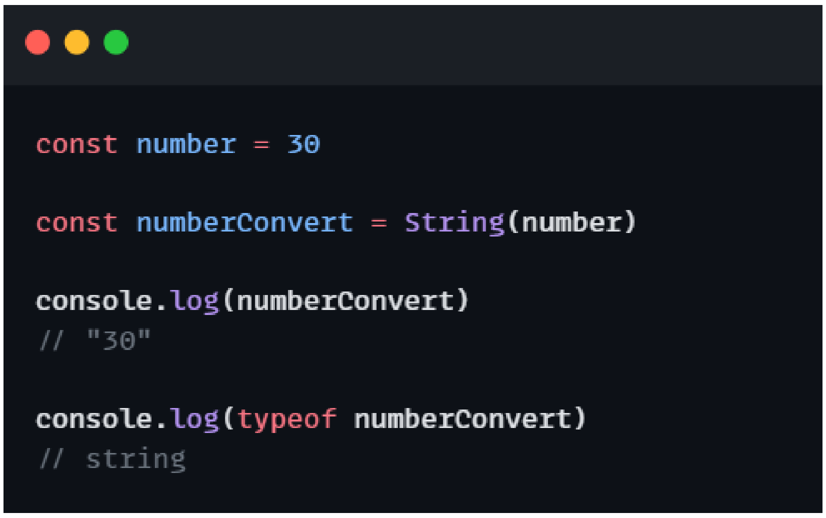
```
const variable1 = 20
const variable2 = "20"

console.log(variable1 === variable2)
// false

const variable3 = false
const variable4 = ""

console.log(variable3 === variable4)
// false
```

Even though the values look the same, 5 is a number and "5" is a string — so the comparison 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.

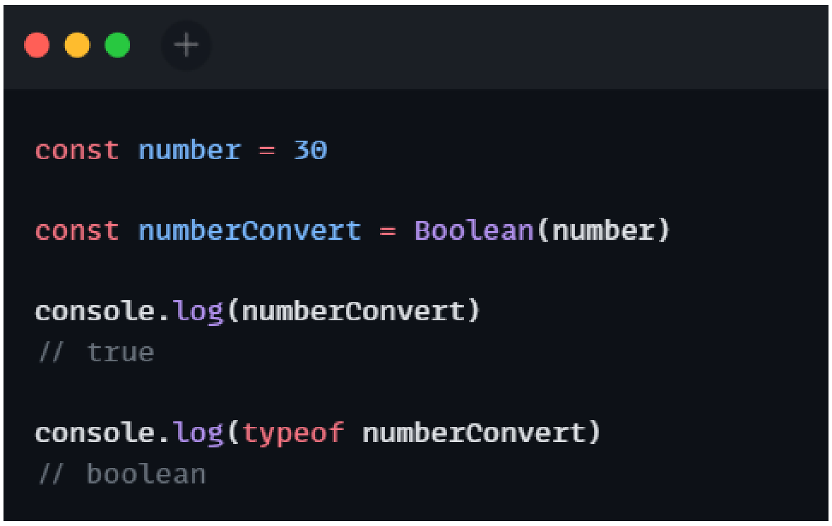
Explicit Conversion (Type Conversion):

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**:



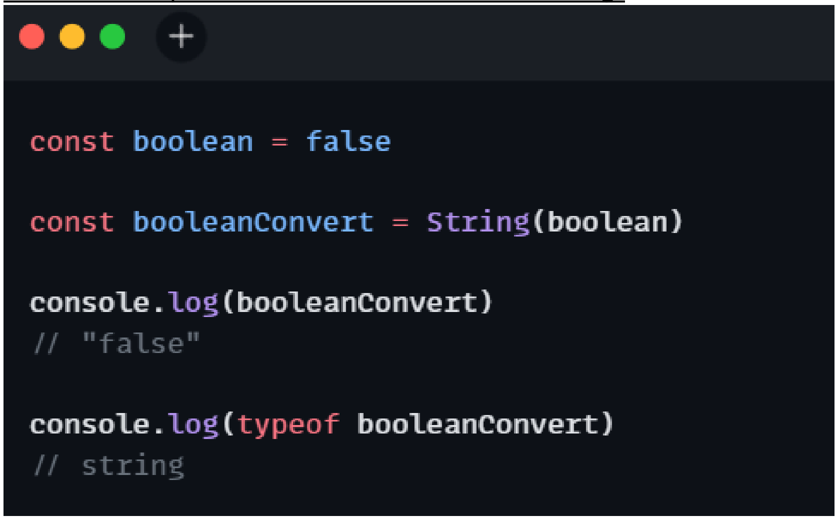
```
const number = 30

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:

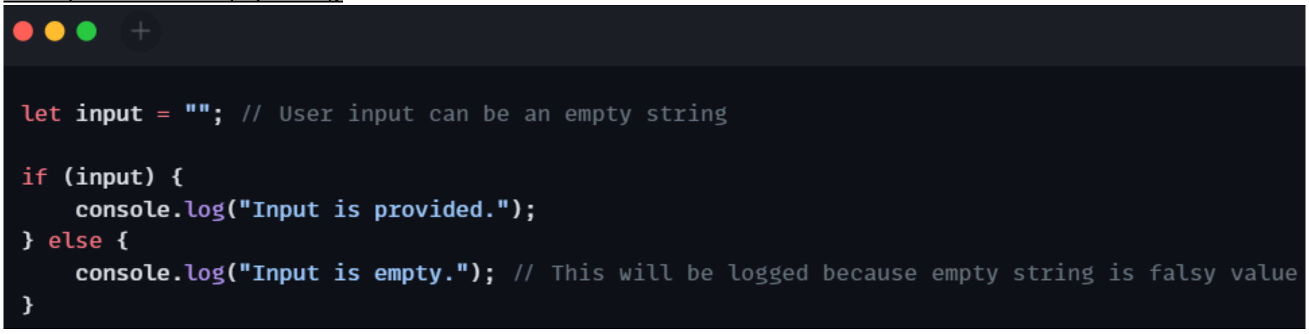
In JavaScript, the concepts of **truthy** and **falsy** values refer to how different values are evaluated in a **boolean** context, such as in conditions for **if statements** or **loops**.

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:

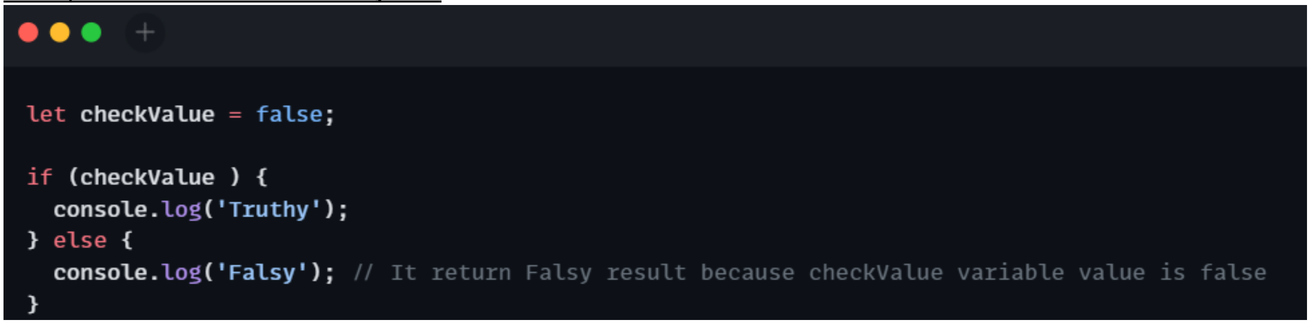
1. **False**: The Boolean value false.
2. **0:** The number zero - both positive (+0) and negative (-0) are falsy.
3. **"", '' or ``:** An empty string.
4. **Null**: The **null** keyword represents the **intentional absence of any object or value**. It is often used to explicitly indicate that a variable has no value or that an object is empty.
5. **Undefined**: The undefined keyword, representing an **uninitialized** value.
6. **NaN**: NaN stands for "**Not-a-Number**." It represents a special value in JavaScript that indicates an operation did not produce a valid numeric result. NaN is returned when a **mathematical operation fails** or when a value cannot be converted into a number.

Example 1 – An empty string



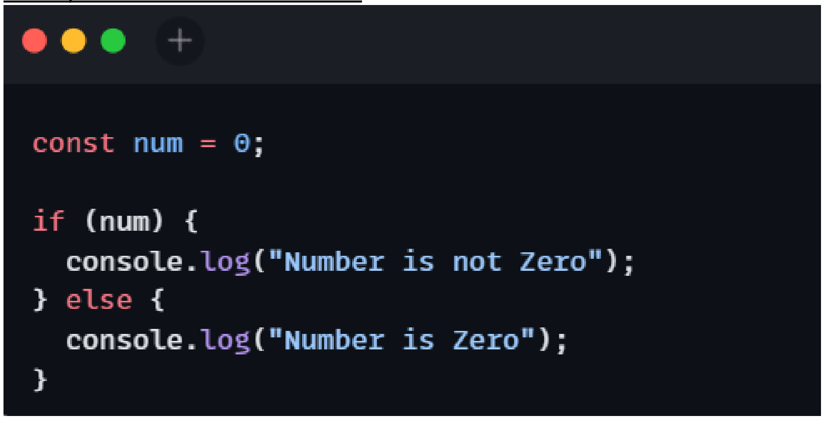
```javascript
let input = ""; // User input can be an empty string

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.



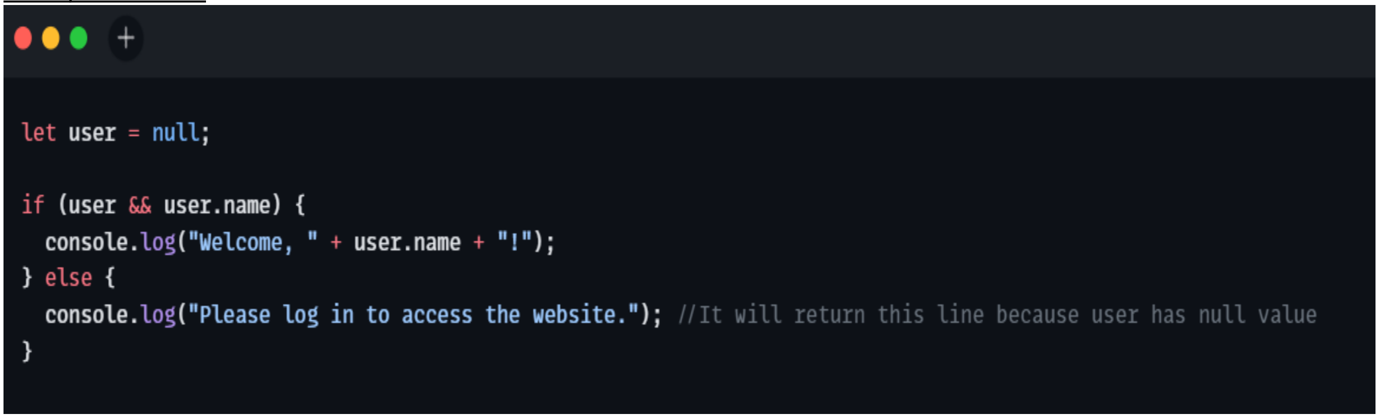
```javascript
let checkValue = false;

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

Example 3 – The number zero.



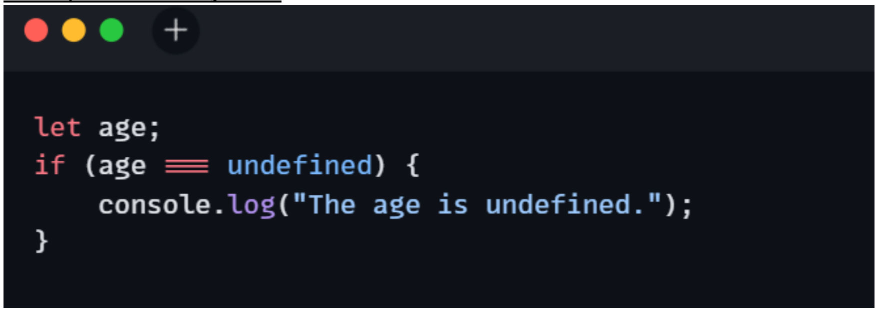
```javascript
const num = 0;

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

Example 4 – null:



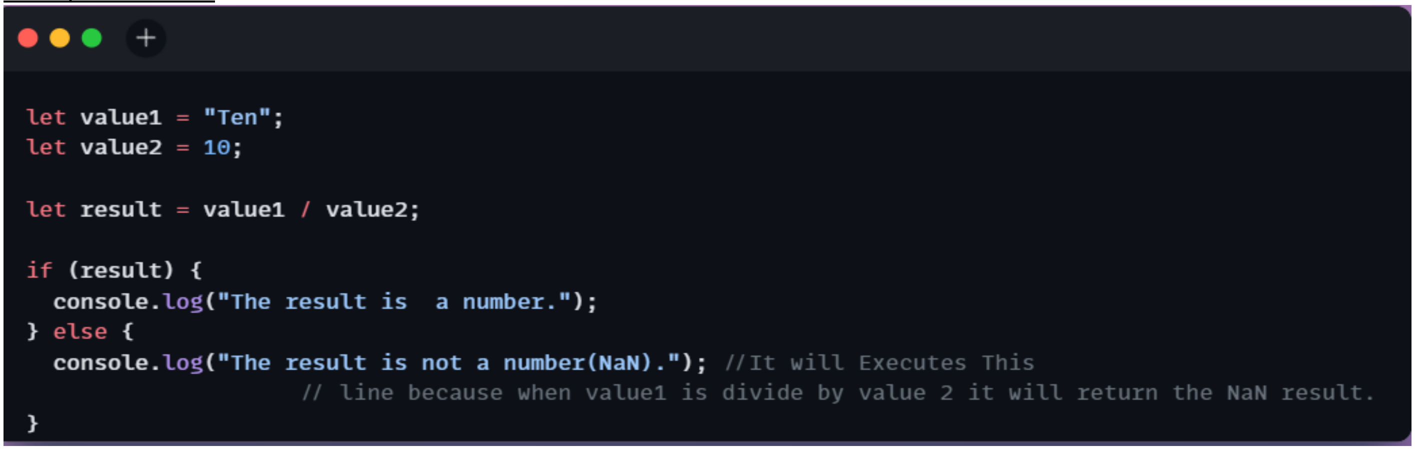
```
let user = null;

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:



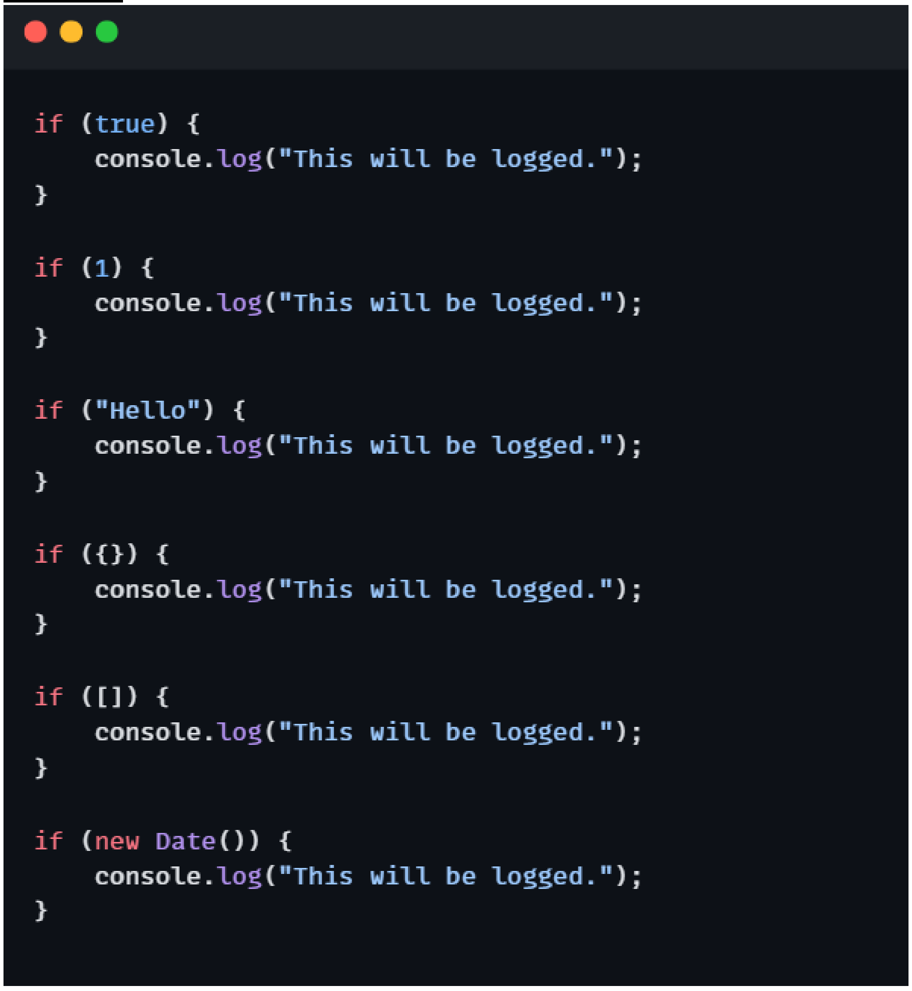
```
let value1 = "Ten";
let value2 = 10;

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:

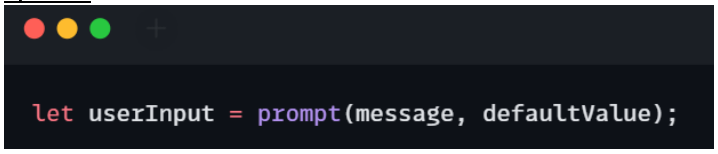


```
if (true) {
    console.log("This will be logged.");
}

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() method in JavaScript is used to **ask the user for input** through a small dialog box in the browser. When it's called, a popup appears with a message and a text field where the user can type something.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:



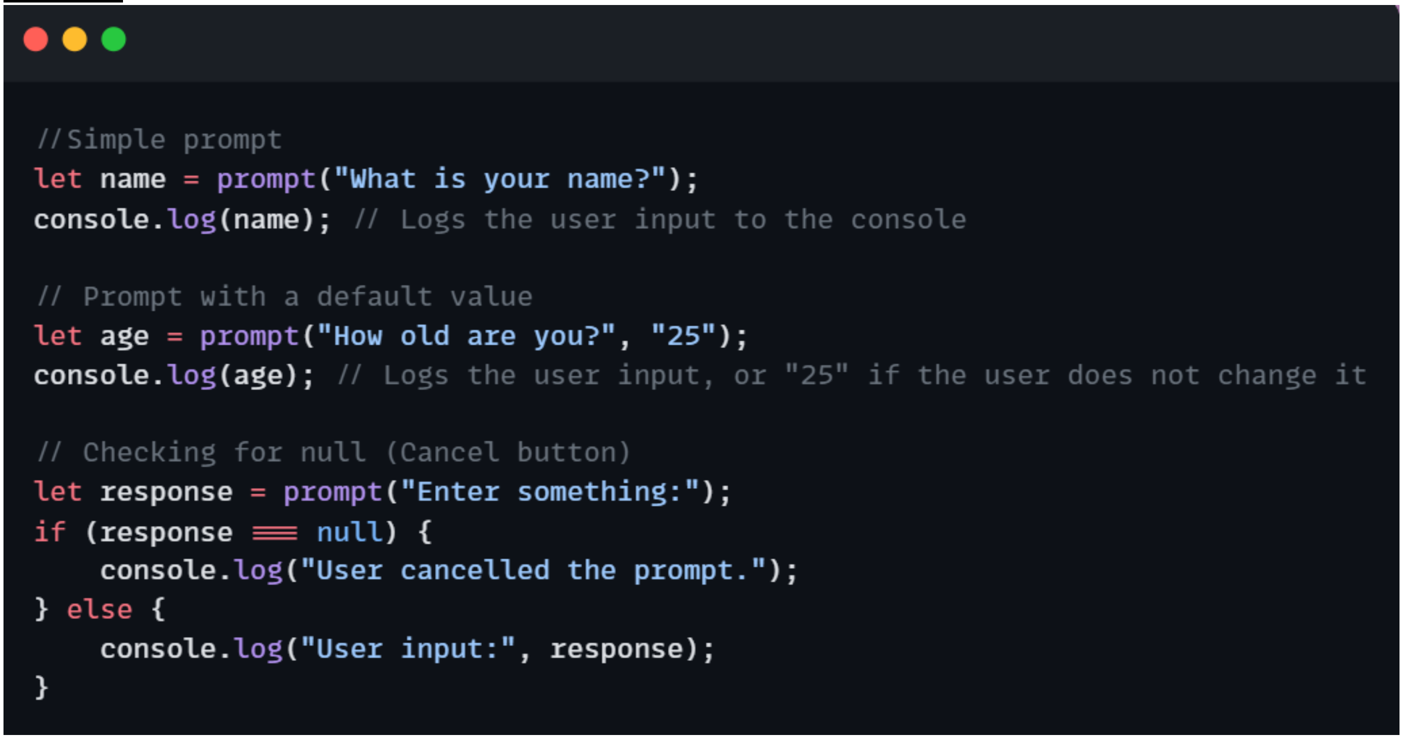
```
let userInput = prompt(message, defaultValue);
```

- **message***(optional)*: A string that appears in the prompt window — usually used to tell the user what to enter. If left out, the prompt will still appear, but it won't show any message.
- **default** *(optional)*: A string that appears as the **default value** in the input box. The user can keep it, change it, or delete it.

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**.

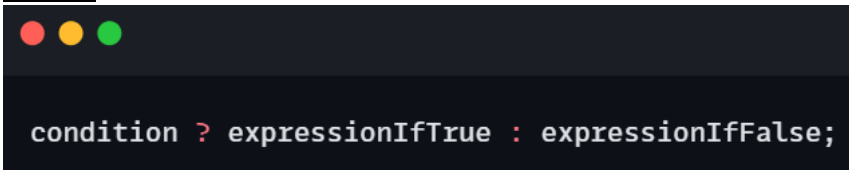
Example:



```
//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:

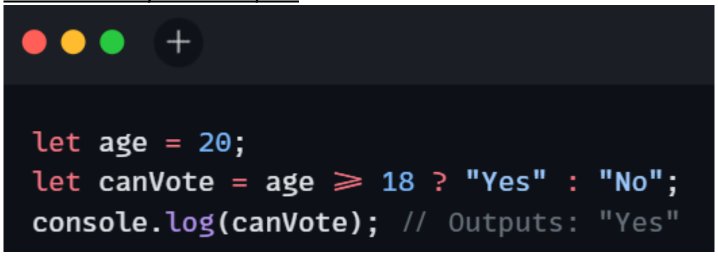
The ternary operator is a concise way to execute one of **two expressions** based on a condition. It is often used as a shorter alternative to an **if-else statement**. The ternary operator is written with the following *syntax:*



```
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:

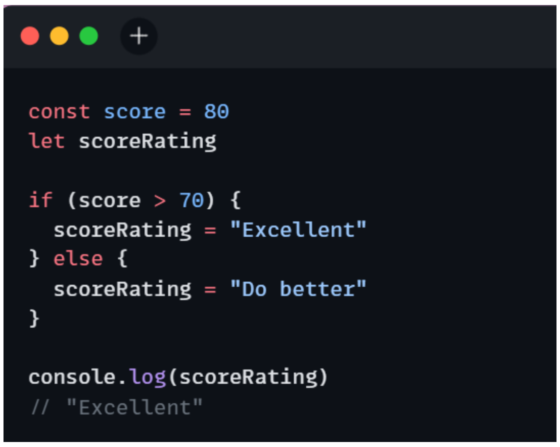


```
let age = 20;
let canVote = age >= 18 ? "Yes" : "No";
console.log(canVote); // Outputs: "Yes"
```

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.

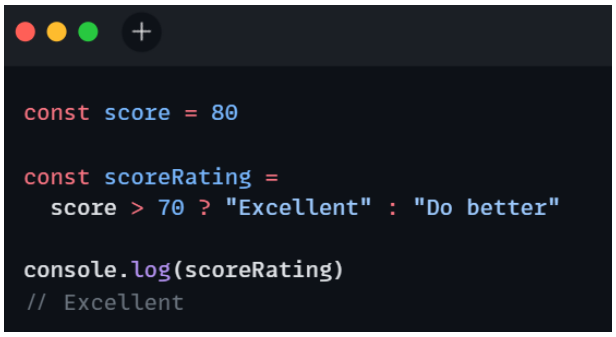
Let's see an example:



```
const score = 80
let scoreRating

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

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

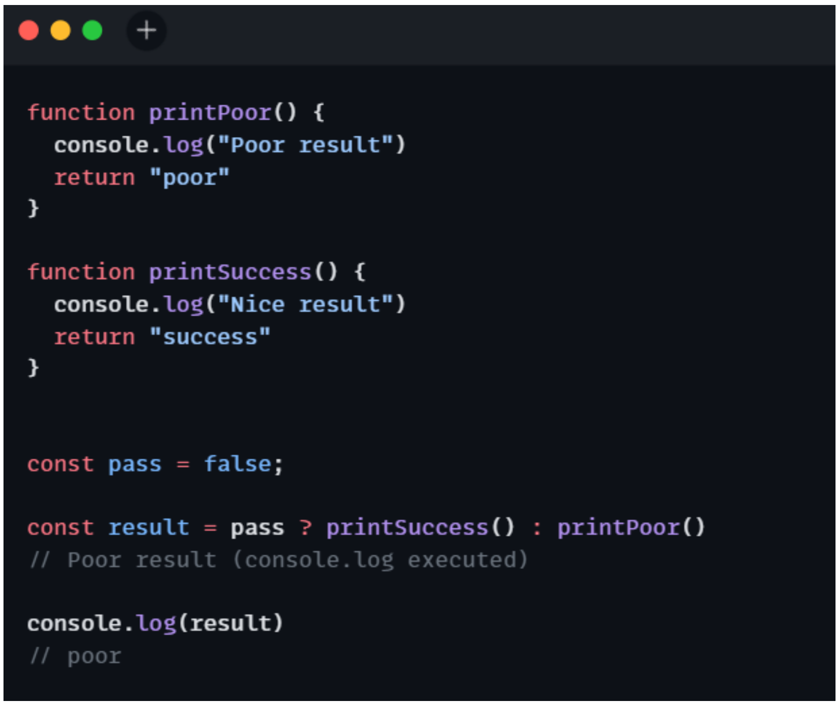
In this example, we have a **score** variable with value of 80 and a **scoreRating** variable. Then we have an if statement that checks **score > 70**. If this condition evaluates to **true**, the scoreRating variable is assigned "Excellent", else, it is assigned "Do better".

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:



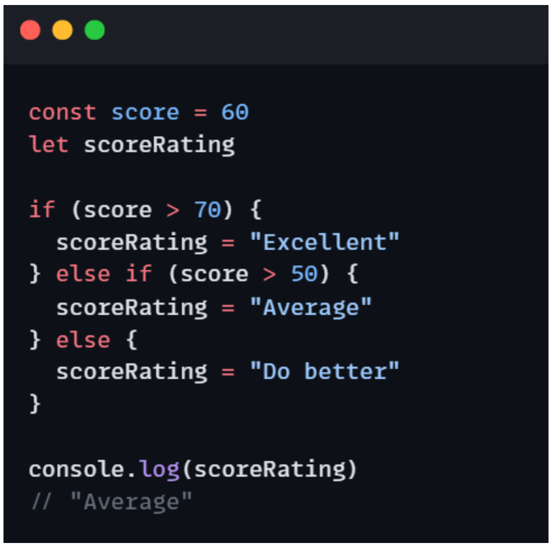
```
function printPoor() {
  console.log("Poor result")
  return "poor"
}

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.

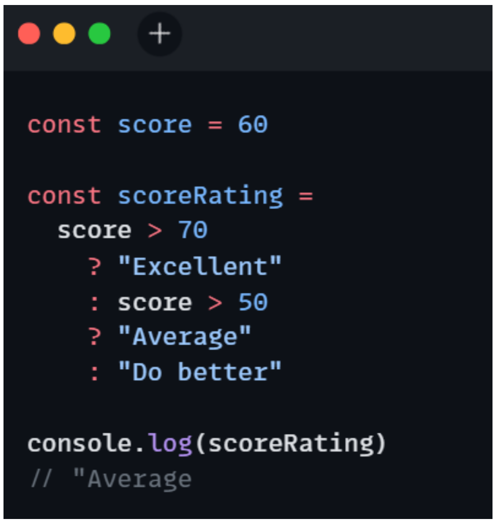
Let's see an example:



```
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 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:

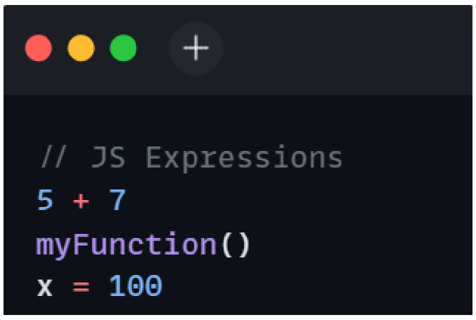
JavaScript Expression:

An **expression** is a piece of code that **produces a value**.

- It can be a number, string, boolean, object, or even a function call.
- Expressions can be used **wherever a value is needed** — like inside variables, conditions, or return statements.

Examples of expressions include:

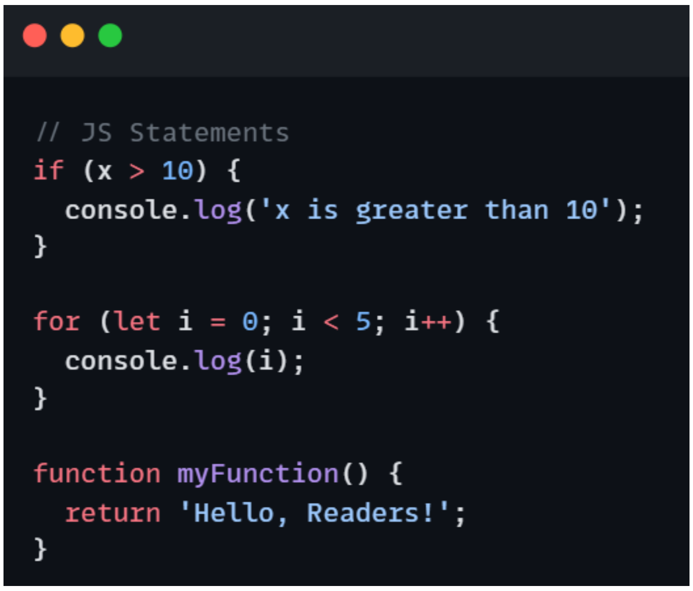
- **Arithmetic operations**: 5 + 3 (evaluates to 8).
- **Function calls**: Math.max(10, 20) (evaluates to 20).
- **Variable assignments**: let x = 10 * 2 (the expression 10 * 2 evaluates to 20).



```
// 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!';
}
```