JavaScript Notes

JavaScript is the world's most popular programming language. JavaScript is the programming language of the Web. JavaScript is easy to learn. This tutorial will teach you JavaScript from basic to advanced.

JavaScript and Java are completely different languages, both in concept and design. JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is getElementById().

The example below "finds" an HTML element (with id="demo"), and changes the element content (innerHTML) to "Hello JavaScript":

Example

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

JavaScript accepts both double and single quotes

JavaScript Can Change HTML Styles (CSS)

Changing the style of an HTML element, is a variant of changing an HTML attribute:

Example

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

JavaScript Can Hide HTML Elements

Hiding HTML elements can be done by changing the display style:

Example

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

JavaScript Can Show HTML Elements

Showing hidden HTML elements can also be done by changing the display style:

Example

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

The <script> Tag

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

Example

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

JavaScript Functions and Events

A JavaScript function is a block of JavaScript code, that can be executed when "called" for.

For example, a function can be called when an **event** occurs, like when the user clicks a button.

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>
```

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>
```

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 inner HTML

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>
```

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>
```

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>
```

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.

Example

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
alert(5 + 6);
</script>

</body>
</html>
```

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>
```

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.";
```

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
```

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";
```

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?";
}
```

JavaScript Keywords

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 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
```

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;
```

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.

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";
```

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

JavaScript Character Set

JavaScript uses the **Unicode** character set. Unicode covers (almost) all the characters, punctuations, and symbols in the world.

JavaScript Comments

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.";
```

JavaScript Variables

4 Ways to Declare a JavaScript Variable:

- Using var
- Using let
- Using const
- Using nothing

- What are Variables?
- Variables are containers for storing data (storing data values).
- In this example, x, y, and z, are variables, declared with the var keyword:

Example

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

In this example, x, y, and z, are variables, declared with the let keyword:

Example

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

In this example, x, y, and z, are undeclared variables:

Example

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

When to Use JavaScript var?

Always declare JavaScript variables with var, let, or const. The var keyword is used in all JavaScript code from 1995 to 2015. The let and const keywords were added to JavaScript in 2015. If you want your code to run in older browsers, you must use var.

When to Use JavaScript const?

If you want a general rule: always declare variables with const. If you think the value of the variable can change, use let. In this example, price1, price2, and total, are variables:

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. This is a value that can be changed.

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.

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;
```

JavaScript Let

The let keyword was introduced in 2015.

Variables defined with let cannot be Re-declared. Variables defined with let must be Declared before use. Variables defined with let have Block Scope. Variables defined with let cannot be **redeclared**.

With let you can not do this:

Example

```
let x = "John Doe";

let x = 0;

// SyntaxError: 'x' has already been declared
```

With var you can:

Example

```
var x = "John Doe";

var x = 0;
```

JavaScript Const

Variables defined with const cannot be Re-declared. Variables defined with const cannot be Re-assigned. A const variable 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
```

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 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");
```

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";
```

JavaScript Operators

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;
```

Adding

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

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
- Logical Operators
- Conditional Operators
- Type Operators

JavaScript Arithmetic Operators

Arithmetic Operators are used to perform arithmetic on numbers:

Arthmetic Operators Example

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

Example:

```
<!DOCTYPE html>

<html>

<body>


<h1>JavaScript Arithmetic</h1>

<h2>Arithmetic Operations</h2>

<p>A typical arithmetic operation takes two numbers (or expressions) and produces a new number.</p>


<p id="demo"></p>


<script>

let a = 3;

let x = (100 + 50) * a;

document.getElementById("demo").innerHTML = x;

</script></body></html>
```

Output: 450

Operator	Example	Same As
=	x = y	x = y
+=	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
**=	x **= y	x = x ** y

Adding JavaScript Strings

The + operator can also be used to add (concatenate) strings.

Example

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

The result of text3 will be:

John Doe

JavaScript Comparison Operators

Operator	Description
==	equal to
===	equal value and equal type Strict equality using === **If the values have the same type, are not numbers, and have the same value, they're considered equal**. Finally, if both values are numbers, they're considered equal if they're both not NaN and are the same value, or if one is +0 and one is -0 .

!=	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

JavaScript Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

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

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

JavaScript Arithmetic Operators

Example

```
let x = 100 + 50;
```

Example

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

Adding

The **addition** operator (+) adds numbers:

Example

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

Subtracting

The **subtraction** operator (-) subtracts numbers.

Example

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

Multiplying

The **multiplication** operator (*) multiplies numbers.

Example

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

Dividing

The **division** operator (/) divides numbers.

Example

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

Remainder

The **modulus** operator (%) returns the division remainder.

Example

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

Note: In arithmetic, the division of two integers produces a **quotient** and a **remainder**. In mathematics, the result of a **modulo operation** is the **remainder** of an arithmetic division.

Incrementing

The **increment** operator (++) increments numbers.

Example

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

Decrementing

The **decrement** operator (--) decrements numbers.

Example

```
let x = 5;
x--;
let z = x;
```

Exponentiation

The **exponentiation** operator (**) raises the first operand to the power of the second operand.

Example

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

x ** y produces the same result as Math.pow(x,y):

Example

```
let x = 5;
let z = Math.pow(x,2);
```

The += Operator

The **Addition Assignment Operator** adds a value to a variable.

Addition Assignment Examples

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

The -= Operator

The **Subtraction Assignment Operator** subtracts a value from a variable.

Subtraction Assignment Example

```
let x = 10;
x -= 5;
```

The *= Operator

The **Multiplication Assignment Operator** multiplies a variable.

Multiplication Assignment Example

```
let x = 10;
x *= 5;
```

The **= Operator

The **Exponentiation Assignment Operator** raises a variable to the power of the operand.

Exponentiation Assignment Example

```
let x = 10;
x **= 5;
```

The /= Operator

The **Division Assignment Operator** divides a variable.

Division Assignment Example

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

The %= Operator

The **Remainder Assignment Operator** assigns a remainder to a variable.

Remainder Assignment Example

```
let x = 10;
x %= 5;
```

The <<= Operator

The **Left Shift Assignment Operator** left shifts a variable.

Left Shift Assignment Example

```
let x = -100;
x <<= 5;
```

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Assignments</h1>
<h2>Left Shift Assignment</h2>
<h3>The <<= Operator</h3>
<p id="demo"></p>
<script>
let x = -100;
x <<= 5;
document.getElementById("demo").innerHTML = "Value of x is: " + x;
</script></body></html>
```

Output: Value of x is: -3200

The >>= Operator

The **Right Shift Assignment Operator** right shifts a variable (signed).

Right Shift Assignment Example

```
let x = -100;
x >>= 5;
```

```
<!DOCTYPE html>

<html><body>

<h1>JavaScript Assignments</h1>

<h2>Right Shift Assignment</h2>

<h3>The >>= Operator</h3>

<p id="demo"></p>

<script>

let x = -100;

x >>= 5;

document.getElementById("demo").innerHTML = "Value of x is: " + x;

</script></body></html>
```

Output: Value of x is: -4

The >>>= Operator

The **Unsigned Right Shift Assignment Operator** right shifts a variable (unsigned). The unsigned right shift (>>>) (zero-fill right shift) operator **evaluates the left-hand operand as an unsigned number, and shifts the binary representation of that number by the number of bits, modulo 32, specified by the right-hand operand**.

Unsigned Right Shift Assignment Example

```
let x = -100;
x >>>= 5;
```

Example:

```
<!DOCTYPE html>

<html><body>

<h1>JavaScript Assignments</h1>
```

<h2>Right Shift Assignment</h2>

<h3>The >>>= Operator</h3>

<script>

let x = -100;

x >>>= 5;

document.getElementById("demo").innerHTML = "Value of x is: " + x;

</script></body></html>

Output: Value of x is: 134217724

The &= Operator

The **Bitwise AND Assignment Operator** does a bitwise AND operation on two operands and assigns the result to the the variable.

Bitwise AND Assignment Example

```
let x = 10;
x &= 5;
```

Example:

<script>

let x = 100;

x &= 5;

document.getElementById("demo").innerHTML = "Value of x is: " + x;

</script>

The output: Value of x is: 4

The |= Operator

The **Bitwise OR Assignment Operator** does a bitwise OR operation on two operands and assigns the result to the variable.

Bitwise OR Assignment Example

```
let x = 10;
x |= 5;
```

The output is: Value of x is: 101

The ^= Operator

The **Bitwise XOR Assignment Operator** does a bitwise XOR operation on two operands and assigns the result to the variable.

Bitwise XOR Assignment Example

```
let x = 10;
x ^= 5;
```

The output is: Value of x is: 97

The &&= Operator

The **Logical AND assignment operator** is used between two values.

If the first value is true, the second value is assigned.

Logical AND Assignment Example

```
let x = 10;
x &&= 5;
```

The output: If the first value is true, the second value is assigned. Value of x is: 5

The ||= Operator

The **Logical OR assignment operator** is used between two values.

If the first value is false, the second value is assigned.

Logical OR Assignment Example

```
let x = 10;
x ||= 5;
```

The output: If the first value is false, the second value is assigned. The value of x is: 5

The ??= Operator

The **Nullish coalescing assignment operator** is used between two values.

If the first value is undefined or null, the second value is assigned.

Nullish Coalescing Assignment Example

```
let x = 10;
x ??= 5;
```

The output: Value of x is: 100

JavaScript has 8 Datatypes

1. String
2. Number
3. Bigint
4. Boolean
5. Undefined
6. Null
7. Symbol
8. Object

The Object Datatype

The object data type can contain:

1. An object
2. An array
3. A date

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");
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:

Example:

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

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
```

Javascript are always one type. Double (64-bit floating point).

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"];
```

Add new elements at the beginning of an array using JavaScript

```
<body style="text-align:center;">
        <h1 style="color:green;">
                Hello World
        </h1>
        <h3>Add element using unshift()</h3>
        <p id="up"></p>
        <button onclick="myGFG()">
                Click to insert
        </button>
        <p id="down" style="color: green"></p>
        <!-- Script to add element at beginning of array -->
        <script>
                var GFG_Array = [1, 2, 3, 4, 5];
                var up = document.getElementById("up");
                up.innerHTML = GFG_Array;
                var down = document.getElementById("down");
                down.innerHTML = "elements of GFG_Array = "
```

```
                + GFG_Array;


        function myGFG() {
                GFG_Array.unshift("6");
                down = document.getElementById("down");
                down.innerHTML = "element of GFG_Array = "
                        + GFG_Array;          }
</script> </body>
```

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 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"
```

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
```

JavaScript Functions

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

```
// 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 **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:

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

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

Why Functions?

You can reuse code: Define the code once, and use it many times. You can use the same code many times with different arguments, to produce different results.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius(77);
```

The () Operator Invokes the Function

Using the example above, toCelsius refers to the function object, and toCelsius() refers to the function result. Accessing a function without () will return the function object instead of the function result.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius;
```

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";
```

Local Variables

Variables declared within a JavaScript function, become **LOCAL** to the function. Local variables can only be accessed from within the function.

```
// code here can NOT use carName

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

// code here can NOT use carName
```

Note: 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.

JavaScript Objects

Real Life Objects, Properties, and Methods

In real life, a car is an **object**. A car has **properties** like weight and color, and **methods** like start and stop:

Properties	**Methods**
car.name = Fiat	car.start()
car.model = 500	car.drive()
car.weight = 850kg	car.brake()

```
car.color = white    car.stop()
```

All cars have the same **properties**, but the property **values** differ from car to car. All cars have the same **methods**, but the methods are performed **at different times**.

JavaScript Objects

You have already learned that JavaScript variables are containers for data values.

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

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

Objects are variables too. But objects can contain many values. This code assigns **many values** (Fiat, 500, white) to a **variable** named car. The values are written as **name:value** pairs (name and value separated by a colon).

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

Object Definition

You define (and create) a JavaScript object with an object literal:

Example:

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

or

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

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName

or

objectName["propertyName"]

Example:

person.lastName; or person["lastName"];

Object Methods

Objects can also have **methods**. Methods are **actions** that can be performed on objects. Methods are stored in properties as **function definitions**.

Note: A method is a function stored as a property.

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

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

What is **this**?

In JavaScript, the this keyword refers to an **object**. **Which** object depends on how this is being invoked (used or called).

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.

The **this** Keyword

In a function definition, this refers to the "owner" of the function. In the example above, this is the **person object** that "owns" the fullName function.

In other words, this.firstName means the firstName property of **this object**.

Accessing Object Methods

You access an object method with the following syntax:

Example:

objectName.methodName()

name = person.fullName();

If you access a method **without** the () parentheses, it will return the **function definition**:

Example:

name = person.fullName;

Do Not Declare Strings, Numbers, and Booleans as Objects!

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

```
x = new String();        // Declares x as a String object
y = new Number();        // Declares y as a Number object
z = new Boolean();       // Declares z as a Boolean object
```

Note: Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

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:

<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):

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

<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

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

JavaScript String Methods

String length	String trim()
String slice()	String trimStart()
String substring()	String trimEnd()
String substr()	String padStart()
String replace()	String padEnd()
String replaceAll()	String charAt()
String toUpperCase()	String charCodeAt()
String toLowerCase()	String split()
String concat()	

JavaScript String Length

The length property returns the length of a string:

Example:

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

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.

Example:

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);
```

Example: 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);
```

Example: 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 substring()

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);
```

Note: 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);
```

Note: 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);
```

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:

```
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:

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

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

```
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):

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

JavaScript String ReplaceAll()

Note: It does not work in Internet Explorer

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

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

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()

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

JavaScript String toLowerCase()

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

JavaScript String concat()

concat() joins two or more strings:

```
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:

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

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

JavaScript String trim()

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

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

JavaScript String trimStart()

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

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

JavaScript String trimEnd()

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

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

JavaScript String Padding

JavaScript String padStart()

The padStart() method pads a string with another string:

```
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.

Try this:

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

Example:

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

JavaScript String padEnd()

The padEnd() method pads a string with another string:

```
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.

Try this:

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

Example:

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

Extracting String Characters

There are 3 methods for extracting string characters:

- charAt(*position*)
- charCodeAt(*position*)
- Property access [ ]

JavaScript String charAt()

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

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

JavaScript String charCodeAt()

The charCodeAt() method returns the unicode of the character at a specified index in a string. The method returns a UTF-16 code (an integer between 0 and 65535).

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

Property Access

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!)

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:

```
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:

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

JavaScript Booleans

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:

(10 > 9)
10 > 9

Comparisons and Conditions

The chapter JS Comparisons gives a full overview of comparison operators.

The chapter JS Conditions gives a full overview of conditional statements.

Here are some examples:

Operator	Description	Example
==	equal to	if (day == "Monday")
>	greater than	if (salary > 9000)
<	less than	if (age < 18)

Everything With a "Value" is True

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);
```

JavaScript Comparison and Logical Operators

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";
```

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	(x < 10 && y > 1) is true
||	or	(x == 5 || y == 5) is false
!	not	!(x == y) 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";
```

Note: 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 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;
```

JavaScript if, else, and else if

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. Note that if is in lowercase letters. Uppercase letters IF will generate a JavaScript error.

Example:

```
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 (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

Example:

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

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:

```
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 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. 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**.

The default Keyword

The default keyword specifies the code to run if there is no case match:

Example:

```
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";
}
```

Note: 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";
}
```

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:

```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
```

```
  break;
 case 1:
  text = "On";
  break;
 default:
  text = "No value found";
}
```

JavaScript For Loop

Example:

```
for (let i = 0; i < cars.length; i++) {
 text += cars[i] + "<br>";
}
```

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>";
}
```

Example:

```
let i = 2;
let len = cars.length;
let text = "";
for (; i < len; i++) {
```

```
  text += cars[i] + "<br>";
}
```

Note: 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.

Example:

```
var i = 5;

for (var i = 0; i < 10; i++) {
  // some code
}

// Here i is 10
```

Example:

```
let i = 5;

for (let i = 0; i < 10; i++) {
  // some code
}

// Here i is 5
```

The For In Loop

Example:

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
```

For In used in arrays

Example:

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

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;
}
```

JavaScript For Of

Example:

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

let text = "";
for (let x of cars) {
  text += x;
}
```

Example:

```
let language = "JavaScript";

let text = "";
for (let x of language) {
text += x;
}
```

The While Loop: The while loop loops through a block of code as long as a specified condition is true.

Example:

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

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.

Example:

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

JavaScript Break and Continue

The Break Statement: The break statement can also be used to jump out of a loop:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
```

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:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
```

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";
```

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

```
const array_name = [item1, item2, ...];
```

Note: It is a common practice arrays with the const keyword.

Accessing Array Elements

You access an array element by referring to the **index number**:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

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";
```

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

```
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. Arrays use **numbers** to access its "elements". In this example, person[0] returns John:

Example:

```
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
```

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;
```

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**.

```
const person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
person.length;    // Will return 3
person[0];        // Will return "John"
```

Note: 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**.

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];
```

JavaScript Array Methods

Converting Arrays to Strings

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();
```

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(" * ");
```

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:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();
```

The pop() method returns the value that was "popped out":

```
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):

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");
```

The push() method returns the new array length:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.push("Kiwi");
```

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":

```
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:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
```

The unshift() method returns the new array length:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
```

Changing Elements

```
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:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Kiwi";
```

JavaScript Array delete()

Note: Array elements can be deleted using the JavaScript operator delete.
Using delete leaves undefined holes in the array. Use pop() or shift() instead.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];
```

Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing 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.

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:

```
const arr1 = ["Emil", "Tobias", "Linus"];
const myChildren = arr1.concat("Peter");
```

Splicing and Slicing Arrays

The splice() method adds new items to an array. The slice() method slices out a piece of 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:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 2, "Lemon", "Kiwi");
```

JavaScript Array slice()

The slice() method slices out a piece of an array into a new array. This example slices out a part of an array starting from array element 1 ("Orange"):

Example:

```
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.

```
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.

```
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:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
```

Note: All JavaScript objects have a toString() method.

Sorting an Array

The sort() method sorts an array alphabetically:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
```

Reversing an Array

The reverse() method reverses the elements in an array. You can use it to sort an array in descending order:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
fruits.reverse();
```

Numeric Sort

By default, the sort() function sorts values as **strings**. This works well for strings ("Apple" comes before "Banana"). However, 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**:

```
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:

```
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>
```

Find the Highest (or Lowest) Array Value

There are no built-in functions for finding the max or min value in an array. However, after you have sorted an array, you can use the index to obtain the highest and lowest values. Sorting ascending:

```
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
```

Sorting descending:

```
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
```

Using Math.max() on an Array

You can use Math.max.apply to find the highest number in an array:

```
function myArrayMax(arr) {
  return Math.max.apply(null, arr);
}
```

Math.max.apply(null, [1, 2, 3]) is equivalent to Math.max(1, 2, 3).

Using Math.min() on an Array

You can use Math.min.apply to find the lowest number in an array:

```
function myArrayMin(arr) {
  return Math.min.apply(null, arr);
}
```

My Min / Max JavaScript Methods

```
function myArrayMax(arr) {
  let len = arr.length;
  let max = -Infinity;
  while (len--) {
    if (arr[len] > max) {
      max = arr[len];
    }
  }
  return max;
}
```

This function loops through an array comparing each value with the lowest value found:

```
function myArrayMin(arr) {
  let len = arr.length;
  let min = Infinity;
  while (len--) {
    if (arr[len] < min) {
      min = arr[len];
    }
  }
  return min;
}
```

Sorting Object Arrays

```
const cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
  {type:"BMW", year:2010}
];

cars.sort(function(a, b){return a.year - b.year});
```

```
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;
});
```

JavaScript Array Iteration

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:

```
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 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:

```
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:

```
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. The reduce() method does not reduce the original 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:

```
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:

```
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. The reduceRight() method does not reduce the original 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:

```
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:

```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Example 1:

```
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;
}
```

JavaScript Array indexOf()

The indexOf() method searches an array for an element value and returns its position.

Example:

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.indexOf("Apple") + 1;
```

Note: 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.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.lastIndexOf("Apple") + 1;
```

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:

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

JavaScript Math Object

```
Math.PI;
```

Example:

```
<!DOCTYPE html>

<html><body>

<h2>JavaScript Math.PI</h2>

<p>Math.PI returns the ratio of a circle's circumference to its diameter:</p>
```

<script>

document.getElementById("demo").innerHTML = Math.PI;</script></body></html>

The output: Math.PI returns the ratio of a circle's circumference to its diameter.
3.141592653589793

The Math Object

Unlike other objects, the Math object has no constructor. The Math object is static.

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
```

Number to Integer

There are 4 common methods to round a number to an integer:

Math.round(x)	Returns x rounded to its nearest integer
Math.ceil(x)	Returns x rounded up to its nearest integer
Math.floor(x)	Returns x rounded down to its nearest integer
Math.trunc(x)	Returns the integer part of x

Math.round()

Math.round(x) returns the nearest integer:

Example:

```
<p id="demo"></p>


<script>

document.getElementById("demo").innerHTML = Math.round(4.6);

</script>
```

The output: 5

Math.ceil()

Math.ceil(x) returns the value of x rounded **up** to its nearest integer:

Example:

Math.ceil(4.9);

Math.floor()

Math.floor(x) returns the value of x rounded **down** to its nearest integer:

Example:

Math.floor(4.9);

Math.trunc()

Math.trunc(x) returns the integer part of x:

Example: Math.trunc(4.9);

Math.sign()

Math.sign(x) returns if x is negative, null or positive:

Example: Math.sign(-4);

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).

Example: Math.sin(90 * Math.PI / 180); // returns 1 (the sine of 90 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);

Math.random()

Math.random() returns a random number between 0 (inclusive), and 1 (exclusive):

Example: Math.random();

JavaScript Object toString()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let text = fruits.toString();
```

Example:

```
<p id="demo"></p><script>

const fruits = ["Banana", "Orange", "Apple", "Mango"];

const myArray = fruits.toString();

document.getElementById("demo").innerHTML = myArray;

</script>
```

The output: The result of using toString() on an array. Banana,Orange,Apple,Mango

1. Using toString() on an object:

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
const keys = person.toString();
```

Example:

```
<html>

<body>

<h1>JavaScript Objects</h1>

<h2>The toString() Method</h2>

<p>The toString() method returns "[object Type]" if it cannot return a proper string:</p>

<p id="demo"></p>
```

```
<script>

const person = {

  firstName: "John",

  lastName: "Doe",

  age: 50,

  eyeColor: "blue"

};

const keys = person.toString();

document.getElementById("demo").innerHTML = keys;

</script></body></html>
```

2. Using Object.toString() on an object:

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
const keys = Object.toString(person);
```

Example

```
<html><body>

<h1>JavaScript Objects</h1>

<h2>The Object.toString() Method</h2>

<p>Using Object.toString() returns the object constructor:</p>

<p id="demo"></p>

<script>
```

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
const keys = Object.toString(person);
document.getElementById("demo").innerHTML = keys;
</script></body></html>
```

The output: Using Object.toString() returns the object constructor.

```
function Object() { [native code] }
```

Definition and Usage

The toString() method returns an object as a string and it returns "[object Object]" if it cannot return a string. The toString() method does not change the original object. Whereas Object.toString() always returns the object constructor.

JavaScript Date Objects

```
const d = new Date();
```

or

```
const d = new Date("2022-03-25");
```

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*)

new Date(*date string*)

new Date(*date string*) creates a date object from a **date string**:

```
const d = new Date("October 13, 2014 11:13:00");
```