Introduction to JavaScript

What is JavaScript?

JavaScript is a programming language initially designed to interact with elements of web pages. In web browsers, JavaScript consists of three main parts:

- ✓ ECMAScript provides the core functionality.
- ✓ The Document Object Model (DOM) provides interfaces for interacting with elements on web pages
- ✓ The Browser Object Model (BOM) provides the browser API for interacting with the web browser.

JavaScript allows you to add interactivity to a web page. Typically, you use JavaScript with HTML and CSS to enhance a web page's functionality, such as validating forms, creating interactive maps, and displaying animated charts.

When a web page is loaded, i.e., after HTML and CSS have been downloaded, the JavaScript engine in the web browser executes the JavaScript code. The JavaScript code then modifies the HTML and CSS to update the user interface dynamically.

The JavaScript engine is a program that executes JavaScript code. In the beginning, JavaScript engines were implemented as interpreters.

JavaScript History

In 1995, JavaScript was created by a Netscape developer named Brendan Eich. First, its name was Mocha. And then, its name was changed to LiveScript.

Netscape decided to change LiveScript to JavaScript to leverage Java's fame, which was popular. The decision was made just before Netscape released its web browser product Netscape Navigator 2. As a result, JavaScript entered version 1.0.

Netscape released JavaScript 1.1 in Netscape Navigator 3. In the meantime, Microsoft introduced a web browser product called Internet Explorer 3 (IE 3), which competed with Netscape. However, IE came with its own JavaScript implementation called JScript. Microsoft used the name JScript to avoid possible license issues with Netscape.

Hence, two different JavaScript versions were in the market:

- ✓ JavaScript in Netscape Navigator
- ✓ JScript in Internet Explorer.

JavaScript Hello World Example

To insert JavaScript into an HTML page, you use the <script> element. There are two ways to use the <script> element in an HTML page:

- ✓ Embed JavaScript code directly into the HTML page.
- ✓ Reference an external JavaScript code file.

Embed JavaScript code on an HTML page

Placing JavaScript code inside the <script> element directly is not recommended and should be used only for proof of concept or testing purposes.

The JavaScript code in the <script> element is interpreted from top to bottom.

For example:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>JavaScript Hello World Example</title>

    <script>

        alert('Hello, World!');

    </script>

</head>

<body>

</body>

</html>
```

In the <script> element, we use the alert() function to display the Hello, World! Message.

Include an external JavaScript file

To include a JavaScript from an external file:

- ✓ First, create a file whose extension is .js e.g., app.js and place it in the js subfolder. Note that placing the JavaScript file in the js folder is not required however it is a good practice.
- ✓ Then, use the URL to the JavasScript source code file in the src attribute of the <script> element.

And the following shows the helloworld.html file:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>JavaScript Hello World Example</title>

    <script src="js/app.js"></script>

</head>

<body>

</body>

</html>
```

JavaScript Fundamentals

JavaScript Syntax:

Statements

A statement is a code that declares a variable or instructs the JavaScript engine to do a task. A simple statement is terminated by a semicolon (;).

Although the semicolon (;) is optional; you should always use it to terminate a statement. For example, the following declares a variable and shows it to the console:

Example

```
let message = "Welcome to JavaScript";

console.log(message);
```

Identifiers

An identifier is a name you choose for variables, parameters, functions, classes, etc. An identifier name starts with a letter (a-z, or A-Z), an underscore (_), or a dollar sign ($) and is followed by a sequence of characters including (a-z, A-Z), numbers (0-9), underscores (_), and dollar signs ($).

Note that the letter is not limited to the ASCII character and may include extended ASCII or Unicode though not recommended.

Identifiers are case-sensitive. For example, the **message** is different from the **Message**.

Comments

Comments allow you to add notes or hints to JavaScript code. When executing the code, the JavaScript engine ignores the comments.

JavaScript supports single-line and block comments.

Single-line comments

A single-line comment starts with two forward-slashes characters (//). A single-line comment makes all the text following the // on the same line into a comment. For example:

// this is a single-line comment

Block comments

A delimited comment begins with a forward slash and asterisk /* and ends with the opposite */ as in the following example:

/* This is a block comment

that can span multiple lines ***/**

Keywords & Reserved words

JavaScript defines a list of reserved keywords that have specific uses. Therefore, you cannot use the reserved keywords as identifiers or property names by rules.

The following table shows the JavaScript reserved words defined in ECMA-262:

`break`	`case`	`catch`
`continue`	`debugger`	`default`
`else`	`export`	`extends`
`function`	`if`	`import`
`new`	`return`	`super`
`throw`	`try`	`null`
`void`	`while`	`with`
`class`	`delete`	`finally`
`in`	`switch`	`typeof`
`yield`	`const`	`do`
`for`	`instanceof`	`this`
`var`		

JavaScript Variables:

A variable is a label that references a value like a number or string. Before using a variable, you need to declare it.

Declare a variable

To declare a variable, you use the **var** keyword followed by the variable name as follows:

var message;

A variable name can be any valid identifier. By default, the message variable has a special value **undefined** if you have not assigned a value to it.

Variable names follow these rules:

- ✓ Variable names are case-sensitive. This means that the message and Message are different variables.
- ✓ Variable names can only contain letters, numbers, underscores, or dollar signs and cannot contain spaces. Also, variable names must begin with a letter, an underscore (_) or a dollar sign ($).
- ✓ Variable names cannot use the reserved words.

By convention, variable names use camelcase like **message, yourAge, and myName**.

JavaScript is a dynamically typed language. This means that you don't need to specify the variable's type in the declaration like other static-typed languages such as Java or C#.

Starting in ES6, you can use the **let** keyword to declare a variable like this:

let message;

It's a good practice to use the let keyword to declare a variable.

Initialize a variable

Once you have declared a variable, you can initialize it with a value. To initialize a variable, you specify the variable name, followed by an equals sign (=) and a value.

For example, the following declares the message variable and initializes it with a literal string "Hello":

let message;

message = "Hello";

To declare and initialize a variable at the same time, you use the following syntax:

```
let variableName = value;
```

JavaScript allows you to declare two or more variables using a single statement. To separate two variable declarations, you use a comma (,) like this:

```
let message = "Hello", counter = 100;
```

Undefined vs. undeclared variables

It's important to distinguish between undefined and undeclared variables.

An undefined variable is a variable that has been declared but has not been initialized with a value. For example:

```
let message;

console.log(message); // undefined
```

In this example, the message variable is declared but not initialized. Therefore, the message variable is undefined.

In contrast, an undeclared variable is a variable that has not been declared. For example:

```
console.log (counter);
```

Output:

```
console.log(counter);

ReferenceError: counter is not defined
```

Constants

A constant holds a value that doesn't change. To declare a constant, you use the **const** keyword. When defining a constant, you need to initialize it with a value. **For example:**

```
const workday = 5;
```

Once you define a constant, you cannot change its value.

The following example attempts to change the value of the workday constant to 4 and causes an error:

```
workday = 2;
```

Error:

Uncaught TypeError: Assignment to constant variable.

JavaScript Data Types:

JavaScript has the primitive data types:

1. null
2. undefined
3. boolean
4. number
5. string
6. symbol – available from ES2015
7. bigint – available from ES2020

and a complex data type **object.**

JavaScript is a dynamically typed language. It means that a variable doesn't associate with a type. 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
```

To get the current type of the value that the variable stores, you use the **typeof** operator:

```
let counter = 120;

console.log(typeof(counter)); // "number"
```

The undefined type

The undefined type is a primitive type that has only one value undefined. By default, when a variable is declared but not initialized, it is assigned the value of undefined.

Consider the following example:

```
let counter;

console.log(counter);       // undefined

console.log(typeof counter); // undefined
```

The null type

The null type is the second primitive data type that also has only one value null. **For example:**

```
let obj = null;

console.log(typeof obj); // object
```

Note that,

The typeof null returns object is a known bug in JavaScript. A proposal to fix this was proposed but rejected. The reason was the fix would break a lot of existing sites.

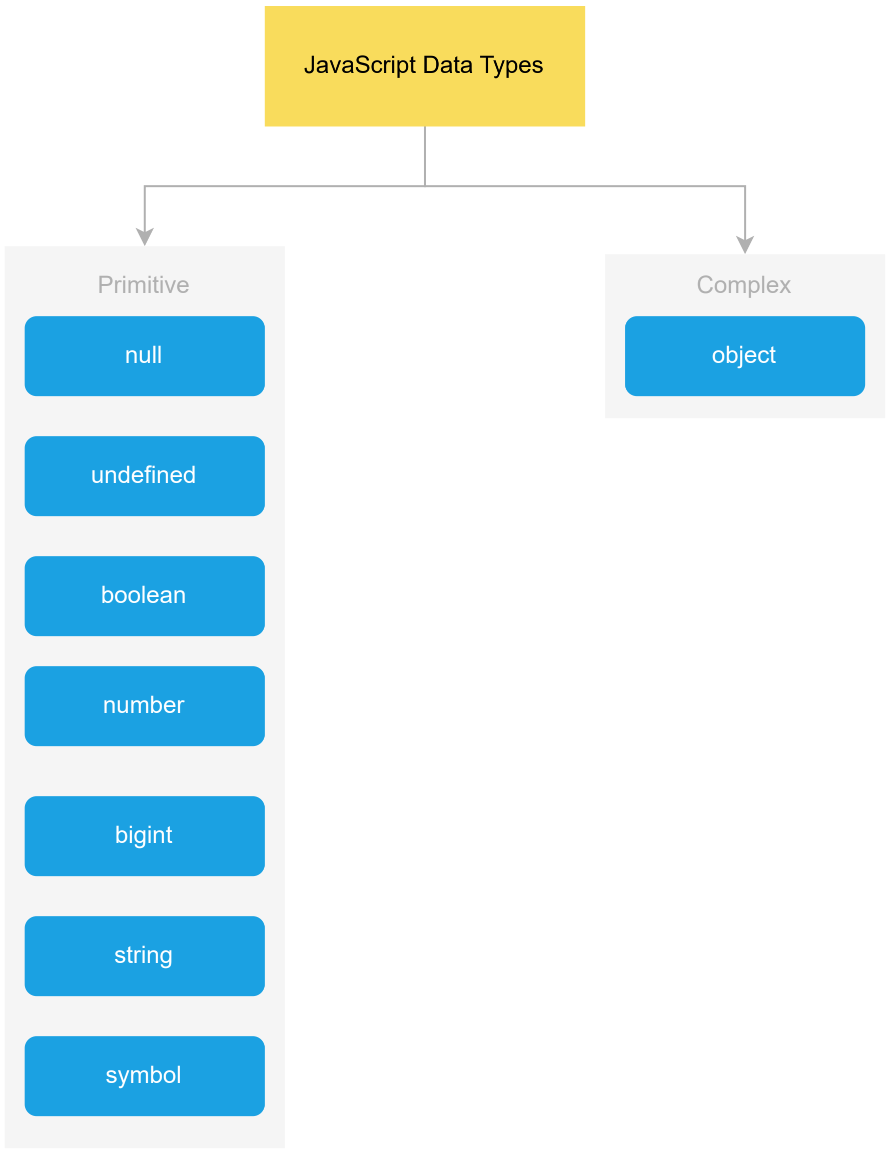
JavaScript Data Types

Summary: in this tutorial, you will learn about the JavaScript data types and their unique characteristics.

JavaScript has the primitive data types:

1. `null`
2. `undefined`
3. `boolean`
4. `number`
5. `string`
6. `symbol` – available from ES2015
7. `bigint` – available from ES2020

and a complex data type `object` .



JavaScript is a dynamically typed language. It means that a variable doesn't associate with a type. 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
```

To get the current type of the value that the variable stores, you use the `typeof` operator:

```
let counter = 120;
console.log(typeof(counter)); // "number"

counter = false;
console.log(typeof(counter)); // "boolean"
```

```
counter = "Hi";
console.log(typeof(counter)); // "string"
```

Output:

```
"number"
"boolean"
"string"
```

The undefined type

The `undefined` type is a primitive type that has only one value `undefined`. By default, when a variable is declared but not initialized, it is assigned the value of `undefined`.

Consider the following example:

```
let counter;
console.log(counter);        // undefined
console.log(typeof counter); // undefined
```

In this example, the `counter` is a variable. Since `counter` hasn't been initialized, it is assigned the value `undefined`. The type of `counter` is also `undefined`.

It's important to note that the `typeof` operator also returns `undefined` when you call it on a variable that hasn't been declared:

```
console.log(typeof undeclaredVar); // undefined
```

The null type

The `null` type is the second primitive data type that also has only one value `null`. For example:

```
let obj = null;
console.log(typeof obj); // object
```

> The typeof null returns object is a known bug in JavaScript. A proposal to fix this was proposed but rejected. The reason was the that fix would break a lot of existing sites.

JavaScript defines that `null` is equal to `undefined` as follows:

```
console.log(null == undefined); // true
```

The number type

JavaScript uses the `number` type to represent both integer and floating-point numbers.

The following statement declares a variable and initializes its value with an integer:

```
let num = 100;
```

To represent a floating-point number, you include a decimal point followed by at least one number. For example:

```
let price= 12.5;
let discount = 0.05;
```

Note that JavaScript automatically converts a floating-point number into an integer number if the number appears to be a whole number.

The reason is that Javascript always wants to use less memory since a floating-point value uses twice as much memory as an integer value. For example:

```
let price = 200.00; // interpreted as an integer 200
```

To get the range of the number type, you use `Number.MIN_VALUE` and `Number.MAX_VALUE` . For example:

```
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
console.log(Number.MIN_VALUE); // 5e-324
```

Also, you can use `Infinity` and `-Infinity` to represent the infinite number. For example:

```
console.log(Number.MAX_VALUE + Number.MAX_VALUE); // Infinity
console.log(-Number.MAX_VALUE - Number.MAX_VALUE); // -Infinity
```

NaN

`NaN` stands for Not a Number. It is a special numeric value that indicates an invalid number. For example, the division of a string by a number returns `NaN` :.

```
console.log('a'/2); // NaN;
```

The `NaN` has two special characteristics:

- Any operation with `NaN` returns `NaN` .
- The `NaN` does not equal any value, including itself.

Here are some examples:

```
console.log(NaN/2); // NaN
console.log(NaN == NaN); // false
```

The string type

In JavaScript, a string is a sequence of zero or more characters. A string literal begins and ends with either a single quote(`'`) or a double quote (`"`).

A string that begins with a double quote must end with a double quote. Likewise, a string that begins with a single quote must also end with a single quote:

```
let greeting = 'Hi';
let message  = "Bye";
```

If you want to single quote or double quotes in a literal string, you need to use the backslash to escape it. For example:

```
let message = 'I\'m also a valid string'; // use \ to escape the single quote (')
```

JavaScript strings are immutable. This means that it cannot be modified once created. However, you can create a new string from an existing string. For example:

```
let str = 'JavaScript';
str = str + ' String';
```

In this example:

- First, declare the `str` variable and initialize it to a string of `'JavaScript'`.
- Second, use the `+` operator to combine `'JavaScript'` with `' String'` to make its value as `'Javascript String'`.

Behind the scene, the JavaScript engine creates a new string that holds the new string `'JavaScript String'` and destroys the original strings `'JavaScript'` and `' String'`.

The following example attempts to change the first character of the string JavaScript:

```
let s = 'JavaScript';
s[0] = 'j';
console.log(s)
```

The output is:

```
'JavaScript'
```

But not:

```
'javaScript'
```

The boolean type

The `boolean` type has two literal values: `true` and `false` in lowercase. The following example declares two variables that hold the boolean values.

```
let inProgress = true;
let completed = false;

console.log(typeof completed); // boolean
```

JavaScript allows values of other types to be converted into boolean values of `true` or `false` .

To convert a value of another data type into a boolean value, you use the `Boolean()` function. The following table shows the conversion rules:

Type	true	false
string	non-empty string	empty string
number	non-zero number and Infinity	0, NaN
object	non-null object	null
undefined		undefined

For example:

```
console.log(Boolean('Hi'));// true
console.log(Boolean(''));   // false

console.log(Boolean(20));   // true
console.log(Boolean(Infinity));   // true
console.log(Boolean(0));   // false
console.log(Boolean({foo: 100}));  // true on non-empty object
console.log(Boolean(null));// false
```

The symbol type

JavaScript added a primitive type in ES6: the `symbol`. Different from other primitive types, the `symbol` type does not have a literal form.

To create a symbol, you call the `Symbol` function as follows:

```
let s1 = Symbol();
```

The `Symbol` function creates a new unique value every time you call it.

```
console.log(Symbol() == Symbol()); // false
```

Note that you'll learn more about symbols in the symbol tutorial.

The bigint type

The `bigint` type represents the whole numbers that are larger than $2^{53} – 1$. To form a `bigint` literal number, you append the letter `n` at the end of the number:

```
let pageView = 9007199254740991n;
console.log(typeof(pageView)); // 'bigint'
```

And you'll learn more about the `bigint` type here.

The object type

In JavaScript, an object is a collection of properties, where each property is defined as a key-value pair.

The following example defines an empty object using the object literal syntax:

```
let emptyObject = {};
```

The following example defines the `person` object with two properties: `firstName` and `lastName` .

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

A property name of an object can be any string. You can use quotes around the property name if it is not a valid identifier.

For example, if the person object has a property `first-name` , you must place it in the quotes such as `"first-name"` .

A property of an object can hold an object. For example:

```
let contact = {
    firstName: 'John',
    lastName: 'Doe',
    email: 'john.doe@example.com',
    phone: '(408)-555-9999',
    address: {
        building: '4000',
        street: 'North 1st street',
        city: 'San Jose',
        state: 'CA',
        country: 'USA'
    }
}
```

The `contact` object has the `firstName` , `lastName` , `email` , `phone` , and `address` properties.

The `address` property itself holds an object that has `building` , `street` , `city` , `state` , and `country` properties.

To access a object’s property, you can use

- The dot notation (`.`)
- The array-like notation (`[]`).

The following example uses the dot notation (`.`) to access the `firstName` and `lastName` properties of the `contact` object.

```
console.log(contact.firstName);
console.log(contact.lastName);
```

If you reference a property that does not exist, you’ll get an `undefined` value. For example:

```
console.log(contact.age); // undefined
```

The following example uses the array-like notation to access the `email` and `phone` properties of the `contact` object.

```
console.log(contact['phone']); // '(408)-555-9999'
console.log(contact['email']); // 'john.doe@example.com'
```

Summary

- JavaScript has the primitive types: `number` , `string` , `boolean` , `null` , `undefined` , `symbol` and `bigint` and a complex type: `object` .

JavaScript Arithmetic Operators

Summary: in this tutorial, you will learn how to use JavaScript arithmetic operators to perform arithmetic calculations.

Introduction to the JavaScript Arithmetic Operators

JavaScript supports the following standard arithmetic operators:

Operator	**Sign**
Addition	+
Subtraction	-
Multiplication	*
Division	/

An arithmetic operator accepts numerical values as operands and returns a single numerical value. The numerical values can be literals or variables.

Addition operator (+)

The addition operator returns the sum of two values. For example, the following uses the addition operator to calculate the sum of two numbers:

```
let sum = 10 + 20;
console.log(sum); // 30
```

Also, you can use the addition operator with two variables. For example:

```
let netPrice    = 9.99,
    shippingFee = 1.99;
let grossPrice  = netPrice + shippingFee;

console.log(grossPrice);
```

Output:

```
11.98
```

If either value is a string, the addition operator uses the following rules:

- If both values are strings, it concatenates the second string to the first one.
- If one value is a string, it implicitly converts the numeric value into a string and concatenates two strings.

For example, the following uses the addition operator to concatenate two strings:

```
let x = '10',
    y = '20';
let result = x + y;

console.log(result);
```

Output:

```
1020
```

The following example shows how to use the addition operator to calculate the sum of a number and a string:

```
let result = 10 + '20';

console.log(result);
```

Output:

```
1020
```

In this example, JavaScript converts the number `10` into a string `'10'` and concatenates the second string `'20'` to it.

The following table shows the result when using the addition operator with special numbers:

First Value	Second Value	Result	Explanation
NaN		NaN	If either value is NaN, the result is NaN
Infinity	Infinity	Infinity	Infinity + Infinity = Infinity
-Infinity	-Infinity	-Infinity	-Infinity + (-Infinity) = – Infinity
Infinity	-Infinity	NaN	Infinity + -Infinity = NaN
+0	+0	+0	+0 + (+0) = +0
-0	+0	+0	-0 + (+0) = +0
-0	-0	-0	-0 + (-0) = -0

Subtraction operator (-)

The subtraction operator (`-`) subtracts one number from another. For example:

```
let result = 30 - 10;
console.log(result); // 20
```

If a value is a string, a boolean, null, or undefined, the JavaScript engine will:

- First, convert the value to a number using the `Number()` function.
- Second, perform the subtraction.

The following table shows how to use the subtraction operator with special values:

First Value	**Second Value**	**Result**	**Explanation**
NaN		NaN	If either value is NaN, the result is NaN
Infinity	Infinity	NaN	Infinity – Infinity = NaN
-Infinity	-Infinity	-Infinity	-Infinity – (-Infinity) = NaN
Infinity	-Infinity	Infinity	Infinity
+0	+0	+0	+0 – (+0) = 0
+0	-0	-0	+0 – (-0) = 0
-0	-0	+0	-0 – (-0) = 0

Multiplication operator (*)

JavaScript uses the asterisk (*) to represent the multiplication operator. The multiplication operator multiplies two numbers and returns a single value. For example:

```
let result = 2 * 3;
console.log(result);
```

Output:

```
6
```

If either value is not a number, the JavaScript engine implicitly converts it into a number using the `Number()` function and perform the multiplication. For example:

```
let result = '5' * 2;

console.log(result);
```

Output:

```
10
```

The following table shows how the multiply operator behaves with special values:

First Value	Second Value	Result	Explanation
NaN		NaN	If either value is NaN, the result is NaN
Infinity	0	NaN	Infinity * 0 = NaN
Infinity	Positive number	Infinity	-Infinity * 100 = -Infinity
Infinity	Negative number	-Infinity	Infinity * (-100) = -Infinity
Infinity	Infinity	Infinity	Infinity * Infinity = Infinity

Divide operator (/)

Javascript uses the slash (`/`) character to represent the divide operator. The divide operator divides the first value by the second one. For example:

```
let result = 20 / 10;

console.log(result); // 2
```

If either value is not a number, the JavaScript engine converts it into a number for division. For example:

```
let result = '20' / 2;
console.log(result); // 10;
```

The following table shows the divide operators' behavior when applying to special values:

First Value	Second Value	Result	Explanation
NaN		NaN	If either value is NaN, the result is NaN
A number	0	Infinity	1/0 = Infinity
Infinity	Infinity	NaN	Infinity / Infinity = NaN
0	0	NaN	0/0 = NaN
Infinity	A positive number	Infinity	Infinity / 2 = Infinity
Infinity	A negative number	-Infinity	Infinity / -2 = -Infinity

Using JavaScript arithmetic operators with objects

If a value is an object, the JavaScript engine will call the `valueOf()` method of the object to get the value for calculation. For example:

```
let energy = {
  valueOf() {
    return 100;
  },
};
```

```
let currentEnergy = energy - 10;
console.log(currentEnergy);

currentEnergy = energy + 100;
console.log(currentEnergy);

currentEnergy = energy / 2;
console.log(currentEnergy);

currentEnergy = energy * 1.5;
console.log(currentEnergy);
```

Output:

```
90
200
50
150
```

If the object doesn't have the `valueOf()` method but has the `toString()` method, the JavaScript engine will call the `toString()` method to get the value for calculation. For example:

```
let energy = {
  toString() {
    return 50;
  },
};

let currentEnergy = energy - 10;
console.log(currentEnergy);

currentEnergy = energy + 100;
console.log(currentEnergy);

currentEnergy = energy / 2;
```

```
console.log(currentEnergy);

currentEnergy = energy * 1.5;
console.log(currentEnergy);
```

Output:

```
40
150
25
75
```

Summary

- Use the JavaScript arithmetic operators including addition (`+`), subtraction (`-`), multiply (`*`) and divide (`/`) to perform arithmetic operations.

JavaScript Assignment Operators

Summary: in this tutorial, you will learn how to use JavaScript assignment operators to assign a value to a variable.

Introduction to JavaScript assignment operators

An assignment operator (`=`) assigns a value to a variable. The syntax of the assignment operator is as follows:

```
let a = b;
```

In this syntax, JavaScript evaluates the expression `b` first and assigns the result to the variable `a` .

The following example declares the `counter` variable and initializes its value to zero:

```
let counter = 0;
```

The following example increases the `counter` variable by one and assigns the result to the `counter` variable:

```
let counter = 0;
counter = counter + 1;
```

When evaluating the second statement, JavaScript evaluates the expression on the right-hand first (`counter + 1`) and assigns the result to the `counter` variable. After the second assignment, the `counter` variable is `1` .

To make the code more concise, you can use the `+=` operator like this:

```
let counter = 0;
counter += 1;
```

In this syntax, you don't have to repeat the `counter` variable twice in the assignment.

The following table illustrates assignment operators that are shorthand for another operator and the assignment:

Operator	Meaning	Description
`a = b`	`a = b`	Assigns the value of b to a.
`a += b`	`a = a + b`	Assigns the result of a plus b to a.
`a -= b`	`a = a - b`	Assigns the result of a minus b to a.
`a *= b`	`a = a * b`	Assigns the result of a times b to a.
`a /= b`	`a = a / b`	Assigns the result of a divided by b to a.
`a %= b`	`a = a % b`	Assigns the result of a modulo b to a.
`a &=b`	`a = a & b`	Assigns the result of a AND b to a.
`a |=b`	`a = a | b`	Assigns the result of a OR b to a.
`a ^=b`	`a = a ^ b`	Assigns the result of a XOR b to a.

Operator	Meaning	Description
`a <<= b`	`a = a << b`	Assigns the result of a shifted left by `b` to a.
`a >>= b`	`a = a >> b`	Assigns the result of a shifted right (sign preserved) by `b` to a.
`a >>>= b`	`a = a >>> b`	Assigns the result of a shifted right by `b` to a.

Chaining JavaScript assignment operators

If you want to assign a single value to multiple variables, you can chain the assignment operators. For example:

```
let a = 10, b = 20, c = 30;
a = b = c; // all variables are 30
```

In this example, JavaScript evaluates from right to left. Therefore, it does the following:

```
let a = 10, b = 20, c = 30;

b = c; // b is 30
a = b; // a is also 30
```

Summary

- Use the assignment operator (`=`) to assign a value to a variable.
- Chain the assignment operators if you want to assign a single value to multiple variables.

JavaScript Comparison Operators

Summary: in this tutorial, you will learn how to use JavaScript comparison operators to compare two values.

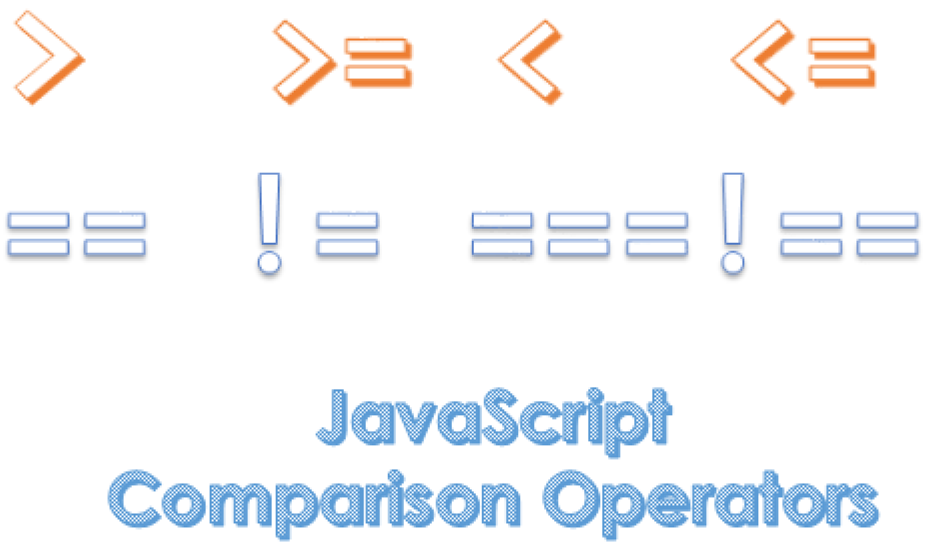
Introduction to JavaScript comparison operators

To compare two values, you use a comparison operator. The following table shows the comparison operators in JavaScript:

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

A comparison operator returns a Boolean value indicating whether the comparison is true or not. See the following example:

```
let r1 = 20 > 10; // true
let r2 = 20 < 10; // false
let r3 = 10 == 10; // true
```



A comparison operator takes two values. If the types of values are not comparable, the comparison operator converts them into values of comparable types according to specific rules.

Compare numbers

If values are numbers, the comparison operators perform a numeric comparison. For example:

```
let a = 10,
    b = 20;

console.log(a >= b);  // false
console.log(a == 10); // true
```

This example is straightforward. The variable `a` is `10`, `b` is `20`. The expression `a >= b` expression returns `false` and the expression `a == 10` expression returns `true`.

Compare strings

If the operands are strings, JavaScript compares the character codes numerically one by one in the string.

```
let name1 = 'alice',
    name2 = 'bob';

let result = name1 < name2;
console.log(result); // true
console.log(name1 == 'alice'); // true
```

Because JavaScript compares the character codes in the strings numerically, you may receive an unexpected result, for example:

```
let f1 = 'apple',
    f2 = 'Banana';
let result = f2 < f1;
console.log(result); // true
```

In this example, `f2` is less than `f1` because the letter `B` has the character code `66` while the letter `a` has the character code `97`.

To fix this, you need to:

- First, convert the strings into a common format, either lowercase or uppercase
- Second, compare the converted values

For example:

```
let f1 = 'apple',
    f2 = 'Banana';

let result = f2.toLowerCase() < f1.toLowerCase();
console.log(result); // false
```

Note that the `toLowerCase()` is a method of the String object that converts the string to lowercase.

Compare a number with a value of another type

If one value is a number and the other is not, the comparison operator will convert the non-numeric value into a number and compare them numerically. For example:

```
console.log(10 < '20'); // true
```

In this example, the comparison operator converts the string `'20'` into the number `20` and compares with the number 10. Here is an example:

```
console.log(10 == '10'); // true
```

In this example, the comparison operator converts the string `'10'` into the number `10` and compares them numerically.

Compare an object with a non-object

If a value is an object, the `valueOf()` method of that object is called to return the value for comparison. If the object doesn't have the `valueOf()` method, the `toString()` method is called instead. For example:

```
let apple = {
  valueOf: function () {
    return 10;
  },
};

let orange = {
  toString: function () {
    return '20';
  },
};
console.log(apple > 10); // false
console.log(orange == 20); // true
```

In this first comparison, the `apple` object has the `valueOf()` method that returns `10` . Therefore, the comparison operator uses the number 10 for comparison.

In the second comparison, JavaScript first calls the `valueOf()` method. However, the `orange` object doesn't have the `valueOf()` method. So JavaScript calls the `toString()` method to get the returned value of `20` for comparison.

Compare a Boolean with another value

If a value is a Boolean value, JavaScript converts it to a number and compares the converted value with the other value; `true` is converted to `1` and `false` is converted to `0` . For example:

```
console.log(true > 0); // true
console.log(false < 1); // true
console.log(true > false); // true
console.log(false > true); // false
console.log(true >= true); // true
console.log(true <= true); // true
console.log(false <= false); // true
console.log(false >= false); // true
```

In addition to the above rules, the equal (`==`) and not equal (`!=`) operators also have the following rules.

Compare `null` and `undefined`

In JavaScript, `null` equals `undefined` . It means that the following expression returns `true` .

```
console.log(null == undefined); // true
```

Compare NaN with other values

If either value is `NaN` , then the equal operator(`==`) returns `false` .

```
console.log(NaN == 1); // false
```

Even

```
console.log(NaN == NaN); // false
```

The not-equal (`!=`) operator returns `true` when comparing the `NaN` with another value:

```
console.log(NaN != 1); // true
```

And also

```
console.log(NaN != NaN); // true
```

Strict equal (===) and not strict equal (!==)

Besides the comparison operators above, JavaScript provides strict equal (`===`) and not strict equal (`!==`) operators.

Operator	**Meaning**
===	strict equal
!==	not strict equal

The strict equal and not strict equal operators behave like the equal and not equal operators except that they don't convert the operand before comparison. See the following example:

```
console.log("10" == 10); // true
console.log("10" === 10); // false
```

In the first comparison, since we use the equality operator, JavaScript converts the string into a number and performs the comparison.

However, in the second comparison, we use the strict equal operator (`===`), JavaScript doesn't convert the string before comparison, therefore the result is `false` .

In this tutorial, you have learned how to use the JavaScript comparison operators to compare values.

An Introduction to JavaScript Logical Operators

Summary: in this tutorial, you will learn how to use the JavaScript logical operators including the logical NOT operator(`!`), the logical AND operator (`&&`) and the logical OR operator (`||`).

The logical operators are important in JavaScript because they allow you to compare variables and do something based on the result of that comparison.

For example, if the result of the comparison is `true` , you can run a block of code; if it's `false` , you can execute another code block.

JavaScript provides three logical operators:

- ! (Logical NOT)
- || (Logical OR)
- && (Logical AND)

1) The Logical NOT operator (!)

JavaScript uses an exclamation point `!` to represent the logical NOT operator. The `!` operator can be applied to a single value of any type, not just a Boolean value.

When you apply the `!` operator to a boolean value, the `!` returns `true` if the value is `false` and vice versa. For example:

```
let eligible = false,
    required = true;

console.log(!eligible);
console.log(!required);
```

Output:

```
true
false
```

In this example, the `eligible` is `true` so `!eligible` returns `false`. And since the `required` is `true`, the `!required` returns `false`.

When you apply the `!` operator to a non-Boolean value. The `!` operator first converts the value to a boolean value and then negates it.

The following example shows how to use the `!` operator:

```
!a
```

The logical `!` operator works based on the following rules:

- If `a` is `undefined`, the result is `true`.
- If `a` is `null`, the result is `true`.
- If `a` is a number other than `0`, the result is `false`.
- If `a` is `NaN`, the result is `true`.
- If a is an object, the result is false.

- If a is an empty string, the result is true. In the case `a` is a non-empty string, the result is `false`

The following demonstrates the results of the logical `!` operator when applying to a non-boolean value:

```
console.log(!undefined); // true
console.log(!null); // true
console.log(!20); //false
console.log(!0); //true
console.log(!NaN); //true
console.log(!{}); // false
console.log(!''); //true
console.log(!'OK'); //false
console.log(!false); //true
console.log(!true); //false
```

Double-negation (`!!`)

Sometimes, you may see the double negation (`!!`) in the code. The `!!` uses the logical NOT operator (`!`) twice to convert a value to its real boolean value.

The result is the same as using the Boolean() function. For example:

```
let counter = 10;
console.log(!!counter); // true
```

The first `!` operator negates the Boolean value of the `counter` variable. If the `counter` is `true` , then the `!` operator makes it false and vice versa.

The second `!` operator negates that result of the first `!` operator and returns the real boolean value of the `counter` variable.

2) The Logical AND operator (&&)

JavaScript uses the double ampersand (`&&`) to represent the logical AND operator. The following expression uses the `&&` operator:

```
let result = a && b;
```

If `a` can be converted to `true`, the `&&` operator returns the `b`; otherwise, it returns the `a`. In fact, this rule is applied to all boolean values.

The following truth table illustrates the result of the `&&` operator when it is applied to two Boolean values:

a	**b**	**a && b**
true	true	true
true	false	false
false	true	false
false	false	false

The result of the `&&` operator is true only if both values are `true`, otherwise, it is `false`. For example:

```
let eligible = false,
    required = true;

console.log(eligible && required); // false
```

In this example, the `eligible` is `false`, therefore, the value of the expression `eligible && required` is `false`.

See the following example:

```
let eligible = true,
    required = true;

console.log(eligible && required); // true
```

In this example, both `eligible` and `required` are `true`, therefore, the value of the expression `eligible && required` is `true`.

Short-circuit evaluation

The `&&` operator is short-circuited. It means that the `&&` operator evaluates the second value only if the first one doesn't suffice to determine the value of an expression. For example:

```
let b = true;
let result = b && (1 / 0);
console.log(result);
```

Output:

```
Infinity
```

In this example, `b` is `true` therefore the `&&` operator could not determine the result without further evaluating the second expression (`1/0`).

The result is `Infinity` which is the result of the expression (`1/0`). However:

```
let b = false;
let result = b && (1 / 0);
console.log(result);
```

Output:

```
false
```

In this case, `b` is `false`, the `&&` operator doesn't need to evaluate the second expression because it can determine the final result as `false` based value of the first value.

The chain of && operators

The following expression uses multiple `&&` operators:

```
let result = value1 && value2 && value3;
```

The `&&` operator carries the following:

- Evaluates values from left to right.
- For each value, convert it to a boolean. If the result is `false`, stops and returns the original value.
- If all values are truthy values, return the last value.

In other words, The `&&` operator returns the first falsy value or the last value if none were found.

> If a value can be converted to `true`, it is so-called a truthy value. If a value can be converted to `false`, it is a so-called falsy value.

3) The Logical OR operator (||)

JavaScript uses the double pipe `||` to represent the logical OR operator. You can apply the `||` operator to two values of any type:

```
let result = a || b;
```

If `a` can be converted to `true`, returns `a`; else, returns `b`. This rule is also applied to boolean values.

The following truth table illustrates the result of the `||` operator based on the value of the operands:

a	**b**	**a || b**
true	true	true
true	false	true

a	b	a || b
false	true	true
false	false	false

The `||` operator returns `false` if both values evaluate to `false`. In case either value is `true`, the `||` operator returns `true`. For example:

```
let eligible = true,
    required = false;

console.log(eligible || required); // true
```

See another example:

```
let eligible = false,
    required = false;

console.log(eligible || required); // false
```

In this example, the expression `eligible || required` returns `false` because both values are `false`.

The || operator is also short-circuited

Similar to the `&&` operator, the `||` operator is short-circuited. It means that if the first value evaluates to `true`, the `&&` operator doesn't evaluate the second one.

The chain of || operators

The following example shows how to use multiple || operators in an expression:

```
let result = value1 || value2 || value3;
```

The `||` operator does the following:

- Evaluates values from left to right.
- For each value, converts it to a boolean value. If the result of the conversion is `true`, stops and returns the value.
- If all values have been evaluated to `false`, returns the last value.

In other words, the chain of the `||` operators returns the first truthy value or the last one if no truthy value was found.

Logical operator precedence

When you mix logical operators in an expression, the JavaScript engine evaluates the operators based on a specified order. And this order is called the *operator precedence*.

In other words, the operator precedence is the order of evaluation of logical operators in an expression.

The precedence of the logical operator is in the following order from the highest to the lowest:

1. Logical NOT (!)
2. Logical AND (&&)
3. Logical OR (||)

Summary

- The NOT operator (`!`) negates a boolean value. The (`!!`) converts a value into its real boolean value.
- The AND operator (`&&`) is applied to two Boolean values and returns true if both values are true.
- The OR operator (`||`) is applied to two Boolean values and returns `true` if one of the operands is `true`.
- Both `&&` and `||` operator are short-circuited. They can be also applied to non-Boolean values.
- The logical operator precedence from the highest to the lowest is `!`, `&&` and `||`.

JavaScript Remainder Operator

Summary: in this tutorial, you'll learn about the JavaScript remainder operator (`%`) to get the remainder of a number divided by another number.

Introduction to the JavaScript remainder operator

JavaScript uses the `%` to represent the remainder operator. The remainder operator returns the remainder left over when one value is divided by another value.

Here's the syntax of the remainder operator:

```
dividend % divisor
```

The following shows the equation for the remainder:

```
dividend = divisor * quotient + remainder
where |remainder| < |divisor|
```

In this equation, the `dividend` , `divisor` , `quotient` , and `remainder` are all integers. The sign of the `remainder` is the same as the sign of the `dividend` .

The sign of the `remainder` is the same as the sign of the `dividend` .

JavaScript remainder operator examples

Let's take some examples of using the JavaScript remainder operator.

1) Using the remainder operator with a positive dividend example

The following example shows how to use the remainder operator with a positive dividend:

```
let remainder = 5 % -2;
console.log(remainder); // 1

remainder = 5 % 2;
console.log(remainder); // 1
```

2) Using the remainder operator with a negative dividend example

The following example uses the remainder operator with a negative dividend:

```
let remainder = -5 % 3;
console.log(remainder); // -2

remainder = -5 % -3;
console.log(remainder); // -2
```

3) Using the remainder operator special values

If a dividend is an `Infinity` and a divisor is a finite number, the remainder is `NaN` . For example:

```
let remainder = Infinity % 2;
console.log(remainder); // NaN
```

If a dividend is a finite number and a divisor is zero, the remainder is `NaN` :

```
let remainder = 10 % 0;
```

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

If both dividend and divisor are `Infinity` , the remainder is `NaN` :

```
let remainder = Infinity % Infinity;
console.log(remainder); // NaN
```

If a dividend is a finite number and the divisor is an `Infinity` , the remainder is the dividend. For example:

```
let remainder = 10 % Infinity;
console.log(remainder); // 10
```

If the dividend is zero and the divisor is non-zero, the remainder is zero:

```
let remainder = 0 % 10;
console.log(remainder); // 0
```

If either dividend or divisor is not a number, it's converted to a number using the `Number()` function and applied the above rules. For example:

```
let remainder = '10' % 3;
console.log(remainder); // 1
```

Using the remainder operator to check if a number is an odd number

To check if a number is an odd number, you use the remainder operator (`%`) like the following example:

```
let num = 13;
let isOdd = num % 2;
```

In this example, if the `num` is an odd number, the remainder is one. But if the `num` is an even number, the remainder is zero.

Later, you'll learn how to define a function that returns `true` if a number is odd or `false` otherwise like this:

```
function isOdd(num) {
  return num % 2;
}
```

Or using an arrow function in ES6:

```
const isOdd = (num) => num % 2;
```

Remainder vs Modulo operator

In JavaScript, the remainder operator (%) is not the modulo operator.

If you have been working with Python, you may find the `%` represents the modulo operator in this language. However, this is not the case in JavaScript.

To get a modulo in JavaScript, you use the following expression:

```
((dividend % divisor) + divisor) % divisor
```

Or wrap it in a function:

```
const mod = (dividend, divisor) => ((dividend % divisor) + divisor) % divisor;
```

If the division and divisor have the same sign, the remainder and modulo operators return the same result. Otherwise, they return different results.

For example:

```
const mod = (dividend, divisor) => ((dividend % divisor) + divisor) % divisor;

// dividen and divisor have the same sign
console.log('remainder:', 5 % 3); // 2
console.log('modulo:', mod(5, 3)); // 2

// dividen and divisor have the different signs
console.log('remainder:', -5 % 3); // -2
console.log('modulo:', mod(-5, 3)); // 1
```

Output:

```
remainder: 2
modulo: 2
remainder: -2
modulo: 1
```

Summary

- Use the JavaScript remainder operator (`%`) get the remainder of a value divided by another value.