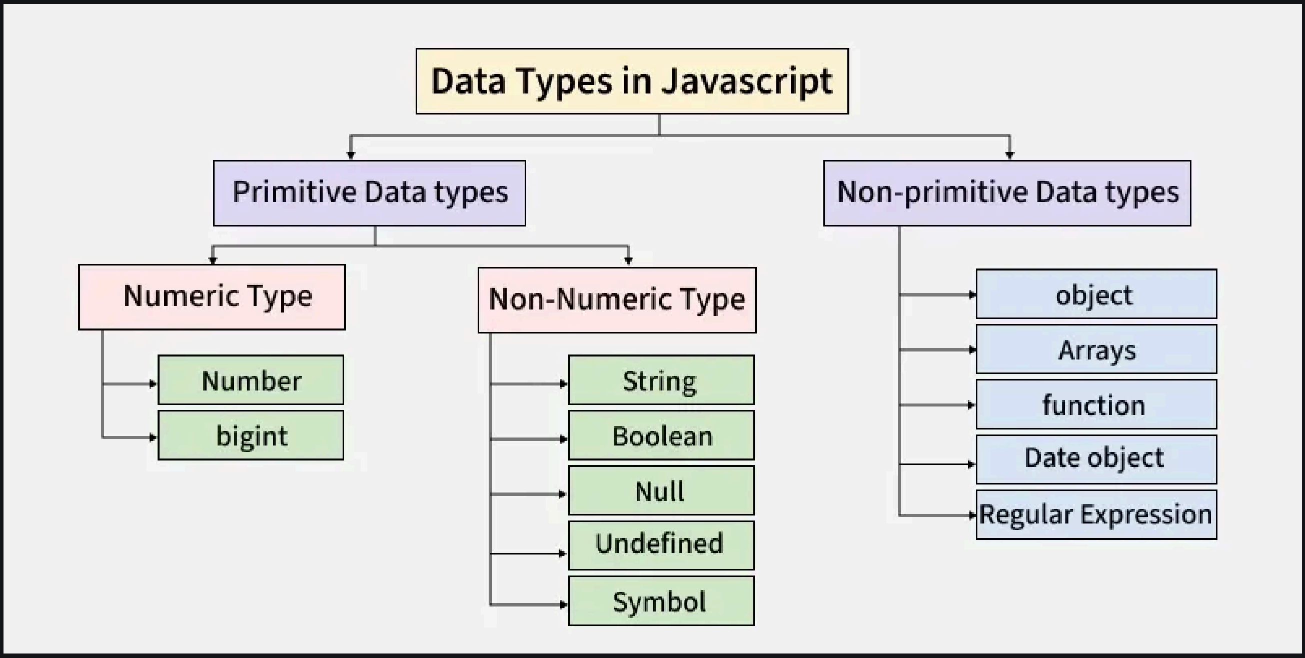
Variables and Datatypes in JavaScript

Last Updated : 19 Nov, 2025

Variables and data types are foundational concepts in programming, serving as the building blocks for storing and manipulating information within a program. In JavaScript, getting a good grasp of these concepts is important for writing code that works well and is easy to understand.



Variables

A variable is like a container that holds data that can be reused or updated later in the program. In JavaScript, variables are declared using the keywords var, let, or const.

1. var Keyword

The var keyword is used to declare a variable. It has a function-scoped or globally-scoped behaviour.

```
var n = 5;
console.log(n);

var n = 20; // reassigning is allowed
console.log(n);
```

Output

```
5
20
```

2. let Keyword

The let keyword is introduced in ES6, has block scope and cannot be re-declared in the same scope.

```
let  n= 10;
n = 20; // Value can be updated
// let n = 15; //can not redeclare
console.log(n)
```

Output

```
20
```

3. const Keyword

The const keyword declares variables that cannot be reassigned. It's block-scoped as well.

```
const n = 100;
// n = 200; This will throw an error
console.log(n)
```

Output

```
100
```

> *For more details read the article - JavaScript Variables*

Data Types

JavaScript supports various datatypes, which can be broadly categorized into primitive and non-primitive types.

Primitive Datatypes

Primitive datatypes represent single values and are immutable.

1. Number: Represents numeric values (integers and decimals).

```
let n = 42;
let pi = 3.14;
```

2. String: Represents text enclosed in single or double quotes.

```
let s = "Hello, World!";
```

3. Boolean: Represents a logical value (true or false).

```
let bool= true;
```

4. Undefined: A variable that has been declared but not assigned a value.

```
let notAssigned;
```

```
console.log(notAssigned);
```

```
undefined
```

5. Null: Represents an intentional absence of any value.

```
let empty = null;
```

6. Symbol: Represents unique and immutable values, often used as object keys.

```
let sym = Symbol('unique');
```

7. BigInt: Represents integers larger than Number.MAX_SAFE_INTEGER.

```
let bigNumber = 123456789012345678901234567890n;
```

Non-Primitive Datatypes

Non-primitive types are objects and can store collections of data or more complex entities.

1. Object: Represents key-value pairs.

```
let obj = {
    name: "Amit",
    age: 25
};
```

2. Array: Represents an ordered list of values.

```
let a = ["red", "green", "blue"];
```

3. Function: Represents reusable blocks of code.

```
function fun() {
    console.log("GeeksforGeeks");
}
```

Exploring JavaScript Datatypes and Variables: Understanding Common Expressions

```
console.log(null === undefined)
```

- **Expression**: null === undefined
- **Result**: false

In JavaScript, both null and undefined represent "empty" values but are distinct types. null is a special object representing the intentional absence of a value, while undefined signifies that a variable has been declared but not assigned a value. Despite their similar purpose, they are not strictly equal (===) to each other.

- `null === undefined` evaluates to `false` because JavaScript does not perform type coercion with `===`.

```
console.log(5 > 3 > 2)
```

- **Expression**: `5 > 3 > 2`
- **Result**: `false`

At first glance, this expression may appear to be checking if 5 is greater than 3 and 3 is greater than 2, but JavaScript evaluates it left-to-right due to its operator precedence.

- First, `5 > 3` evaluates to `true`.
- Then, `true > 2` is evaluated, which in JavaScript results in `1 > 2` (since `true` is coerced to 1), which evaluates to `false`.

So, `5 > 3 > 2` evaluates to `false`.

```
console.log([] === [])
```

- **Expression**: `[] === []`
- **Result**: `false`

In JavaScript, arrays are objects. Even if two arrays have the same content, they are still different objects in memory.

- When you compare two arrays with `===`, you are comparing their references, not their contents.
- Since `[]` and `[]` are different instances in memory, the result is `false`.

```
console.log("10" < "9")
```

- **Expression**: `"10" < "9"`
- **Result**: `true`

When JavaScript compares strings, it compares their Unicode values lexicographically (character by character).

- `"10"` is compared to `"9"`. Since `"1"` has a lower Unicode value than `"9"`, JavaScript determines that `"10"` is less than `"9"`.
- This comparison might seem counterintuitive, but it's due to JavaScript's string comparison mechanism.

```
console.log(NaN === NaN)
```

- **Expression**: `NaN === NaN`
- **Result**: `false`

In JavaScript, `NaN` (Not-a-Number) is a special value that represents an invalid number or the result of an operation that cannot produce a valid number.

- One of the most unusual aspects of `NaN` is that it is not equal to itself. This behavior exists due to the design of the IEEE 754 standard, which JavaScript follows for floating-point arithmetic.
- As a result, `NaN === NaN` returns `false`.

To check if a value is NaN, use Number.isNaN().

```
console.log(true == 1)
```

- **Expression**: true == 1
- **Result**: true

JavaScript uses type coercion with the loose equality operator (==). When comparing true and 1, JavaScript converts true to 1 and then compares the values.

- Since 1 == 1 is true, the overall expression evaluates to true.

This behavior might lead to unexpected results in some cases, so it's often recommended to use the strict equality operator (===) to avoid implicit type coercion.

```
console.log(undefined > 0)
```

- **Expression**: undefined > 0
- **Result**: false

When JavaScript attempts to compare undefined with 0, it converts undefined to NaN (Not-a-Number). Any comparison involving NaN returns false.

- undefined > 0 becomes NaN > 0, which evaluates to false.

```
console.log("5" === 5)
```

- **Expression**: "5" === 5
- **Result**: false

The strict equality operator (===) checks both value and type. Since "5" is a string and 5 is a number, the types are different, and the comparison returns false.

- If you used the loose equality operator (==), JavaScript would perform type coercion, converting the string "5" to the number 5, and the comparison would return true.

```
console.log([1, 2] == [1, 2])
```

- **Expression**: [1, 2] == [1, 2]
- **Result**: false

Even though both arrays contain the same elements, JavaScript compares arrays by reference, not by value.

- Since each array is a separate object in memory, their references are different, and thus the comparison returns false.

To check if two arrays are equal, you must compare their contents element by element.

```
console.log(Infinity > 1000)
```

- **Expression**: Infinity > 1000
- **Result**: true