

JavaScript Notes

1. JavaScript Introduction

- JavaScript Can Change HTML Content

```
document.getElementById("demo").innerHTML = "Hello JavaScript";  
document.getElementById('demo').innerHTML = 'Hello JavaScript';  
JavaScript accepts both double and single quotes:
```

- JavaScript Can Change HTML Attribute Values

```
<button onclick="document.getElementById('myImage').src='pic_bulbon.gif'">Turn on the  
light</button>
```

- JavaScript Can Change HTML Styles (CSS)

```
document.getElementById("demo").style.fontSize = "35px";
```

- JavaScript Can Hide HTML Elements

```
document.getElementById("demo").style.display = "none";
```

- JavaScript Can Show HTML Element

```
document.getElementById("demo").style.display = "block";
```

2. JavaScript Where To

In HTML, JavaScript code is inserted between `<script>` and `</script>` tags.

Example

```
<script>  
document.getElementById("demo").innerHTML = "My First JavaScript";  
</script>
```

Old JavaScript examples may use a type attribute: `<script type="text/javascript">`.
The type attribute is not required. JavaScript is the default scripting language in HTML.

JavaScript in <head> or <body>

You can place any number of scripts in an HTML document.

Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page, or in both.

JavaScript in <head>

In this example, a JavaScript `function` is placed in the `<head>` section of an HTML page.

The function is invoked (called) when a button is clicked:

Example

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body><h2>Demo JavaScript in Head</h2>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>
</body>
</html>
```

JavaScript in <body>

In this example, a JavaScript `function` is placed in the `<body>` section of an HTML page.

The function is invoked (called) when a button is clicked:

Example

```
<!DOCTYPE html>
<html>
<body>
<h2>Demo JavaScript in Body</h2>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</body>
</html>
```

Placing scripts at the bottom of the `<body>` element improves the display speed, because script interpretation slows down the display.

External JavaScript

Scripts can also be placed in external files:

External file: myScript.js

```
function myFunction() {  
    document.getElementById("demo").innerHTML = "Paragraph changed.";  
}
```

External scripts are practical when the same code is used in many different web pages.

JavaScript files have the file extension **.js**.

To use an external script, put the name of the script file in the **src** (source) attribute of a **<script>** tag:

Example

```
<script src="myScript.js"></script>
```

You can place an external script reference in **<head>** or **<body>** as you like.

The script will behave as if it was located exactly where the **<script>** tag is located.

External scripts cannot contain **<script>** tags.

External JavaScript Advantages

Placing scripts in external files has some advantages:

- It separates HTML and code
- It makes HTML and JavaScript easier to read and maintain
- Cached JavaScript files can speed up page loads

To add several script files to one page - use several script tags:

Example

```
<script src="myScript1.js"></script>  
<script src="myScript2.js"></script>
```

External References

An external script can be referenced in 3 different ways:

- With a full URL (a full web address)
- With a file path (like /js/)
- Without any path

This example uses a **full URL** to link to myScript.js:

Example

```
<script src="https://www.w3schools.com/js/myScript.js"></script>
```

This example uses a **file path** to link to myScript.js:

Example

```
<script src="/js/myScript.js"></script>
```

This example uses no path to link to myScript.js:

Example

```
<script src="myScript.js"></script>
```

3. JavaScript Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

Using `innerHTML`

To access an HTML element, JavaScript can use the `document.getElementById(id)` method.

The `id` attribute defines the HTML element. The `innerHTML` property defines the HTML content:

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My First Paragraph</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

Changing the `innerHTML` property of an HTML element is a common way to display data in HTML.

Using document.write()

For testing purposes, it is convenient to use `document.write()`:

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
document.write(5 + 6);
</script>
</body>
</html>
```

Using `document.write()` after an HTML document is loaded, will **delete all existing HTML**:

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<button type="button" onclick="document.write(5 + 6)">Try it</button>
</body>
</html>
```

The `document.write()` method should only be used for testing.

Using window.alert()

You can use an alert box to display data:

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
window.alert(5 + 6);
</script>
</body>
</html>
```

You can skip the `window` keyword.

In JavaScript, the `window` object is the global scope object. This means that variables, properties, and methods by default belong to the `window` object. This also means that specifying the `window` keyword is optional:

Using console.log()

For debugging purposes, you can call the `console.log()` method in the browser to display data.

Example

```
<!DOCTYPE html>
<html>
<body>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

JavaScript Print

JavaScript does not have any print object or print methods.

You cannot access output devices from JavaScript.

The only exception is that you can call the `window.print()` method in the browser to print the content of the current window.

Example

```
<!DOCTYPE html>
<html>
<body>
<button onclick="window.print()">Print this page</button>
</body>
</html>
```

4. JavaScript Statements

Example

```
let x, y, z;      // Statement 1
x = 5;           // Statement 2
y = 6;           // Statement 3
z = x + y;       // Statement 4
```

JavaScript Programs

A **computer program** is a list of "instructions" to be "executed" by a computer. In a programming language, these programming instructions are called **statements**. A **JavaScript program** is a list of programming **statements**.

In HTML, JavaScript programs are executed by the web browser.

JavaScript Statements

JavaScript statements are composed of: Values, Operators, Expressions, Keywords, and Comments. This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

Example

```
document.getElementById("demo").innerHTML = "Hello Dolly.;"
```

Most JavaScript programs contain many JavaScript statements. The statements are executed, one by one, in the same order as they are written.

JavaScript programs (and JavaScript statements) are often called JavaScript code.

Semicolons ;

Semicolons separate JavaScript statements.

Add a semicolon at the end of each executable statement:

Examples

```
let a, b, c; // Declare 3 variables
a = 5;        // Assign the value 5 to a
b = 6;        // Assign the value 6 to b
c = a + b;    // Assign the sum of a and b to c
```

When separated by semicolons, multiple statements on one line are allowed:

```
a = 5; b = 6; c = a + b;
```

On the web, you might see examples without semicolons.

Ending statements with semicolon is not required, but highly recommended.

JavaScript White Space

JavaScript ignores multiple spaces. You can add white space to your script to make it more readable.

The following lines are equivalent:

```
let person = "Hege";
let person="Hege";
```

A good practice is to put spaces around operators (= + - * /):

```
let x = y + z;
```

JavaScript Line Length and Line Breaks

For best readability, programmers often like to avoid code lines longer than 80 characters.

If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

Example

```
document.getElementById("demo").innerHTML =
"Hello Dolly!";
```

JavaScript Code Blocks

JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.

The purpose of code blocks is to define statements to be executed together.

One place you will find statements grouped together in blocks, is in JavaScript functions:

Example

```
function myFunction() {
  document.getElementById("demo1").innerHTML = "Hello Dolly!";
  document.getElementById("demo2").innerHTML = "How are you?";
}
```

In this tutorial we use 2 spaces of indentation for code blocks.

JavaScript Keywords

JavaScript statements often start with a **keyword** to identify the JavaScript action to be performed. Our [Reserved Words Reference](#) lists all JavaScript keywords.

Here is a list of some of the keywords you will learn about in this tutorial:

Keyword	Description
var	Declares a variable
let	Declares a block variable
const	Declares a block constant
if	Marks a block of statements to be executed on a condition
switch	Marks a block of statements to be executed in different cases
for	Marks a block of statements to be executed in a loop
function	Declares a function
return	Exits a function
try	Implements error handling to a block of statements

JavaScript keywords are reserved words. Reserved words cannot be used as names for variables.

5. JavaScript Syntax

JavaScript syntax is the set of rules, how JavaScript programs are constructed:

```
// How to create variables:  
var x;  
let y;  
// How to use variables:  
x = 5;  
y = 6;  
let z = x + y;
```

JavaScript Values

The JavaScript syntax defines two types of values:

- Fixed values
- Variable values

Fixed values are called **Literals**.

Variable values are called **Variables**.

JavaScript Literals

The two most important syntax rules for fixed values are:

1. **Numbers** are written with or without decimals:

10.50

1001

2. **Strings** are text, written within double or single quotes:

"John Doe"

'John Doe'

JavaScript Variables

In a programming language, **variables** are used to **store** data values.

JavaScript uses the keywords **var**, **let** and **const** to **declare** variables.

An **equal sign** is used to **assign values** to variables.

In this example, x is defined as a variable. Then, x is assigned (given) the value 6:

```
let x;  
x = 6;
```

JavaScript Operators

JavaScript uses **arithmetic operators** (+ - * /) to **compute** values:

```
(5 + 6) * 10
```

JavaScript uses an **assignment operator** (=) to **assign** values to variables:

```
let x, y;  
x = 5;  
y = 6;
```

JavaScript Expressions

An expression is a combination of values, variables, and operators, which computes to a value.

The computation is called an evaluation.

For example, 5 * 10 evaluates to 50:

```
5 * 10
```

Expressions can also contain variable values:

```
x * 10
```

The values can be of various types, such as numbers and strings.

For example, "John" + " " + "Doe", evaluates to "John Doe":

```
"John" + " " + "Doe"
```

JavaScript Keywords

JavaScript **keywords** are used to identify actions to be performed.

The `let` keyword tells the browser to create variables:

```
let x, y;  
x = 5 + 6;  
y = x * 10;
```

The `var` keyword also tells the browser to create variables:

```
var x, y;  
x = 5 + 6;  
y = x * 10;
```

In these examples, using `var` or `let` will produce the same result.

JavaScript Comments

Not all JavaScript statements are "executed".

Code after double slashes `//` or between `/*` and `*/` is treated as a **comment**.

Comments are ignored, and will not be executed:

```
let x = 5;    // I will be executed  
// x = 6;    I will NOT be executed
```

JavaScript Identifiers / Names

Identifiers are JavaScript names.

Identifiers are used to name variables and keywords, and functions.

The rules for legal names are the same in most programming languages.

A JavaScript name must begin with:

- A letter (A-Z or a-z)
- A dollar sign (\$)
- Or an underscore (_)

Subsequent characters may be letters, digits, underscores, or dollar signs.

Note

Numbers are not allowed as the first character in names.

This way JavaScript can easily distinguish identifiers from numbers.

JavaScript is Case Sensitive

All JavaScript identifiers are **case sensitive**.

The variables `lastName` and `lastname`, are two different variables:

```
let lastname, lastName;  
lastName = "Doe";  
lastname = "Peterson";
```

JavaScript does not interpret **LET** or **Let** as the keyword **let**.

JavaScript and Camel Case

Historically, programmers have used different ways of joining multiple words into one variable name:

Hyphens:

first-name, last-name, master-card, inter-city.

Hyphens are not allowed in JavaScript. They are reserved for subtractions.

Underscore:

first_name, last_name, master_card, inter_city.

Upper Camel Case (Pascal Case):

FirstName, LastName, MasterCard, InterCity.

Lower Camel Case:

JavaScript programmers tend to use camel case that starts with a lowercase letter:

firstName, lastName, masterCard, interCity.

JavaScript Character Set

JavaScript uses the **Unicode** character set.

Unicode covers (almost) all the characters, punctuations, and symbols in the world.

For a closer look, please study our [Complete Unicode Reference](#).

6. JavaScript Comments

JavaScript comments can be used to explain JavaScript code, and to make it more readable. JavaScript comments can also be used to prevent execution, when testing alternative code.

Single Line Comments

Single line comments start with `//`. Any text between `//` and the end of the line will be ignored by JavaScript (will not be executed).

This example uses a single-line comment before each code line:

Example

```
// Change heading:  
document.getElementById("myH").innerHTML = "My First Page";  
// Change paragraph:  
document.getElementById("myP").innerHTML = "My first paragraph.;"
```

This example uses a single line comment at the end of each line to explain the code:

Example

```
let x = 5;      // Declare x, give it the value of 5  
let y = x + 2; // Declare y, give it the value of x + 2
```

Multi-line Comments

Multi-line comments start with `/*` and end with `*/`.

Any text between `/*` and `*/` will be ignored by JavaScript.

This example uses a multi-line comment (a comment block) to explain the code:

Example

```
/*  
The code below will change  
the heading with id = "myH"  
and the paragraph with id = "myP"  
in my web page:  
*/  
document.getElementById("myH").innerHTML = "My First Page";  
document.getElementById("myP").innerHTML = "My first paragraph.;"
```

It is most common to use single line comments.

Block comments are often used for formal documentation.

7. JavaScript Variables

Variables are Containers for Storing Data

JavaScript Variables can be declared in 4 ways:

- Automatically
- Using `var`
- Using `let`
- Using `const`

In this first example, `x`, `y`, and `z` are undeclared variables.

They are automatically declared when first used:

Example

```
x = 5;  
y = 6;  
z = x + y;
```

Note

It is considered good programming practice to always declare variables before use.

From the examples you can guess:

- `x` stores the value 5
- `y` stores the value 6
- `z` stores the value 11

Example using var

```
var x = 5;  
var y = 6;  
var z = x + y;
```

Note

The `var` keyword was used in all JavaScript code from 1995 to 2015.

The `let` and `const` keywords were added to JavaScript in 2015.

The `var` keyword should only be used in code written for older browsers.

Example using let

```
let x = 5;  
let y = 6;  
let z = x + y;
```

Example using const

```
const x = 5;  
const y = 6;  
const z = x + y;
```

Mixed Example

```
const price1 = 5;  
const price2 = 6;  
let total = price1 + price2;
```

The two variables `price1` and `price2` are declared with the `const` keyword.

These are constant values and cannot be changed.

The variable `total` is declared with the `let` keyword.

The value `total` can be changed.

When to Use var, let, or const?

1. Always declare variables
2. Always use `const` if the value should not be changed
3. Always use `const` if the type should not be changed (Arrays and Objects)
4. Only use `let` if you can't use `const`
5. Only use `var` if you MUST support old browsers.

Just Like Algebra

Just like in algebra, variables hold values:

```
let x = 5;  
let y = 6;
```

Just like in algebra, variables are used in expressions:

```
let z = x + y;
```

From the example above, you can guess that the total is calculated to be 11.

JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with \$ and _ (but we will not use it in this tutorial).
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

Note

JavaScript identifiers are case-sensitive.

The Assignment Operator

In JavaScript, the equal sign (=) is an "assignment" operator, not an "equal to" operator.

This is different from algebra. The following does not make sense in algebra:

`x = x + 5`

In JavaScript, however, it makes perfect sense: it assigns the value of $x + 5$ to x.

(It calculates the value of $x + 5$ and puts the result into x. The value of x is incremented by 5.)

Note

The "equal to" operator is written like == in JavaScript.

JavaScript Data Types

JavaScript variables can hold numbers like 100 and text values like "John Doe".

In programming, text values are called text strings.

JavaScript can handle many types of data, but for now, just think of numbers and strings.

Strings are written inside double or single quotes. Numbers are written without quotes.

If you put a number in quotes, it will be treated as a text string.

Example

```
const pi = 3.14;
let person = "John Doe";
let answer = 'Yes I am!';
```

Declaring a JavaScript Variable

Creating a variable in JavaScript is called "declaring" a variable.

You declare a JavaScript variable with the `var` or the `let` keyword:

```
var carName;
```

or:

```
let carName;
```

After the declaration, the variable has no value (technically it is `undefined`).

To **assign** a value to the variable, use the equal sign:

```
carName = "Volvo";
```

You can also assign a value to the variable when you declare it:

```
let carName = "Volvo";
```

In the example below, we create a variable called `carName` and assign the value "Volvo" to it. Then we "output" the value inside an HTML paragraph with `id="demo"`:

Example

```
<p id="demo"></p>
<script>
let carName = "Volvo";
document.getElementById("demo").innerHTML = carName;
</script>
```

Note

It's a good programming practice to declare all variables at the beginning of a script.

One Statement, Many Variables

You can declare many variables in one statement.

Start the statement with `let` and separate the variables by **comma**:

Example

```
let person = "John Doe", carName = "Volvo", price = 200;
```

A declaration can span multiple lines:

Example

```
let person = "John Doe",
carName = "Volvo",
price = 200;
```

Value = undefined

In computer programs, variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input.

A variable declared without a value will have the value `undefined`.

The variable `carName` will have the value `undefined` after the execution of this statement:

Example

```
let carName;
```

Re-Declaring JavaScript Variables

If you re-declare a JavaScript variable declared with `var`, it will not lose its value.

The variable `carName` will still have the value "Volvo" after the execution of these statements:

Example

```
var carName = "Volvo";
var carName;
```

Note

You cannot re-declare a variable declared with `let` or `const`.

This will not work:

```
let carName = "Volvo";
let carName;
```

JavaScript Arithmetic

As with algebra, you can do arithmetic with JavaScript variables, using operators like `=` and `+`:

Example

```
let x = 5 + 2 + 3;
```

You can also add strings, but strings will be concatenated:

Example

```
let x = "John" + " " + "Doe";
```

Also try this:

Example

```
let x = "5" + 2 + 3;
```

Note

If you put a number in quotes, the rest of the numbers will be treated as strings, and concatenated.

Now try this:

Example

```
let x = 2 + 3 + "5";
```

JavaScript Dollar Sign \$

Since JavaScript treats a dollar sign as a letter, identifiers containing `$` are valid variable names:

Example

```
let $ = "Hello World";
let $$ = 2;
let $myMoney = 5;
```

Using the dollar sign is not very common in JavaScript, but professional programmers often use it as an alias for the main function in a JavaScript library.

In the JavaScript library jQuery, for instance, the main function `$` is used to select HTML elements. In jQuery `$(“p”);` means "select all p elements".

JavaScript Underscore (_)

Since JavaScript treats underscore as a letter, identifiers containing `_` are valid variable names:

Example

```
let _lastName = "Johnson";
let _x = 2;
let _100 = 5;
```

Using the underscore is not very common in JavaScript, but a convention among professional programmers is to use it as an alias for "private (hidden)" variables.

8. JavaScript Let

The `let` keyword was introduced in [ES6 \(2015\)](#)

Variables declared with `let` have **Block Scope**

Variables declared with `let` must be **Declared** before use

Variables declared with `let` cannot be **Redeclared** in the same scope

Block Scope

Before ES6 (2015), JavaScript did not have **Block Scope**.

JavaScript had **Global Scope** and **Function Scope**.

ES6 introduced the two new JavaScript keywords: `let` and `const`.

These two keywords provided **Block Scope** in JavaScript:

Example

Variables declared inside a `{ }` block cannot be accessed from outside the block:

```
{  
  let x = 2;  
}  
// x can NOT be used here
```

Global Scope

Variables declared with the `var` always have **Global Scope**.

Variables declared with the `var` keyword can NOT have block scope:

Example

Variables declared with `var` inside a `{ }` block can be accessed from outside the block:

```
{  
  var x = 2;  
}  
// x CAN be used here
```

Cannot be Redeclared

Variables defined with `let` **can not** be redeclared.

You can not accidentally redeclare a variable declared with `let`.

With `let` you **can not** do this:

```
let x = "John Doe";  
let x = 0;
```

Variables defined with `var` **can** be redeclared.

With `var` you **can** do this:

```
var x = "John Doe";  
var x = 0;
```

Redeclaring Variables

Redeclaring a variable using the `var` keyword can impose problems.

Redeclaring a variable inside a block will also redeclare the variable outside the block:

Example

```
var x = 10;  
// Here x is 10  
{  
var x = 2;  
// Here x is 2  
}  
// Here x is 2
```

Redeclaring a variable using the `let` keyword can solve this problem.

Redeclaring a variable inside a block will not redeclare the variable outside the block:

Example

```
let x = 10;  
// Here x is 10  
{  
let x = 2;  
// Here x is 2  
}  
  
// Here x is 10
```

Difference Between var, let and const

	Scope	Redeclare	Reassign	Hoisted	Binds this
var	No	Yes	Yes	Yes	Yes
let	Yes	No	Yes	No	No
const	Yes	No	No	No	No

What is Good?

`let` and `const` have **block scope**.

`let` and `const` can not be **redeclared**.

`let` and `const` must be **declared** before use.

`let` and `const` does **not bind** to `this`.

`let` and `const` are **not hoisted**.

What is Not Good?

`var` does not have to be declared.

`var` is hoisted.

`var` binds to this.

Browser Support

The `let` and `const` keywords are not supported in Internet Explorer 11 or earlier.

The following table defines the first browser versions with full support:

				
Chrome 49	Edge 12	Firefox 36	Safari 11	Opera 36
Mar, 2016	Jul, 2015	Jan, 2015	Sep, 2017	Mar, 2016

Redeclaring

Redeclaring a JavaScript variable with `var` is allowed anywhere in a program:

Example

```
var x = 2;  
// Now x is 2  
var x = 3;  
// Now x is 3
```

With `let`, redeclaring a variable in the same block is NOT allowed:

Example

```
var x = 2;    // Allowed  
let x = 3;    // Not allowed  
{  
let x = 2;    // Allowed  
let x = 3;    // Not allowed  
}  
{  
let x = 2;    // Allowed  
var x = 3;    // Not allowed  
}
```

Redeclaring a variable with `let`, in another block, IS allowed:

Example

```
let x = 2;    // Allowed  
{  
let x = 3;    // Allowed  
}  
{  
let x = 4;    // Allowed  
}
```

Let Hoisting

Variables defined with `var` are **hoisted** to the top and can be initialized at any time.

Meaning: You can use the variable before it is declared:

Example

This is OK:

```
carName = "Volvo";
var carName;
```

If you want to learn more about hoisting, study the chapter [JavaScript Hoisting](#).

Variables defined with `let` are also hoisted to the top of the block, but not initialized.

Meaning: Using a `let` variable before it is declared will result in a `ReferenceError`:

Example

```
carName = "Saab";
let carName = "Volvo";
```

9. JavaScript Const

The `const` keyword was introduced in [ES6 \(2015\)](#)

Variables defined with `const` cannot be **Redeclared**

Variables defined with `const` cannot be **Reassigned**

Variables defined with `const` have **Block Scope**

Cannot be Reassigned

A variable defined with the `const` keyword cannot be reassigned:

Example

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

Must be Assigned

JavaScript `const` variables must be assigned a value when they are declared:

Correct

```
const PI = 3.14159265359;
```

Incorrect

```
const PI;
PI = 3.14159265359;
```

When to use JavaScript const?

Always declare a variable with `const` when you know that the value should not be changed.

Use `const` when you declare:

- A new Array
- A new Object
- A new Function
- A new RegExp

Constant Objects and Arrays

The keyword `const` is a little misleading.

It does not define a constant value. It defines a constant reference to a value.

Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

But you CAN:

- Change the elements of constant array
- Change the properties of constant object

Constant Arrays

You can change the elements of a constant array:

Example

```
// You can create a constant array:  
const cars = ["Saab", "Volvo", "BMW"];  
  
// You can change an element:  
cars[0] = "Toyota";  
  
// You can add an element:  
cars.push("Audi");
```

But you can NOT reassign the array:

Example

```
const cars = ["Saab", "Volvo", "BMW"];  
  
cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

Constant Objects

You can change the properties of a constant object:

Example

```
// You can create a const object:  
const car = {type:"Fiat", model:"500", color:"white"};  
  
// You can change a property:  
car.color = "red";  
  
// You can add a property:  
car.owner = "Johnson";
```

But you can NOT reassign the object:

Example

```
const car = {type:"Fiat", model:"500", color:"white"};  
  
car = {type:"Volvo", model:"EX60", color:"red"}; // ERROR
```

Difference Between var, let and const

	Scope	Redeclare	Reassign	Hoisted	Binds this
var	No	Yes	Yes	Yes	Yes
let	Yes	No	Yes	No	No
const	Yes	No	No	No	No

What is Good?

`let` and `const` have **block scope**.

`let` and `const` can not be **redeclared**.

`let` and `const` must be **declared** before use.

`let` and `const` does **not bind** to `this`.

`let` and `const` are **not hoisted**.

What is Not Good?

`var` does not have to be declared.

`var` is hoisted.

`var` binds to this.

Browser Support

The `let` and `const` keywords are not supported in Internet Explorer 11 or earlier.

The following table defines the first browser versions with full support:

				
Chrome 49	Edge 12	Firefox 36	Safari 11	Opera 36
Mar, 2016	Jul, 2015	Jan, 2015	Sep, 2017	Mar, 2016

Block Scope

Declaring a variable with `const` is similar to `let` when it comes to **Block Scope**.

The `x` declared in the block, in this example, is not the same as the `x` declared outside the block:

Example

```
const x = 10;
// Here x is 10
{const x = 2; // Here x is 2} // Here x is 10
```

Redeclaring

Redeclaring a JavaScript `var` variable is allowed anywhere in a program:

Example

```
var x = 2;      // Allowed
var x = 3;      // Allowed
x = 4;          // Allowed
```

Redeclaring an existing `var` or `let` variable to `const`, in the same scope, is not allowed:

Example

```
var x = 2;      // Allowed
const x = 2;    // Not allowed

{
let x = 2;      // Allowed
const x = 2;    // Not allowed
}

{
const x = 2;    // Allowed
const x = 2;    // Not allowed
}
```

Reassigning an existing `const` variable, in the same scope, is not allowed:

Example

```
const x = 2;    // Allowed
x = 2;          // Not allowed
var x = 2;      // Not allowed
let x = 2;      // Not allowed
const x = 2;    // Not allowed

{
  const x = 2;  // Allowed
  x = 2;        // Not allowed
  var x = 2;    // Not allowed
  let x = 2;    // Not allowed
  const x = 2;  // Not allowed
}
```

Redeclaring a variable with `const`, in another scope, or in another block, is allowed:

Example

```
const x = 2;          // Allowed  
  
{  
  const x = 3;      // Allowed  
}  
  
{  
  const x = 4;      // Allowed  
}
```

Hoisting

Variables defined with `var` are **hoisted** to the top and can be initialized at any time.

Meaning: You can use the variable before it is declared:

Example

This is OK:

```
carName = "Volvo";  
var carName;
```

If you want to learn more about hoisting, study the chapter [JavaScript Hoisting](#).

Variables defined with `const` are also hoisted to the top, but not initialized.

Meaning: Using a `const` variable before it is declared will result in a [ReferenceError](#):

Example

```
alert (carName);  
const carName = "Volvo";
```

10. JavaScript Operators

Javascript operators are used to perform different types of mathematical and logical computations.

Examples:

The **Assignment Operator** **=** assigns values

The **Addition Operator** **+** adds values

The **Multiplication Operator** ***** multiplies values

The **Comparison Operator** **>** compares values

JavaScript Assignment

The **Assignment Operator** (**=**) assigns a value to a variable:

Assignment Examples

```
let x = 10;  
  
// Assign the value 5 to x  
let x = 5;  
// Assign the value 2 to y  
let y = 2;  
// Assign the value x + y to z:  
let z = x + y;
```

JavaScript Addition

The **Addition Operator** (**+**) adds numbers:

Adding

```
let x = 5;  
let y = 2;  
let z = x + y;
```

JavaScript Multiplication

The **Multiplication Operator** (*) multiplies numbers:

Multiplying

```
let x = 5;  
let y = 2;  
let z = x * y;
```

Types of JavaScript Operators

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- String Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators
- Type Operators

JavaScript Arithmetic Operators

Arithmetic Operators are used to perform arithmetic on numbers:

Arithmetic Operators Example

```
let a = 3;  
let x = (100 + 50) * a;
```

Operator	Description
+	Addition
-	Subtraction
*	Multiplication

**	Exponentiation (ES2016)
	x ** y produces the same result as <code>Math.pow(x,y)</code>
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Note

Arithmetic operators are fully described in the [JS Arithmetic](#) chapter.

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

The **Addition Assignment Operator** (`+=`) adds a value to a variable.

Assignment

```
let x = 10;
x += 5;
```

Operator	Example	Same As
=	<code>x = y</code>	<code>x = y</code>
<code>+=</code>	<code>x += y</code>	<code>x = x + y</code>

`-=` `x -= y` `x = x - y`

`*=` `x *= y` `x = x * y`

`/=` `x /= y` `x = x / y`

`%=` `x %= y` `x = x % y`

`**=` `x **= y` `x = x ** y`

Note

Assignment operators are fully described in the [JS Assignment](#) chapter.

JavaScript Comparison Operators

Operator Description

`==` equal to

`===` equal value and equal type

`!=` not equal

`!==` not equal value or not equal type

`>` greater than

< less than

>= greater than or equal to

<= less than or equal to

? ternary operator

Note

Comparison operators are fully described in the [JS Comparisons](#) chapter.

JavaScript String Comparison

All the comparison operators above can also be used on strings:

Example

```
let text1 = "A";
let text2 = "B";
let result = text1 < text2;
```

Note that strings are compared alphabetically:

Example

```
let text1 = "20";
let text2 = "5";
let result = text1 < text2;
```

JavaScript String Addition

The `+` can also be used to add (concatenate) strings:

Example

```
let text1 = "John";
let text2 = "Doe";
let text3 = text1 + " " + text2;
```

The `+=` assignment operator can also be used to add (concatenate) strings:

Example

```
let text1 = "What a very ";
text1 += "nice day";
```

The result of `text1` will be:

What a very nice day

Note

When used on strings, the `+` operator is called the concatenation operator.

Adding Strings and Numbers

Adding two numbers, will return the sum, but adding a number and a string will return a string:

Example

```
let x = 5 + 5;
let y = "5" + 5;
let z = "Hello" + 5;
```

The result of `x`, `y`, and `z` will be:

10
55
Hello5

Note

If you add a number and a string, the result will be a string!

JavaScript Logical Operators

Operator	Description
&&	logical and
	logical or
!	logical not

Note

Logical operators are fully described in the [JS Comparisons](#) chapter.

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Note

Type operators are fully described in the [JS Type Conversion](#) chapter.

JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
	OR	5 1	0101 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	left shift	5 << 1	0101 << 1	1010	10
>>	right shift	5 >> 1	0101 >> 1	0010	2
>>>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

The examples above uses 4 bits unsigned examples. But JavaScript uses 32-bit signed numbers.

Because of this, in JavaScript, `~ 5` will not return 10. It will return -6.

`~000000000000000000000000000000101` will return

`111111111111111111111111111111010`

Bitwise operators are fully described in the [JS Bitwise](#) chapter.

11. JavaScript Data Types

JavaScript has 8 Datatypes

String
Number
Bigint
Boolean
Undefined
Null
Symbol
Object

The Object Datatype

The object data type can contain both **built-in objects**, and **user defined objects**:

Built-in object types can be:

objects, arrays, dates, maps, sets, intarrays, floatarrays, promises, and more.

Examples

```
// Numbers:  
let length = 16;  
let weight = 7.5;  
  
// Strings:  
let color = "Yellow";  
let lastName = "Johnson";  
  
// Booleans  
let x = true;  
let y = false;  
  
// Object:  
const person = {firstName:"John", lastName:"Doe"};  
  
// Array object:  
const cars = ["Saab", "Volvo", "BMW"];  
  
// Date object:  
const date = new Date("2022-03-25");
```

Note

A JavaScript variable can hold any type of data.

The Concept of Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

Without data types, a computer cannot safely solve this:

```
let x = 16 + "Volvo";
```

Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?

JavaScript will treat the example above as:

```
let x = "16" + "Volvo";
```

Note

When adding a number and a string, JavaScript will treat the number as a string.

Example

```
let x = 16 + "Volvo";
```

Example

```
let x = "Volvo" + 16;
```

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
let x = 16 + 4 + "Volvo";
```

Result:

20Volvo

```
let x = "Volvo" + 16 + 4;
```

Result:

Volvo164

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".

In the second example, since the first operand is a string, all operands are treated as strings.

JavaScript Types are Dynamic

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

Example

```
let x;          // Now x is undefined
x = 5;         // Now x is a Number
x = "John";    // Now x is a String
```

JavaScript Strings

A string (or a text string) is a series of characters like "John Doe".

Strings are written with quotes. You can use single or double quotes:

Example

```
// Using double quotes:
let carName1 = "Volvo XC60";
// Using single quotes:
let carName2 = 'Volvo XC60';
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

Example

```
// Single quote inside double quotes:
let answer1 = "It's alright";
// Single quotes inside double quotes:
let answer2 = "He is called 'Johnny'";
// Double quotes inside single quotes:
let answer3 = 'He is called "Johnny"';
```

JavaScript Numbers

All JavaScript numbers are stored as decimal numbers (floating point).

Numbers can be written with, or without decimals:

Example

```
// With decimals:
let x1 = 34.00;
// Without decimals:
let x2 = 34;
```

Exponential Notation

Extra large or extra small numbers can be written with scientific (exponential) notation:

Example

```
let y = 123e5;      // 12300000
let z = 123e-5;    // 0.00123
```

Note

Most programming languages have many number types:

Whole numbers (integers):

byte (8-bit), short (16-bit), int (32-bit), long (64-bit)

Real numbers (floating-point):

float (32-bit), double (64-bit).

**Javascript numbers are always one type:
double (64-bit floating point).**

JavaScript BigInt

All JavaScript numbers are stored in a 64-bit floating-point format.

JavaScript BigInt is a new datatype ([ES2020](#)) that can be used to store integer values that are too big to be represented by a normal JavaScript Number.

Example

```
let x = BigInt("123456789012345678901234567890");
```

JavaScript Booleans

Booleans can only have two values: `true` or `false`.

Example

```
let x = 5;
let y = 5;
let z = 6;
(x == y)      // Returns true
(x == z)      // Returns false
```

Booleans are often used in conditional testing.

JavaScript Arrays

JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called `cars`, containing three items (car names):

Example

```
const cars = ["Saab", "Volvo", "BMW"];
```

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

JavaScript Objects

JavaScript objects are written with curly braces `{}`.

Object properties are written as name:value pairs, separated by commas.

Example

```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

The object (`person`) in the example above has 4 properties: `firstName`, `lastName`, `age`, and `eyeColor`.

The `typeof` Operator

You can use the JavaScript `typeof` operator to find the type of a JavaScript variable.

The `typeof` operator returns the type of a variable or an expression:

Example

```
typeof ""           // Returns "string"  
typeof "John"      // Returns "string"  
typeof "John Doe"  // Returns "string"
```

Example

```
typeof 0            // Returns "number"  
typeof 314          // Returns "number"  
typeof 3.14         // Returns "number"  
typeof (3)          // Returns "number"  
typeof (3 + 4)      // Returns "number"
```

Undefined

In JavaScript, a variable without a value, has the value `undefined`. The type is also `undefined`.

Example

```
let car; // Value is undefined, type is undefined
```

Any variable can be emptied, by setting the value to `undefined`. The type will also be `undefined`.

Example

```
car = undefined; // Value is undefined, type is undefined
```

Empty Values

An empty value has nothing to do with `undefined`.

An empty string has both a legal value and a type.

Example

```
let car = ""; // The value is "", the typeof is "string"
```

12. JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

Example

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
    return p1 * p2;
}
```

JavaScript Function Syntax

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses `()`.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(parameter1, parameter2, ...)

The code to be executed, by the function, is placed inside curly brackets: `{}`

```
function name(parameter1, parameter2, parameter3) {
    // code to be executed
}
```

Function **parameters** are listed inside the parentheses `()` in the function definition.

Function **arguments** are the **values** received by the function when it is invoked.

Inside the function, the arguments (the parameters) behave as local variables.

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Function Return

When JavaScript reaches a `return` statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

Example

Calculate the product of two numbers, and return the result:

```
// Function is called, the return value will end up in x
let x = myFunction(4, 3);

function myFunction(a, b) {
// Function returns the product of a and b
  return a * b;
}
```

Why Functions?

With functions you can reuse code

You can write code that can be used many times.

You can use the same code with different arguments, to produce different results.

The () Operator

The () operator invokes (calls) the function:

Example

Convert Fahrenheit to Celsius:

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}

let value = toCelsius(77);
```

Accessing a function with incorrect parameters can return an incorrect answer:

Example

```
function toCelsius(fahrenheit) {  
    return (5/9) * (fahrenheit-32);  
}  
  
let value = toCelsius();
```

Accessing a function without () returns the function and not the function result:

Example

```
function toCelsius(fahrenheit) {  
    return (5/9) * (fahrenheit-32);  
}  
  
let value = toCelsius;
```

Note

As you see from the examples above, `toCelsius` refers to the function object, and `toCelsius()` refers to the function result.

Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

Example

Instead of using a variable to store the return value of a function:

```
let x = toCelsius(77);  
let text = "The temperature is " + x + " Celsius";
```

You can use the function directly, as a variable value:

```
let text = "The temperature is " + toCelsius(77) + " Celsius";
```

You will learn a lot more about functions later in this tutorial.

Local Variables

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables can only be accessed from within the function.

Example

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

13. JavaScript Objects

Real Life Objects

In real life, **objects** are things like: houses, cars, people, animals, or any other subjects.

Here is a **car object** example:

Car Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

Object Properties

A real life car has **properties** like weight and color:

car.name = Fiat, car.model = 500, car.weight = 850kg, car.color = white.

Car objects have the same **properties**, but the **values** differ from car to car.

Object Methods

A real life car has **methods** like start and stop:

car.start(), car.drive(), car.brake(), car.stop().

Car objects have the same **methods**, but the methods are performed **at different times**.

JavaScript Variables

JavaScript variables are containers for data values.

This code assigns a **simple value** (Fiat) to a **variable** named car:

Example

```
let car = "Fiat";
```

JavaScript Objects

Objects are variables too. But objects can contain many values.

This code assigns **many values** (Fiat, 500, white) to an **object** named car:

Example

```
const car = {type:"Fiat", model:"500", color:"white"};
```

Note:

It is a common practice to declare objects with the **const** keyword.

JavaScript Object Definition

How to Define a JavaScript Object

- Using an Object Literal
- Using the **new** Keyword
- Using an Object Constructor

JavaScript Object Literal

An object literal is a list of **name:value** pairs inside curly braces **{}**.

```
{firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}
```

Note:

name:value pairs are also called **key:value pairs**.

object literals are also called **object initializers**.

Creating a JavaScript Object

These examples create a JavaScript object with 4 properties:

Examples

```
// Create an Object
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Spaces and line breaks are not important. An object initializer can span multiple lines:

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

This example creates an empty JavaScript object, and then adds 4 properties:

```
// Create an Object
const person = {};

// Add Properties
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

Using the new Keyword

This example create a new JavaScript object using `new Object()`, and then adds 4 properties:

Example

```
// Create an Object
const person = new Object();

// Add Properties
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

Note:

The examples above do exactly the same.

But, there is no need to use `new Object()`.

For readability, simplicity and execution speed, use the **object literal** method.

Object Properties

The **named values**, in JavaScript objects, are called **properties**.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Objects written as name value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName

objectName["propertyName"]

Examples

```
person.lastName;
```

```
person["lastName"];
```

JavaScript Object Methods

Methods are **actions** that can be performed on objects.

Methods are **function definitions** stored as **property values**.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

Example

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

In the example above, `this` refers to the **person object**:

this.firstName means the **firstName** property of **person**.

this.lastName means the **lastName** property of **person**.

In JavaScript, Objects are King.

If you Understand Objects, you Understand JavaScript.

Objects are containers for **Properties** and **Methods**.

Properties are named **Values**.

Methods are **Functions** stored as **Properties**.

Properties can be primitive values, functions, or even other objects.

In JavaScript, almost "everything" is an object.

- Objects are objects
- Maths are objects
- Functions are objects
- Dates are objects
- Arrays are objects
- Maps are objects
- Sets are objects

All JavaScript values, except primitives, are objects.

JavaScript Primitives

A **primitive value** is a value that has no properties or methods.

3.14 is a primitive value

A **primitive data type** is data that has a primitive value.

JavaScript defines 7 types of primitive data types:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `symbol`
- `bigint`

Immutable

Primitive values are immutable (they are hardcoded and cannot be changed).

if `x = 3.14`, you can change the value of `x`, but you cannot change the value of `3.14`.

Value	Type	Comment
"Hello"	string	"Hello" is always "Hello"
3.14	number	3.14 is always 3.14
true	boolean	true is always true
false	boolean	false is always false
null	null (object)	null is always null
undefined	undefined	undefined is always undefined

JavaScript Objects are Mutable

Objects are mutable: They are addressed by reference, not by value.

If person is an object, the following statement will not create a copy of person:

```
const x = person;
```

The object x is **not a copy** of person. The object x **is** person.

The object x and the object person share the same memory address.

Any changes to x will also change person:

Example

```
//Create an Object
const person = {
  firstName:"John",
  lastName:"Doe",
  age:50, eyeColor:"blue"
}
// Create a copy
const x = person;
// Change Age in both
x.age = 10;
```

14. JavaScript Object Properties

An Object is an Unordered Collection of Properties

Properties are the most important part of JavaScript objects.

Properties can be changed, added, deleted, and some are read only.

Accessing JavaScript Properties

The syntax for accessing the property of an object is:

```
// objectName.property  
let age = person.age;  
or  
//objectName["property"]  
let age = person["age"];  
or  
//objectName[expression]  
let age = person[x];
```

Examples

```
person.firstname + " is " + person.age + " years old.";  
person["firstname"] + " is " + person["age"] + " years old.";  
let x = "firstname";  
let y = "age";  
person[x] + " is " + person[y] + " years old.;"
```

Adding New Properties

You can add new properties to an existing object by simply giving it a value:

Example

```
person.nationality = "English";
```

Deleting Properties

The `delete` keyword deletes a property from an object:

Example

```
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};  
delete person.age;
```

```
or delete person["age"];
```

Example

```
const person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};  
  
delete person["age"];
```

Note:

The `delete` keyword deletes both the value of the property and the property itself.

After deletion, the property cannot be used before it is added back again.

Nested Objects

Property values in an object can be other objects:

Example

```
myObj = {  
    name: "John",  
    age: 30,  
    myCars: {  
        car1: "Ford",  
        car2: "BMW",  
        car3: "Fiat"  
    }  
}
```

You can access nested objects using the dot notation or the bracket notation:

Examples

```
myObj.myCars.car2;  
  
myObj.myCars["car2"];  
  
myObj["myCars"]["car2"];  
  
let p1 = "myCars";  
let p2 = "car2";  
myObj[p1][p2];
```

15. JavaScript Object Methods

Object methods are actions that can be performed on objects.

A method is a **function definition** stored as a **property value**.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

Example

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

In the example above, `this` refers to the **person object**:

this.firstName means the **firstName** property of **person**.

this.lastName means the **lastName** property of **person**.

Accessing Object Methods

You access an object method with the following syntax:

```
objectName.methodName()
```

If you invoke the **fullName property** with (), it will execute as a **function**:

Example

```
name = person.fullName();
```

If you access the **fullName property** without (), it will return the **function definition**:

Example

```
name = person.fullName;
```

Adding a Method to an Object

Adding a new method to an object is easy:

Example

```
person.name = function () {  
    return this.firstName + " " + this.lastName;  
};
```

Using JavaScript Methods

This example uses the JavaScript `toUpperCase()` method to convert a text to uppercase:

Example

```
person.name = function () {  
    return (this.firstName + " " + this.lastName).toUpperCase();  
};
```

16. JavaScript Display Objects

How to Display JavaScript Objects?

Displaying a JavaScript object will output **[object Object]**.

Example

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

document.getElementById("demo").innerHTML = person;
```

Some solutions to display JavaScript objects are:

- Displaying the Object Properties by name
- Displaying the Object Properties in a Loop
- Displaying the Object using Object.values()
- Displaying the Object using JSON.stringify()

Displaying Object Properties

The properties of an object can be displayed as a string:

Example

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// Display Properties
document.getElementById("demo").innerHTML =
person.name + "," + person.age + "," + person.city;
```

Displaying Properties in a Loop

The properties of an object can be collected in a loop:

Example

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// Build a Text
let text = "";
for (let x in person) {
  text += person[x] + " ";
}

// Display the Text
document.getElementById("demo").innerHTML = text;
```

Note:

You must use **person[x]** in the loop.

person.x will not work (Because **x** is the loop variable).

Using Object.values()

`Object.values()` creates an array from the property values:

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// Create an Array
const myArray = Object.values(person);

// Display the Array
document.getElementById("demo").innerHTML = myArray;
```

Using Object.entries()

`Object.entries()` makes it simple to use objects in loops:

Example

```
const fruits = {Bananas:300, Oranges:200, Apples:500};

let text = "";
for (let [fruit, value] of Object.entries(fruits)) {
  text += fruit + ": " + value + "<br>";
}
```

Using JSON.stringify()

JavaScript objects can be converted to a string with `JSON` method `JSON.stringify()`.

`JSON.stringify()` is included in JavaScript and supported in all major browsers.

Note:

The result will be a string written in JSON notation:

```
{"name":"John","age":50,"city":"New York"}
```

Example

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// Stringify Object
let myString = JSON.stringify(person);

// Display String
document.getElementById("demo").innerHTML = myString;
```

17. JavaScript Object Constructors

Object Constructor Functions

Sometimes we need to create many objects of the same **type**.

To create an **object type** we use an **object constructor function**.

It is considered good practice to name constructor functions with an upper-case first letter.

Object Type Person

```
function Person(first, last, age, eye) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eye;  
}
```

Note:

In the constructor function, `this` has no value.

The value of `this` will become the new object when a new object is created.

Now we can use `new Person()` to create many new Person objects:

Example

```
const myFather = new Person("John", "Doe", 50, "blue");  
const myMother = new Person("Sally", "Rally", 48, "green");  
const mySister = new Person("Anna", "Rally", 18, "green");  
const mySelf = new Person("Johnny", "Rally", 22, "green");
```

Property Default Values

A **value** given to a property will be a **default value** for all objects created by the constructor:

Example

```
function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
    this.nationality = "English";  
}
```

Adding a Property to an Object

Adding a property to a created object is easy:

Example

```
myFather.nationality = "English";
```

Note:

The new property will be added to **myFather**. Not to any other **Person Objects**.

Adding a Property to a Constructor

You can **NOT** add a new property to an object constructor:

Example

```
Person.nationality = "English";
```

To add a new property, you must add it to the constructor function prototype:

Example

```
Person.prototype.nationality = "English";
```

Constructor Function Methods

A constructor function can also have **methods**:

Example

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
  this.fullName = function() {
    return this.firstName + " " + this.lastName;
  };
}
```

Adding a Method to an Object

Adding a method to a created object is easy:

Example

```
myMother.changeName = function (name) {  
    this.lastName = name;  
}
```

Note:

The new method will be added to **myMother**. Not to any other **Person Objects**.

Adding a Method to a Constructor

You cannot add a new method to an object constructor function.

This code will produce a TypeError:

Example

```
Person.changeName = function (name) {  
    this.lastName = name;  
}  
  
myMother.changeName("Doe");
```

```
TypeError: myMother.changeName is not a function
```

Adding a new method must be done to the constructor function prototype:

Example

```
Person.prototype.changeName = function (name) {  
    this.lastName = name;  
}  
  
myMother.changeName("Doe");
```

Note:

The changeName() function assigns the value of `name` to the person's `lastName` property, substituting `this` with `myMother`.

Built-in JavaScript Constructors

JavaScript has built-in constructors for all native objects:

```
new Object()    // A new Object object
new Array()     // A new Array object
new Map()       // A new Map object
new Set()       // A new Set object
new Date()      // A new Date object
new RegExp()    // A new RegExp object
new Function()  // A new Function object
```

Note:

The `Math()` object is not in the list. `Math` is a global object. The `new` keyword cannot be used on `Math`.

Did You Know?

Use object literals `{}` instead of `new Object()`.

Use array literals `[]` instead of `new Array()`.

Use pattern literals `/()/.exec("")` instead of `new RegExp()`.

Use function expressions `() {}` instead of `new Function()`.

Example

```
"";           // primitive string
0;            // primitive number
false;        // primitive boolean

{};           // object object
[];           // array object
/()/.exec(""); // regexp object
function(){}; // function
```

18. JavaScript Events

HTML events are "**things**" that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can "**react**" on these events.

HTML Events

An HTML event can be something the browser does, or something a user does.
Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

```
<element event='some JavaScript'>
```

With double quotes:

```
<element event="some JavaScript">
```

In the following example, an `onclick` attribute (with code), is added to a `<button>` element:

Example

```
<button onclick="document.getElementById('demo').innerHTML = Date()">The time  
is?</button>
```

In the example above, the JavaScript code changes the content of the element with `id="demo"`.

In the next example, the code changes the content of its own element (using `this.innerHTML`):

Example

```
<button onclick="this.innerHTML = Date()">The time is?</button>
```

JavaScript code is often several lines long. It is more common to see event attributes calling functions:

Example

```
<button onclick="displayDate()">The time is?</button>
```

Common HTML Events

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

The list is much longer: [W3Schools JavaScript Reference HTML DOM Events](https://www.w3schools.com/jsref/dom_event.asp)

JavaScript Event Handlers

Event handlers can be used to handle and verify user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user inputs data
- And more ...

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign your own event handler functions to HTML elements
- You can prevent events from being sent or being handled
- And more ...

HTML DOM Events

DOM Events allow JavaScript to add **event listener** or **event handlers** to HTML elements.

Examples

In HTML onclick is the event listener, myFunction is the event handler:

```
<button onclick="myFunction()">Click me</button>
```

In JavaScript click is the event, myFunction is the event handler:

```
button.addEventListener("click", myFunction);
```

Event	Occurs When	Belongs To
abort	The loading of a media is aborted	UiEvent , Event
afterprint	A page has started printing	Event
animationend	A CSS animation has completed	AnimationEvent
animationiteration	A CSS animation is repeated	AnimationEvent
animationstart	A CSS animation has started	AnimationEvent
beforeprint	A page is about to be printed	Event
beforeunload	Before a document is about to be unloaded	UiEvent , Event

blur	An element loses focus	FocusEvent
canplay	The browser can start playing a media (has buffered enough to begin)	Event
canplaythrough	The browser can play through a media without stopping for buffering	Event
change	The content of a form element has changed	Event
click	An element is clicked on	MouseEvent
contextmenu	An element is right-clicked to open a context menu	MouseEvent
copy	The content of an element is copied	ClipboardEvent
cut	The content of an element is cut	ClipboardEvent
dblclick	An element is double-clicked	MouseEvent
drag	An element is being dragged	DragEvent
dragend	Dragging of an element has ended	DragEvent

dragenter	A dragged element enters the drop target	DragEvent
dragleave	A dragged element leaves the drop target	DragEvent
dragover	A dragged element is over the drop target	DragEvent
dragstart	Dragging of an element has started	DragEvent
drop	A dragged element is dropped on the target	DragEvent
durationchange	The duration of a media is changed	Event
ended	A media has reach the end ("thanks for listening")	Event
error	An error has occurred while loading a file	ProgressEvent, UiEvent, Event
focus	An element gets focus	FocusEvent
focusin	An element is about to get focus	FocusEvent
focusout	An element is about to lose focus	FocusEvent

fullscreenchange	An element is displayed in fullscreen mode	Event
fullscreenerror	An element can not be displayed in fullscreen mode	Event
hashchange	There has been changes to the anchor part of a URL	HashChangeEvent
input	An element gets user input	InputEvent , Event
invalid	An element is invalid	Event
keydown	A key is down	KeyboardEvent
keypress	A key is pressed	KeyboardEvent
keyup	A key is released	KeyboardEvent
load	An object has loaded	UiEvent , Event
loadeddata	Media data is loaded	Event
loadedmetadata	Meta data (like dimensions and duration) are loaded	Event

loadstart	The browser starts looking for the specified media	ProgressEvent
message	A message is received through the event source	Event
mousedown	The mouse button is pressed over an element	MouseEvent
mouseenter	The pointer is moved onto an element	MouseEvent
mouseleave	The pointer is moved out of an element	MouseEvent
mousemove	The pointer is moved over an element	MouseEvent
mouseover	The pointer is moved onto an element	MouseEvent
mouseout	The pointer is moved out of an element	MouseEvent
mouseup	A user releases a mouse button over an element	MouseEvent

mousewheel	<p>Deprecated. Use the wheel event instead</p>	WheelEvent
offline	The browser starts working offline	Event
online	The browser starts working online	Event
open	A connection with the event source is opened	Event
pagehide	User navigates away from a webpage	PageTransitionEvent
pageshow	User navigates to a webpage	PageTransitionEvent
paste	Some content is pasted in an element	ClipboardEvent
pause	A media is paused	Event
play	The media has started or is no longer paused	Event
playing	The media is playing after being paused or buffered	Event
popstate	The window's history changes	PopStateEvent

progress	The browser is downloading media data	Event
ratechange	The playing speed of a media is changed	Event
resize	The document view is resized	UiEvent , Event
reset	A form is reset	Event
scroll	An scrollbar is being scrolled	UiEvent , Event
search	Something is written in a search field	Event
seeked	Skipping to a media position is finished	Event
seeking	Skipping to a media position is started	Event
select	User selects some text	UiEvent , Event
show	A <menu> element is shown as a context menu	Event
stalled	The browser is trying to get unavailable media data	Event

storage	A Web Storage area is updated	StorageEvent
submit	A form is submitted	Event
suspend	The browser is intentionally not getting media data	Event
timeupdate	The playing position has changed (the user moves to a different point in the media)	Event
toggle	The user opens or closes the <details> element	Event
touchcancel	The touch is interrupted	TouchEvent
touchend	A finger is removed from a touch screen	TouchEvent
touchmove	A finger is dragged across the screen	TouchEvent
touchstart	A finger is placed on a touch screen	TouchEvent
transitionend	A CSS transition has completed	TransitionEvent
unload	A page has unloaded	UiEvent , Event

<u>volumechange</u>	The volume of a media is changed (includes muting)	<u>Event</u>
<u>waiting</u>	A media is paused but is expected to resume (e.g. buffering)	<u>Event</u>
<u>wheel</u>	The mouse wheel rolls up or down over an element	<u>WheelEvent</u>

19. JavaScript Strings

Strings are for **storing text**

Strings are written **with quotes**

Using Quotes

A JavaScript string is zero or more characters written inside quotes.

Example

```
let text = "John Doe";
```

You can use single or double quotes:

Example

```
let carName1 = "Volvo XC60"; // Double quotes
let carName2 = 'Volvo XC60'; // Single quotes
```

Note

Strings created with single or double quotes works the same.

There is no difference between the two.

Quotes Inside Quotes

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

Example

```
let answer1 = "It's alright";
let answer2 = "He is called 'Johnny'";
let answer3 = 'He is called "Johnny"';
```

Template Strings

Templates were introduced with ES6 (JavaScript 2016).

Templates are strings enclosed in backticks (`This is a template string`).

Templates allow single and double quotes inside a string:

Example

```
let text = `He's often called "Johnny"`;
```

Note

Templates are not supported in Internet Explorer.

String Length

To find the length of a string, use the built-in `length` property:

Example

```
let text = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
let length = text.length;
```

Escape Characters

Because strings must be written within quotes, JavaScript will misunderstand this string:

```
let text = "We are the so-called "Vikings" from the north.;"
```

The string will be chopped to "We are the so-called ".

To solve this problem, you can use an **backslash escape character**.

The backslash escape character (\) turns special characters into string characters:

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\"	\	Backslash

Examples

\" inserts a double quote in a string:

```
let text = "We are the so-called \"Vikings\" from the north.";
```

\' inserts a single quote in a string:

```
let text= 'It\'s alright.';
```

\\\ inserts a backslash in a string:

```
let text = "The character \\ is called backslash.";
```

Six other escape sequences are valid in JavaScript:

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

Note

The 6 escape characters above were originally designed to control typewriters, teletypes, and fax machines. They do not make any sense in HTML.

Breaking Long Lines

For readability, programmers often like to avoid long code lines.

A safe way to break up a **statement** is after an operator:

Example

```
document.getElementById("demo").innerHTML =  
"Hello Dolly!";
```

A safe way to break up a **string** is by using string addition:

Example

```
document.getElementById("demo").innerHTML = "Hello " +  
"Dolly!";
```

Template Strings

Templates were introduced with ES6 (JavaScript 2016).

Templates are strings enclosed in backticks (`This is a template string`).

Templates allow multiline strings:

Example

```
let text =  
`The quick  
brown fox  
jumps over  
the lazy dog`;
```

Note

Templates are not supported in Internet Explorer.

JavaScript Strings as Objects

Normally, JavaScript strings are primitive values, created from literals:

```
let x = "John";
```

But strings can also be defined as objects with the keyword `new`:

```
let y = new String("John");
```

Example

```
let x = "John";
let y = new String("John");
```

Do not create Strings objects.

The `new` keyword complicates the code and slows down execution speed.

String objects can produce unexpected results:

When using the `==` operator, x and y are **equal**:

```
let x = "John";
let y = new String("John");
```

When using the `===` operator, x and y are **not equal**:

```
let x = "John";
let y = new String("John");
```

Note the difference between `(x==y)` and `(x===y)`.

`(x == y)` true or false?

```
let x = new String("John");
let y = new String("John");
```

`(x === y)` true or false?

```
let x = new String("John");
let y = new String("John");
```

Comparing two JavaScript objects **always** returns **false**.

20. JavaScript String Methods

Basic String Methods

Javascript strings are primitive and immutable: All string methods produce a new string without altering the original string.

[String length](#)
[String charAt\(\)](#)
[String charCodeAt\(\)](#)
[String at\(\)](#)
[String \[\]](#)
[String slice\(\)](#)
[String substring\(\)](#)
[String substr\(\)](#)

[String toUpperCase\(\)](#)
[String toLowerCase\(\)](#)
[String concat\(\)](#)
[String trim\(\)](#)
[String trimStart\(\)](#)
[String trimEnd\(\)](#)
[String padStart\(\)](#)
[String padEnd\(\)](#)
[String repeat\(\)](#)
[String replace\(\)](#)
[String replaceAll\(\)](#)
[String split\(\)](#)

See Also:

[String Search Methods](#)
[String Templates](#)

JavaScript String Length

The `length` property returns the length of a string:

Example

```
let text = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
let length = text.length;
```

Extracting String Characters

There are 4 methods for extracting string characters:

- The `at(position)` Method
- The `charAt(position)` Method
- The `charCodeAt(position)` Method
- Using property access `[]` like in arrays

JavaScript String `charAt()`

The `charAt()` method returns the character at a specified index (position) in a string:

Example

```
let text = "HELLO WORLD";
let char = text.charAt(0);
```

JavaScript String charCodeAt()

The `charCodeAt()` method returns the code of the character at a specified index in a string:

The method returns a UTF-16 code (an integer between 0 and 65535).

Example

```
let text = "HELLO WORLD";
let char = text.charCodeAt(0);
```

JavaScript String at()

[ES2022](#) introduced the string method `at()`:

Examples

Get the third letter of name:

```
const name = "W3Schools";
let letter = name.at(2);
```

Get the third letter of name:

```
const name = "W3Schools";
let letter = name[2];
```

The `at()` method returns the character at a specified index (position) in a string.

The `at()` method is supported in all modern browsers since March 2022:

Note

The `at()` method is a new addition to JavaScript.

It allows the use of negative indexes while `charAt()` do not.

Now you can use `myString.at(-2)` instead of `charAt(myString.length-2)`.

Browser Support

`at()` is an ES2022 feature.

JavaScript 2022 (ES2022) is supported in all modern browsers since March 2023:

				
Chrome 94	Edge 94	Firefox 93	Safari 16.4	Opera 79
Sep 2021	Sep 2021	Oct 2021	Mar 2023	Oct 2021

Property Access []

Example

```
let text = "HELLO WORLD";
let char = text[0];
```

Note

Property access might be a little **unpredictable**:

- It makes strings look like arrays (but they are not)
- If no character is found, [] returns undefined, while charAt() returns an empty string.
- It is read only. str[0] = "A" gives no error (but does not work!)

Example

```
let text = "HELLO WORLD";
text[0] = "A";    // Gives no error, but does not work
```

Extracting String Parts

There are 3 methods for extracting a part of a string:

- `slice(start, end)`
- `substring(start, end)`
- `substr(start, length)`

JavaScript String slice()

`slice()` extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: start position, and end position (end not included).

Example

Slice out a portion of a string from position 7 to position 13:

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(7, 13);
```

Note

JavaScript counts positions from zero.

First position is 0.

Second position is 1.

Examples

If you omit the second parameter, the method will slice out the rest of the string:

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(7);
```

If a parameter is negative, the position is counted from the end of the string:

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(-12);
```

This example slices out a portion of a string from position -12 to position -6:

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(-12, -6);
```

JavaScript String substr()

`substring()` is similar to `slice()`.

The difference is that start and end values less than 0 are treated as 0 in `substring()`.

Example

```
let str = "Apple, Banana, Kiwi";
let part = str.substring(7, 13);
```

If you omit the second parameter, `substring()` will slice out the rest of the string.

JavaScript String substr()

`substr()` is similar to `slice()`.

The difference is that the second parameter specifies the **length** of the extracted part.

Example

```
let str = "Apple, Banana, Kiwi";
let part = str.substr(7, 6);
```

If you omit the second parameter, `substr()` will slice out the rest of the string.

Example

```
let str = "Apple, Banana, Kiwi";
let part = str.substr(7);
```

If the first parameter is negative, the position counts from the end of the string.

Example

```
let str = "Apple, Banana, Kiwi";
let part = str.substr(-4);
```

Converting to Upper and Lower Case

A string is converted to upper case with `toUpperCase()`:

A string is converted to lower case with `toLowerCase()`:

JavaScript String `toUpperCase()`

Example

```
let text1 = "Hello World!";
let text2 = text1.toUpperCase();
```

JavaScript String `toLowerCase()`

Example

```
let text1 = "Hello World!";      // String
let text2 = text1.toLowerCase(); // text2 is text1 converted to lower
```

JavaScript String `concat()`

`concat()` joins two or more strings:

Example

```
let text1 = "Hello";
let text2 = "World";
let text3 = text1.concat(" ", text2);
```

The `concat()` method can be used instead of the plus operator. These two lines do the same:

Example

```
text = "Hello" + " " + "World!";
text = "Hello".concat(" ", "World!");
```

Note

All string methods return a new string. They don't modify the original string.

Formally said:

Strings are immutable: Strings cannot be changed, only replaced.

JavaScript String trim()

The `trim()` method removes whitespace from both sides of a string:

Example

```
let text1 = "      Hello World!      ";
let text2 = text1.trim();
```

JavaScript String trimStart()

[ECMAScript 2019](#) added the String method `trimStart()` to JavaScript.

The `trimStart()` method works like `trim()`, but removes whitespace only from the start of a string.

Example

```
let text1 = "      Hello World!      ";
let text2 = text1.trimStart();
```

JavaScript String `trimStart()` is supported in all modern browsers since January 2020:

				
Chrome 66	Edge 79	Firefox 61	Safari 12	Opera 50
Apr 2018	Jan 2020	Jun 2018	Sep 2018	May 2018

JavaScript String trimEnd()

[ECMAScript 2019](#) added the string method `trimEnd()` to JavaScript.

The `trimEnd()` method works like `trim()`, but removes whitespace only from the end of a string.

Example

```
let text1 = "    Hello World!    ";
let text2 = text1.trimEnd();
```

JavaScript String `trimEnd()` is supported in all modern browsers since January 2020:

				
Chrome 66	Edge 79	Firefox 61	Safari 12	Opera 50
Apr 2018	Jan 2020	Jun 2018	Sep 2018	May 2018

JavaScript String Padding

[ECMAScript 2017](#) added two new string methods to JavaScript: `padStart()` and `padEnd()` to support padding at the beginning and at the end of a string.

JavaScript String padStart()

The `padStart()` method pads a string from the start.

It pads a string with another string (multiple times) until it reaches a given length.

Examples

Pad a string with "0" until it reaches the length 4:

```
let text = "5";
let padded = text.padStart(4, "0");
```

Pad a string with "x" until it reaches the length 4:

```
let text = "5";
let padded = text.padStart(4, "x");
```

Note

The `padStart()` method is a string method.

To pad a number, convert the number to a string first.

See the example below.

Example

```
let numb = 5;  
let text = numb.toString();  
let padded = text.padStart(4,"0");
```

Browser Support

`padStart()` is an [ECMAScript 2017](#) feature.

ES2017 is supported in all modern browsers since September 2017:

				
Chrome 58	Edge 15	Firefox 52	Safari 11	Opera 45
Apr 2017	Apr 2017	Mar 2017	Sep 2017	May 2017

`padStart()` is not supported in Internet Explorer.

JavaScript String padEnd()

The `padEnd()` method pads a string from the end.

It pads a string with another string (multiple times) until it reaches a given length.

Examples

```
let text = "5";  
let padded = text.padEnd(4,"0");  
  
let text = "5";  
let padded = text.padEnd(4,"x");
```

Note

The `padEnd()` method is a string method.

To pad a number, convert the number to a string first.

See the example below.

Example

```
let numb = 5;  
let text = numb.toString();  
let padded = text.padEnd(4, "0");
```

Browser Support

`padEnd()` is an [ECMAScript 2017](#) feature.

ES2017 is supported in all modern browsers since September 2017:

				
Chrome 58	Edge 15	Firefox 52	Safari 11	Opera 45
Apr 2017	Apr 2017	Mar 2017	Sep 2017	May 2017

`padEnd()` is not supported in Internet Explorer.

JavaScript String repeat()

The `repeat()` method returns a string with a number of copies of a string.

The `repeat()` method returns a new string.

The `repeat()` method does not change the original string.

Examples

Create copies of a text:

```
let text = "Hello world!";  
let result = text.repeat(2);  
  
let text = "Hello world!";  
let result = text.repeat(4);
```

Syntax

`string.repeat(count)`

Parameters

Parameter	Description
<code>count</code>	Required. The number of copies wanted.

Return Value

Type	Description
String	A new string containing the copies.

Browser Support

`repeat()` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`repeat()` is not supported in Internet Explorer.

Replacing String Content

The `replace()` method replaces a specified value with another value in a string:

Example

```
let text = "Please visit Microsoft!";
let newText = text.replace("Microsoft", "W3Schools");
```

Note

The `replace()` method does not change the string it is called on.

The `replace()` method returns a new string.

The `replace()` method replaces **only the first** match

If you want to replace all matches, use a regular expression with the /g flag set. See examples below.

By default, the `replace()` method replaces **only the first** match:

Example

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace("Microsoft", "W3Schools");
```

By default, the `replace()` method is case sensitive. Writing MICROSOFT (with upper-case) will not work:

Example

```
let text = "Please visit Microsoft!";
let newText = text.replace("MICROSOFT", "W3Schools");
```

To replace case insensitive, use a **regular expression** with an `/i` flag (insensitive):

Example

```
let text = "Please visit Microsoft!";
let newText = text.replace(/MICROSOFT/i, "W3Schools");
```

Note

Regular expressions are written without quotes.

To replace all matches, use a **regular expression** with a `/g` flag (global match):

Example

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace(/Microsoft/g, "W3Schools");
```

JavaScript String ReplaceAll()

In 2021, JavaScript introduced the string method `replaceAll()`:

Example

```
text = text.replaceAll("Cats", "Dogs");
text = text.replaceAll("cats", "dogs");
```

The `replaceAll()` method allows you to specify a regular expression instead of a string to be replaced.

If the parameter is a regular expression, the global flag (g) must be set, otherwise a `TypeError` is thrown.

Example

```
text = text.replaceAll(/Cats/g, "Dogs");
text = text.replaceAll(/cats/g, "dogs");
```

Note

`replaceAll()` is an [ES2021](#) feature.

`replaceAll()` does not work in Internet Explorer.

Converting a String to an Array

If you want to work with a string as an array, you can convert it to an array.

JavaScript String split()

A string can be converted to an array with the `split()` method:

Example

```
text.split(",") // Split on commas
text.split(" ") // Split on spaces
text.split("|") // Split on pipe
```

If the separator is omitted, the returned array will contain the whole string in index [0]. If the separator is "", the returned array will be an array of single characters:

Example

```
text.split("")
```

21. JavaScript String Search

String Search Methods

[String indexOf\(\)](#)

[String lastIndexOf\(\)](#)

[String search\(\)](#)

[String match\(\)](#)

[String matchAll\(\)](#)

[String includes\(\)](#)

[String startsWith\(\)](#)

[String endsWith\(\)](#)

See Also:

[Basic String Methods](#)

[String Templates](#)

JavaScript String indexOf()

The `indexOf()` method returns the **index** (position) of the **first** occurrence of a string in a string, or it returns -1 if the string is not found:

Example

```
let text = "Please locate where 'locate' occurs!";
let index = text.indexOf("locate");
```

Note

JavaScript counts positions from zero.

0 is the first position in a string, 1 is the second, 2 is the third, ...

JavaScript String lastIndexOf()

The `lastIndexOf()` method returns the **index** of the **last** occurrence of a specified text in a string:

Example

```
let text = "Please locate where 'locate' occurs!";
let index = text.lastIndexOf("locate");
```

Both `indexOf()`, and `lastIndexOf()` return -1 if the text is not found:

Example

```
let text = "Please locate where 'locate' occurs!";
let index = text.lastIndexOf("John");
```

Both methods accept a second parameter as the starting position for the search:

Example

```
let text = "Please locate where 'locate' occurs!";
let index = text.indexOf("locate", 15);
```

The `lastIndexOf()` methods searches backwards (from the end to the beginning), meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.

Example

```
let text = "Please locate where 'locate' occurs!";
text.lastIndexOf("locate", 15);
```

JavaScript String search()

The `search()` method searches a string for a string (or a regular expression) and returns the position of the match:

Examples

```
let text = "Please locate where 'locate' occurs!";
text.search("locate");

let text = "Please locate where 'locate' occurs!";
text.search(/locate/);
```

Did You Notice?

The two methods, `indexOf()` and `search()`, are **equal?**

They accept the same arguments (parameters), and return the same value?

The two methods are **NOT** equal. These are the differences:

- The `search()` method cannot take a second start position argument.
- The `indexOf()` method cannot take powerful search values (regular expressions).

You will learn more about regular expressions in a later chapter.

JavaScript String match()

The `match()` method returns an array containing the results of matching a string against a string (or a regular expression).

Examples

Perform a search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match("ain");
```

Perform a search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/);
```

Perform a global search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/g);
```

Perform a global, case-insensitive search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/gi);
```

Note

If a regular expression does not include the `g` modifier (global search), `match()` will return only the first match in the string.

JavaScript String matchAll()

The `matchAll()` method returns an iterator containing the results of matching a string against a string (or a regular expression).

Example

```
const iterator = text.matchAll("Cats");
```

If the parameter is a regular expression, the global flag (`g`) must be set, otherwise a `TypeError` is thrown.

Example

```
const iterator = text.matchAll(/Cats/g);
```

If you want to search case insensitive, the insensitive flag (i) must be set:

Example

```
const iterator = text.matchAll(/Cats/gi);
```

Notes

`matchAll()` is an [ES2020](#) feature.

`matchAll()` does not work in Internet Explorer.

JavaScript String includes()

The `includes()` method returns true if a string contains a specified value.

Otherwise it returns `false`.

Examples

Check if a string includes "world":

```
let text = "Hello world, welcome to the universe.";
text.includes("world");
```

Check if a string includes "world". Start at position 12:

```
let text = "Hello world, welcome to the universe.";
text.includes("world", 12);
```

Notes

`includes()` is case sensitive.

`includes()` is an [ES6 feature](#).

`includes()` is not supported in Internet Explorer.

JavaScript String startsWith()

The `startsWith()` method returns `true` if a string begins with a specified value.

Otherwise it returns `false`:

Examples

Returns true:

```
let text = "Hello world, welcome to the universe.";
text.startsWith("Hello");
```

Returns false:

```
let text = "Hello world, welcome to the universe.";
text.startsWith("world")
```

A start position for the search can be specified:

Returns false:

```
let text = "Hello world, welcome to the universe.";
text.startsWith("world", 5)
```

Returns true:

```
let text = "Hello world, welcome to the universe.";
text.startsWith("world", 6)
```

Notes

`startsWith()` is case sensitive.

`startsWith()` is an [ES6 feature](#).

`startsWith()` is not supported in Internet Explorer.

JavaScript String endsWith()

The `endsWith()` method returns `true` if a string ends with a specified value.

Otherwise it returns `false`:

Examples

Check if a string ends with "Doe":

```
let text = "John Doe";
text.endsWith("Doe");
```

Check if the 11 first characters of a string ends with "world":

```
let text = "Hello world, welcome to the universe.";
text.endsWith("world", 11);
```

Notes

`endsWith()` is case sensitive.

`endsWith()` is an [ES6 feature](#).

`endsWith()` is not supported in Internet Explorer.

22. JavaScript Template Strings

String Templates

Template Strings

Template Literals

Beloved child has many names

Back-Ticks Syntax

Template Strings use back-ticks (``) rather than the quotes ("") to define a string:

Example

```
let text = `Hello World!`;
```

Quotes Inside Strings

Template Strings allow both single and double quotes inside a string:

Example

```
let text = `He's often called "Johnny"`;
```

Multiline Strings

Template Strings allow multiline strings:

Example

```
let text =  
`The quick  
brown fox  
jumps over  
the lazy dog`;
```

Interpolation

Template String provide an easy way to interpolate variables and expressions into strings.
The method is called string interpolation.

The syntax is:

``${...}``

Variable Substitutions

Template Strings allow variables in strings:

Example

```
let firstName = "John";
let lastName = "Doe";
let text = `Welcome ${firstName}, ${lastName}!`;
```

Automatic replacing of variables with real values is called **string interpolation**.

Expression Substitution

Template Strings allow expressions in strings:

Example

```
let price = 10;
let VAT = 0.25;
let total = `Total: ${price * (1 + VAT).toFixed(2)}`;
```

Automatic replacing of expressions with real values is called **string interpolation**.

HTML Templates

Example

```
let header = "Template Strings";
let tags = ["template strings", "javascript", "es6"];

let html = `<h2>${header}</h2><ul>`;
for (const x of tags) {
    html += `<li>${x}</li>`;
}
html += `</ul>`;
```

Browser Support

Template Strings is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

Template Strings is not supported in Internet Explorer.

23. JavaScript Numbers

JavaScript has only one type of number. Numbers can be written with or without decimals.

Example

```
let x = 3.14;      // A number with decimals  
let y = 3;        // A number without decimals
```

Extra large or extra small numbers can be written with scientific (exponent) notation:

Example

```
let x = 123e5;    // 12300000  
let y = 123e-5;   // 0.00123
```

JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:

Value (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

Integer Precision

Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

Example

```
let x = 999999999999999;  // x will be 999999999999999  
let y = 999999999999999; // y will be 1000000000000000000000000
```

The maximum number of decimals is 17.

Floating Precision

Floating point arithmetic is not always 100% accurate:

```
let x = 0.2 + 0.1;
```

To solve the problem above, it helps to multiply and divide:

```
let x = (0.2 * 10 + 0.1 * 10) / 10;
```

Adding Numbers and Strings

WARNING !!

JavaScript uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
let x = 10;
let y = 20;
let z = x + y;
```

If you add two strings, the result will be a string concatenation:

Example

```
let x = "10";
let y = "20";
let z = x + y;
```

If you add a number and a string, the result will be a string concatenation:

Example

```
let x = 10;
let y = "20";
let z = x + y;
```

If you add a string and a number, the result will be a string concatenation:

Example

```
let x = "10";
let y = 20;
let z = x + y;
```

A common mistake is to expect this result to be 30:

Example

```
let x = 10;  
let y = 20;  
let z = "The result is: " + x + y;
```

A common mistake is to expect this result to be 102030:

Example

```
let x = 10;  
let y = 20;  
let z = "30";  
let result = x + y + z;
```

The JavaScript interpreter works from left to right.

First $10 + 20$ is added because x and y are both numbers.

Then $30 + "30"$ is concatenated because z is a string.

Numeric Strings

JavaScript strings can have numeric content:

```
let x = 100;           // x is a number  
let y = "100";        // y is a string
```

JavaScript will try to convert strings to numbers in all numeric operations:

This will work:

```
let x = "100";  
let y = "10";  
let z = x / y;
```

This will also work:

```
let x = "100";  
let y = "10";  
let z = x * y;
```

And this will work:

```
let x = "100";  
let y = "10";  
let z = x - y;
```

But this will **not** work:

```
let x = "100";  
let y = "10";  
let z = x + y;
```

In the last example JavaScript uses the `+` operator to concatenate the strings.

NaN - Not a Number

NaN is a JavaScript reserved word indicating that a number is not a legal number.

Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

Example

```
let x = 100 / "Apple";
```

However, if the string is numeric, the result will be a number:

Example

```
let x = 100 / "10";
```

You can use the global JavaScript function isNaN() to find out if a value is not a number:

Example

```
let x = 100 / "Apple";
isNaN(x);
```

Watch out for NaN. If you use NaN in a mathematical operation, the result will also be NaN:

Example

```
let x = NaN;
let y = 5;
let z = x + y;
```

Or the result might be a concatenation like NaN5:

Example

```
let x = NaN;
let y = "5";
let z = x + y;
```

NaN is a number: typeof NaN returns number:

Example

```
typeof NaN;
```

Infinity

`Infinity` (or `-Infinity`) is the value JavaScript will return if you calculate a number outside the largest possible number.

Example

```
let myNumber = 2;  
// Execute until Infinity  
while (myNumber != Infinity) {  
    myNumber = myNumber * myNumber;  
}
```

Division by 0 (zero) also generates `Infinity`:

Example

```
let x = 2 / 0;  
let y = -2 / 0;  
Infinity is a number: typeof Infinity returns number.
```

Example

```
typeof Infinity;
```

Hexadecimal

JavaScript interprets numeric constants as hexadecimal if they are preceded by `0x`.

Example

```
let x = 0xFF;
```

Never write a number with a leading zero (like `07`).

Some JavaScript versions interpret numbers as octal if they are written with a leading zero.

By default, JavaScript displays numbers as **base 10** decimals.

But you can use the `toString()` method to output numbers from **base 2** to **base 36**.

Hexadecimal is **base 16**. Decimal is **base 10**. Octal is **base 8**. Binary is **base 2**.

Example

```
let myNumber = 32;  
myNumber.toString(32);  
myNumber.toString(16);  
myNumber.toString(12);  
myNumber.toString(10);  
myNumber.toString(8);  
myNumber.toString(2);
```

JavaScript Numbers as Objects

Normally JavaScript numbers are primitive values created from literals:

```
let x = 123;
```

But numbers can also be defined as objects with the keyword `new`:

```
let y = new Number(123);
```

Example

```
let x = 123;  
let y = new Number(123);
```

Do not create Number objects.

The `new` keyword complicates the code and slows down execution speed.

Number Objects can produce unexpected results:

When using the `==` operator, x and y are **equal**:

```
let x = 500;  
let y = new Number(500);
```

When using the `===` operator, x and y are **not equal**.

```
let x = 500;  
let y = new Number(500);
```

Note the difference between `(x==y)` and `(x===y)`.

`(x == y)` true or false?

```
let x = new Number(500);  
let y = new Number(500);
```

`(x === y)` true or false?

```
let x = new Number(500);  
let y = new Number(500);
```

Comparing two JavaScript objects **always** returns **false**.

24. JavaScript BigInt

JavaScript **BigInt** variables are used to store big integer values that are too big to be represented by a normal JavaScript **Number**.

JavaScript Integer Accuracy

JavaScript integers are only accurate up to 15 digits:

Integer Precision

```
let x = 999999999999999;
let y = 999999999999999;
```

In JavaScript, all numbers are stored in a 64-bit floating-point format (IEEE 754 standard).

With this standard, large integer cannot be exactly represented and will be rounded.

Because of this, JavaScript can only safely represent integers:

Up to **9007199254740991** $+(2^{53}-1)$

and

Down to **-9007199254740991** $-(2^{53}-1)$.

Integer values outside this range lose precision.

How to Create a BigInt

To create a **BigInt**, append n to the end of an integer or call **BigInt()**:

Examples

```
let x = 999999999999999;
let y = 999999999999999n;

let x = 1234567890123456789012345n;
let y = BigInt(1234567890123456789012345)
```

BigInt: A new JavaScript Datatype

The JavaScript **typeof** a **BigInt** is "bigint":

Example

```
let x = BigInt(999999999999999);
let type = typeof x;
```

`BigInt` is the second numeric data type in JavaScript (after `Number`). With `BigInt` the total number of supported data types in JavaScript is 8:

1. String
 2. Number
 3. Bigint
 4. Boolean
 5. Undefined
 6. Null
 7. Symbol
 8. Object

BigInt Operators

Operators that can be used on a JavaScript `Number` can also be used on a `BigInt`.

BigInt Multiplication Example

```
let x = 9007199254740995n;  
let y = 9007199254740995n;  
let z = x * y;
```

Notes

Arithmetic between a `BigInt` and a `Number` is not allowed (type conversion lose information).

Unsigned right shift (`>>>`) can not be done on a `BigInt` (it does not have a fixed width).

BigInt Decimals

A **BigInt** can not have decimals.

BigInt Division Example

```
let x = 5n;
let y = x / 2;
// Error: Cannot mix BigInt and other types, use explicit conversion.

let x = 5n;
let y = Number(x) / 2;
```

BigInt Hex, Octal and Binary

BigInt can also be written in hexadecimal, octal, or binary notation:

BigInt Hex Example

Precision Curiosity

Rounding can compromise program security:

MAX_SAFE_INTEGER Example

```
9007199254740992 === 9007199254740993; // is true !!!
```

Browser Support

`BigInt` is supported in all browsers since September 2020:

				
Chrome 67	Edge 79	Firefox 68	Safari 14	Opera 54
May 2018	Jan 2020	Jul 2019	Sep 2020	Jun 2018

Minimum and Maximum Safe Integers

[ES6](#) added `max` and `min` properties to the `Number` object:

- `MAX_SAFE_INTEGER`
- `MIN_SAFE_INTEGER`

MAX_SAFE_INTEGER Example

```
let x = Number.MAX_SAFE_INTEGER;
```

MIN_SAFE_INTEGER Example

```
let x = Number.MIN_SAFE_INTEGER;
```

New Number Methods

[ES6](#) also added 2 new methods to the `Number` object:

- `Number.isInteger()`
- `Number.isSafeInteger()`

The Number.isInteger() Method

The `Number.isInteger()` method returns `true` if the argument is an integer.

Example: isInteger()

```
Number.isInteger(10);  
Number.isInteger(10.5);
```

The Number.isSafeInteger() Method

A safe integer is an integer that can be exactly represented as a double precision number.

The `Number.isSafeInteger()` method returns `true` if the argument is a safe integer.

Example isSafeInteger()

```
Number.isSafeInteger(10);  
Number.isSafeInteger(12345678901234567890);
```

Safe integers are all integers from $-(2^{53} - 1)$ to $+(2^{53} - 1)$.

This is safe: 9007199254740991. This is not safe: 9007199254740992

25. JavaScript Number Methods

JavaScript Number Methods

These **number methods** can be used on all JavaScript numbers:

Method	Description
toString()	Returns a number as a string
toExponential()	Returns a number written in exponential notation
toFixed()	Returns a number written with a number of decimals
toPrecision()	Returns a number written with a specified length
valueOf()	Returns a number as a number

The **toString()** Method

The **toString()** method returns a number as a string.

All number methods can be used on any type of numbers (literals, variables, or expressions):

Example

```
let x = 123;
x.toString();
(123).toString();
(100 + 23).toString();
```

The **toExponential()** Method

toExponential() returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of characters behind the decimal point:

Example

```
let x = 9.656;  
x.toExponential(2);  
x.toExponential(4);  
x.toExponential(6);
```

The parameter is optional. If you don't specify it, JavaScript will not round the number.

The **toFixed()** Method

toFixed() returns a string, with the number written with a specified number of decimals:

Example

```
let x = 9.656;  
x.toFixed(0);  
x.toFixed(2);  
x.toFixed(4);  
x.toFixed(6);
```

toFixed(2) is perfect for working with money.

The **toPrecision()** Method

toPrecision() returns a string, with a number written with a specified length:

Example

```
let x = 9.656;  
x.toPrecision();  
x.toPrecision(2);  
x.toPrecision(4);  
x.toPrecision(6);
```

The valueOf() Method

`valueOf()` returns a number as a number.

Example

```
let x = 123;  
x.valueOf();  
(123).valueOf();  
(100 + 23).valueOf();
```

In JavaScript, a number can be a primitive value (`typeof = number`) or an object (`typeof = object`).

The `valueOf()` method is used internally in JavaScript to convert Number objects to primitive values.

There is no reason to use it in your code.

All JavaScript data types have a `valueOf()` and a `toString()` method.

Converting Variables to Numbers

There are 3 JavaScript methods that can be used to convert a variable to a number:

Method	Description
Number()	Returns a number converted from its argument.
parseFloat()	Parses its argument and returns a floating point number
parseInt()	Parses its argument and returns a whole number

The methods above are not **number** methods. They are **global** JavaScript methods.

The Number() Method

The `Number()` method can be used to convert JavaScript variables to numbers:

Example

```
Number(true);
Number(false);
Number("10");
Number(" 10");
Number("10  ");
Number(" 10  ");
Number("10.33");
Number("10,33");
Number("10 33");
Number("John");
```

If the number cannot be converted, `NaN` (Not a Number) is returned.

The Number() Method Used on Dates

`Number()` can also convert a date to a number.

Example

```
Number(new Date("1970-01-01"))
```

Note

The `Date()` method returns the number of milliseconds since 1.1.1970.

The number of milliseconds between 1970-01-02 and 1970-01-01 is 86400000:

Example

```
Number(new Date("1970-01-02"))
```

Example

```
Number(new Date("2017-09-30"))
```

The parseInt() Method

`parseInt()` parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

Example

```
parseInt("-10");
parseInt("-10.33");
parseInt("10");
parseInt("10.33");
parseInt("10 20 30");
parseInt("10 years");
parseInt("years 10");
```

If the number cannot be converted, `NaN` (Not a Number) is returned.

The parseFloat() Method

`parseFloat()` parses a string and returns a number. Spaces are allowed. Only the first number is returned:

Example

```
parseFloat("10");
parseFloat("10.33");
parseFloat("10 20 30");
parseFloat("10 years");
parseFloat("years 10");
```

If the number cannot be converted, `NaN` (Not a Number) is returned.

Number Object Methods

These **object methods** belong to the **Number** object:

Method	Description
Number.isInteger()	Returns true if the argument is an integer
Number.isSafeInteger()	Returns true if the argument is a safe integer
Number.parseFloat()	Converts a string to a number
Number.parseInt()	Converts a string to a whole number

Number Methods Cannot be Used on Variables

The number methods above belong to the JavaScript **Number Object**.

These methods can only be accessed like `Number.isInteger()`.

Using `X.isInteger()` where X is a variable, will result in an error:

`TypeError X.isInteger is not a function.`

The Number.isInteger() Method

The `Number.isInteger()` method returns `true` if the argument is an integer.

Example

```
Number.isInteger(10);
Number.isInteger(10.5);
```

The Number.isSafeInteger() Method

A safe integer is an integer that can be exactly represented as a double precision number. The `Number.isSafeInteger()` method returns `true` if the argument is a safe integer.

Example

```
Number.isSafeInteger(10);  
Number.isSafeInteger(12345678901234567890);
```

Safe integers are all integers from $-(2^{53} - 1)$ to $+(2^{53} - 1)$.
This is safe: 9007199254740991. This is not safe: 9007199254740992.

The Number.parseFloat() Method

`Number.parseFloat()` parses a string and returns a number.
Spaces are allowed. Only the first number is returned:

Example

```
Number.parseFloat("10");  
Number.parseFloat("10.33");  
Number.parseFloat("10 20 30");  
Number.parseFloat("10 years");  
Number.parseFloat("years 10");
```

If the number cannot be converted, `NaN` (Not a Number) is returned.

Note

The **Number** methods `Number.parseInt()` and `Number.parseFloat()`

are the same as the

Global methods `parseInt()` and `parseFloat()`.

The purpose is modularization of globals (to make it easier to use the same JavaScript code outside the browser).

The Number.parseInt() Method

`Number.parseInt()` parses a string and returns a whole number.
Spaces are allowed. Only the first number is returned:

Example

```
Number.parseInt("-10");  
Number.parseInt("-10.33");  
Number.parseInt("10");  
Number.parseInt("10.33");  
Number.parseInt("10 20 30");  
Number.parseInt("10 years");  
Number.parseInt("years 10");
```

If the number cannot be converted, `NaN` (Not a Number) is returned.

26. JavaScript Number Properties

Property	Description
EPSILON	The difference between 1 and the smallest number > 1.
MAX_VALUE	The largest number possible in JavaScript
MIN_VALUE	The smallest number possible in JavaScript
MAX_SAFE_INTEGER	The maximum safe integer ($2^{53} - 1$)
MIN_SAFE_INTEGER	The minimum safe integer $-(2^{53} - 1)$
POSITIVE_INFINITY	Infinity (returned on overflow)
NEGATIVE_INFINITY	Negative infinity (returned on overflow)
NaN	A "Not-a-Number" value

JavaScript EPSILON

Number.EPSILON is the difference between the smallest floating point number greater than 1 and 1.

Example

```
let x = Number.EPSILON;
```

Note

Number.EPSILON is an [ES6](#) feature.
It does not work in Internet Explorer.

JavaScript MAX_VALUE

`Number.MAX_VALUE` is a constant representing the largest possible number in JavaScript.

Example

```
let x = Number.MAX_VALUE;
```

Number Properties Cannot be Used on Variables

Number properties belong to the JavaScript **Number Object**.

These properties can only be accessed as `Number.MAX_VALUE`.

Using `x.MAX_VALUE`, where `x` is a variable or a value, will return `undefined`:

Example

```
let x = 6;  
x.MAX_VALUE
```

JavaScript MIN_VALUE

`Number.MIN_VALUE` is a constant representing the lowest possible number in JavaScript.

Example

```
let x = Number.MIN_VALUE;
```

JavaScript MAX_SAFE_INTEGER

`Number.MAX_SAFE_INTEGER` represents the maximum safe integer in JavaScript.

`Number.MAX_SAFE_INTEGER` is $(2^{53} - 1)$.

Example

```
let x = Number.MAX_SAFE_INTEGER;
```

JavaScript MIN_SAFE_INTEGER

`Number.MIN_SAFE_INTEGER` represents the minimum safe integer in JavaScript.

`Number.MIN_SAFE_INTEGER` is $-(2^{53} - 1)$.

Example

```
let x = Number.MIN_SAFE_INTEGER;
```

Note

`MAX_SAFE_INTEGER` and `MIN_SAFE_INTEGER` are [ES6](#) features.

They do not work in Internet Explorer.

JavaScript **POSITIVE_INFINITY**

Example

```
let x = Number.POSITIVE_INFINITY;
```

POSITIVE_INFINITY is returned on overflow:

```
let x = 1 / 0;
```

JavaScript **NEGATIVE_INFINITY**

Example

```
let x = Number.NEGATIVE_INFINITY;
```

NEGATIVE_INFINITY is returned on overflow:

```
let x = -1 / 0;
```

JavaScript **NaN - Not a Number**

NaN is a JavaScript reserved word for a number that is not a legal number.

Examples

```
let x = Number.NaN;
```

Trying to do arithmetic with a non-numeric string will result in **NaN** (Not a Number):

```
let x = 100 / "Apple";
```

27. JavaScript Arrays

An array is a special variable, which can hold more than one value:

```
const cars = ["Saab", "Volvo", "BMW"];
```

Why Use Arrays?

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
let car1 = "Saab";
let car2 = "Volvo";
let car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

```
const array_name = [item1, item2, ...];
```

It is a common practice to declare arrays with the `const` keyword.

Learn more about `const` with arrays in the chapter: [JS Array Const.](#)

Example

```
const cars = ["Saab", "Volvo", "BMW"];
```

Spaces and line breaks are not important. A declaration can span multiple lines:

Example

```
const cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

You can also create an array, and then provide the elements:

Example

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example

```
const cars = new Array("Saab", "Volvo", "BMW");
```

The two examples above do exactly the same.

There is no need to use `new Array()`.

For simplicity, readability and execution speed, use the array literal method.

Accessing Array Elements

You access an array element by referring to the **index number**:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

Note: Array indexes start with 0.

[0] is the first element. [1] is the second element.

Changing an Array Element

This statement changes the value of the first element in `cars`:

```
cars[0] = "Opel";
```

Example

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
```

Converting an Array to a String

The JavaScript method `toString()` converts an array to a string of (comma separated) array values.

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Result:

Banana,Orange,Apple,Mango

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

Example

```
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

Arrays are Objects

Arrays are a special type of objects. The `typeof` operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

Arrays use **numbers** to access its "elements". In this example, `person[0]` returns John:

Array:

```
const person = ["John", "Doe", 46];
```

Objects use **names** to access its "members". In this example, `person.firstName` returns John:

Object:

```
const person = {firstName:"John", lastName:"Doe", age:46};
```

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.

Because of this, you can have variables of different types in the same Array.

You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;  
myArray[1] = myFunction;  
myArray[2] = myCars;
```

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

```
cars.length // Returns the number of elements  
cars.sort() // Sorts the array
```

Array methods are covered in the next chapters.

The length Property

The `length` property of an array returns the length of an array (the number of array elements).

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let length = fruits.length;
```

The `length` property is always one more than the highest array index.

Accessing the First Array Element

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits[0];
```

Accessing the Last Array Element

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits[fruits.length - 1];
```

Looping Array Elements

One way to loop through an array, is using a `for` loop:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;
let text = "<ul>";
for (let i = 0; i < fLen; i++) {
    text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

You can also use the `Array.forEach()` function:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
function myFunction(value) {
    text += "<li>" + value + "</li>";
}
```

Adding Array Elements

The easiest way to add a new element to an array is using the `push()` method:

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits.push("Lemon"); // Adds a new element (Lemon) to fruits
```

New element can also be added to an array using the `length` property:

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[fruits.length] = "Lemon"; // Adds "Lemon" to fruits
```

WARNING !

Adding elements with high indexes can create undefined "holes" in an array:

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[6] = "Lemon"; // Creates undefined "holes" in fruits
```

Associative Arrays

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** always use **numbered indexes**.

Example

```
const person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
person.length;      // Will return 3
person[0];          // Will return "John"
```

WARNING !!

If you use named indexes, JavaScript will redefine the array to an object.

After that, some array methods and properties will produce **incorrect results**.

Example:

```
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
person.length;      // Will return 0
person[0];          // Will return undefined
```

The Difference Between Arrays and Objects

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.

Arrays are a special kind of objects, with numbered indexes.

When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

JavaScript new Array()

JavaScript has a built-in array constructor `new Array()`.

But you can safely use `[]` instead.

These two different statements both create a new empty array named `points`:

```
const points = new Array();
const points = [];
```

These two different statements both create a new array containing 6 numbers:

```
const points = new Array(40, 100, 1, 5, 25, 10);
const points = [40, 100, 1, 5, 25, 10];
```

The `new` keyword can produce some unexpected results:

```
// Create an array with three elements:
const points = new Array(40, 100, 1);

// Create an array with two elements:
const points = new Array(40, 100);

// Create an array with one element ????
const points = new Array(40);
```

A Common Error

```
const points = [40];
```

is not the same as:

```
const points = new Array(40);

// Create an array with one element:
const points = [40];

// Create an array with 40 undefined elements:
const points = new Array(40);
```

How to Recognize an Array

A common question is: How do I know if a variable is an array?

The problem is that the JavaScript operator `typeof` returns "object":

```
const fruits = ["Banana", "Orange", "Apple"];
let type = typeof fruits;
```

The `typeof` operator returns object because a JavaScript array is an object.

Solution 1:

To solve this problem ECMAScript 5 (JavaScript 2009) defined a new method `Array.isArray()`:

```
Array.isArray(fruits);
```

Solution 2:

The `instanceof` operator returns true if an object is created by a given constructor:

```
const fruits = ["Banana", "Orange", "Apple"];  
(fruits instanceof Array);
```

Nested Arrays and Objects

Values in objects can be arrays, and values in arrays can be objects:

Example

```
const myObj = {  
    name: "John",  
    age: 30,  
    cars: [  
        {name: "Ford", models:["Fiesta", "Focus", "Mustang"]},  
        {name: "BMW", models:["320", "X3", "X5"]},  
        {name: "Fiat", models:["500", "Panda"]}  
    ]  
}
```

To access arrays inside arrays, use a for-in loop for each array:

Example

```
for (let i in myObj.cars) {  
    x += "<h1>" + myObj.cars[i].name + "</h1>";  
    for (let j in myObj.cars[i].models) {  
        x += myObj.cars[i].models[j];  
    }  
}
```

28. JavaScript Array Methods

Basic Array Methods

[Array length](#)
[Array toString\(\)](#)
[Array at\(\)](#)
[Array join\(\)](#)
[Array pop\(\)](#)
[Array push\(\)](#)

[Array shift\(\)](#)
[Array unshift\(\)](#)
[Array delete\(\)](#)
[Array concat\(\)](#)
[Array copyWithin\(\)](#)
[Array flat\(\)](#)
[Array splice\(\)](#)
[Array toSpliced\(\)](#)
[Array slice\(\)](#)

See Also:

[Search Methods](#)
[Sort Methods](#)
[Iteration Methods](#)

JavaScript Array length

The `length` property returns the length (size) of an array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let size = fruits.length;
```

JavaScript Array toString()

The JavaScript method `toString()` converts an array to a string of (comma separated) array values.

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Result:

Banana,Orange,Apple,Mango

JavaScript Array at()

[ES2022](#) introduced the array method `at()`:

Examples

Get the third element of fruits using `at()`:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits.at(2);
```

Get the third element of fruits using `[]`:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[2];
```

The `at()` method returns an indexed element from an array.

The `at()` method returns the same as `[]`.

The `at()` method is supported in all modern browsers since March 2022:

				
Chrome 92	Edge 92	Firefox 90	Safari 15.4	Opera 78
Apr 2021	Jul 2021	Jul 2021	Mar 2022	Aug 2021

Note

Many languages allows `negative bracket indexing` like `[-1]` to access elements from the end of an object / array / string.

This is not possible in JavaScript, because `[]` is used for accessing both arrays and objects. `obj[-1]` refers to the value of key `-1`, not to the last property of the object.

The `at()` method was introduced in ES2022 to solve this problem.

JavaScript Array join()

The `join()` method also joins all array elements into a string.

It behaves just like `toString()`, but in addition you can specify the separator:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
```

Result:

Banana * Orange * Apple * Mango

Popping and Pushing

When you work with arrays, it is easy to remove elements and add new elements.

This is what popping and pushing is:

Popping items **out** of an array, or pushing items **into** an array.

JavaScript Array pop()

The `pop()` method removes the last element from an array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();
```

The `pop()` method returns the value that was "popped out":

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits.pop();
```

JavaScript Array push()

The `push()` method adds a new element to an array (at the end):

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");
```

The `push()` method returns the new array length:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.push("Kiwi");
```

Shifting Elements

Shifting is equivalent to popping, but working on the first element instead of the last.

JavaScript Array shift()

The `shift()` method removes the first array element and "shifts" all other elements to a lower index.

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();
```

The `shift()` method returns the value that was "shifted out":

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits.shift();
```

JavaScript Array unshift()

The `unshift()` method adds a new element to an array (at the beginning), and "unshifts" older elements:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
```

The `unshift()` method returns the new array length:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
```

Changing Elements

Array elements are accessed using their **index number**:

Array **indexes** start with 0:
[0] is the first array element
[1] is the second
[2] is the third ...

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[0] = "Kiwi";
```

JavaScript Array length

The `length` property provides an easy way to append a new element to an array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Kiwi";
```

JavaScript Array delete()

Warning !

Using `delete()` leaves `undefined` holes in the array.
Use `pop()` or `shift()` instead.

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];
```

Merging Arrays (Concatenating)

In programming languages, concatenation means joining strings end-to-end. Concatenation "snow" and "ball" gives "snowball". Concatenating arrays means joining arrays end-to-end.

JavaScript Array concat()

The `concat()` method creates a new array by merging (concatenating) existing arrays:

Example (Merging Two Arrays)

```
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];

const myChildren = myGirls.concat(myBoys);
```

Note

The `concat()` method does not change the existing arrays. It always returns a new array. The `concat()` method can take any number of array arguments.

Example (Merging Three Arrays)

```
const arr1 = ["Cecilie", "Lone"];
const arr2 = ["Emil", "Tobias", "Linus"];
const arr3 = ["Robin", "Morgan"];
const myChildren = arr1.concat(arr2, arr3);
```

The `concat()` method can also take strings as arguments:

Example (Merging an Array with Values)

```
const arr1 = ["Emil", "Tobias", "Linus"];
const myChildren = arr1.concat("Peter");
```

Array copyWithin()

The `copyWithin()` method copies array elements to another position in an array:

Examples

Copy to index 2, all elements from index 0:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.copyWithin(2, 0);
```

Copy to index 2, the elements from index 0 to 2:

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi"];
fruits.copyWithin(2, 0, 2);
```

Note

The `copyWithin()` method overwrites the existing values.

The `copyWithin()` method does not add items to the array.

The `copyWithin()` method does not change the length of the array.

Flattening an Array

Flattening an array is the process of reducing the dimensionality of an array.

Flattening is useful when you want to convert a multi-dimensional array into a one-dimensional array.

JavaScript Array flat()

[ES2019](#) Introduced the Array flat() method.

The `flat()` method creates a new array with sub-array elements concatenated to a specified depth.

Example

```
const myArr = [[1,2],[3,4],[5,6]];
const newArr = myArr.flat();
```

Browser Support

JavaScript Array `flat()` is supported in all modern browsers since January 2020:

				
Chrome 69	Edge 79	Firefox 62	Safari 12	Opera 56
Sep 2018	Jan 2020	Sep 2018	Sep 2018	Sep 2018

JavaScript Array flatMap()

ES2019 added the Array `flatMap()` method to JavaScript.

The `flatMap()` method first maps all elements of an array and then creates a new array by flattening the array.

Example

```
const myArr = [1, 2, 3, 4, 5, 6];
const newArr = myArr.flatMap(x => [x, x * 10]);
```

Browser Support

JavaScript Array `flatMap()` is supported in all modern browsers since January 2020:

				
Chrome 69	Edge 79	Firefox 62	Safari 12	Opera 56
Sep 2018	Jan 2020	Sep 2018	Sep 2018	Sep 2018

Splicing and Slicing Arrays

The `splice()` method adds new items to an array.

The `slice()` method slices out a piece of an array.

JavaScript Array `splice()`

The `splice()` method can be used to add new items to an array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

The `splice()` method returns an array with the deleted items:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 2, "Lemon", "Kiwi");
```

Using `splice()` to Remove Elements

With clever parameter setting, you can use `splice()` to remove elements without leaving "holes" in the array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);
```

The first parameter (0) defines the position where new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

JavaScript Array toSpliced()

[ES2023](#) added the Array `toSpliced()` method as a safe way to splice an array without altering the original array.

The difference between the new **toSpliced()** method and the old **splice()** method is that the new method creates a new array, keeping the original array unchanged, while the old method altered the original array.

Example

```
const months = ["Jan", "Feb", "Mar", "Apr"];
const spliced = months.toSpliced(0, 1);
```

JavaScript Array slice()

The `slice()` method slices out a piece of an array into a new array:

Example

Slice out a part of an array starting from array element 1 ("Orange"):

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1);
```

Note

The `slice()` method creates a new array.

The `slice()` method does not remove any elements from the source array.

Example

Slice out a part of an array starting from array element 3 ("Apple"):

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(3);
```

The `slice()` method can take two arguments like `slice(1, 3)`.

The method then selects elements from the start argument, and up to (but not including) the end argument.

Example

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3);
```

If the end argument is omitted, like in the first examples, the `slice()` method slices out the rest of the array.

Example

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(2);
```

Automatic `toString()`

JavaScript automatically converts an array to a comma separated string when a primitive value is expected.

This is always the case when you try to output an array.

These two examples will produce the same result:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
```

Note

All JavaScript objects have a `toString()` method.

29. JavaScript Array Search

Array Find and Search Methods

[Array indexOf\(\)](#)
[Array lastIndexOf\(\)](#)
[Array includes\(\)](#)

[Array find\(\)](#)
[Array findIndex\(\)](#)
[Array findLast\(\)](#)
[Array findLastIndex\(\)](#)

See Also:

[Basic Methods](#)
[Sort Methods](#)
[Iteration Methods](#)

JavaScript Array indexOf()

The `indexOf()` method searches an array for an element value and returns its position.

Note: The first item has position 0, the second item has position 1, and so on.

Example

Search an array for the item "Apple":

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.indexOf("Apple") + 1;
```

Syntax

`array.indexOf(item, start)`

item Required. The item to search for.

start Optional. Where to start the search. Negative values will start at the given position counting from the end, and search to the end.

`Array.indexOf()` returns -1 if the item is not found.

If the item is present more than once, it returns the position of the first occurrence.

JavaScript Array lastIndexOf()

`Array.lastIndexOf()` is the same as `Array.indexOf()`, but returns the position of the last occurrence of the specified element.

Example

Search an array for the item "Apple":

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.lastIndexOf("Apple") + 1;
```

Syntax

`array.lastIndexOf(item, start)`

item Required. The item to search for

start Optional. Where to start the search. Negative values will start at the given position counting from the end, and search to the beginning

JavaScript Array includes()

ECMAScript 2016 introduced `Array.includes()` to arrays. This allows us to check if an element is present in an array (including NaN, unlike indexOf).

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.includes("Mango"); // is true
```

Syntax

`array.includes(search-item)`

`Array.includes()` allows to check for NaN values. Unlike `Array.indexOf()`.

Browser Support

`includes()` is an [ECMAScript 2016](#) feature.

ES 2016 is fully supported in all modern browsers since March 2017:

				
Chrome 52	Edge 15	Firefox 52	Safari 10.1	Opera 39
Jul 2016	Apr 2017	Mar 2017	May 2017	Aug 2016

`includes()` is not supported in Internet Explorer.

JavaScript Array `find()`

The `find()` method returns the value of the first array element that passes a test function.

This example finds (returns the value of) the first element that is larger than 18:

Example

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Browser Support

`find()` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`find()` is not supported in Internet Explorer.

JavaScript Array `findIndex()`

The `findIndex()` method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

Example

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Browser Support

`findIndex()` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`findIndex()` is not supported in Internet Explorer.

JavaScript Array `findLast()` Method

ES2023 added the `findLast()` method that will start from the end of an array and return the value of the first element that satisfies a condition.

Example

```
const temp = [27, 28, 30, 40, 42, 35, 30];
let high = temp.findLast(x => x > 40);
```

Browser Support

`findLast()` is an ES2023 feature.

It is supported in all modern browsers since July 2023:

				
Chrome 110	Edge 110	Firefox 115	Safari 16.4	Opera 96
Feb 2023	Feb 2023	Jul 2023	Mar 2023	May 2023

JavaScript Array `findLastIndex()` Method

The `findLastIndex()` method finds the index of the last element that satisfies a condition.

Example

```
const temp = [27, 28, 30, 40, 42, 35, 30];
let pos = temp.findLastIndex(x => x > 40);
```

Browser Support

`findLastIndex()` is an ES2023 feature.

It is supported in all modern browsers since July 2023:

				
Chrome 110	Edge 110	Firefox 115	Safari 16.4	Opera 96
Feb 2023	Feb 2023	Jul 2023	Mar 2023	May 2023

30. JavaScript Sorting Arrays

Array Sort Methods

Alphabetic Sort

[Array sort\(\)](#)
[Array reverse\(\)](#)
[Array toSorted\(\)](#)
[Array toReversed\(\)](#)
[Sorting Objects](#)

Numeric Sort

[Numeric Sort](#)
[Random Sort](#)
[Math.min\(\)](#)
[Math.max\(\)](#)
[Home made Min\(\)](#)
[Home made Max\(\)](#)

See Also:

[Basic Methods](#)
[Search Methods](#)
[Iteration Methods](#)

Sorting an Array

The `sort()` method sorts an array alphabetically:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
```

Reversing an Array

The `reverse()` method reverses the elements in an array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.reverse();
```

By combining `sort()` and `reverse()`, you can sort an array in descending order:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
fruits.reverse();
```

JavaScript Array toSorted() Method

[ES2023](#) added the `toSorted()` method as a safe way to sort an array without altering the original array.

The difference between `toSorted()` and `sort()` is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

Example

```
const months = ["Jan", "Feb", "Mar", "Apr"];
const sorted = months.toSorted();
```

JavaScript Array toReversed() Method

[ES2023](#) added the `toReversed()` method as a safe way to reverse an array without altering the original array.

The difference between `toReversed()` and `reverse()` is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

Example

```
const months = ["Jan", "Feb", "Mar", "Apr"];
const reversed = months.toReversed();
```

Numeric Sort

By default, the `sort()` function sorts values as **strings**.

This works well for strings ("Apple" comes before "Banana").

If numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the `sort()` method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

Use the same trick to sort an array descending:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
```

The Compare Function

The purpose of the compare function is to define an alternative sort order.

The compare function should return a negative, zero, or positive value, depending on the arguments:

```
function(a, b){return a - b}
```

When the `sort()` function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative, `a` is sorted before `b`.

If the result is positive, `b` is sorted before `a`.

If the result is 0, no changes are done with the sort order of the two values.

Example:

The compare function compares all the values in the array, two values at a time (`a, b`).

When comparing 40 and 100, the `sort()` method calls the compare function(40, 100).

The function calculates $40 - 100$ (`a - b`), and since the result is negative (-60), the sort function will sort 40 as a value lower than 100.

You can use this code snippet to experiment with numerically and alphabetically sorting:

```
<button onclick="myFunction1()">Sort Alphabetically</button>
<button onclick="myFunction2()">Sort Numerically</button>

<p id="demo"></p>

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction1() {
  points.sort();
  document.getElementById("demo").innerHTML = points;
}

function myFunction2() {
  points.sort(function(a, b){return a - b});
  document.getElementById("demo").innerHTML = points;
}
</script>
```

Sorting an Array in Random Order

Using a sort function, like explained above, you can sort an numeric array in random order

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(){return 0.5 - Math.random()});
```

The Fisher Yates Method

The points.sort() method in the example above is not accurate. It will favor some numbers over others.

The most popular correct method, is called the Fisher Yates shuffle, and was introduced in data science as early as 1938!

In JavaScript the method can be translated to this:

Example

```
const points = [40, 100, 1, 5, 25, 10];

for (let i = points.length -1; i > 0; i--) {
  let j = Math.floor(Math.random() * (i+1));
  let k = points[i];
  points[i] = points[j];
  points[j] = k;
}
```

Find the Lowest (or Highest) Array Value

There are no built-in functions for finding the max or min value in an array.

To find the lowest or highest value you have 3 options:

- Sort the array and read the first or last element
- Use Math.min() or Math.max()
- Write a home made function

Find Min or Max with sort()

After you have sorted an array, you can use the index to obtain the highest and lowest values.

Sort Ascending:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
// now points[0] contains the lowest value
// and points[points.length-1] contains the highest value
```

Sort Descending:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
// now points[0] contains the highest value
// and points[points.length-1] contains the lowest value
```

Note

Sorting a whole array is a very inefficient method if you only want to find the highest (or lowest) value.

Using Math.min() on an Array

You can use `Math.min.apply` to find the lowest number in an array:

Example

```
function myArrayMin(arr) {  
    return Math.min.apply(null, arr);  
}
```

`Math.min.apply(null, [1, 2, 3])` is equivalent to `Math.min(1, 2, 3)`.

Using Math.max() on an Array

You can use `Math.max.apply` to find the highest number in an array:

Example

```
function myArrayMax(arr) {  
    return Math.max.apply(null, arr);  
}
```

`Math.max.apply(null, [1, 2, 3])` is equivalent to `Math.max(1, 2, 3)`.

JavaScript Array Minimum Method

There is no built-in function for finding the lowest value in a JavaScript array.

The fastest code to find the lowest number is to use a **home made** method.

This function loops through an array comparing each value with the lowest value found:

Example (Find Min)

```
function myArrayMin(arr) {  
    let len = arr.length;  
    let min = Infinity;  
    while (len--) {  
        if (arr[len] < min) {  
            min = arr[len];  
        }  
    }  
    return min;  
}
```

JavaScript Array Maximum Method

There is no built-in function for finding the highest value in a JavaScript array.

The fastest code to find the highest number is to use a **home made** method.

This function loops through an array comparing each value with the highest value found:

Example (Find Max)

```
function myArrayMax(arr) {  
    let len = arr.length;  
    let max = -Infinity;  
    while (len--) {  
        if (arr[len] > max) {  
            max = arr[len];  
        }  
    }  
    return max;  
}
```

Sorting Object Arrays

JavaScript arrays often contain objects:

Example

```
const cars = [  
    {type:"Volvo", year:2016},  
    {type:"Saab", year:2001},  
    {type:"BMW", year:2010}  
];
```

Even if objects have properties of different data types, the `sort()` method can be used to sort the array.

The solution is to write a compare function to compare the property values:

Example

```
cars.sort(function(a, b){return a.year - b.year});
```

Comparing string properties is a little more complex:

Example

```
cars.sort(function(a, b){  
    let x = a.type.toLowerCase();  
    let y = b.type.toLowerCase();  
    if (x < y) {return -1;}  
    if (x > y) {return 1;}  
    return 0;  
});
```

Stable Array sort()

[ES2019 revised](#) the Array `sort()` method.

Before 2019, the specification allowed unstable sorting algorithms such as QuickSort.

After ES2019, browsers must use a stable sorting algorithm:

When sorting elements on a value, the elements must keep their relative position to other elements with the same value.

Example

```
const myArr = [
  {name:"X00",price:100 },
  {name:"X01",price:100 },
  {name:"X02",price:100 },
  {name:"X03",price:100 },
  {name:"X04",price:110 },
  {name:"X05",price:110 },
  {name:"X06",price:110 },
  {name:"X07",price:110 }
];
```

In the example above, when sorting on price, the result is not allowed to come out with the names in an other relative position like this:

```
X01 100
X03 100
X00 100
X03 100
X05 110
X04 110
X06 110
X07 110
```

31. JavaScript Array Iteration

Array Iteration Methods

Array iteration methods operate on every array item:

[Array forEach](#)
[Array map\(\)](#)
[Array flatMap\(\)](#)
[Array filter\(\)](#)
[Array reduce\(\)](#)
[Array reduceRight\(\)](#)

[Array every\(\)](#)
[Array some\(\)](#)
[Array from\(\)](#)
[Array keys\(\)](#)
[Array entries\(\)](#)
[Array with\(\)](#)
[Array Spread \(...\)](#)

See Also:

[Basic Array Methods](#)
[Array Search Methods](#)
[Array Sort Methods](#)

JavaScript Array forEach()

The `forEach()` method calls a function (a callback function) once for each array element.

Example

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);
function myFunction(value, index, array) {
  txt += value + "<br>";
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

The example above uses only the value parameter. The example can be rewritten to:

Example

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);

function myFunction(value) {
  txt += value + "<br>";
}
```

JavaScript Array map()

The `map()` method creates a new array by performing a function on each array element.

The `map()` method does not execute the function for array elements without values.

The `map()` method does not change the original array.

This example multiplies each array value by 2:

Example

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

function myFunction(value, index, array) {
  return value * 2;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

When a callback function uses only the value parameter, the index and array parameters can be omitted:

Example

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

function myFunction(value) {
  return value * 2;
}
```

JavaScript Array flatMap()

[ES2019](#) added the Array `flatMap()` method to JavaScript.

The `flatMap()` method first maps all elements of an array and then creates a new array by flattening the array.

Example

```
const myArr = [1, 2, 3, 4, 5, 6];
const newArr = myArr.flatMap((x) => x * 2);
```

Browser Support

JavaScript Array `flatMap()` is supported in all modern browsers since January 2020:

				
Chrome 69	Edge 79	Firefox 62	Safari 12	Opera 56
Sep 2018	Jan 2020	Sep 2018	Sep 2018	Sep 2018

JavaScript Array filter()

The `filter()` method creates a new array with array elements that pass a test.

This example creates a new array from elements with a value larger than 18:

Example

```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

In the example above, the callback function does not use the index and array parameters, so they can be omitted:

Example

```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

function myFunction(value) {
  return value > 18;
}
```

JavaScript Array reduce()

The `reduce()` method runs a function on each array element to produce (reduce it to) a single value.

The `reduce()` method works from left-to-right in the array. See also `reduceRight()`.

The `reduce()` method does not reduce the original array.

This example finds the sum of all numbers in an array:

Example

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);

function myFunction(total, value, index, array) {
  return total + value;
}
```

Note that the function takes 4 arguments:

- The total (the initial value / previously returned value)
- The item value
- The item index
- The array itself

The example above does not use the index and array parameters. It can be rewritten to:

Example

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);

function myFunction(total, value) {
  return total + value;
}
```

The `reduce()` method can accept an initial value:

Example

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction, 100);

function myFunction(total, value) {
  return total + value;
}
```

JavaScript Array reduceRight()

The `reduceRight()` method runs a function on each array element to produce (reduce it to) a single value.

The `reduceRight()` works from right-to-left in the array. See also `reduce()`.

The `reduceRight()` method does not reduce the original array.

This example finds the sum of all numbers in an array:

Example

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduceRight(myFunction);

function myFunction(total, value, index, array) {
  return total + value;
}
```

Note that the function takes 4 arguments:

- The total (the initial value / previously returned value)
- The item value
- The item index
- The array itself

The example above does not use the index and array parameters. It can be rewritten to:

Example

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduceRight(myFunction);

function myFunction(total, value) {
  return total + value;
}
```

JavaScript Array every()

The `every()` method checks if all array values pass a test.

This example checks if all array values are larger than 18:

Example

```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

When a callback function uses the first parameter only (value), the other parameters can be omitted:

Example

```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction);

function myFunction(value) {
  return value > 18;
}
```

JavaScript Array some()

The `some()` method checks if some array values pass a test.

This example checks if some array values are larger than 18:

Example

```
const numbers = [45, 4, 9, 16, 25];
let someOver18 = numbers.some(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

JavaScript Array.from()

The `Array.from()` method returns an Array object from any object with a length property or any iterable object.

Example

Create an Array from a String:

```
Array.from("ABCDEFG");
```

Browser Support

`from()` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`from()` is not supported in Internet Explorer.

JavaScript Array keys()

The `Array.keys()` method returns an Array Iterator object with the keys of an array.

Example

Create an Array Iterator object, containing the keys of the array:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const keys = fruits.keys();

for (let x of keys) {
  text += x + "<br>";
}
```

Browser Support

`keys()` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`keys()` is not supported in Internet Explorer.

JavaScript Array entries()

Example

Create an Array Iterator, and then iterate over the key/value pairs:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const f = fruits.entries();

for (let x of f) {
  document.getElementById("demo").innerHTML += x;
}
```

The `entries()` method returns an Array Iterator object with key/value pairs:

```
[0, "Banana"]
[1, "Orange"]
[2, "Apple"]
[3, "Mango"]
```

The `entries()` method does not change the original array.

Browser Support

`entries()` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`entries()` is not supported in Internet Explorer.

JavaScript Array with() Method

[ES2023](#) added the Array `with()` method as a safe way to update elements in an array without altering the original array.

Example

```
const months = ["Januar", "Februar", "Mar", "April"];
const myMonths = months.with(2, "March");
```

JavaScript Array Spread (...)

The ... operator expands an iterable (like an array) into more elements:

Example

```
const q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const q3 = ["Jul", "Aug", "Sep"];
const q4 = ["Oct", "Nov", "May"];

const year = [...q1, ...q2, ...q3, ...q4];
```

Browser Support

... is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

... is not supported in Internet Explorer.

32. JavaScript Array Const

ECMAScript 2015 (ES6)

In 2015, JavaScript introduced an important new keyword: `const`.

It has become a common practice to declare arrays using `const`:

Example

```
const cars = ["Saab", "Volvo", "BMW"];
```

Cannot be Reassigned

An array declared with `const` cannot be reassigned:

Example

```
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"]; // ERROR
```

Arrays are Not Constants

The keyword `const` is a little misleading.

It does NOT define a constant array. It defines a constant reference to an array.

Because of this, we can still change the elements of a constant array.

Elements Can be Reassigned

You can change the elements of a constant array:

Example

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

Browser Support

The `const` keyword is not supported in Internet Explorer 10 or earlier.
The following table defines the first browser versions with full support for the `const` keyword:

				
Chrome 49	IE 11 / Edge	Firefox 36	Safari 10	Opera 36
Mar, 2016	Oct, 2013	Feb, 2015	Sep, 2016	Mar, 2016

Assigned when Declared

JavaScript `const` variables must be assigned a value when they are declared:

Meaning: An array declared with `const` must be initialized when it is declared.

Using `const` without initializing the array is a syntax error:

Example

This will not work:

```
const cars;  
cars = ["Saab", "Volvo", "BMW"];
```

Arrays declared with `var` can be initialized at any time.

You can even use the array before it is declared:

Example

This is OK:

```
cars = ["Saab", "Volvo", "BMW"];  
var cars;
```

Const Block Scope

An array declared with `const` has **Block Scope**.

An array declared in a block is not the same as an array declared outside the block:

Example

```
const cars = ["Saab", "Volvo", "BMW"];  
// Here cars[0] is "Saab"  
{  
  const cars = ["Toyota", "Volvo", "BMW"];  
  // Here cars[0] is "Toyota"  
}  
// Here cars[0] is "Saab"
```

An array declared with `var` does not have block scope:

Example

```
var cars = ["Saab", "Volvo", "BMW"];
// Here cars[0] is "Saab"
{
  var cars = ["Toyota", "Volvo", "BMW"];
  // Here cars[0] is "Toyota"
}
// Here cars[0] is "Toyota"
```

Redeclaring Arrays

Redeclaring an array declared with `var` is allowed anywhere in a program:

Example

```
var cars = ["Volvo", "BMW"]; // Allowed
var cars = ["Toyota", "BMW"]; // Allowed
cars = ["Volvo", "Saab"]; // Allowed
```

Redeclaring or reassigning an array to `const`, in the same scope, or in the same block, is not allowed:

Example

```
var cars = ["Volvo", "BMW"]; // Allowed
const cars = ["Volvo", "BMW"]; // Not allowed
{
  var cars = ["Volvo", "BMW"]; // Allowed
  const cars = ["Volvo", "BMW"]; // Not allowed
}
```

Redeclaring or reassigning an existing `const` array, in the same scope, or in the same block, is not allowed:

Example

```
const cars = ["Volvo", "BMW"]; // Allowed
const cars = ["Volvo", "BMW"]; // Not allowed
var cars = ["Volvo", "BMW"]; // Not allowed
cars = ["Volvo", "BMW"]; // Not allowed
{
  const cars = ["Volvo", "BMW"]; // Allowed
  const cars = ["Volvo", "BMW"]; // Not allowed
  var cars = ["Volvo", "BMW"]; // Not allowed
  cars = ["Volvo", "BMW"]; // Not allowed
}
```

Redeclaring an array with `const`, in another scope, or in another block, is allowed:

Example

```
const cars = ["Volvo", "BMW"]; // Allowed
{
  const cars = ["Volvo", "BMW"]; // Allowed
}
{
  const cars = ["Volvo", "BMW"]; // Allowed
}
```

33. JavaScript Date Objects

JavaScript **Date Objects** let us work with dates:

Sun Oct 13 2024 20:17:33 GMT+0530 (India Standard Time)

Year: 2024 Month: 10 Day: 13 Hours: 20 Minutes: 17 Seconds: 33

Examples

```
const d = new Date();
const d = new Date("2022-03-25");
```

Note

Date objects are static. The "clock" is not "running".

The computer clock is ticking, date objects are not.

JavaScript Date Output

By default, JavaScript will use the browser's time zone and display a date as a full text string:

Sun Oct 13 2024 20:17:33 GMT+0530 (India Standard Time)

Creating Date Objects

Date objects are created with the `new Date()` constructor.

There are **9 ways** to create a new date object:

```
new Date()
new Date(date string)

new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)
```

JavaScript new Date()

`new Date()` creates a date object with the **current date and time**:

Example

```
const d = new Date();
```

new Date(*date string*)

`new Date(date string)` creates a date object from a **date string**:

Examples

```
const d = new Date("October 13, 2014 11:13:00");
const d = new Date("2022-03-25");
```

new Date(*year, month, ...*)

`new Date(year, month, ...)` creates a date object with a **specified date and time**.

7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

Example

```
const d = new Date(2018, 11, 24, 10, 33, 30, 0);
```

Note

JavaScript counts months from **0** to **11**:

January = 0.

December = 11.

Specifying a month higher than 11, will not result in an error but add the overflow to the next year:

Specifying:

```
const d = new Date(2018, 15, 24, 10, 33, 30);
```

Is the same as:

```
const d = new Date(2019, 3, 24, 10, 33, 30);
```

Specifying a day higher than max, will not result in an error but add the overflow to the next month:

Specifying:

```
const d = new Date(2018, 5, 35, 10, 33, 30);
```

Is the same as:

```
const d = new Date(2018, 6, 5, 10, 33, 30);
```

Using 6, 4, 3, or 2 Numbers

6 numbers specify year, month, day, hour, minute, second:

Example

```
const d = new Date(2018, 11, 24, 10, 33, 30);
```

5 numbers specify year, month, day, hour, and minute:

Example

```
const d = new Date(2018, 11, 24, 10, 33);
```

4 numbers specify year, month, day, and hour:

Example

```
const d = new Date(2018, 11, 24, 10);
```

3 numbers specify year, month, and day:

Example

```
const d = new Date(2018, 11, 24);
```

2 numbers specify year and month:

Example

```
const d = new Date(2018, 11);
```

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

Example

```
const d = new Date(2018);
```

Previous Century

One and two digit years will be interpreted as 19xx:

Example

```
const d = new Date(99, 11, 24);
```

Example

```
const d = new Date(9, 11, 24);
```

JavaScript Stores Dates as Milliseconds

JavaScript stores dates as number of milliseconds since January 01, 1970.

Zero time is January 01, 1970 00:00:00 UTC.

One day (24 hours) is 86 400 000 milliseconds.

Now the time is: **1728830853834** milliseconds past January 01, 1970

new Date(milliseconds)

`new Date(milliseconds)` creates a new date object as **milliseconds** plus zero time:

Examples

01 January 1970 **plus** 100 000 000 000 milliseconds is:

```
const d = new Date(1000000000000);
```

January 01 1970 **minus** 100 000 000 000 milliseconds is:

```
const d = new Date(-1000000000000);
```

January 01 1970 **plus** 24 hours is:

```
const d = new Date(24 * 60 * 60 * 1000);
```

// or

```
const d = new Date(86400000);
```

01 January 1970 **plus** 0 milliseconds is:

```
const d = new Date(0);
```

Date Methods

When a date object is created, a number of **methods** allow you to operate on it.

Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of date objects, using either local time or UTC (universal, or GMT) time.

Displaying Dates

JavaScript will (by default) output dates using the **toString()** method. This is a string representation of the date, including the time zone. The format is specified in the ECMAScript specification:

Example

Sun Oct 13 2024 20:17:33 GMT+0530 (India Standard Time)

When you display a date object in HTML, it is automatically converted to a string, with the **toString()** method.

Example

```
const d = new Date();
d.toString();
```

The **toDateString()** method converts a date to a more readable format:

Example

```
const d = new Date();
d.toDateString();
```

The **toUTCString()** method converts a date to a string using the UTC standard:

Example

```
const d = new Date();
d.toUTCString();
```

The **toISOString()** method converts a date to a string using the ISO standard:

Example

```
const d = new Date();
d.toISOString();
```

34. JavaScript Date Formats

JavaScript Date Input

There are generally 3 types of JavaScript date input formats:

Type	Example
ISO Date	"2015-03-25" (The International Standard)
Short Date	"03/25/2015"
Long Date	"Mar 25 2015" or "25 Mar 2015"

The ISO format follows a strict standard in JavaScript.

The other formats are not so well defined and might be browser specific.

JavaScript Date Output

Independent of input format, JavaScript will (by default) output dates in full text string format:

Sun Oct 13 2024 21:01:17 GMT+0530 (India Standard Time)

JavaScript ISO Dates

ISO 8601 is the international standard for the representation of dates and times.

The ISO 8601 syntax (YYYY-MM-DD) is also the preferred JavaScript date format:

Example (Complete date)

```
const d = new Date("2015-03-25");
```

The computed date will be relative to your time zone.

Depending on your time zone, the result above will vary between March 24 and March 25.

ISO Dates (Year and Month)

ISO dates can be written without specifying the day (YYYY-MM):

Example

```
const d = new Date("2015-03");
```

Time zones will vary the result above between February 28 and March 01.

ISO Dates (Only Year)

ISO dates can be written without month and day (YYYY):

Example

```
const d = new Date("2015");
```

Time zones will vary the result above between December 31 2014 and January 01 2015.

ISO Dates (Date-Time)

ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):

Example

```
const d = new Date("2015-03-25T12:00:00Z");
```

Date and time is separated with a capital T.

UTC time is defined with a capital letter Z.

If you want to modify the time relative to UTC, remove the Z and add +HH:MM or -HH:MM instead:

Example

```
const d = new Date("2015-03-25T12:00:00-06:30");
```

UTC (Universal Time Coordinated) is the same as GMT (Greenwich Mean Time).

Omitting T or Z in a date-time string can give different results in different browsers.

Time Zones

When setting a date, without specifying the time zone, JavaScript will use the browser's time zone.

When getting a date, without specifying the time zone, the result is converted to the browser's time zone.

In other words: If a date/time is created in GMT (Greenwich Mean Time), the date/time will be converted to CDT (Central US Daylight Time) if a user browses from central US.

JavaScript Short Dates.

Short dates are written with an "MM/DD/YYYY" syntax like this:

Example

```
const d = new Date("03/25/2015");
```

WARNINGS !

In some browsers, months or days with no leading zeroes may produce an error:

```
const d = new Date("2015-3-25");
```

The behavior of "YYYY/MM/DD" is undefined.

Some browsers will try to guess the format. Some will return NaN.

```
const d = new Date("2015/03/25");
```

The behavior of "DD-MM-YYYY" is also undefined.

Some browsers will try to guess the format. Some will return NaN.

```
const d = new Date("25-03-2015");
```

JavaScript Long Dates.

Long dates are most often written with a "MMM DD YYYY" syntax like this:

Example

```
const d = new Date("Mar 25 2015");
```

Month and day can be in any order:

Example

```
const d = new Date("25 Mar 2015");
```

And, month can be written in full (January), or abbreviated (Jan):

Example

```
const d = new Date("January 25 2015");
```

Example

```
const d = new Date("Jan 25 2015");
```

Commas are ignored. Names are case insensitive:

Example

```
const d = new Date("JANUARY, 25, 2015");
```

Date Input - Parsing Dates

If you have a valid date string, you can use the `Date.parse()` method to convert it to milliseconds.

`Date.parse()` returns the number of milliseconds between the date and January 1, 1970:

Example

```
let msec = Date.parse("March 21, 2012");
```

You can then use the number of milliseconds to **convert it to a date** object:

Example

```
let msec = Date.parse("March 21, 2012");
const d = new Date(msec);
```

35. JavaScript Get Date Methods

The new Date() Constructor

In JavaScript, date objects are created with `new Date()`.

`new Date()` returns a date object with the current date and time.

Get the Current Time

```
const date = new Date();
```

Date Get Methods

Method	Description
getFullYear()	Get year as a four digit number (yyyy)
getMonth()	Get month as a number (0-11)
getDate()	Get day as a number (1-31)
getDay()	Get weekday as a number (0-6)
getHours()	Get hour (0-23)
getMinutes()	Get minute (0-59)
getSeconds()	Get second (0-59)

`getMilliseconds()`

Get **millisecond** (0-999)

`getTime()`

Get **time** (milliseconds since January 1, 1970)

Note 1

The get methods above return **Local time**.

Universal time (UTC) is documented at the bottom of this page.

Note 2

The get methods return information from existing date objects.

In a date object, the time is static. The "clock" is not "running".

The time in a date object is NOT the same as current time.

The **getFullYear()** Method

The `getFullYear()` method returns the year of a date as a four digit number:

Examples

```
const d = new Date("2021-03-25");
d.getFullYear();
```

```
const d = new Date();
d.getFullYear();
```

Warning !

Old JavaScript code might use the non-standard method `getYear()`.

`getYear()` is supposed to return a 2-digit year.

`getYear()` is deprecated. Do not use it!

The getMonth() Method

The `getMonth()` method returns the month of a date as a number (0-11).

Note

In JavaScript, January is month number 0, February is number 1, ...

Finally, December is month number 11.

Examples

```
const d = new Date("2021-03-25");
d.getMonth();

const d = new Date();
d.getMonth();
```

Note

You can use an array of names to return the month as a name:

Examples

```
const months =
["January", "February", "March", "April", "May", "June", "July", "August", "September",
", "October", "November", "December"];

const d = new Date("2021-03-25");
let month = months[d.getMonth()];

const months =
["January", "February", "March", "April", "May", "June", "July", "August", "September",
", "October", "November", "December"];
const d = new Date();
let month = months[d.getMonth()];
```

The getDate() Method

The `getDate()` method returns the day of a date as a number (1-31):

Examples

```
const d = new Date("2021-03-25");
d.getDate();

const d = new Date();
d.getDate();
```

The getHours() Method

The `getHours()` method returns the hours of a date as a number (0-23):

Examples

```
const d = new Date("2021-03-25");
d.getHours();

const d = new Date();
d.getHours();
```

The getMinutes() Method

The `getMinutes()` method returns the minutes of a date as a number (0-59):

Examples

```
const d = new Date("2021-03-25");
d.getMinutes();

const d = new Date();
d.getMinutes();
```

The getSeconds() Method

The `getSeconds()` method returns the seconds of a date as a number (0-59):

Examples

```
const d = new Date("2021-03-25");
d.getSeconds();

const d = new Date();
d.getSeconds();
```

The getMilliseconds() Method

The `getMilliseconds()` method returns the milliseconds of a date as a number (0-999):

Examples

```
const d = new Date("2021-03-25");
d.getMilliseconds();

const d = new Date();
d.getMilliseconds();
```

The `getDay()` Method

The `getDay()` method returns the weekday of a date as a number (0-6).

Note

In JavaScript, the first day of the week (day 0) is Sunday.

Some countries in the world consider the first day of the week to be Monday.

Examples

```
const d = new Date("2021-03-25");
d.getDay();

const d = new Date();
d.getDay();
```

Note

You can use an array of names, and `getDay()` to return weekday as a name:

Examples

```
const days =
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];
const d = new Date("2021-03-25");
let day = days[d.getDay()];

const days =
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];

const d = new Date();
let day = days[d.getDay()];
```

The `getTime()` Method

The `getTime()` method returns the number of milliseconds since January 1, 1970:

Examples

```
const d = new Date("1970-01-01");
d.getTime();

const d = new Date("2021-03-25");
d.getTime();

const d = new Date();
d.getTime();
```

The Date.now() Method

`Date.now()` returns the number of milliseconds since January 1, 1970.

Examples

```
let ms = Date.now();
```

Calculate the number of years since 1970/01/01:

```
const minute = 1000 * 60;
const hour = minute * 60;
const day = hour * 24;
const year = day * 365;

let years = Math.round(Date.now() / year);
```

`Date.now()` is a static method of the `Date` object.

You cannot use it on a date object like `myDate.now()`.

The syntax is always `Date.now()`.

UTC Date Get Methods

Method	Same As	Description
<code>getUTCDate()</code>	<code>getDate()</code>	Returns the UTC date
<code>getUTCFullYear()</code>	<code>getFullYear()</code>	Returns the UTC year
<code>getUTCMonth()</code>	<code>getMonth()</code>	Returns the UTC month
<code>getUTCDay()</code>	<code>getDay()</code>	Returns the UTC day
<code>getUTCHours()</code>	<code>getHours()</code>	Returns the UTC hour

getUTCMinutes()	getMinutes()	Returns the UTC minutes
getUTCSCONDS()	getSeconds()	Returns the UTC seconds
getUTCMilliseconds()	getMilliseconds()	Returns the UTC milliseconds

UTC methods use UTC time (Coordinated Universal Time).

UTC time is the same as GMT (Greenwich Mean Time).

The difference between Local time and UTC time can be up to 24 hours.

Local Time?

UTC Time?

The **getTimezoneOffset()** Method

The **getTimezoneOffset()** method returns the difference (in minutes) between local time and UTC time:

Example

```
let diff = d.getTimezoneOffset();
```

36. JavaScript Set Date Methods

Set Date methods let you set date values (years, months, days, hours, minutes, seconds, milliseconds) for a Date Object.

Set Date Methods

Set Date methods are used for setting a part of a date:

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

The `setFullYear()` Method

The `setFullYear()` method sets the year of a date object. In this example to 2020:

Example

```
const d = new Date();
d.setFullYear(2020);
```

The `setFullYear()` method can **optionally** set month and day:

Example

```
const d = new Date();
d.setFullYear(2020, 11, 3);
```

The `setMonth()` Method

The `setMonth()` method sets the month of a date object (0-11):

Example

```
const d = new Date();
d.setMonth(11);
```

The `setDate()` Method

The `setDate()` method sets the day of a date object (1-31):

Example

```
const d = new Date();
d.setDate(15);
```

The `setDate()` method can also be used to **add days** to a date:

Example

```
const d = new Date();
d.setDate(d.getDate() + 50);
```

If adding days shifts the month or year, the changes are handled automatically by the Date object.

The setHours() Method

The `setHours()` method sets the hours of a date object (0-23):

Example

```
const d = new Date();
d.setHours(22);
```

The setMinutes() Method

The `setMinutes()` method sets the minutes of a date object (0-59):

Example

```
const d = new Date();
d.setMinutes(30);
```

The setSeconds() Method

The `setSeconds()` method sets the seconds of a date object (0-59):

Example

```
const d = new Date();
d.setSeconds(30);
```

Compare Dates

Dates can easily be compared.

The following example compares today's date with January 14, 2100:

Example

```
let text = "";
const today = new Date();
const someday = new Date();
someday.setFullYear(2100, 0, 14);

if (someday > today) {
  text = "Today is before January 14, 2100.";
} else {
  text = "Today is after January 14, 2100.";
}
```

JavaScript counts months from 0 to 11. January is 0. December is 11.

37. JavaScript Math Object

The JavaScript Math object allows you to perform mathematical tasks on numbers.

Example

```
Math.PI;
```

The Math Object

Unlike other objects, the Math object has no constructor.

The Math object is static.

All methods and properties can be used without creating a Math object first.

Math Properties (Constants)

The syntax for any Math property is : `Math.property`.

JavaScript provides 8 mathematical constants that can be accessed as Math properties:

Example

```
Math.E      // returns Euler's number  
Math.PI     // returns PI  
Math.SQRT2  // returns the square root of 2  
Math.SQRT1_2 // returns the square root of 1/2  
Math.LN2    // returns the natural logarithm of 2  
Math.LN10   // returns the natural logarithm of 10  
Math.LOG2E  // returns base 2 logarithm of E  
Math.LOG10E // returns base 10 logarithm of E
```

Math Methods

The syntax for Math any methods is : `Math.method(number)`

Number to Integer

There are 4 common methods to round a number to an integer:

<code>Math.round(x)</code>	Returns x rounded to its nearest integer
----------------------------	--

<code>Math.ceil(x)</code>	Returns x rounded up to its nearest integer
---------------------------	---

<code>Math.floor(x)</code>	Returns x rounded down to its nearest integer
----------------------------	---

<code>Math.trunc(x)</code>	Returns the integer part of x (new in ES6)
----------------------------	--

Math.round()

`Math.round(x)` returns the nearest integer:

Examples

```
Math.round(4.6);  
Math.round(4.5);  
Math.round(4.4);
```

Math.ceil()

`Math.ceil(x)` returns the value of x rounded **up** to its nearest integer:

Example

```
Math.ceil(4.9);  
Math.ceil(4.7);  
Math.ceil(4.4);  
Math.ceil(4.2);  
Math.ceil(-4.2);
```

Math.floor()

`Math.floor(x)` returns the value of x rounded **down** to its nearest integer:

Example

```
Math.floor(4.9);  
Math.floor(4.7);  
Math.floor(4.4);  
Math.floor(4.2);  
Math.floor(-4.2);
```

Math.trunc()

`Math.trunc(x)` returns the integer part of x:

Example

```
Math.trunc(4.9);  
Math.trunc(4.7);  
Math.trunc(4.4);  
Math.trunc(4.2);  
Math.trunc(-4.2);
```

Math.sign()

`Math.sign(x)` returns if x is negative, null or positive:

Example

```
Math.sign(-4);  
Math.sign(0);  
Math.sign(4);
```

Math.trunc() and Math.sign() were added to [JavaScript 2015 - ES6](#).

Math.pow()

`Math.pow(x, y)` returns the value of x to the power of y:

Example

```
Math.pow(8, 2);
```

Math.sqrt()

`Math.sqrt(x)` returns the square root of x:

Example

```
Math.sqrt(64);
```

Math.abs()

`Math.abs(x)` returns the absolute (positive) value of x:

Example

```
Math.abs(-4.7);
```

Math.sin()

`Math.sin(x)` returns the sine (a value between -1 and 1) of the angle x (given in radians).

If you want to use degrees instead of radians, you have to convert degrees to radians:

Angle in radians = Angle in degrees x PI / 180.

Example

```
Math.sin(90 * Math.PI / 180);      // returns 1 (the sine of 90 degrees)
```

Math.cos()

`Math.cos(x)` returns the cosine (a value between -1 and 1) of the angle x (given in radians). If you want to use degrees instead of radians, you have to convert degrees to radians:
Angle in radians = Angle in degrees $\times \pi / 180$.

Example

```
Math.cos(0 * Math.PI / 180);      // returns 1 (the cos of 0 degrees)
```

Math.min() and Math.max()

`Math.min()` and `Math.max()` can be used to find the lowest or highest value in a list of arguments:

Example

```
Math.min(0, 150, 30, 20, -8, -200);
```

Example

```
Math.max(0, 150, 30, 20, -8, -200);
```

Math.random()

`Math.random()` returns a random number between 0 (inclusive), and 1 (exclusive):

Example

```
Math.random();
```

The Math.log() Method

`Math.log(x)` returns the natural logarithm of x.

The natural logarithm returns the time needed to reach a certain level of growth:

Examples

```
Math.log(1);
```

```
Math.log(2);
```

```
Math.log(3);
```

Math.E and Math.log() are twins.

How many times must we multiply Math.E to get 10?

```
Math.log(10);
```

The Math.log2() Method

`Math.log2(x)` returns the base 2 logarithm of x.

How many times must we multiply 2 to get 8?

```
Math.log2(8);
```

The Math.log10() Method

`Math.log10(x)` returns the base 10 logarithm of x.

How many times must we multiply 10 to get 1000?

```
Math.log10(1000);
```

JavaScript Math Methods

Method	Description
<u>abs(x)</u>	Returns the absolute value of x
<u>acos(x)</u>	Returns the arccosine of x, in radians
<u>acosh(x)</u>	Returns the hyperbolic arccosine of x
<u>asin(x)</u>	Returns the arcsine of x, in radians
<u>asinh(x)</u>	Returns the hyperbolic arcsine of x
<u>atan(x)</u>	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians

[atan2\(y, x\)](#) Returns the arctangent of the quotient of its arguments

[atanh\(x\)](#) Returns the hyperbolic arctangent of x

[cbrt\(x\)](#) Returns the cubic root of x

[ceil\(x\)](#) Returns x, rounded upwards to the nearest integer

[cos\(x\)](#) Returns the cosine of x (x is in radians)

[cosh\(x\)](#) Returns the hyperbolic cosine of x

[exp\(x\)](#) Returns the value of E^x

[floor\(x\)](#) Returns x, rounded downwards to the nearest integer

[log\(x\)](#) Returns the natural logarithm (base E) of x

[max\(x, y, z, ..., n\)](#) Returns the number with the highest value

[min\(x, y, z, ..., n\)](#) Returns the number with the lowest value

[pow\(x, y\)](#) Returns the value of x to the power of y

[random\(\)](#) Returns a random number between 0 and 1

[round\(x\)](#) Rounds x to the nearest integer

[sign\(x\)](#) Returns if x is negative, null or positive (-1, 0, 1)

[sin\(x\)](#) Returns the sine of x (x is in radians)

[sinh\(x\)](#) Returns the hyperbolic sine of x

[sqrt\(x\)](#) Returns the square root of x

[tan\(x\)](#) Returns the tangent of an angle

[tanh\(x\)](#) Returns the hyperbolic tangent of a number

[trunc\(x\)](#) Returns the integer part of a number (x)

38. JavaScript Random

Math.random()

`Math.random()` returns a random number between 0 (inclusive), and 1 (exclusive):

Example

```
// Returns a random number:  
Math.random();
```

`Math.random()` always returns a number lower than 1.

JavaScript Random Integers

`Math.random()` used with `Math.floor()` can be used to return random integers.

There is no such thing as JavaScript integers.

We are talking about numbers with no decimals here.

Example

```
// Returns a random integer from 0 to 9:  
Math.floor(Math.random() * 10);
```

Example

```
// Returns a random integer from 0 to 10:  
Math.floor(Math.random() * 11);
```

Example

```
// Returns a random integer from 0 to 99:  
Math.floor(Math.random() * 100);
```

Example

```
// Returns a random integer from 0 to 100:  
Math.floor(Math.random() * 101);
```

Example

```
// Returns a random integer from 1 to 10:  
Math.floor(Math.random() * 10) + 1;
```

Example

```
// Returns a random integer from 1 to 100:  
Math.floor(Math.random() * 100) + 1;
```

A Proper Random Function

As you can see from the examples above, it might be a good idea to create a proper random function to use for all random integer purposes.

This JavaScript function always returns a random number between min (included) and max (excluded):

Example

```
function getRndInteger(min, max) {  
    return Math.floor(Math.random() * (max - min) ) + min;  
}
```

This JavaScript function always returns a random number between min and max (both included):

Example

```
function getRndInteger(min, max) {  
    return Math.floor(Math.random() * (max - min + 1) ) + min;  
}
```

39. JavaScript Booleans

A JavaScript Boolean represents one of two values: **true** or **false**.

Boolean Values

Very often, in programming, you will need a data type that can only have one of two values, like

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, JavaScript has a **Boolean** data type. It can only take the values **true** or **false**.

The Boolean() Function

You can use the **Boolean()** function to find out if an expression (or a variable) is true:

Example

```
Boolean(10 > 9)
```

Or even easier:

Example

```
(10 > 9)
```

```
10 > 9
```

Comparisons and Conditions

The chapter [JS Comparisons](#) gives a full overview of comparison operators.

The chapter [JS If Else](#) gives a full overview of conditional statements.

Here are some examples:

Operator	Description	Example
<code>==</code>	equal to	<code>if (day == "Monday")</code>
<code>></code>	greater than	<code>if (salary > 9000)</code>
<code><</code>	less than	<code>if (age < 18)</code>

The Boolean value of an expression is the basis for all JavaScript comparisons and conditions.

Everything With a "Value" is True

Examples

100

3.14

-15

"Hello"

"false"

7 + 1 + 3.14

Everything Without a "Value" is False

The Boolean value of **0** (zero) is **false**:

```
let x = 0;  
Boolean(x);
```

The Boolean value of **-0** (minus zero) is **false**:

```
let x = -0;  
Boolean(x);
```

The Boolean value of **""** (empty string) is **false**:

```
let x = "";  
Boolean(x);
```

The Boolean value of **undefined** is **false**:

```
let x;  
Boolean(x);
```

The Boolean value of **null** is **false**:

```
let x = null;  
Boolean(x);
```

The Boolean value of **false** is (you guessed it) **false**:

```
let x = false;  
Boolean(x);
```

The Boolean value of **NaN** is **false**:

```
let x = 10 / "Hallo";  
Boolean(x);
```

JavaScript Booleans as Objects

Normally JavaScript booleans are primitive values created from literals:

```
let x = false;
```

But booleans can also be defined as objects with the keyword `new`:

```
let y = new Boolean(false);
```

Example

```
let x = false;
let y = new Boolean(false);

// typeof x returns boolean
// typeof y returns object
```

Do not create Boolean objects.

The `new` keyword complicates the code and slows down execution speed.

Boolean objects can produce unexpected results:

When using the `==` operator, x and y are **equal**:

```
let x = false;
let y = new Boolean(false);
```

When using the `===` operator, x and y are **not equal**:

```
let x = false;
let y = new Boolean(false);
```

Note the difference between `(x==y)` and `(x===y)`.

`(x == y)` true or false?

```
let x = new Boolean(false);
let y = new Boolean(false);
```

`(x === y)` true or false?

```
let x = new Boolean(false);
let y = new Boolean(false);
```

Comparing two JavaScript objects **always** return **false**.

40. JavaScript Comparison and Logical Operators

Comparison and Logical operators are used to test for **true or false**.

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Given that `x = 5`, the table below explains the comparison operators:

Operator	Description	Comparing	Returns
<code>==</code>	equal to	<code>x == 8</code>	false
		<code>x == 5</code>	true
		<code>x == "5"</code>	true
<code>====</code>	equal value and equal type	<code>x === 5</code>	true
		<code>x === "5"</code>	false
<code>!=</code>	not equal	<code>x != 8</code>	true
<code>!==</code>	not equal value or not equal type	<code>x !== 5</code>	false
		<code>x !== "5"</code>	true

		x != 8	true
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

How Can it be Used

Comparison operators can be used in conditional statements to compare values and take action depending on the result:

```
if (age < 18) text = "Too young to buy alcohol";
```

You will learn more about the use of conditional statements in the next chapter of this tutorial.

Logical Operators

Logical operators are used to determine the logic between variables or values.

Given that `x = 6` and `y = 3`, the table below explains the logical operators:

Operator	Description	Example
&&	and	<code>(x < 10 && y > 1)</code> is true
	or	<code>(x == 5 y == 5)</code> is false
!	not	<code>!(x == y)</code> is true

Conditional (Ternary) Operator

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

Syntax

```
variablename = (condition) ? value1:value2
```

Example

```
let voteable = (age < 18) ? "Too young":"Old enough";
```

If the variable age is a value below 18, the value of the variable voteable will be "Too young", otherwise the value of voteable will be "Old enough".

Comparing Different Types

Comparing data of different types may give unexpected results.

When comparing a string with a number, JavaScript will convert the string to a number when doing the comparison. An empty string converts to 0. A non-numeric string converts to `Nan` which is always `false`.

Case	Value
<code>2 < 12</code>	true
<code>2 < "12"</code>	true
<code>2 < "John"</code>	false
<code>2 > "John"</code>	false
<code>2 == "John"</code>	false

"2" < "12"	false
------------	-------

| "2" > "12" | true |
| "2" == "12" | false |

When comparing two strings, "2" will be greater than "12", because (alphabetically) 1 is less than 2.

To secure a proper result, variables should be converted to the proper type before comparison:

```
age = Number(age);
if (isNaN(age)) {
  voteable = "Input is not a number";
} else {
  voteable = (age < 18) ? "Too young" : "Old enough";
}
```

The Nullish Coalescing Operator (??)

The `??` operator returns the first argument if it is not **nullish** (`null` or `undefined`).

Otherwise it returns the second argument.

Example

```
let name = null;
let text = "missing";
let result = name ?? text;
```

The nullish operator is supported in all browsers since March 2020:



Chrome 80	Edge 80	Firefox 72	Safari 13.1	Opera 67
-----------	---------	------------	-------------	----------

The Optional Chaining Operator (?.)

The `?.` operator returns `undefined` if an object is `undefined` or `null` (instead of throwing an error).

Example

```
// Create an object:  
const car = {type:"Fiat", model:"500", color:"white"};  
// Ask for car name:  
document.getElementById("demo").innerHTML = car?.name;
```

The optional chaining operator is supported in all browsers since March 2020:

				
Chrome 80	Edge 80	Firefox 72	Safari 13.1	Opera 67
Feb 2020	Feb 2020	Jan 2020	Mar 2020	Mar 2020

41. JavaScript if, else, and else if

Conditional statements are used to perform different actions based on different conditions.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax

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

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

Example

Make a "Good day" greeting if the hour is less than 18:00:

```
if (hour < 18) {  
    greeting = "Good day";  
}
```

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is false.

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

Example

If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

```
if (hour < 18) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

The result of greeting will be:

Good evening

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {  
    // block of code to be executed if condition1 is true  
} else if (condition2) {  
    // block of code to be executed if the condition1 is false and condition2 is true  
} else {  
    // block of code to be executed if the condition1 is false and condition2 is false  
}
```

Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
if (time < 10) {  
    greeting = "Good morning";  
} else if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

The result of greeting will be:

Good evening

42. JavaScript Switch Statement

The `switch` statement is used to perform different actions based on different conditions.

The JavaScript Switch Statement

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {  
    case x:  
        // code block  
        break;  
    case y:  
        // code block  
        break;  
    default:  
        // code block  
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

(Sunday=0, Monday=1, Tuesday=2 ..)

This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {  
    case 0:  
        day = "Sunday";  
        break;  
    case 1:  
        day = "Monday";  
        break;  
    case 2:  
        day = "Tuesday";  
        break;  
    case 3:  
        day = "Wednesday";  
        break;  
    case 4:
```

```
day = "Thursday";
break;
case 5:
    day = "Friday";
    break;
case 6:
    day = "Saturday";
}
```

The result of day will be:

Tuesday

The break Keyword

When JavaScript reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution inside the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

Note: If you omit the `break` statement, the next case will be executed even if the evaluation does not match the case.

The default Keyword

The `default` keyword specifies the code to run if there is no case match:

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

If today is neither Saturday (6) nor Sunday (0), write a default message:

```
switch (new Date().getDay()) {
    case 6:
        text = "Today is Saturday";
        break;
    case 0:
        text = "Today is Sunday";
        break;
    default:
        text = "Looking forward to the Weekend";
}
```

The result of text will be:

Looking forward to the Weekend

The `default` case does not have to be the last case in a switch block:

Example

```
switch (new Date().getDay()) {  
    default:  
        text = "Looking forward to the Weekend";  
        break;  
    case 6:  
        text = "Today is Saturday";  
        break;  
    case 0:  
        text = "Today is Sunday";  
}  
}
```

If `default` is not the last case in the switch block, remember to end the default case with a `break`.

Common Code Blocks

Sometimes you will want different switch cases to use the same code.

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

Example

```
switch (new Date().getDay()) {  
    case 4:  
    case 5:  
        text = "Soon it is Weekend";  
        break;  
    case 0:  
    case 6:  
        text = "It is Weekend";  
        break;  
    default:  
        text = "Looking forward to the Weekend";  
}  
}
```

Switching Details

If multiple cases matches a case value, the **first** case is selected.

If no matching cases are found, the program continues to the **default** label.

If no default label is found, the program continues to the statement(s) **after the switch**.

Strict Comparison

Switch cases use **strict** comparison (`==`).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

In this example there will be no match for x:

Example

```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
    break;
  case 1:
    text = "On";
    break;
  default:
    text = "No value found";
}
```

43. JavaScript For Loop

Loops can execute a block of code a number of times.

JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

Instead of writing:

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

You can write:

```
for (let i = 0; i < cars.length; i++) {
    text += cars[i] + "<br>";
}
```

Different Kinds of Loops

JavaScript supports different kinds of loops:

- `for` - loops through a block of code a number of times
- `for/in` - loops through the properties of an object
- `for/of` - loops through the values of an iterable object
- `while` - loops through a block of code while a specified condition is true
- `do/while` - also loops through a block of code while a specified condition is true

The For Loop

The `for` statement creates a loop with 3 optional expressions:

```
for (expression 1; expression 2; expression 3) {
    // code block to be executed
}
```

Expression 1 is executed (one time) before the execution of the code block.

Expression 2 defines the condition for executing the code block.

Expression 3 is executed (every time) after the code block has been executed.

Example

```
for (let i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}
```

From the example above, you can read:

Expression 1 sets a variable before the loop starts (let *i* = 0).

Expression 2 defines the condition for the loop to run (*i* must be less than 5).

Expression 3 increases a value (*i*++) each time the code block in the loop has been executed.

Expression 1

Normally you will use expression 1 to initialize the variable used in the loop (let *i* = 0).

This is not always the case. JavaScript doesn't care. Expression 1 is optional.

You can initiate many values in expression 1 (separated by comma):

Example

```
for (let i = 0, len = cars.length, text = ""; i < len; i++) {  
    text += cars[i] + "<br>";  
}
```

And you can omit expression 1 (like when your values are set before the loop starts):

Example

```
let i = 2;  
let len = cars.length;  
let text = "";  
for (; i < len; i++) {  
    text += cars[i] + "<br>";  
}
```

Expression 2

Often expression 2 is used to evaluate the condition of the initial variable.

This is not always the case. JavaScript doesn't care. Expression 2 is also optional.

If expression 2 returns true, the loop will start over again. If it returns false, the loop will end.

If you omit expression 2, you must provide a **break** inside the loop. Otherwise the loop will never end. This will crash your browser. Read about breaks in a later chapter of this tutorial.

Expression 3

Often expression 3 increments the value of the initial variable.

This is not always the case. JavaScript doesn't care. Expression 3 is optional.

Expression 3 can do anything like negative increment (`i--`), positive increment (`i = i + 15`), or anything else.

Expression 3 can also be omitted (like when you increment your values inside the loop):

Example

```
let i = 0;
let len = cars.length;
let text = "";
for (; i < len; ) {
    text += cars[i] + "<br>";
    i++;
}
```

Loop Scope

Using `var` in a loop:

Example

```
var i = 5;
for (var i = 0; i < 10; i++) {
    // some code
}
// Here i is 10
```

Using `let` in a loop:

Example

```
let i = 5;
for (let i = 0; i < 10; i++) {
    // some code
}
// Here i is 5
```

In the first example, using `var`, the variable declared in the loop redeclares the variable outside the loop.

In the second example, using `let`, the variable declared in the loop does not redeclare the variable outside the loop.

When `let` is used to declare the `i` variable in a loop, the `i` variable will only be visible within the loop.

44. JavaScript For In

The For In Loop

The JavaScript `for in` statement loops through the properties of an Object:

Syntax

```
for (key in object) {  
    // code block to be executed  
}
```

Example

```
const person = {fname:"John", lname:"Doe", age:25};  
let text = "";  
for (let x in person) {  
    text += person[x];  
}
```

Example Explained

- The `for in` loop iterates over a **person** object
- Each iteration returns a **key** (x)
- The key is used to access the **value** of the key
- The value of the key is **person[x]**

For In Over Arrays

The JavaScript `for in` statement can also loop over the properties of an Array:

Syntax

```
for (variable in array) {  
    code  
}
```

Example

```
const numbers = [45, 4, 9, 16, 25];  
let txt = "";  
for (let x in numbers) {  
    txt += numbers[x];  
}
```

Do not use `for in` over an Array if the index **order** is important.

The index order is implementation-dependent, and array values may not be accessed in the order you expect.

It is better to use a `for` loop, a `for of` loop, or `Array.forEach()` when the order is important.

Array.forEach()

The `forEach()` method calls a function (a callback function) once for each array element.

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);

function myFunction(value, index, array) {
  txt += value;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

The example above uses only the value parameter. It can be rewritten to:

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);

function myFunction(value) {
  txt += value;
}
```

45. JavaScript For Of

The For Of Loop

The JavaScript `for of` statement loops through the values of an iterable object. It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

Syntax

```
for (variable of iterable) {  
    // code block to be executed  
}
```

variable - For every iteration the value of the next property is assigned to the variable. *Variable* can be declared with `const`, `let`, or `var`.

iterable - An object that has iterable properties.

Browser Support

For/of was added to JavaScript in 2015 ([ES6](#))

Safari 7 was the first browser to support for of:

				
Chrome 38	Edge 12	Firefox 51	Safari 7	Opera 25
Oct 2014	Jul 2015	Oct 2016	Oct 2013	Oct 2014

For/of is not supported in Internet Explorer.

Looping over an Array

Example

```
const cars = ["BMW", "Volvo", "Mini"];  
  
let text = "";  
for (let x of cars) {  
    text += x;  
}
```

Looping over a String

Example

```
let language = "JavaScript";  
  
let text = "";  
for (let x of language) {  
    text += x;  
}
```

46. JavaScript While Loop

Loops can execute a block of code as long as a specified condition is true.

The While Loop

The `while` loop loops through a block of code as long as a specified condition is true.

Syntax

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

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (`i`) is less than 10:

Example

```
while (i < 10) {  
    text += "The number is " + i;  
    i++;  
}
```

If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The Do While Loop

The `do while` loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {  
    // code block to be executed  
}  
while (condition);
```

Example

The example below uses a `do while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
do {  
    text += "The number is " + i;  
    i++;  
}  
while (i < 10);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Comparing For and While

If you have read the previous chapter, about the for loop, you will discover that a while loop is much the same as a for loop, with statement 1 and statement 3 omitted.

The loop in this example uses a `for` loop to collect the car names from the cars array:

Example

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];  
let i = 0;  
let text = "";  
  
for (;cars[i]); {  
    text += cars[i];  
    i++;  
}
```

The loop in this example uses a `while` loop to collect the car names from the cars array:

Example

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];  
let i = 0;  
let text = "";  
  
while (cars[i]) {  
    text += cars[i];  
    i++;  
}
```

47. JavaScript Break and Continue

The **break** statement "jumps out" of a loop.

The **continue** statement "jumps over" one iteration in the loop.

The Break Statement

You have already seen the **break** statement used in an earlier chapter of this tutorial. It was used to "jump out" of a **switch()** statement.

The **break** statement can also be used to jump out of a loop:

Example

```
for (let i = 0; i < 10; i++) {  
    if (i === 3) { break; }  
    text += "The number is " + i + "<br>;  
}
```

In the example above, the **break** statement ends the loop ("breaks" the loop) when the loop counter (i) is 3.

The Continue Statement

The **continue** statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 3:

Example

```
for (let i = 0; i < 10; i++) {  
    if (i === 3) { continue; }  
    text += "The number is " + i + "<br>;
```

JavaScript Labels

To label JavaScript statements you precede the statements with a label name and a colon:
label:

statements

The **break** and the **continue** statements are the only JavaScript statements that can "jump out of" a code block.

Syntax:

```
break Labelname;  
continue Labelname;
```

The **continue** statement (with or without a label reference) can only be used to **skip one loop iteration**.

The **break** statement, without a label reference, can only be used to **jump out of a loop or a switch**.

With a label reference, the break statement can be used to **jump out of any code block**:

Example

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];  
list: {  
    text += cars[0] + "<br>;  
    text += cars[1] + "<br>;  
    break list;  
    text += cars[2] + "<br>;  
    text += cars[3] + "<br>;}
```

A code block is a block of code between { and }.

48. JavaScript Iterables

Iterables are iterable objects (like Arrays).

Iterables can be accessed with simple and efficient code.

Iterables can be iterated over with `for..of` loops

The For Of Loop

The JavaScript `for..of` statement loops through the elements of an iterable object.

Syntax

```
for (variable of iterable) {  
    // code block to be executed  
}
```

Iterating

Iterating is easy to understand.

It simply means looping over a sequence of elements.

Here are some easy examples:

- Iterating over a String
- Iterating over an Array

Iterating Over a String

You can use a `for..of` loop to iterate over the elements of a string:

Example

```
const name = "W3Schools";  
for (const x of name) {  
    // code block to be executed  
}
```

Iterating Over an Array

You can use a `for..of` loop to iterate over the elements of an Array:

Example 1

```
const letters = ["a", "b", "c"];  
  
for (const x of letters) {  
    // code block to be executed  
}
```

Example 2

```
const numbers = [2,4,6,8];
for (const x of numbers) {
    // code block to be executed
}
```

Iterating Over a Set

You can use a `for..of` loop to iterate over the elements of a Set:

Example

```
const letters = new Set(["a","b","c"]);
for (const x of letters) {
    // code block to be executed
}
```

Iterating Over a Map

You can use a `for..of` loop to iterate over the elements of a Map:

Example

```
const fruits = new Map([
    ["apples", 500],
    ["bananas", 300],
    ["oranges", 200]
]);
for (const x of fruits) {
    // code block to be executed
}
```

JavaScript Iterators

The **iterator protocol** defines how to produce a **sequence of values** from an object.

An object becomes an **iterator** when it implements a `next()` method.

The `next()` method must return an object with two properties:

- `value` (the next value)
- `done` (true or false)

value The value returned by the iterator
(Can be omitted if done is true)

done *true* if the iterator has completed
false if the iterator has produced a new value

Note

Technically, iterables must implement the `Symbol.iterator` method.

String, Array, TypedArray, Map and Set are all iterables, because their prototype objects have a `Symbol.iterator` method.

Home Made Iterable

This iterable returns never ending: 10,20,30,40,... Everytime `next()` is called:

Example

```
// Home Made Iterable
function myNumbers() {
  let n = 0;
  return {
    next: function() {
      n += 10;
      return {value:n, done:false};
    }
}
// Create Iterable
const n = myNumbers();
n.next(); // Returns 10
n.next(); // Returns 20
n.next(); // Returns 30
```

The problem with a home made iterable:

It does not support the JavaScript `for..of` statement.

A JavaScript iterable is an object that has a **Symbol.iterator**.

The **Symbol.iterator** is a function that returns a `next()` function.

An iterable can be iterated over with the code: `for (const x of iterable) { }`

Example

```
// Create an Object
myNumbers = {};
// Make it Iterable
myNumbers[Symbol.iterator] = function() {
  let n = 0;
  done = false;
  return {
    next() {
      n += 10;
      if (n == 100) {done = true}
      return {value:n, done:done};
    }
  };
}
```

Now you can use `for..of`

```
for (const num of myNumbers) {
  // Any Code Here}
```

The `Symbol.iterator` method is called automatically by `for..of`.

But we can also do it "manually":

Example

```
let iterator = myNumbers[Symbol.iterator]();
while (true) {
  const result = iterator.next();
  if (result.done) break;
  // Any Code Here
}
```

49. JavaScript Sets

A JavaScript Set is a collection of unique values.

Each value can only occur once in a Set.

The values can be of any type, primitive values or objects.

How to Create a Set

You can create a JavaScript Set by:

- Passing an array to `new Set()`
- Create an empty set and use `add()` to add values

The `new Set()` Method

Pass an array to the `new Set()` constructor:

Example

```
// Create a Set
const letters = new Set(["a", "b", "c"]);
```

Create a Set and add values:

Example

```
// Create a Set
const letters = new Set();
// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
```

Create a Set and add variables:

Example

```
// Create a Set
const letters = new Set();
// Create Variables
const a = "a";
const b = "b";
const c = "c";
// Add Variables to the Set
letters.add(a);
letters.add(b);
letters.add(c);
```

The add() Method

Example

```
letters.add("d");
letters.add("e");
```

If you add equal elements, only the first will be saved:

Example

```
letters.add("a");
letters.add("b");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
```

Listing the Elements

You can list all Set elements (values) with a **for..of** loop:

Example

```
// Create a Set
const letters = new Set(["a", "b", "c"]);
// List all Elements
let text = "";
for (const x of letters) {
  text += x;
}
```

Sets are Objects

`typeof` returns object:

```
typeof letters; // Returns object
```

`instanceof Set` returns true:

```
letters instanceof Set; // Returns true
```

Browser Support

`Set` is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

`Set` is not supported in Internet Explorer.

50. JavaScript Set Methods

The new Set() Method

Pass an array to the `new Set()` constructor:

Example

```
// Create a Set
const letters = new Set(["a", "b", "c"]);
```

The add() Method

Example

```
letters.add("d");
letters.add("e");
```

If you add equal elements, only the first will be saved:

Example

```
letters.add("a");
letters.add("b");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
```

Listing Set Elements

You can list all Set elements (values) with a `for..of` loop:

Example

```
// Create a Set
const letters = new Set(["a", "b", "c"]);

// List all Elements
let text = "";
for (const x of letters) {
  text += x;
}
```

The has() Method

The `has()` method returns `true` if a specified value exists in a set.

Example

```
// Create a Set
const letters = new Set(["a", "b", "c"]);

// Does the Set contain "d"?
answer = letters.has("d");
```

The forEach() Method

The `forEach()` method invokes a function for each Set element:

Example

```
// Create a Set
const letters = new Set(["a", "b", "c"]);

// List all entries
let text = "";
letters.forEach (function(value) {
  text += value;
})
```

The values() Method

The `values()` method returns an Iterator object with the values in a Set:

Example 1

```
// Create a Set
const letters = new Set(["a", "b", "c"]);

// Get all Values
const myIterator = letters.values();

// List all Values
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

Example 2

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Values
let text = "";
for (const entry of letters.values()) {
  text += entry;
}
```

The keys() Method

The `keys()` method returns an Iterator object with the values in a Set:

Note

A Set has no keys, so `keys()` returns the same as `values()`.

This makes Sets compatible with Maps.

Example 1

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Create an Iterator
const myIterator = letters.keys();

// List all Elements
let text = "";
for (const x of myIterator) {
  text += x;
}
```

Example 2

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Elements
let text = "";
for (const x of letters.keys()) {
  text += x;
}
```

The entries() Method

The `entries()` method returns an Iterator with [value,value] pairs from a Set.

Note

The `entries()` method is supposed to return a [key,value] pair from an object.

A Set has no keys, so the `entries()` method returns [value,value].

This makes Sets compatible with Maps.

Example 1

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Get all Entries
const myIterator = letters.entries();

// List all Entries
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

Example 2

```
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Entries
let text = "";
for (const entry of letters.entries()) {
  text += entry;
}
```

51. JavaScript Maps

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

How to Create a Map

You can create a JavaScript Map by:

- Passing an Array to `new Map()`
- Create a Map and use `Map.set()`

The new Map() Method

You can create a Map by passing an Array to the `new Map()` constructor:

Example

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

The set() Method

You can add elements to a Map with the `set()` method:

Example

```
// Create a Map
const fruits = new Map();
// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);
```

The `set()` method can also be used to change existing Map values:

Example

```
fruits.set("apples", 200);
```

The get() Method

The `get()` method gets the value of a key in a Map:

Example

```
fruits.get("apples"); // Returns 500
```

Maps are Objects

`typeof` returns object:

Example

```
// Returns object:
typeof fruits;
instanceof Map returns true:
```

Example

```
// Returns true:
fruits instanceof Map;
```

JavaScript Objects vs Maps

Differences between JavaScript Objects and Maps:

Object	Map
Not directly iterable	Directly iterable
Do not have a size property	Have a size property
Keys must be Strings (or Symbols)	Keys can be any datatype
Keys are not well ordered	Keys are ordered by insertion
Have default keys	Do not have default keys

Browser Support

Map is an [ES6 feature](#) (JavaScript 2015).

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

Map is not supported in Internet Explorer.

52. JavaScript Map Methods

The new Map() Method

You can create a map by passing an array to the `new Map()` constructor:

Example

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

Map.get()

You get the value of a key in a map with the `get()` method

Example

```
fruits.get("apples");
```

Map.set()

You can add elements to a map with the `set()` method:

Example

```
// Create a Map
const fruits = new Map();
// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);
```

The `set()` method can also be used to change existing map values:

Example

```
fruits.set("apples", 500);
```

Map.size

The `size` property returns the number of elements in a map:

Example

```
fruits.size;
```

Map.delete()

The `delete()` method removes a map element:

Example

```
fruits.delete("apples");
```

Map.clear()

The `clear()` method removes all the elements from a map:

Example

```
fruits.clear();
```

Map.has()

The `has()` method returns true if a key exists in a map:

Example

```
fruits.has("apples");
```

Try This:

```
fruits.delete("apples");
fruits.has("apples");
```

Map.forEach()

The `forEach()` method invokes a callback for each key/value pair in a map:

Example

```
// List all entries
let text = "";
fruits.forEach (function(value, key) {
  text += key + ' = ' + value;
})
```

Map.entries()

The `entries()` method returns an iterator object with the [key,value] in a map:

Example

```
// List all entries
let text = "";
for (const x of fruits.entries()) {
  text += x;
}
```

Map.keys()

The `keys()` method returns an iterator object with the keys in a map:

Example

```
// List all keys
let text = "";
for (const x of fruits.keys()) {
  text += x;
}
```

Map.values()

The `values()` method returns an iterator object with the values in a map:

Example

```
// List all values
let text = "";
for (const x of fruits.values()) {
  text += x;
}
```

You can use the `values()` method to sum the values in a map:

Example

```
// Sum all values
let total = 0;
for (const x of fruits.values()) {
  total += x;
}
```

Objects as Keys

Being able to use objects as keys is an important Map feature.

Example

```
// Create Objects
const apples = {name: 'Apples'};
const bananas = {name: 'Bananas'};
const oranges = {name: 'Oranges'};
// Create a Map
const fruits = new Map();
// Add new Elements to the Map
fruits.set(apples, 500);
fruits.set(bananas, 300);
fruits.set(oranges, 200);
```

Remember: The key is an object (apples), not a string ("apples"):

Example

```
fruits.get("apples"); // Returns undefined
```

JavaScript Map.groupBy()

ES2024 added the `Map.groupBy()` method to JavaScript.

The `Map.groupBy()` method groups elements of an object according to string values returned from a callback function.

The `Map.groupBy()` method does not change the original object.

Example

```
// Create an Array
const fruits = [
  {name:"apples", quantity:300},
  {name:"bananas", quantity:500},
  {name:"oranges", quantity:200},
  {name:"kiwi", quantity:150}
];
// Callback function to Group Elements
function myCallback({ quantity }) {
  return quantity > 200 ? "ok" : "low";
}
// Group by Quantity
const result = Map.groupBy(fruits, myCallback);
```

Browser Support

`Map.groupby()` is an ES2024 feature.

It is supported in new browsers since March 2024:

				
Chrome 117	Edge 117	Firefox 119	Safari 17.4	Opera 103
Sep 2023	Sep 2023	Oct 2023	Okt 2024	May 2023

Warning

ES2024 features are relatively new.

Older browsers may need an alternative code (Polyfill)

Object.groupBy() vs Map.groupBy()

The difference between `Object.groupBy()` and `Map.groupBy()` is:

`Object.groupBy()` groups elements into a JavaScript object.

`Map.groupBy()` groups elements into a Map object.

53. JavaScript `typeof`

The `typeof` Operator

The `typeof` operator returns the **data type** of a JavaScript variable.

Primitive Data Types

In JavaScript, a primitive value is a single value with no properties or methods.

JavaScript has 7 primitive data types:

- string
- number
- boolean
- bigint
- symbol
- null
- undefined

The `typeof` operator returns the type of a variable or an expression.

Examples

```
typeof "John"           // Returns string
typeof ("John"+"Doe")  // Returns string
typeof 3.14             // Returns number
typeof 33               // Returns number
typeof (33 + 66)       // Returns number
typeof true             // Returns boolean
typeof false            // Returns boolean
typeof 1234n           // Returns bigint
typeof Symbol()         // Returns symbol
typeof x                // Returns undefined
typeof null             // Returns object
```

Note:

In JavaScript, `null` is a primitive value. However, `typeof` returns "object".

This is a well-known bug in JavaScript and has historical reasons.

Complex Data Types

A complex data type can store multiple values and/or different data types together.

JavaScript has one complex data type:

- object

All other complex types like arrays, functions, sets, and maps are just different types of objects.

The `typeof` operator returns only two types:

- object
- function

Example

```
typeof {name:'John'}   // Returns object
typeof [1,2,3,4]      // Returns object
typeof new Map()       // Returns object
typeof new Set()        // Returns object

typeof function (){}  // Returns function
```

Note:

The `typeof` operator returns object for all types of objects:
objects
arrays
sets
maps

You cannot use `typeof` to determine if a JavaScript object is an array or a date.

How to Recognize an Array

How to know if a variable is an array?

ECMAScript 5 (2009) defined a new method for this: `Array.isArray()`:

Example

```
// Create an Array
const fruits = ["apples", "bananas", "oranges"];
Array.isArray(fruits);
```

The `instanceof` Operator

The `instanceof` operator returns `true` if an object is an instance of a specified object type:

Examples

```
// Create a Date
const time = new Date();
(time instanceof Date);
// Create an Array
const fruits = ["apples", "bananas", "oranges"];
(fruits instanceof Array);
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
(fruits instanceof Map);
// Create a Set
const fruits = new Set(["apples", "bananas", "oranges"]);
(fruits instanceof Set);
```

Undefined Variables

The `typeof` of an undefined variable is `undefined`.

Example

```
typeof car;
```

The `typeof` of a variable with no value is `undefined`. The value is also `undefined`.

Example

```
let car;
typeof car;
```

Any variable can be emptied, by setting the value to `undefined`.

The type will also be `undefined`.

Example

```
let car = "Volvo";
car = undefined;
```

Empty Values

An empty value has nothing to do with `undefined`.

An empty string has both a legal value and a type.

Example

```
let car = "";
typeof car;
```

Null

In JavaScript `null` is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of `null` is an object.

You can empty an object by setting it to `null`:

Example

```
// Create an Object
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;
// Now value is null, but type is still an object
```

You can also empty an object by setting it to `undefined`:

Example

```
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined;
// Now both value and type is undefined
```

Difference Between Undefined and Null

`undefined` and `null` are equal in value but different in type:

```
typeof undefined      // undefined
typeof null          // object
null === undefined   // false
null == undefined    // true
```

The constructor Property

The `constructor` property returns the constructor function for all JavaScript variables.

Example

```
// Returns function Object() {[native code]}:
{name:'John',age:34}.constructor

// Returns function Array() {[native code]}:
[1,2,3,4].constructor

// Returns function Date() {[native code]}:
new Date().constructor

// Returns function Set() {[native code]}:
new Set().constructor

// Returns function Map() {[native code]}:
new Map().constructor

// Returns function Function() {[native code]}:
function () {}.constructor
```

With the constructor, you can check if an object is an **Array**:

Example

```
(myArray.constructor === Array);
```

With the constructor, you can check if an object is a **Date**:

Example

```
(myDate.constructor === Date);
```

All Together

```
typeof "John"           // Returns "string"
typeof ("John"+"Doe")  // Returns "string"
typeof 3.14             // Returns "number"
typeof (33 + 66)       // Returns "number"
typeof NaN              // Returns "number"
typeof 1234n           // Returns "bigint"
typeof true             // Returns "boolean"
typeof false            // Returns "boolean"
typeof {name: 'John'}   // Returns "object"
typeof [1,2,3,4]        // Returns "object"
typeof {}               // Returns "object"
typeof []               // Returns "object"
typeof new Object()     // Returns "object"
typeof new Array()      // Returns "object"
typeof new Date()       // Returns "object"
typeof new Set()         // Returns "object"
typeof new Map()         // Returns "object"
typeof function () {}  // Returns "function"
typeof x                 // Returns "undefined"
typeof null              // Returns "object"
```

Note:

The data type of **NaN** (Not a Number) is **number** !

The void Operator

The **void** operator evaluates an expression and returns **undefined**. This operator is often used to obtain the undefined primitive value, using "void(0)" (useful when evaluating an expression without using the return value).

Example

```
<a href="javascript:void(0);">
```

Useless link

```
</a>
```

```
<a href="javascript:void(document.body.style.backgroundColor='red');">
```

Click me to change the background color of body to red

```
</a>
```

54. JavaScript Type Conversion

- Converting Strings to Numbers
- Converting Numbers to Strings
- Converting Dates to Numbers
- Converting Numbers to Dates
- Converting Booleans to Numbers
- Converting Numbers to Booleans

JavaScript Type Conversion

JavaScript variables can be converted to a new variable and another data type:

- By the use of a JavaScript function
- **Automatically** by JavaScript itself

Converting Strings to Numbers

The global method `Number()` converts a variable (or a value) into a number.

A numeric string (like "3.14") converts to a number (like 3.14).

An empty string (like "") converts to 0.

A non numeric string (like "John") converts to `NaN` (Not a Number).

Examples

These will convert:

```
Number("3.14")
Number(Math.PI)
Number(" ")
Number("")
```

These will not convert:

```
Number("99 88")
Number("John")
```

Number Methods

In the chapter [Number Methods](#), you will find more methods that can be used to convert strings to numbers:

Method	Description
Number()	Returns a number, converted from its argument
parseFloat()	Parses a string and returns a floating point number
parseInt()	Parses a string and returns an integer

The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

Example

```
let y = "5";      // y is a string
let x = + y;      // x is a number
```

If the variable cannot be converted, it will still become a number, but with the value **NaN** (Not a Number):

Example

```
let y = "John";   // y is a string
let x = + y;      // x is a number (NaN)
```

Converting Numbers to Strings

The global method `String()` can convert numbers to strings.

It can be used on any type of numbers, literals, variables, or expressions:

Example

```
String(x)      // returns a string from a number variable x  
String(123)    // returns a string from a number literal 123  
String(100 + 23) // returns a string from a number from an expression
```

The Number method `toString()` does the same.

Example

```
x.toString()  
(123).toString()  
(100 + 23).toString()
```

More Methods

In the chapter [Number Methods](#), you will find more methods that can be used to convert numbers to strings:

Method	Description
<code>toExponential()</code>	Returns a string, with a number rounded and written using exponential notation.
<code>toFixed()</code>	Returns a string, with a number rounded and written with a specified number of decimals.
<code>toPrecision()</code>	Returns a string, with a number written with a specified length

Converting Dates to Numbers

The global method `Number()` can be used to convert dates to numbers.

```
d = new Date();
Number(d)           // returns 1404568027739
```

The date method `getTime()` does the same.

```
d = new Date();
d.getTime()         // returns 1404568027739
```

Converting Dates to Strings

The global method `String()` can convert dates to strings.

```
String(Date()) // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight
Time)"
```

The Date method `toString()` does the same.

Example

```
Date().toString() // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight
Time)"
```

In the chapter [Date Methods](#), you will find more methods that can be used to convert dates to strings:

Method	Description
<code>getDate()</code>	Get the day as a number (1-31)
<code>getDay()</code>	Get the weekday a number (0-6)
<code>getFullYear()</code>	Get the four digit year (yyyy)
<code>getHours()</code>	Get the hour (0-23)

getMilliseconds()	Get the milliseconds (0-999)
getMinutes()	Get the minutes (0-59)
getMonth()	Get the month (0-11)
getSeconds()	Get the seconds (0-59)
getTime()	Get the time (milliseconds since January 1, 1970)

Converting Booleans to Numbers

The global method `Number()` can also convert booleans to numbers.

```
Number(false)      // returns 0
Number(true)       // returns 1
```

Converting Booleans to Strings

The global method `String()` can convert booleans to strings.

```
String(false)      // returns "false"
String(true)       // returns "true"
```

The Boolean method `toString()` does the same.

```
false.toString()  // returns "false"
true.toString()    // returns "true"
```

Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

The result is not always what you expect:

```
5 + null      // returns 5          because null is converted to 0
"5" + null    // returns "5null"   because null is converted to "null"
"5" + 2       // returns "52"        because 2 is converted to "2"
"5" - 2       // returns 3          because "5" is converted to 5
"5" * "2"     // returns 10         because "5" and "2" are converted to 5 and 2
```

Automatic String Conversion

JavaScript automatically calls the variable's `toString()` function when you try to "output" an object or a variable:

```
document.getElementById("demo").innerHTML = myVar;  
  
// if myVar = {name:"Fjohn"} // toString converts to "[object Object]"  
// if myVar = [1,2,3,4]      // toString converts to "1,2,3,4"  
// if myVar = new Date()     // toString converts to "Fri Jul 18 2014 09:08:55  
                           // GMT+0200"
```

Numbers and booleans are also converted, but this is not very visible:

```
// if myVar = 123           // toString converts to "123"  
// if myVar = true          // toString converts to "true"  
// if myVar = false         // toString converts to "false"
```

JavaScript Type Conversion Table

This table shows the result of converting different JavaScript values to Number, String, and Boolean:

Original Value	Converted to Number	Converted to String	Converted to Boolean
false	0	"false"	false
true	1	"true"	true
0	0	"0"	false
1	1	"1"	true
"0"	0	"0"	true

"000"	0	"000"	true
-------	---	-------	-------------

"1"	1	"1"	true
NaN	NaN	"NaN"	false
Infinity	Infinity	"Infinity"	true
-Infinity	-Infinity	"-Infinity"	true
""	**0**	""	**false**
"20"	20	"20"	true
"twenty"	NaN	"twenty"	true
[]	**0**	""	true
[20]	**20**	"20"	true
[10,20]	NaN	"10,20"	true
["twenty"]	NaN	"twenty"	true

["ten","twenty"]	NaN	"ten,twenty"	true
------------------	-----	--------------	------

function(){} {} null undefined	NaN NaN 0 NaN	"function(){}" "[object Object]" "null" "undefined"	true true false false
---	------------------------	--	--------------------------------

Values in quotes indicate string values.

Red values indicate values (some) programmers might not expect.

55. JavaScript Destructuring

Destructuring Assignment Syntax

The destructuring assignment syntax unpack object properties into variables:

```
let {firstName, lastName} = person;
```

It can also unpack arrays and any other iterables:

```
let [firstName, lastName] = person;
```

Object Destructuring

Example

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
// Destructuring
let {firstName, lastName} = person;
```

The order of the properties does not matter:

Example

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
// Destructuring
let {lastName, firstName} = person;
```

Note:

Destructuring is not destructive.

Destructuring does not change the original object.

Object Default Values

For potentially missing properties we can set default values:

Example

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
// Destructuring
let {firstName, lastName, country = "US"} = person;
```

Object Property Alias

Example

```
// Create an Object
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
// Destructuring
let {lastName : name} = person;
```

String Destructuring

One use for destructuring is unpacking string characters.

Example

```
// Create a String
let name = "W3Schools";
// Destructuring
let [a1, a2, a3, a4, a5] = name;
```

Note:

Destructuring can be used with any iterables.

Array Destructuring

We can pick up array variables into our own variables:

Example

```
// Create an Array
const fruits = ["Bananas", "Oranges", "Apples", "Mangos"];

// Destructuring
let [fruit1, fruit2] = fruits;
```

Skipping Array Values

We can skip array values using two or more commas:

Example

```
// Create an Array
const fruits = ["Bananas", "Oranges", "Apples", "Mangos"];
// Destructuring
let [fruit1,,,fruit2] = fruits;
```

Array Position Values

We can pick up values from specific index locations of an array:

Example

```
// Create an Array
const fruits = ["Bananas", "Oranges", "Apples", "Mangos"];
// Destructuring
let {[0]:fruit1 ,[1]:fruit2} = fruits;
```

The Rest Property

You can end a destructuring syntax with a rest property.

This syntax will store all remaining values into a new array:

Example

```
// Create an Array
const numbers = [10, 20, 30, 40, 50, 60, 70];

// Destructuring
const [a,b, ...rest] = numbers
```

Destructuring Maps

Example

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
// Destructing
let text = "";
for (const [key, value] of fruits) {
  text += key + " is " + value;
}
```

Swapping JavaScript Variables

You can swap the values of two variables using a destructuring assignment:

Example

```
let firstName = "John";
let lastName = "Doe";

// Destructing
[firstName, lastName] = [lastName, firstName];
```

56. JavaScript Bitwise Operations

JavaScript Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Examples

Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
5 1	5	0101 0001	0101
~ 5	10	~ 0101	1010
$5 \ll 1$	10	$0101 \ll 1$	1010
$5 \wedge 1$	4	$0101 \wedge 0001$	0100
$5 \gg 1$	2	$0101 \gg 1$	0010
$5 \ggg 1$	2	$0101 \ggg 1$	0010

JavaScript Uses 32 bits Bitwise Operands

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.

Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers.

After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.

The examples above uses 4 bits unsigned binary numbers. Because of this ~ 5 returns 10. Since JavaScript uses 32 bits signed integers, it will not return 10. It will return -6.

0000000000000000000000000000101 (5)

11111111111111111111111111010 ($\sim 5 = -6$)

A signed integer uses the leftmost bit as the minus sign.

JavaScript Bitwise AND

When a bitwise AND is performed on a pair of bits, it returns 1 if both bits are 1.

One bit example:

Operation	Result
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

4 bits example:

Operation	Result
1111 & 0000	0000
1111 & 0001	0001
1111 & 0010	0010
1111 & 0100	0100

JavaScript Bitwise OR

When a bitwise OR is performed on a pair of bits, it returns 1 if one of the bits is 1:

One bit example:

Operation	Result
0 0	0
0 1	1
1 0	1
1 1	1

4 bits example:

Operation	Result
1111 0000	1111
1111 0001	1111
1111 0010	1111
1111 0100	1111

JavaScript Bitwise XOR

When a bitwise XOR is performed on a pair of bits, it returns 1 if the bits are different:

One bit example:

Operation	Result
$0 \wedge 0$	0
$0 \wedge 1$	1
$1 \wedge 0$	1
$1 \wedge 1$	0

4 bits example:

Operation	Result
$1111 \wedge 0000$	1111
$1111 \wedge 0001$	1110
$1111 \wedge 0010$	1101
$1111 \wedge 0100$	1011

JavaScript Bitwise AND (&)

Bitwise AND returns 1 only if both bits are 1:

Decimal	Binary
5	00000000000000000000000000000000101
1	00000000000000000000000000000000001
5 & 1	00000000000000000000000000000000001 (1)

Example

```
let x = 5 & 1;
```

JavaScript Bitwise OR (|)

Bitwise OR returns 1 if one of the bits is 1:

Decimal	Binary
5	00000000000000000000000000000000101
1	00000000000000000000000000000000001
5 1	00000000000000000000000000000000101 (5)

Example

```
let x = 5 | 1;
```

JavaScript Bitwise XOR (^)

Bitwise XOR returns 1 if the bits are different:

Decimal	Binary
5	00000000000000000000000000000000101
1	00000000000000000000000000000000001
5 ^ 1	00000000000000000000000000000000100 (4)

Example

```
let x = 5 ^ 1;
```

JavaScript Bitwise NOT (~)

Decimal	Binary
5	00000000000000000000000000000000101
~5	111111111111111111111111111111010 (-6)

Example

```
let x = ~5;
```

JavaScript (Zero Fill) Bitwise Left Shift (<<)

This is a zero fill left shift. One or more zero bits are pushed in from the right, and the leftmost bits fall off:

Example

```
let x = 5 << 1;
```

JavaScript (Sign Preserving) Bitwise Right Shift (>>)

This is a sign preserving right shift. Copies of the leftmost bit are pushed in from the left, and the rightmost bits fall off:

Example

```
let x = -5 >> 1;
```

JavaScript (Zero Fill) Right Shift (>>>)

This is a zero fill right shift. One or more zero bits are pushed in from the left, and the rightmost bits fall off:

Example

```
let x = 5 >>> 1;
```

Binary Numbers

Binary numbers with only one bit set are easy to understand:

Setting a few more bits reveals the binary pattern:

JavaScript binary numbers are stored in two's complement format.

This means that a negative number is the bitwise NOT of the number plus 1:

6

-6

40

11111111111111111111111111111111011000

-40

Joke:

There are only 10 types of people in the world: those who understand binary and those who don't.

Converting Decimal to Binary

Example

```
function dec2bin(dec){  
    return (dec >>> 0).toString(2);  
}
```

Converting Binary to Decimal

Example

```
function bin2dec(bin){  
    return parseInt(bin, 2).toString(10);  
}
```

57. JavaScript Regular Expressions

A regular expression is a sequence of characters that forms a search pattern.

The search pattern can be used for text search and text replace operations.

What Is a Regular Expression?

A regular expression is a sequence of characters that forms a **search pattern**.

When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Syntax

/pattern/modifiers;

Example

```
/w3schools/i;
```

Example explained:

/w3schools/i is a regular expression.

w3schools is a pattern (to be used in a search).

i is a modifier (modifies the search to be case-insensitive).

Using String Methods

In JavaScript, regular expressions are often used with the two **string methods**: `search()` and `replace()`.

The `search()` method uses an expression to search for a match, and returns the position of the match.

The `replace()` method returns a modified string where the pattern is replaced.

Using String search() With a String

The `search()` method searches a string for a specified value and returns the position of the match:

Example

Use a string to do a search for "W3schools" in a string:

```
let text = "Visit W3Schools!";
let n = text.search("W3Schools");
```

The result in `n` will be:

6

Using String search() With a Regular Expression

Example

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

```
let text = "Visit W3Schools";
let n = text.search(/w3schools/i);
```

The result in `n` will be:

6

Using String replace() With a String

The `replace()` method replaces a specified value with another value in a string:

```
let text = "Visit Microsoft!";
let result = text.replace("Microsoft", "W3Schools");
```

Use String replace() With a Regular Expression

Example

Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

```
let text = "Visit Microsoft!";
let result = text.replace(/microsoft/i, "W3Schools");
```

The result in `res` will be:

Visit W3Schools!

Did You Notice?

Regular expression arguments (instead of string arguments) can be used in the methods above.

Regular expressions can make your search much more powerful (case insensitive for example).

Regular Expression Modifiers

Modifiers can be used to perform case-insensitive more global searches:

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all)
m	Perform multiline matching
d	Perform start and end matching (New in ES2022)

Regular Expression Patterns

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find any of the characters between the brackets
[0-9]	Find any of the digits between the brackets
(x y)	Find any of the alternatives separated with

Metacharacters are characters with a special meaning:

Metacharacter	Description
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one <i>n</i>
n*	Matches any string that contains zero or more occurrences of <i>n</i>
n?	Matches any string that contains zero or one occurrences of <i>n</i>

Using the RegExp Object

In JavaScript, the `RegExp` object is a regular expression object with predefined properties and methods.

Using test()

The `test()` method is a RegExp expression method.

It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":

Example

```
const pattern = /e/;  
pattern.test("The best things in life are free!");
```

Since there is an "e" in the string, the output of the code above will be:

true

You don't have to put the regular expression in a variable first. The two lines above can be shortened to one:

```
/e/.test("The best things in life are free!");
```

Using exec()

The `exec()` method is a RegExp expression method.

It searches a string for a specified pattern, and returns the found text as an object.

If no match is found, it returns an empty (`null`) object.

The following example searches a string for the character "e":

Example

```
/e/.exec("The best things in life are free!");
```

58. JavaScript Operator Precedence

Operator precedence describes the order in which operations are performed in an arithmetic expression.

Multiplication (*) and division (/) have higher **precedence** than addition (+) and subtraction (-).

As in traditional mathematics, multiplication is done first:

```
let x = 100 + 50 * 3;
```

When using parentheses, operations inside the parentheses are computed first:

```
let x = (100 + 50) * 3;
```

Operations with the same precedence (like * and /) are computed from left to right:

```
let x = 100 / 50 * 3;
```

Operator Precedence Values

Expressions in parentheses are computed **before** the rest of the expression

Function are executed **before** the result is used in the rest of the expression

Val	Operator	Description	Example
18	()	Expression Grouping	(100 + 50) * 3
17	.	Member Of	person.name
17	[]	Member Of	person["name"]
17	?.	Optional Chaining ES2020	x ?. y
17	()	Function Call	myFunction()
17	new	New with Arguments	new Date("June 5,2022")
16	new	New without Arguments	new Date()

Increment Operators

Postfix increments are executed **before** prefix increments

15	++	Postfix Increment	i++
15	--	Postfix Decrement	i--
14	++	Prefix Increment	++i
14	--	Prefix Decrement	--i

NOT Operators

14 ! [Logical NOT](#) !(x==y)

14 ~ [Bitwise NOT](#) ~x

Unary Operators

14 + [Unary Plus](#) +x

14 - [Unary Minus](#) -x

14 typeof [Data Type](#) typeof x

14 void [Evaluate Void](#) void(0)

14 delete [Property Delete](#) delete myCar.color

Arithmetic Operators

Exponentiations are executed **before** multiplications

Multiplications and divisions are executed **before** additions and subtractions

13 ** [Exponentiation ES2016](#) 10 ** 2

12 * [Multiplication](#) 10 * 5

12 / [Division](#) 10 / 5

12 % [Division Remainder](#) 10 % 5

11 + [Addition](#) 10 + 5

11 - [Subtraction](#) 10 - 5

11 + [Concatenation](#) "John" + "Doe"

Shift Operators

10	<<	Shift Left	x << 2
10	>>	Shift Right (signed)	x >> 2
10	>>>	Shift Right (unsigned)	x >>> 2

Relational Operators

9	in	Property in Object	"PI" in Math
9	instanceof	Instance of Object	x instanceof Array

Comparison Operators

9	<	Less than	x < y
9	<=	Less than or equal	x <= y
9	>	Greater than	x > y
9	>=	Greater than or equal	x >= Array
8	==	Equal	x == y
8	====	Strict equal	x === y
8	!=	Unequal	x != y
8	!==	Strict unequal	x !== y

Bitwise Operators

7	&	Bitwise AND	x & y
6	^	Bitwise XOR	x ^ y

5		Bitwise OR	x y
---	--	----------------------------	-------

Logical Operators

4	&&	Logical AND	x && y
---	----	-----------------------------	--------

3		Logical OR	x y
---	--	----------------------------	--------

3	??	Nullish Coalescing ES2020	x ?? y
---	----	---	--------

Conditional (ternary) Operator

2	? :	Condition	? "yes" : "no"
---	-----	---------------------------	----------------

Assignment Operators

Assignments are executed **after** other operations

2	=	Simple Assignment	x = y
---	---	-----------------------------------	-------

2	:	Colon Assignment	x: 5
---	---	----------------------------------	------

2	+=	Addition Assignment	x += y
---	----	-------------------------------------	--------

2	-=	Subtraction Assignment	x -= y
---	----	--	--------

2	*=	Multiplication Assignment	x *= y
---	----	---	--------

2	**=	Exponentiation Assignment	x **= y
---	-----	---	---------

2	/=	Division Assignment	x /= y
---	----	-------------------------------------	--------

2	%=	Remainder Assignment	x %= y
---	----	--------------------------------------	--------

2	<=	Left Shift Assignment	x <= y
---	----	---------------------------------------	--------

2	>=	Right Shift Assignment	x >= y
---	----	--	--------

2	>>=	Unsigned Right Shift	x >>= y
---	-----	--------------------------------------	---------

2	<code>&=</code>	Bitwise AND Assignment	<code>x &= y</code>
2	<code> =</code>	Bitwise OR Assignment	<code>x = y</code>
2	<code>^=</code>	Bitwise XOR Assignment	<code>x ^= y</code>
2	<code>&&=</code>	Logical AND Assignment	<code>x &&= y</code>
2	<code> =</code>	Logical OR Assignment	<code>x = y</code>
2	<code>=></code>	Arrow	<code>x => y</code>
2	<code>yield</code>	Pause / Resume	<code>yield x</code>
2	<code>yield*</code>	Delegate	<code>yield* x</code>
2	<code>...</code>	Spread	<code>... x</code>
1	<code>,</code>	Comma	<code>x, y</code>

59. JavaScript Errors

Throw, and Try...Catch...Finally

The `try` statement defines a code block to run (to try).

The `catch` statement defines a code block to handle any error.

The `finally` statement defines a code block to run regardless of the result.

The `throw` statement defines a custom error.

Errors Will Happen!

When executing JavaScript code, different errors can occur.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

Example

In this example we misspelled "alert" as "addrlert" to deliberately produce an error:

```
<p id="demo"></p>
<script>
try {
    addrlert("Welcome guest!");
}
catch(err) {
    document.getElementById("demo").innerHTML = err.message;
}
</script>
```

JavaScript catches **addrlert** as an error, and executes the catch code to handle it.

JavaScript try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements `try` and `catch` come in pairs:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
```

JavaScript Throws Errors

When an error occurs, JavaScript will normally stop and generate an error message.

The technical term for this is: JavaScript will **throw an exception (throw an error)**.

JavaScript will actually create an **Error object** with two properties: **name** and **message**.

The throw Statement

The **throw** statement allows you to create a custom error.

Technically you can **throw an exception (throw an error)**.

The exception can be a JavaScript **String**, a **Number**, a **Boolean** or an **Object**:

```
throw "Too big";      // throw a text
throw 500;            // throw a number
```

If you use **throw** together with **try** and **catch**, you can control program flow and generate custom error messages.

Input Validation Example

This example examines input. If the value is wrong, an exception (err) is thrown.

The exception (err) is caught by the catch statement and a custom error message is displayed:

```
<!DOCTYPE html>
<html>
<body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>

<script>
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x.trim() == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>

</body>
</html>
```

HTML Validation

The code above is just an example.

Modern browsers will often use a combination of JavaScript and built-in HTML validation, using predefined validation rules defined in HTML attributes:

```
<input id="demo" type="number" min="5" max="10" step="1">
```

You can read more about forms validation in a later chapter of this tutorial.

The finally Statement

The `finally` statement lets you execute code, after try and catch, regardless of the result:

Syntax

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

Example

```
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x.trim() == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Error: " + err + ".";
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
```

The Error Object

JavaScript has a built in error object that provides error information when an error occurs. The error object provides two useful properties: name and message.

Error Object Properties

Property	Description
name	Sets or returns an error name
message	Sets or returns an error message (a string)

Error Name Values

Six different values can be returned by the error name property:

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

The six different values are described below.

Eval Error

An `EvalError` indicates an error in the `eval()` function.

Newer versions of JavaScript do not throw `EvalError`. Use `SyntaxError` instead.

Range Error

A `RangeError` is thrown if you use a number that is outside the range of legal values.

For example: You cannot set the number of significant digits of a number to 500.

Example

```
let num = 1;
try {
  num.toPrecision(500);    // A number cannot have 500 significant digits
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Reference Error

A `ReferenceError` is thrown if you use (reference) a variable that has not been declared:

Example

```
let x = 5;
try {
  x = y + 1;    // y cannot be used (referenced)
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Syntax Error

A `SyntaxError` is thrown if you try to evaluate code with a syntax error.

Example

```
try {
  eval("alert('Hello')");    // Missing ' will produce an error
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Type Error

A **TypeError** is thrown if an operand or argument is incompatible with the type expected by an operator or function.

Example

```
let num = 1;
try {
    num.toUpperCase(); // You cannot convert a number to upper case
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

URI (Uniform Resource Identifier) Error

A **URIError** is thrown if you use illegal characters in a URI function:

Example

```
try {
    decodeURI("%%%"); // You cannot URI decode percent signs
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

Non-Standard Error Object Properties

Mozilla and Microsoft define some non-standard error object properties:

- fileName (Mozilla)
- lineNumber (Mozilla)
- columnNumber (Mozilla)
- stack (Mozilla)
- description (Microsoft)
- number (Microsoft)

Do not use these properties in public web sites. They will not work in all browsers.

60. JavaScript Scope

Scope determines the accessibility (visibility) of variables.

JavaScript variables have 3 types of scope:

- Block scope
- Function scope
- Global scope

Block Scope

Before ES6 (2015), JavaScript variables had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: `let` and `const`.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a `{ }` block cannot be accessed from outside the block:

Example

```
{  
  let x = 2;  
}  
  
// x can NOT be used here
```

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a `{ }` block can be accessed from outside the block.

Example

```
{  
  var x = 2;  
}  
  
// x CAN be used here
```

Local Scope

Variables declared within a JavaScript function, are **LOCAL** to the function:

Example

```
// code here can NOT use carName  
function myFunction() {  
  let carName = "Volvo";  
  // code here CAN use carName  
}  
  
// code here can NOT use carName
```

Local variables have **Function Scope**:

They can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

Function Scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared with `var`, `let` and `const` are quite similar when declared inside a function.

They all have **Function Scope**:

```
function myFunction() {  
    var carName = "Volvo"; // Function Scope  
}  
function myFunction() {  
    let carName = "Volvo"; // Function Scope  
}  
function myFunction() {  
    const carName = "Volvo"; // Function Scope  
}
```

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

Example

```
let carName = "Volvo";  
// code here can use carName  
  
function myFunction() {  
// code here can also use carName  
}
```

A global variable has **Global Scope**:

All scripts and functions on a web page can access it.

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

Global variables can be accessed from anywhere in a JavaScript program.

Variables declared with `var`, `let` and `const` are quite similar when declared outside a block.

They all have **Global Scope**:

```
var x = 2; // Global scope  
let x = 2; // Global scope  
const x = 2; // Global scope
```

JavaScript Variables

In JavaScript, objects and functions are also variables.

Scope determines the accessibility of variables, objects, and functions from different parts of the code.

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable `carName`, even if the value is assigned inside a function.

Example

```
myFunction();  
  
// code here can use carName  
  
function myFunction() {  
    carName = "Volvo";  
}
```

Strict Mode

All modern browsers support running JavaScript in "Strict Mode".

You will learn more about how to use strict mode in a later chapter of this tutorial.

In "Strict Mode", undeclared variables are not automatically global.

Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the `var` keyword belong to the window object:

Example

```
var carName = "Volvo";
// code here can use window.carName
```

Global variables defined with the `let` keyword do not belong to the window object:

Example

```
let carName = "Volvo";
// code here can not use window.carName
```

Warning

Do NOT create global variables unless you intend to.

Your global variables (or functions) can overwrite window variables (or functions).

Any function, including the window object, can overwrite your global variables and functions.

The Lifetime of JavaScript Variables

The lifetime of a JavaScript variable starts when it is declared.

Function (local) variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser window (or tab).

Function Arguments

Function arguments (parameters) work as local variables inside functions.

61. JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top.

JavaScript Declarations are Hoisted

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

Example 1 gives the same result as **Example 2**:

Example 1

```
x = 5; // Assign 5 to x
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x; // Display x in the element
var x; // Declare x
```

Example 2

```
var x; // Declare x
x = 5; // Assign 5 to x
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x; // Display x in the element
```

To understand this, you have to understand the term "hoisting".

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

The let and const Keywords

Variables defined with `let` and `const` are hoisted to the top of the block, but not *initialized*.

Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a `let` variable before it is declared will result in a [ReferenceError](#).

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Example

This will result in a [ReferenceError](#):

```
carName = "Volvo";
let carName;
```

Using a `const` variable before it is declared, is a syntax error, so the code will simply not run.

Example

This code will not run.

```
carName = "Volvo";
const carName;
```

JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

Example 1 does **not** give the same result as **Example 2**:

Example 1

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y; // Display x and y
```

Example 2

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

var y = 7; // Initialize y
```

Does it make sense that y is undefined in the last example?

This is because only the declaration (var y), not the initialization (=7) is hoisted to the top. Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

Example 2 is the same as writing:

Example

```
var x = 5; // Initialize x
var y;      // Declare y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

y = 7;      // Assign 7 to y
```

Declare Your Variables At the Top !

Hoisting is (to many developers) an unknown or overlooked behavior of JavaScript. If a developer doesn't understand hoisting, programs may contain bugs (errors). To avoid bugs, always declare all variables at the beginning of every scope. Since this is how JavaScript interprets the code, it is always a good rule.

62. JavaScript Use Strict

"`use strict`"; Defines that JavaScript code should be executed in "strict mode".

The "`use strict`" Directive

The "`use strict`" directive was new in ECMAScript version 5.

It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

The purpose of "`use strict`" is to indicate that the code should be executed in "strict mode". With strict mode, you can not, for example, use undeclared variables.

All modern browsers support "use strict" except Internet Explorer 9 and lower:

Directive					
" <code>use strict</code> "	13.0	10.0	4.0	6.0	12.1

The numbers in the table specify the first browser version that fully supports the directive.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"`use strict`" is just a string, so IE 9 will not throw an error even if it does not understand it.

Declaring Strict Mode

Strict mode is declared by adding "`use strict`"; to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

Example

```
"use strict";
x = 3.14;      // This will cause an error because x is not declared
```

Example

```
"use strict";
myFunction();
function myFunction() {
  y = 3.14;  // This will also cause an error because y is not declared
}
```

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

```
x = 3.14;           // This will not cause an error.  
myFunction();  
  
function myFunction() {  
    "use strict";  
    y = 3.14;    // This will cause an error  
}
```

The "use strict"; Syntax

The syntax, for declaring strict mode, was designed to be compatible with older versions of JavaScript.

Compiling a numeric literal (4 + 5;) or a string literal ("John Doe";) in a JavaScript program has no side effects. It simply compiles to a non existing variable and dies.

So "use strict"; only matters to new compilers that "understand" the meaning of it.

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Not Allowed in Strict Mode

Using a variable, without declaring it, is not allowed:

```
"use strict";  
x = 3.14;           // This will cause an error
```

Objects are variables too.

Using an object, without declaring it, is not allowed:

```
"use strict";
x = {p1:10, p2:20};           // This will cause an error
```

Deleting a variable (or object) is not allowed.

```
"use strict";
let x = 3.14;
delete x;                  // This will cause an error
```

Deleting a function is not allowed.

```
"use strict";
function x(p1, p2) {};
delete x;                  // This will cause an error
```

Duplicating a parameter name is not allowed:

```
"use strict";
function x(p1, p1) {};    // This will cause an error
```

Octal numeric literals are not allowed:

```
"use strict";
let x = 010;              // This will cause an error
```

Octal escape characters are not allowed:

```
"use strict";
let x = "\010";           // This will cause an error
```

Writing to a read-only property is not allowed:

```
"use strict";
const obj = {};
Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14;             // This will cause an error
```

Writing to a get-only property is not allowed:

```
"use strict";
const obj = {get x() {return 0} };

obj.x = 3.14;             // This will cause an error
```

Deleting an undeletable property is not allowed:

```
"use strict";
delete Object.prototype; // This will cause an error
```

The word `eval` cannot be used as a variable:

```
"use strict";
let eval = 3.14;           // This will cause an error
```

The word `arguments` cannot be used as a variable:

```
"use strict";
let arguments = 3.14;     // This will cause an error
```

The `with` statement is not allowed:

```
"use strict";
with (Math){x = cos(2)}; // This will cause an error
```

For security reasons, `eval()` is not allowed to create variables in the scope from which it was called.

In strict mode, a variable can not be used before it is declared:

```
"use strict";
eval ("x = 2");
alert (x);      // This will cause an error
```

In strict mode, `eval()` can not declare a variable using the `var` keyword:

```
"use strict";
eval ("var x = 2");
alert (x);      // This will cause an error
```

`eval()` can not declare a variable using the `let` keyword:

```
eval ("let x = 2");
alert (x);      // This will cause an error
```

The `this` keyword in functions behaves differently in strict mode.

The `this` keyword refers to the object that called the function.

If the object is not specified, functions in strict mode will return `undefined` and functions in normal mode will return the global object (`window`):

```
"use strict";
function myFunction() {
  alert(this); // will alert "undefined"
}
myFunction();
```

Future Proof!

Keywords reserved for future JavaScript versions can NOT be used as variable names in strict mode.

These are:

- `implements`
- `interface`
- `let`
- `package`
- `private`
- `protected`
- `public`
- `static`
- `yield`

```
"use strict";
let public = 1500;      // This will cause an error
```

Watch Out!

The `"use strict"` directive is only recognized at the **beginning** of a script or a function.

63. The JavaScript this Keyword

Example

```
const person = {  
    firstName: "John",  
    lastName : "Doe",  
    id       : 5566,  
    fullName : function() {  
        return this.firstName + " " + this.lastName;  
    }  
};
```

What is this?

In JavaScript, the `this` keyword refers to an **object**.

The `this` keyword refers to **different objects** depending on how it is used:

In an object method, `this` refers to the **object**.

Alone, `this` refers to the **global object**.

In a function, `this` refers to the **global object**.

In a function, in strict mode, `this` is **undefined**.

In an event, `this` refers to the **element** that received the event.

Methods like `call()`, `apply()`, and `bind()` can refer `this` to **any object**.

Note

`this` is not a variable. It is a keyword. You cannot change the value of `this`.

this in a Method

When used in an object method, `this` refers to the **object**.

In the example on top of this page, `this` refers to the **person** object.

Because the **fullName** method is a method of the **person** object.

```
fullName : function() {  
    return this.firstName + " " + this.lastName;  
}
```

this Alone

When used alone, `this` refers to the **global object**.

Because `this` is running in the global scope.

In a browser window the global object is `[object Window]`:

Example

```
let x = this;
```

In **strict mode**, when used alone, `this` also refers to the **global object**:

Example

```
"use strict";  
let x = this;
```

this in a Function (Default)

In a function, the **global object** is the default binding for `this`.

In a browser window the global object is `[object Window]`:

Example

```
function myFunction() {  
    return this;  
}
```

this in a Function (Strict)

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, `this` is `undefined`.

Example

```
"use strict";
function myFunction() {
    return this;
}
```

this in Event Handlers

In HTML event handlers, `this` refers to the HTML element that received the event:

Example

```
<button onclick="this.style.display='none'">
    Click to Remove Me!
</button>
```

Object Method Binding

In these examples, `this` is the **person object**:

Example

```
const person = {
    firstName : "John",
    lastName  : "Doe",
    id        : 5566,
    myFunction: function() {
        return this;
    }
};
```

Example

```
const person = {
    firstName: "John",
    lastName : "Doe",
    id       : 5566,
    fullName : function() {
        return this.firstName + " " + this.lastName;
    }
};
```

i.e. `this.firstName` is the **firstName** property of `this` (the person object).

Explicit Function Binding

The `call()` and `apply()` methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.

The example below calls `person1.fullName` with `person2` as an argument, `this` refers to `person2`, even if `fullName` is a method of `person1`:

Example

```
const person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}

const person2 = {
  firstName: "John",
  lastName: "Doe",
}

// Return "John Doe":
person1.fullName.call(person2);
```

Function Borrowing

With the `bind()` method, an object can borrow a method from another object.

This example creates 2 objects (`person` and `member`).

The `member` object borrows the `fullname` method from the `person` object:

Example

```
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}

const member = {
  firstName: "Hege",
  lastName: "Nilsen",
}

let fullName = person.fullName.bind(member);
```

This Precedence

To determine which object `this` refers to; use the following precedence of order.

Precedence	Object
1	<code>bind()</code>
2	<code>apply()</code> and <code>call()</code>
3	Object method
4	Global scope

Is `this` in a function being called using `bind()`?

Is `this` in a function being called using `apply()`?

Is `this` in a function being called using `call()`?

Is `this` in an object function (method)?

Is `this` in a function in the global scope.

64. JavaScript Arrow Function

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

```
let myFunction = (a, b) => a * b;
```

Before Arrow:

```
hello = function() {
    return "Hello World!";
}
```

With Arrow Function:

```
hello = () => {
    return "Hello World!";
}
```

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the `return` keyword:

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Note: This works only if the function has only one statement.

If you have parameters, you pass them inside the parentheses:

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

In fact, if you have only one parameter, you can skip the parentheses as well:

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

What About `this`?

The handling of `this` is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of `this`.

In regular functions the `this` keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the `this` keyword *always* represents the object that defined the arrow function.

Let us take a look at two examples to understand the difference.

Both examples call a method twice, first when the page loads, and once again when the user clicks a button.

The first example uses a regular function, and the second example uses an arrow function.

The result shows that the first example returns two different objects (window and button), and the second example returns the window object twice, because the window object is the "owner" of the function.

Example

With a regular function `this` represents the object that *calls* the function:

```
// Regular Function:  
hello = function() {  
    document.getElementById("demo").innerHTML += this;  
}  
// The window object calls the function:  
window.addEventListener("load", hello);  
// A button object calls the function:  
document.getElementById("btn").addEventListener("click", hello);
```

Example

With an arrow function `this` represents the *owner* of the function:

```
// Arrow Function:  
hello = () => {  
    document.getElementById("demo").innerHTML += this;  
}  
// The window object calls the function:  
window.addEventListener("load", hello);  
// A button object calls the function:  
document.getElementById("btn").addEventListener("click", hello);
```

Remember these differences when you are working with functions. Sometimes the behavior of regular functions is what you want, if not, use arrow functions.

Browser Support

The following table defines the first browser versions with full support for Arrow Functions in JavaScript:

				
Chrome 45	Edge 12	Firefox 22	Safari 10	Opera 32
Sep, 2015	Jul, 2015	May, 2013	Sep, 2016	Sep, 2015

65. JavaScript Classes

ECMAScript 2015, also known as ES6, introduced JavaScript Classes.

JavaScript Classes are templates for JavaScript Objects.

JavaScript Class Syntax

Use the keyword `class` to create a class.

Always add a method named `constructor()`:

Syntax

```
class ClassName {  
    constructor() { ... }  
}
```

Example

```
class Car {  
    constructor(name, year) {  
        this.name = name;  
        this.year = year;  
    }  
}
```

The example above creates a class named "Car".

The class has two initial properties: "name" and "year".

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

Using a Class

When you have a class, you can use the class to create objects:

Example

```
const myCar1 = new Car("Ford", 2014);  
const myCar2 = new Car("Audi", 2019);
```

The example above uses the **Car class** to create two **Car objects**.

The constructor method is called automatically when a new object is created.

The Constructor Method

The constructor method is a special method:

- It has to have the exact name "constructor"
- It is executed automatically when a new object is created
- It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

Class Methods

Class methods are created with the same syntax as object methods.

Use the keyword `class` to create a class.

Always add a `constructor()` method.

Then add any number of methods.

Syntax

```
class ClassName {  
    constructor() { ... }  
    method_1() { ... }  
    method_2() { ... }  
    method_3() { ... }  
}
```

Create a Class method named "age", that returns the Car age:

Example

```
class Car {  
    constructor(name, year) {  
        this.name = name;  
        this.year = year;  
    }  
    age() {  
        const date = new Date();  
        return date.getFullYear() - this.year;  
    }  
}  
  
const myCar = new Car("Ford", 2014);  
document.getElementById("demo").innerHTML =  
"My car is " + myCar.age() + " years old.";
```

You can send parameters to Class methods:

Example

```
class Car {  
    constructor(name, year) {  
        this.name = name;  
        this.year = year;  
    }  
    age(x) {  
        return x - this.year;  
    }  
}  
  
const date = new Date();  
let year = date.getFullYear();  
  
const myCar = new Car("Ford", 2014);  
document.getElementById("demo").innerHTML=  
"My car is " + myCar.age(year) + " years old.";
```

Browser Support

The following table defines the first browser version with full support for Classes in JavaScript:

				
Chrome 49	Edge 12	Firefox 45	Safari 9	Opera 36
Mar, 2016	Jul, 2015	Mar, 2016	Oct, 2015	Mar, 2016

66. JavaScript Modules

Modules

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain a code-base.

Modules are imported from external files with the `import` statement.

Modules also rely on `type="module"` in the `<script>` tag.

Example

```
<script type="module">
import message from "./message.js";
</script>
```

Export

Modules with **functions** or **variables** can be stored in any external file.

There are two types of exports: **Named Exports** and **Default Exports**.

Named Exports

Let us create a file named `person.js`, and fill it with the things we want to export.

You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line individually:

```
person.js

export const name = "Jesse";
export const age = 40;
```

All at once at the bottom:

```
person.js

const name = "Jesse";
const age = 40;

export {name, age};
```

Default Exports

Let us create another file, named `message.js`, and use it for demonstrating default export.

You can only have one default export in a file.

Example

`message.js`

```
const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
};

export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports are constructed using curly braces. Default exports are not.

Import from named exports

Import named exports from the file `person.js`:

```
import { name, age } from "./person.js";
```

Import from default exports

Import a default export from the file `message.js`:

```
import message from "./message.js";
```

Note

Modules only work with the HTTP(s) protocol.

A web-page opened via the file:// protocol cannot use import / export.

67. JavaScript JSON

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

What is JSON?

- JSON stands for **JavaScript Object Notation**
- JSON is a lightweight data interchange format
- JSON is language independent *
- JSON is "self-describing" and easy to understand

* The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

JSON Example

This JSON syntax defines an employees object: an array of 3 employee records (objects):

JSON Example

```
{  
  "employees": [  
    {"firstName": "John", "lastName": "Doe"},  
    {"firstName": "Anna", "lastName": "Smith"},  
    {"firstName": "Peter", "lastName": "Jones"}  
]
```

The JSON Format Evaluates to JavaScript Objects

The JSON format is syntactically identical to the code for creating JavaScript objects.

Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties. A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName": "John"
```

JSON names require double quotes. JavaScript names do not.

JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName": "John", "lastName": "Doe"}
```

JSON Arrays

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

```
"employees": [
    {"firstName": "John", "lastName": "Doe"},
    {"firstName": "Anna", "lastName": "Smith"},
    {"firstName": "Peter", "lastName": "Jones"}
]
```

In the example above, the object "employees" is an array. It contains three objects.

Each object is a record of a person (with a first name and a last name).

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.

For simplicity, this can be demonstrated using a string as input.

First, create a JavaScript string containing JSON syntax:

```
let text = '{ "employees" : [ ' +
'{ "firstName": "John" , "lastName": "Doe" },' +
'{ "firstName": "Anna" , "lastName": "Smith" },' +
'{ "firstName": "Peter" , "lastName": "Jones" } ] }';
```

Then, use the JavaScript built-in function `JSON.parse()` to convert the string into a JavaScript object:

```
const obj = JSON.parse(text);
```

Finally, use the new JavaScript object in your page:

Example

```
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>
```

68. JavaScript Debugging

Errors can (will) happen, every time you write some new computer code.

Code Debugging

Programming code might contain syntax errors, or logical errors.

Many of these errors are difficult to diagnose.

Often, when programming code contains errors, nothing will happen. There are no error messages, and you will get no indications where to search for errors.

Searching for (and fixing) errors in programming code is called code debugging.

JavaScript Debuggers

Debugging is not easy. But fortunately, all modern browsers have a built-in JavaScript debugger.

Built-in debuggers can be turned on and off, forcing errors to be reported to the user.

With a debugger, you can also set breakpoints (places where code execution can be stopped), and examine variables while the code is executing.

Normally (otherwise follow the steps at the bottom of this page), you activate debugging in your browser with the F12 key, and select "Console" in the debugger menu.

The `console.log()` Method

If your browser supports debugging, you can use `console.log()` to display JavaScript values in the debugger window:

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<script>
a = 5;
b = 6;
c = a + b;
console.log(c);
</script>
</body>
</html>
```

Setting Breakpoints

In the debugger window, you can set breakpoints in the JavaScript code.

At each breakpoint, JavaScript will stop executing, and let you examine JavaScript values.

After examining values, you can resume the execution of code (typically with a play button).

The `debugger` Keyword

The `debugger` keyword stops the execution of JavaScript, and calls (if available) the debugging function.

This has the same function as setting a breakpoint in the debugger.

If no debugging is available, the `debugger` statement has no effect.

With the debugger turned on, this code will stop executing before it executes the third line.

Example

```
let x = 15 * 5;  
debugger;  
document.getElementById("demo").innerHTML = x;
```

Major Browsers' Debugging Tools

Normally, you activate debugging in your browser with F12, and select "Console" in the debugger menu.

Otherwise follow these steps:

Chrome

- Open the browser.
- From the menu, select "More tools".
- From tools, choose "Developer tools".
- Finally, select Console.

Firefox

- Open the browser.
- From the menu, select "Web Developer".
- Finally, select "Web Console".

Edge

- Open the browser.
- From the menu, select "Developer Tools".
- Finally, select "Console".

Opera

- Open the browser.
- From the menu, select "Developer".
- From "Developer", select "Developer tools".
- Finally, select "Console".

Safari

- Go to Safari, Preferences, Advanced in the main menu.
- Check "Enable Show Develop menu in menu bar".
- When the new option "Develop" appears in the menu:
Choose "Show Error Console".

Did You Know?

Debugging is the process of testing, finding, and reducing bugs (errors) in computer programs.

The first known computer bug was a real bug (an insect) stuck in the electronics.

69. JavaScript Style Guide

Always use the same coding conventions for all your JavaScript projects.

JavaScript Coding Conventions

Coding conventions are **style guidelines for programming**. They typically cover:

- Naming and declaration rules for variables and functions.
- Rules for the use of white space, indentation, and comments.
- Programming practices and principles.

Coding conventions **secure quality**:

- Improve code readability
- Make code maintenance easier

Coding conventions can be documented rules for teams to follow, or just be your individual coding practice.

This page describes the general JavaScript code conventions used by W3Schools.
You should also read the next chapter "Best Practices", and learn how to avoid coding pitfalls.

Variable Names

At W3schools we use **camelCase** for identifier names (variables and functions).

All names start with a **letter**.

At the bottom of this page, you will find a wider discussion about naming rules.

```
firstName = "John";
lastName = "Doe";
price = 19.90;
tax = 0.20;
fullPrice = price + (price * tax);
```

Spaces Around Operators

Always put spaces around operators (= + - * /), and after commas:

Examples:

```
let x = y + z;
const myArray = ["Volvo", "Saab", "Fiat"];
```

Code Indentation

Always use 2 spaces for indentation of code blocks:

Functions:

```
function toCelsius(fahrenheit) {  
    return (5 / 9) * (fahrenheit - 32);  
}
```

Do not use tabs (tabulators) for indentation. Different editors interpret tabs differently.

Statement Rules

General rules for simple statements:

- Always end a simple statement with a semicolon.

Examples:

```
const cars = ["Volvo", "Saab", "Fiat"];  
const person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};
```

General rules for complex (compound) statements:

- Put the opening bracket at the end of the first line.
- Use one space before the opening bracket.
- Put the closing bracket on a new line, without leading spaces.
- Do not end a complex statement with a semicolon.

Functions:

```
function toCelsius(fahrenheit) {  
    return (5 / 9) * (fahrenheit - 32);  
}
```

Loops:

```
for (let i = 0; i < 5; i++) {  
    x += i;  
}
```

Conditionals:

```
if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

Object Rules

General rules for object definitions:

- Place the opening bracket on the same line as the object name.
- Use colon plus one space between each property and its value.
- Use quotes around string values, not around numeric values.
- Do not add a comma after the last property-value pair.
- Place the closing bracket on a new line, without leading spaces.
- Always end an object definition with a semicolon.

Example

```
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};
```

Short objects can be written compressed, on one line, using spaces only between properties, like this:

```
const person = {firstName: "John", lastName: "Doe", age: 50, eyeColor: "blue"};
```

Line Length < 80

For readability, avoid lines longer than 80 characters.

If a JavaScript statement does not fit on one line, the best place to break it, is after an operator or a comma.

Example

```
document.getElementById("demo").innerHTML =  
"Hello Dolly.;"
```

Naming Conventions

Always use the same naming convention for all your code. For example:

- Variable and function names written as **camelCase**
- Global variables written in **UPPERCASE** (We don't, but it's quite common)
- Constants (like PI) written in **UPPERCASE**

Should you use **hyp-hens**, **camelCase**, or **under_scores** in variable names?

This is a question programmers often discuss. The answer depends on who you ask:

Hyphens in HTML and CSS:

HTML5 attributes can start with data- (data-quantity, data-price).

CSS uses hyphens in property-names (font-size).

Hyphens can be mistaken as subtraction attempts. Hyphens are not allowed in JavaScript names.

Underscores:

Many programmers prefer to use underscores (date_of_birth), especially in SQL databases.

Underscores are often used in PHP documentation.

PascalCase:

PascalCase is often preferred by C programmers.

camelCase:

camelCase is used by JavaScript itself, by jQuery, and other JavaScript libraries.

Do not start names with a \$ sign. It will put you in conflict with many JavaScript library names.

Loading JavaScript in HTML

Use simple syntax for loading external scripts (the type attribute is not necessary):

```
<script src="myscript.js"></script>
```

Accessing HTML Elements

A consequence of using "untidy" HTML styles, might result in JavaScript errors.

These two JavaScript statements will produce different results:

```
const obj = getElementById("Demo")
```

```
const obj = getElementById("demo")
```

If possible, use the same naming convention (as JavaScript) in HTML.

File Extensions

HTML files should have a **.html** extension (**.htm** is allowed).

CSS files should have a **.css** extension.

JavaScript files should have a **.js** extension.

Use Lower Case File Names

Most web servers (Apache, Unix) are case sensitive about file names:

london.jpg cannot be accessed as London.jpg.

Other web servers (Microsoft, IIS) are not case sensitive:

london.jpg can be accessed as London.jpg or london.jpg.

If you use a mix of upper and lower case, you have to be extremely consistent.

If you move from a case insensitive, to a case sensitive server, even small errors can break your web site.

To avoid these problems, always use lower case file names (if possible).

Performance

Coding conventions are not used by computers. Most rules have little impact on the execution of programs.

Indentation and extra spaces are not significant in small scripts.

For code in development, readability should be preferred. Larger production scripts should be minimized.

70. JavaScript Best Practices

Avoid global variables, avoid `new`, avoid `==`, avoid `eval()`

Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead, and learn how to use [closures](#).

Always Declare Local Variables

All variables used in a function should be declared as **local** variables.

Local variables **must** be declared with the `var`, the `let`, or the `const` keyword, otherwise they will become global variables.

Strict mode does not allow undeclared variables.

Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function.

This will:

- Give cleaner code
- Provide a single place to look for local variables
- Make it easier to avoid unwanted (implied) global variables
- Reduce the possibility of unwanted re-declarations

```
// Declare at the beginning
let firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;
fullPrice = price - discount;
```

This also goes for loop variables:

```
for (let i = 0; i < 5; i++) {
```

Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

- Give cleaner code
- Provide a single place to initialize variables
- Avoid undefined values

```
// Declare and initiate at the beginning
let firstName = "";
let lastName = "";
let price = 0;
let discount = 0;
let fullPrice = 0,
const myArray = [];
const myObject = {};
```

Initializing variables provides an idea of the intended use (and intended data type).

Declare Objects with const

Declaring objects with const will prevent any accidental change of type:

Example

```
let car = {type:"Fiat", model:"500", color:"white"};
car = "Fiat"; // Changes object to string
```

```
const car = {type:"Fiat", model:"500", color:"white"};
car = "Fiat"; // Not possible
```

Declare Arrays with const

Declaring arrays with const will prevent any accidental change of type:

Example

```
let cars = ["Saab", "Volvo", "BMW"];
cars = 3; // Changes array to number
```

```
const cars = ["Saab", "Volvo", "BMW"];
cars = 3; // Not possible
```

Don't Use new Object()

- Use `""` instead of `new String()`
- Use `0` instead of `new Number()`
- Use `false` instead of `new Boolean()`
- Use `{}` instead of `new Object()`
- Use `[]` instead of `new Array()`
- Use `/()/.exec("...")` instead of `new RegExp()`
- Use `function (){}()` instead of `new Function()`

Example

```
let x1 = "";           // new primitive string
let x2 = 0;           // new primitive number
let x3 = false;        // new primitive boolean
const x4 = {};         // new object
const x5 = [];          // new array object
const x6 = /()/.exec("..."); // new regexp object
const x7 = function(){}(); // new function object
```

Beware of Automatic Type Conversions

JavaScript is loosely typed.

A variable can contain all data types.

A variable can change its data type:

Example

```
let x = "Hello";      // typeof x is a string
x = 5;                // changes typeof x to a number
```

Beware that numbers can accidentally be converted to strings or `NaN` (Not a Number).

When doing mathematical operations, JavaScript can convert numbers to strings:

Example

```
let x = 5 + 7;        // x.valueOf() is 12, typeof x is a number
let x = 5 + "7";       // x.valueOf() is 57, typeof x is a string
let x = "5" + 7;       // x.valueOf() is 57, typeof x is a string
let x = 5 - 7;        // x.valueOf() is -2, typeof x is a number
let x = 5 - "7";       // x.valueOf() is -2, typeof x is a number
let x = "5" - 7;       // x.valueOf() is -2, typeof x is a number
let x = "5" - "x";     // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns `NaN` (Not a Number):

Example

```
"Hello" - "Dolly" // returns NaN
```

Use === Comparison

The `==` comparison operator always converts (to matching types) before comparison.

The `===` operator forces comparison of values and type:

Example

```
0 == "";      // true
1 == "1";      // true
1 == true;     // true

0 === "";      // false
1 === "1";      // false
1 === true;     // false
```

Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to `undefined`.

Undefined values can break your code. It is a good habit to assign default values to arguments.

Example

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 0;
  }
}
```

[ECMAScript 2015](#) allows default parameters in the function definition:

```
function (a=1, b=1) { /*function code*/ }
```

Read more about function parameters and arguments at [Function Parameters](#)

End Your Switches with Defaults

Always end your `switch` statements with a `default`. Even if you think there is no need for it.

Example

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
```

```
day = "Monday";
break;
case 2:
  day = "Tuesday";
  break;
case 3:
  day = "Wednesday";
  break;
case 4:
  day = "Thursday";
  break;
case 5:
  day = "Friday";
  break;
case 6:
  day = "Saturday";
  break;
default:
  day = "Unknown";
}
```

Avoid Number, String, and Boolean as Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Example

```
let x = "John";
let y = new String("John");
(x === y) // is false because x is a string and y is an object.
```

Or even worse:

Example

```
let x = new String("John");
let y = new String("John");
(x == y) // is false because you cannot compare objects.
```

Avoid Using eval()

The `eval()` function is used to run text as code. In almost all cases, it should not be necessary to use it.

Because it allows arbitrary code to be run, it also represents a security problem.

71. JavaScript Common Mistakes

This chapter points out some common JavaScript mistakes.

Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (`=`), instead of a comparison operator (`==`) in an if statement.

This `if` statement returns `false` (as expected) because `x` is not equal to 10:

```
let x = 0;
if (x == 10)
```

This `if` statement returns `true` (maybe not as expected), because 10 is true:

```
let x = 0;
if (x = 10)
```

This `if` statement returns `false` (maybe not as expected), because 0 is false:

```
let x = 0;
if (x = 0)
```

An assignment always returns the value of the assignment.

Expecting Loose Comparison

In regular comparison, data type does not matter. This `if` statement returns true:

```
let x = 10;
let y = "10";
if (x == y)
```

In strict comparison, data type does matter. This `if` statement returns false:

```
let x = 10;
let y = "10";
if (x === y)
```

It is a common mistake to forget that `switch` statements use strict comparison:

This `case switch` will display an alert:

```
let x = 10;
switch(x) {
  case 10: alert("Hello");
}
```

This `case switch` will not display an alert:

```
let x = 10;
switch(x) {
  case "10": alert("Hello");
}
```

Confusing Addition & Concatenation

Addition is about adding **numbers**.

Concatenation is about adding **strings**.

In JavaScript both operations use the same `+` operator.

Because of this, adding a number as a number will produce a different result from adding a number as a string:

```
let x = 10;
x = 10 + 5;           // Now x is 15
let y = 10;
y += "5";            // Now y is "105"
```

When adding two variables, it can be difficult to anticipate the result:

```
let x = 10;
let y = 5;
let z = x + y;       // Now z is 15
let x = 10;
let y = "5";
let z = x + y;       // Now z is "105"
```

Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits **Floating point numbers** (Floats).

All programming languages, including JavaScript, have difficulties with precise floating point values:

```
let x = 0.1;
let y = 0.2;
let z = x + y           // the result in z will not be 0.3
```

To solve the problem above, it helps to multiply and divide:

Example

```
let z = (x * 10 + y * 10) / 10;      // z will be 0.3
```

Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

Example 1

```
let x =
"Hello World!";
```

But, breaking a statement in the middle of a string will not work:

Example 2

```
let x = "Hello
World!";
```

You must use a "backslash" if you must break a statement in a string:

Example 3

```
let x = "Hello \
World!";
```

Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

```
if (x == 19);  
{  
    // code block  
}
```

Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line. Because of this, these two examples will return the same result:

Example 1

```
function myFunction(a) {  
    let power = 10  
    return a * power  
}
```

Example 2

```
function myFunction(a) {  
    let power = 10;  
    return a * power;  
}
```

JavaScript will also allow you to break a statement into two lines.

Because of this, example 3 will also return the same result:

Example 3

```
function myFunction(a) {  
    let  
    power = 10;  
    return a * power;  
}
```

But, what will happen if you break the return statement in two lines like this:

Example 4

```
function myFunction(a) {  
    let  
    power = 10;  
    return  
        a * power;  
}
```

The function will return `undefined`!

Why? Because JavaScript thought you meant:

Example 5

```
function myFunction(a) {  
    let  
    power = 10;  
    return;  
    a * power;  
}
```

Explanation

If a statement is incomplete like:

```
let
```

JavaScript will try to complete the statement by reading the next line:

```
power = 10;
```

But since this statement is complete:

```
return
```

JavaScript will automatically close it like this:

```
return;
```

This happens because closing (ending) statements with semicolon is optional in JavaScript. JavaScript will close the return statement at the end of the line, because it is a complete statement.

Never break a return statement.

Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** use **numbered indexes**:

Example

```
const person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
person.length;           // person.length will return 3
person[0];              // person[0] will return "John"
```

In JavaScript, **objects** use **named indexes**.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

Example:

```
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
person.length;           // person.length will return 0
person[0];              // person[0] will return undefined
```

Ending Definitions with a Comma

Trailing commas in object and array definition are legal in ECMAScript 5.

Object Example:

```
person = {firstName:"John", lastName:"Doe", age:46,}
```

Array Example:

```
points = [40, 100, 1, 5, 25, 10,];
```

WARNING !!

Internet Explorer 8 will crash.

JSON does not allow trailing commas.

JSON:

```
person = {"firstName":"John", "lastName":"Doe", "age":46}
```

JSON:

```
points = [40, 100, 1, 5, 25, 10];
```

Undefined is Not Null

JavaScript objects, variables, properties, and methods can be `undefined`.

In addition, empty JavaScript objects can have the value `null`.

This can make it a little bit difficult to test if an object is empty.

You can test if an object exists by testing if the type is `undefined`:

Example:

```
if (typeof myObj === "undefined")
```

But you cannot test if an object is `null`, because this will throw an error if the object is `undefined`:

Incorrect:

```
if (myObj === null)
```

To solve this problem, you must test if an object is not `null`, and not `undefined`.

But this can still throw an error:

Incorrect:

```
if (myObj !== null && typeof myObj !== "undefined")
```

Because of this, you must test for not `undefined` before you can test for not `null`:

Correct:

```
if (typeof myObj !== "undefined" && myObj !== null)
```

72. JavaScript Performance

How to speed up your JavaScript code.

Reduce Activity in Loops

Loops are often used in programming.

Each statement in a loop, including the for statement, is executed for each iteration of the loop.

Statements or assignments that can be placed outside the loop will make the loop run faster.

Bad:

```
for (let i = 0; i < arr.length; i++) {
```

Better Code:

```
let l = arr.length;
for (let i = 0; i < l; i++) {
```

The bad code accesses the length property of an array each time the loop is iterated.

The better code accesses the length property outside the loop and makes the loop run faster.

Reduce DOM Access

Accessing the HTML DOM is very slow, compared to other JavaScript statements.

If you expect to access a DOM element several times, access it once, and use it as a local variable:

Example

```
const obj = document.getElementById("demo");
obj.innerHTML = "Hello";
```

Reduce DOM Size

Keep the number of elements in the HTML DOM small.

This will always improve page loading, and speed up rendering (page display), especially on smaller devices.

Every attempt to search the DOM (like `getElementsByName`) will benefit from a smaller DOM.

Avoid Unnecessary Variables

Don't create new variables if you don't plan to save values.

Often you can replace code like this:

```
let fullName = firstName + " " + lastName;  
document.getElementById("demo").innerHTML = fullName;
```

With this:

```
document.getElementById("demo").innerHTML = firstName + " " + lastName;
```

Delay JavaScript Loading

Putting your scripts at the bottom of the page body lets the browser load the page first.

While a script is downloading, the browser will not start any other downloads. In addition all parsing and rendering activity might be blocked.

The HTTP specification defines that browsers should not download more than two components in parallel.

An alternative is to use `defer="true"` in the script tag. The defer attribute specifies that the script should be executed after the page has finished parsing, but it only works for external scripts.

If possible, you can add your script to the page by code, after the page has loaded:

Example

```
<script>  
window.onload = function() {  
    const element = document.createElement("script");  
    element.src = "myScript.js";  
    document.body.appendChild(element);  
};  
</script>
```

Avoid Using with

Avoid using the `with` keyword. It has a negative effect on speed. It also clutters up JavaScript scopes.

The `with` keyword is **not allowed** in strict mode.

73. JavaScript Reserved Words

In JavaScript you cannot use these reserved words as variables, labels, or function names:

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const*	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected

public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Words marked with* was new in [ECMAScript 5](#) and [ECMAScript 6](#).

Removed Reserved Words

The following reserved words have been removed from the ECMAScript 5/6 standard:

abstract	boolean	byte	char
double	final	float	goto
int	long	native	short
synchronized	throws	transient	volatile

Do not use these words as variables. ECMAScript 5/6 does not have full support in all browsers.

JavaScript Objects, Properties, and Methods

You should also avoid using the name of JavaScript built-in objects, properties, and methods:

Array	Date	eval	function
hasOwnProperty	Infinity	isFinite	isNaN
isPrototypeOf	length	Math	NaN
name	Number	Object	prototype
String	toString	undefined	valueOf

Java Reserved Words

JavaScript is often used together with Java. You should avoid using some Java objects and properties as JavaScript identifiers:

getClass	java	JavaArray	javaClass
JavaObject	JavaPackage		

Other Reserved Words

JavaScript can be used as the programming language in many applications.

You should also avoid using the name of HTML and Window objects and properties:

alert	all	anchor	anchors
area	assign	blur	button
checkbox	clearInterval	clearTimeout	clientInformation
close	closed	confirm	constructor
crypto	decodeURI	decodeURIComponent	defaultStatus
document	element	elements	embed
embeds	encodeURI	encodeURIComponent	escape
event	fileUpload	focus	form
forms	frame	innerHeight	innerWidth
layer	layers	link	location
mimeTypes	navigate	navigator	frames
frameRate	hidden	history	image
images	offscreenBuffering	open	opener
option	outerHeight	outerWidth	packages
pageXOffset	pageYOffset	parent	parseFloat
parseInt	password	pkcs11	plugin
prompt	propertyIsEnum	radio	reset
screenX	screenY	scroll	secure
select	self	setInterval	setTimeout
status	submit	taint	text
textarea	top	unescape	untaint
window			

HTML Event Handlers

In addition you should avoid using the name of all HTML event handlers.

Examples:

onblur	onclick	onerror	onfocus
onkeydown	onkeypress	onkeyup	onmouseover
onload	onmouseup	onmousedown	onsubmit