JAVASCRIPT

INTRODUCTION

This page contains some examples of what JavaScript can do.

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:

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

<h2> What Can JavaScript Do?</h2>

<p id="demo"> JavaScript can change HTML content </p>

<button type="button" onclick
document.getElementById("demo").innerHTML = "Hello JavaScript!"'>Click
Me!</button>

</body>
</html>
```

JavaScript accepts both double and single quotes:

Example:

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

<h2> What Can JavaScript Do?</h2>

<p id="demo"> JavaScript can change HTML content </p>
<button type="button" onclick
```

```
document.getElementById("demo").innerHTML = "Hello JavaScript!"'>Click Me!</button>

</body>
</html>
```

JavaScript Can Change HTML Attribute Values

In this example JavaScript changes the value of the src (source) attribute of an <img> tag:

The Light Bulb

Turn on the light Turn off the light

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

<h2> What Can JavaScript Do?</h2>

<p> JavaScript can change HTML attribute values.</p>

<p>In this case JavaScript changes the value of the src (source) attribute of an image.</p>

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

<img id="myImage" src="pic_bulboff.gif" style="width:100px">

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

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

JavaScript Can Change HTML Styles (CSS)

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

Example:

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

<h2> What Can JavaScript Do?</h2>

<p id="demo"> JavaScript can change the style of an HTML element.</p>

<button type="button" onclick
=”document.getElementById("demo").style.fontSize = ‘35px’ ">Click
Me!</button>

</body>
</html>
```

JavaScript Can Hide HTML Elements

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

Example:

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

<h2> What Can JavaScript Do?</h2>

<p id="demo">JavaScript can hide HTML elements.</p>

<button type="button" onclick =”
document.getElementById("demo").style.display = ‘none’ ">Click
Me!</button>
```

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

JavaScript Can Show HTML Elements

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

Example:

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

<h2> What Can JavaScript Do?</h2>

<p> JavaScript can show hidden HTML elements.</p>

<p id="demo" style="display:none">Hello JavaScript!</p>

<button type="button" onclick ="
document.getElementById("demo").style.display = 'block' ">Click
Me!</button>

</body>
</html>
```

Did You Know?

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.

ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

JAVASCRIPT WHERE TO

The <script> Tag

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

Example:

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

<h2> JavaScript in Body</h2>
<p id="demo"> </p>

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

</body>
</html>
```

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 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>
<h1>A Web Page</h1>
<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>

<h1>A Web Page</h1>
<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:

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

<h2>External JavaScript</h2>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

<p>This example links to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

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

</body>
</html>
```


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:

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

<h2>External JavaScript</h2>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Click Me</button>

<p>This example uses a full web URL to link to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

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

</body>
</html>
```

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

Example:

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

<h2>External JavaScript</h2>
<p id="demo">A Paragraph</p>
```

```
<button type="button" onclick="myFunction()">Try it</button>

<p>This example uses a file path to link to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

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

</body>
</html>
```

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

Example:

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

<h2>External JavaScript</h2>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

<p>This example links to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

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

</body>
</html>
```


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, that 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>

<p>F12 on your keybord will activate debugging. </p>
<p>Then select "Console" in the debugger menu. </p>
<p>Then click Run again. </p>

<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>
<h2>The window.print() Method</h2>

<p>Click the button to print the current page.</p>

<button onclick="window.print()">Print this page</button>

</body>
</html>
```

JAVASCRIPT STATEMENTS

Example:

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

<h2>JavaScript Statements</h2>

<p> A <b> JavaScript program</b>is a list of <b> statements</b>to be executed by a computer.</p>

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

<script>
```

```
let x, y, z;    // Statement 1
x = 5;          // Statement 2
y = 6;          // Statement 3
z = x + y;      // Statement 4
document.getElementById("demo").innerHTML =
"The value of z is " + z + ".";
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Statements</h2>

<p>In HTML, JavaScript statements are executed by the browser</p>

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

<script>
document.getElementById("demo").innerHTML = "Hello Dolly.";
</script>
```

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

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

Semicolons ;

Semicolons separate JavaScript statements.
Add a semicolon at the end of each executable statement:
Examples:

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

<h2> JavaScript Statements</h2>

<p> JavaScript statements are separated by semicolons.</p>

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

<script>
let a, b, c;
a = 5;
b = 6;
c = a + b;
document.getElementById("demo1").innerHTML = c;
</script>

</body>
</html>
```

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

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

<h2> JavaScript Statements</h2>
```

```
<p> Multiple statements on one line are allowed.</p>

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

<script>
 let a, b, c;
a = 5; b = 6; c = a + b;

document.getElementById("demo1").innerHTML = c;
</script>

</body>
</html>
```

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:

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

<h2> JavaScript Statements</h2>
```

```
<p> The best place to break a code line is after an operator or a comma..</p>

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

<script>
document.getElementById("demo").innerHTML =
"Hello Dolly!";

</script>

</body>
</html>
```

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:

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

<h2> JavaScript Statements</h2>

<p> JavaScript code blocks are written between { and }</p>

<button type="button" onclick="myFunction()">Click Me!</button>

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

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

```
}
</script>

</body>
</html>
```

In this tutorial we use 2 spaces of indentation for code blocks.
You will learn more about functions later in this tutorial.

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.

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:

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

<h2>JavaScript Numbers</h2>

<p>Number can be written with or without decimals.</p>

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

<script>document.getElementById("demo").innerHTML=10.50;
</script>

</body>
</html>
```


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

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

<h2> JavaScript Strings</h2>

<p> Strings can be written with double or single quotes.</p>

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

```
<script>
document.getElementById("demo").innerHTML ="John Doe";
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Variables</h2>

<p>In this example, x is defined as a variable.
Then, x is assigned the value of 6:</p>

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

<script>
let x;
x = 6;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

JavaScript Operators

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

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

```
<body>

<h2> JavaScript Operators</h2>

<p> JavaScript uses arithmetic operators to compute values (just like algebra).</p>

<p id="demo"></p>
<script>document.getElementById("demo").innerHTML =(5 + 6) * 10;
</script>

</body>
</html>
```

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

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

<h2> Assigning JavaScript Values</h2>

<p> In JavaScript the = operator is used to assign values to variables.</p>

<p id="demo"></p>
<script>
let x, y;
x = 5;
y = 6;
document.getElementById("demo").innerHTML = x + y;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Expressions</h2>

<p>Expressions compute to values..</p>

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

<script>
document.getElementById("demo").innerHTML =5 * 10;
</script>

</body>
</html>
```

Expressions can also contain variable values:

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

<h2>JavaScript Expressions</h2>

<p>Expressions compute to values..</p>

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

<script>
var x;
x = 5;
document.getElementById("demo").innerHTML = x * 10;
</script>

</body>
</html>
```

The values can be of various types, such as numbers and strings.
For example, "John" + " " + "Doe", evaluates to "John Doe":

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

<h2>JavaScript Expressions</h2>

<p>Expressions compute to values..</p>
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "John" + " " + "Doe";
</script>

</body>
</html>
```

JavaScript Keywords

JavaScript **keywords** are used to identify actions to be performed.
The let keyword tells the browser to create variables:

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

<h2>The <b>let</b>Keyword Creates Variables</h2>

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

<script>
let x, y;
x = 5 + 6;
y = x * 10;
document.getElementById("demo").innerHTML = y;
</script>

</body>
</html>
```

The var keyword also tells the browser to create variables:

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

<h2>The var Keyword Creates Variables</h2>

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

<script>
var  x,  y;
x = 5 + 6;
y = x * 10;
document.getElementById("demo").innerHTML = y;
</script>

</body>
</html>
```

In these examples, using var or let will produce the same result.
You will learn more about var and let later in this tutorial.

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:

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

<h2>JavaScript Comments are NOT Executed</h2>

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

<script>
let x;
x = 5;
// x = 6; I will NOT be executed
document.getElementById("demo").innerHTML = x;
```

```
</script>

</body>
</html>
```

JavaScript Identifiers

Identifiers are names.
In JavaScript, identifiers are used to name variables (and keywords, and functions, and labels).
The rules for legal names are much the same in most programming languages.
In JavaScript, the first character must be a letter, or an underscore (_), or a dollar sign ($).
Subsequent characters may be letters, digits, underscores, or dollar signs.
Numbers are not allowed as the first character.
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:

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

<h2>JavaScript is Case Sensitive</h2>

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

<script>
let lastname, lastName;
lastName = "Doe";
lastname = "Peterson";
document.getElementById("demo").innerHTML = LastName;
</script>

</body>
</html>
```

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.

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:

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

<h1 id ="myH"></h1>
```

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

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

</body>
</html>
```

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

Example:

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

<h2>JavaScript Comments</h2>

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

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

// Write y to demo:
document.getElementById("demo").innerHTML = y;
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>
<h1 id ="myH"></h1>

<p id= "myP" ></p>

<script>
/*
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.";
</script>

</body>
</html>
```

It is most common to use single line comments.
Block comments are often used for formal documentation.

Using Comments to Prevent Execution

Using comments to prevent execution of code is suitable for code testing.
Adding // in front of a code line changes the code lines from an executable line to a comment.
This example uses // to prevent execution of one of the code lines:
Example:

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

<h2>JavaScript Comments</h2>
```

```
<h1 id ="myH"></h1>

<p id= "myP" ></p>

<script>
//document.getElementById("myH").innerHTML = "My First Page";
document.getElementById("myP").innerHTML = "My first paragraph.";
</script>

</body>
</html>
```

This example uses a comment block to prevent execution of multiple lines:

Example:

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

<h2>JavaScript Comments</h2>

<h1 id ="myH"></h1>

<p id= "myP" ></p>

<script>
/*
document.getElementById("myH").innerHTML = "My First Page";
document.getElementById("myP").innerHTML = "My first paragraph.";
*/
document.getElementById("myP").innerHTML = "The comment-block is not executed.";
</script>

</body>
</html>
```

JAVASCRIPT VARIABLES

There are 3 ways to declare a JavaScript variable:

- Using var
- Using let
- Using const

This chapter uses var.
The let and const keywords are explained in the next chapters.

Variables

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

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

<h2>JavaScript Variables</h2>

<p>In this example, x, y, and z are variables. </p>

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

<script>
var x = 5;
var y = 6;
var z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>

</body>
</html>
```

From the example above, you can expect:

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

Much Like Algebra

In this example, price1, price2, and total, are variables:

Example:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>

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

<script>
var price1 = 5;
var price2 = 6;
var total = price1 + price2;
document.getElementById("demo").innerHTML =
"The total is: " + total;
</script>

</body>
</html>
```

In programming, just like in algebra, we use variables (like price1) to hold values.

In programming, just like in algebra, we use variables in expressions (total = price1 + price2).

From the example above, you can calculate the total to be 11.

JavaScript variables are containers for storing data values.

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 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.)

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:

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

<h2>JavaScript Variables</h2>

<p>Strings are written with quotes.</p>
<p>Numbers are written without quotes.</p>

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

<script>
var pi = 3.14;
var person = "John Doe";
var answer = 'Yes I am!';
document.getElementById("demo").innerHTML =
```

```
pi + "<br>" + person + "<br>" + answer;
</script>

</body>
</html>
```

Declaring (Creating) JavaScript Variables

Creating a variable in JavaScript is called "declaring" a variable.
You declare a JavaScript variable with the var keyword:

```
var carName;
```

After the declaration, the variable has no value (technically it has the value of 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:

```
var 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:

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

<h2>JavaScript Variables</h2>

<p>Create a variable, assign a value to it, and display it:</p>

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

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

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

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 var and separate the variables by **comma**:

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

<h2>JavaScript Variables</h2>

<p>You can declare many variables in one statement.</p>

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

<script>
var person = "John Doe", carName = "Volvo", price = 200;
document.getElementById("demo").innerHTML = carName;
</script>

</body>
</html>
```


A declaration can span multiple lines:

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

<h2>JavaScript Variables</h2>

<p>You can declare many variables in one statement.</p>

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

```
<script>
var person = "John Doe",
carName = "Volvo",
price = 200;
document.getElementById("demo").innerHTML = carName;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Variables</h2>

<p>A variable declared without a value will have the value undefined.</p>

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

<script>
var carName;
document.getElementById("demo").innerHTML = carName;
</script>

</body>
</html>
```

Re-Declaring JavaScript Variables

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

The variable carName will still have the value "Volvo" after the execution of

these statements:
Example:

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

<h2>JavaScript Variables</h2>

<p> If you re-declare a JavaScript variable, it will not lose its value.</p>

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

<script>
var carName = "Volvo";
var carName;
document.getElementById("demo").innerHTML = carName;
</script>

</body>
</html>
```

JavaScript Arithmetic

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

Example:

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

<h2>JavaScript Variables</h2>

<p>The result of adding 5 + 2 + 3:</p>

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

<script>
var x = 5 + 2 + 3;
document.getElementById("demo").innerHTML = x;
</script>
```

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

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

Example:

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

<h2>JavaScript Variables</h2>

<p>The result of adding "John" + " " + "Doe":</p>

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

<script>
var x = "John" + " " + "Doe";
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

Also try this:

Example:

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

<h2>JavaScript Variables</h2>

<p>The result of adding "5" + 2 + 3:</p>

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

<script>
```

```
var x = "5" + 2 + 3;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

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

Now try this:

Example:

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

<h2>JavaScript Variables</h2>

<p>The result of adding 2 + 3 + "5":</p>

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

<script>
var x = 2 + 3 + "5";
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```


JavaScript Dollar Sign $

Remember that JavaScript identifiers (names) must begin with:

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

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

Example:

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

<h2>JavaScript $</h2>
<p>The dollar sign is treated as a letter in JavaScript names.</p>

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

<script>
var $ = 2;
var $myMoney = 5;
document.getElementById("demo").innerHTML = $ + $myMoney;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript $</h2>

<p>The underscore is treated as a letter in JavaScript names.</p>

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

```
<script>
var _x = 2;
var _100 = 5;
document.getElementById("demo").innerHTML = _x + _100;
</script>

</body>
</html>
```

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.

JAVASCRIPT LET

- The let keyword was introduced in ES6 (2015).
- Variables defined with let cannot be Redeclared.
- Variables defined with let must be Declared before use.
- Variables defined with let have Block Scope.

Cannot be Redeclared

Variables defined with let cannot be **redeclared**.
You cannot accidentally redeclare a variable.
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;
```

Block Scope

Before ES6 (2015), JavaScript 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
```

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:

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

<h2>Redeclaring a Variable Using var</h2>

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

<script>
var x = 10;
// Here x is 10

{
var x = 2;
```

```
// Here x is 2
}

// Here x is 2
document.getElementById("demo").innerHTML =  x ;
</script>

</body>
</html>
```

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:

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

<h2>Redeclaring a Variable Using let</h2>

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

<script>
let x = 10;
// Here x is 10

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

// Here x is 10
document.getElementById("demo").innerHTML =  x ;
</script>

</body>
</html>
```

Browser Support

The let keyword is not fully supported in Internet Explorer 11 or earlier.

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

Chrome 49	Edge 12	Firefox 44	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:

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

<h2>JavaScript let</h2>

<p>Redeclaring a JavaScript variable with <b>var</b> is allowed anywhere in a program:</p>

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

<script>
var x = 2;
// Now x is 2

var x = 3;
// Now x is 3
document.getElementById("demo").innerHTML = x ;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript let</h2>

<p>Redeclaring a variable with <b>let</b>, in another scope, or in another block, is allowed:</p>

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

<script>
let x = 2;    // Allowed

{
let x = 3;    // Allowed
}

{
let x = 4;    // Allowed
}
document.getElementById("demo").innerHTML =  x ;
</script>
```

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

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:

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

<h2>JavaScript Hoisting</h2>

<p>With <b>var</b>, you can use a variable before it is declared:</p>

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

<script>
carName = "Volvo";
document.getElementById("demo").innerHTML = carName;
var carName;
</script>

</body>
</html>
```


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 Reference Error:

Example:

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

```
<h2>JavaScript let</h2>

<p> With <b>let</b>, you cannot use a variable before it is declared.</p>
<p id="demo"></p>

<script>
Try {
}
  carName = "Saab";
   let carName = "Volvo";
}
catch(err)
 { document.getElementById("demo").innerHTML = err;
</script>

</body>
</html>
```

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 const variable cannot be reassigned:

Example:

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

<h2>JavaScript const</h2>

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

<script>
```

```
Try {
}
  const PI = 3.141592653589793;
  PI = 3.14;
}
catch (err)
 { document.getElementById("demo").innerHTML = err;
}
</script>

</body>
</html>
```

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?

As a general rule, always declare a variable with const unless you know that the value will change.
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:

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

<h2>JavaScript const</h2>

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

<script>
// Create an array:
const cars = ["Saab", "Volvo", "BMW"];

// Change an element:
cars[0] = "Toyota";

// Add an element:
cars.push("Audi");

// Display the Array:
  document.getElementById("demo").innerHTML = car;
</script>

</body>
</html>
```

But you can NOT reassign the array:

Example:

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

<h2>JavaScript const</h2>
```

```
<p>You can NOT reassign a constant array:</p>

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

<script>
Try {
}
const cars = ["Saab", "Volvo", "BMW"];

cars = ["Toyota", "Volvo", "Audi"];    // ERROR
}
catch (err)
  { document.getElementById("demo").innerHTML = err;
}
</script>

</body>
</html>
```

Constant Objects

You can change the properties of a constant object:

Example:

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

<h2>JavaScript const</h2>

<p>Declaring a constant object does NOT make the objects properties
unchangeable:</p>
<p id="demo"></p>

<script>
// Create an object:
const car = {type:"Fiat", model:"500", color:"white"};

// Change a property:
car.color = "red";
```

```
// Add a property:
car.owner = "Johnson";

// Display the property:
document.getElementById("demo").innerHTML = "Car owner is " + car.owner;
</script>

</body>
</html>
```

But you can NOT reassign the object:

Example:

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

<h2>JavaScript const</h2>

<p>You can NOT reassign a constant object:</p>

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

<script>
try {
  const car = {type:"Fiat", model:"500", color:"white"};
car = {type:"Volvo", model:"EX60", color:"red"};
}
catch (err)
  { document.getElementById("demo").innerHTML =
  err;
}
</script>

</body>
</html>
```


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	Edge 12	Firefox 44	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:

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

<h2>JavaScropt const variables has block scope</h2>

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

<script>
const x = 10;
// Here x is 10

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

// Here x is 10
document.getElementById("demo").innerHTML = "x is " + x;
</script>

</body>
</html>
```


You can learn more about block scope in the chapter JavaScript Scope.

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
}
```

Const 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:

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

<h2>JavaScript Hoisting</h2>

<p>With <b>var</b>you can use a variable before it is declared:</p>

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

<script>
carName = "Volvo";
document.getElementById("demo").innerHTML = carName;
var carName;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Hoisting</h2>

<p>With <b>var</b>you can use a variable before it is declared:</p>

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

<script>
try
  { alert(carNam
  e);
  const carName = "Volvo";
}
catch (err)
  { document.getElementById("demo").innerHTML = err;
}
</script>

</body>
</html>
```


JAVASCRIPT OPERATORS

Example:
Assign values to variables and add them together:

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

<h2>JavaScript Operators</h2>
<p>x = 5, y = 2, calculate z = x + y, and display z:</p>
```

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

<script>
let x = 5;
let y = 2;
let z = x + y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

The **assignment** operator (=) assigns a value to a variable.
Assignment:

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

<h2>JavaScript Operators</h2>
<h3>The = Operators</h3>

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

<script>
let x = 10;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```


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

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

<h2>JavaScript Arithmeti</h2>
<h3>The + Operators</h3>
```

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

<script>
let x = 5;
let y = 2;
let z = x + y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

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

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

<h2>JavaScript Operators</h2>
<h3>The * Operators</h3>

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

<script>
let x = 5;
let y = 2;
let z = x * y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

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

The **addition assignment** operator (+=) adds a value to a variable.
Assignment:

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

<h2>The += Operators</h2>

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

<script>
let x = 10;
x += 5;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

JavaScript String Operators

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

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

<h2>JavaScript Operators</h2>

<p>The + operator concatenates (adds) strings.</p>

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

<script>
```

```
let text1 = "John";
let text2 = "Doe";
document.getElementById("demo").innerHTML = text1 + " " + text2;
</script>

</body>
</html>
```

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

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

<h2>JavaScript Operators</h2>

<p>The assignment operator += can concatenate strings..</p>

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

<script>
let text1 = "What a very ";
text1 += "nice day";
document.getElementById("demo").innerHTML = text1;
</script>

</body>
</html>
```

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:

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

```
<h2>JavaScript Operators</h2>

<p>Adding a number and a string, returns a string..</p>

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

<script>
let x = 5 + 5;
let y = "5" + 5;
let z = "Hello" + 5;
document.getElementById("demo").innerHTML = x + "<br>" + y + "<br>" + z;
</script>

</body>
</html>
```

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

JAVASCRIPT ARITHMETIC

JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Arithmetic Operations

A typical arithmetic operation operates on two numbers.

The two numbers can be literals:

Example:

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

```
<body>

<h2>JavaScript Arithmetic</h2>
<p>A typical arithmetic operation takes two numbers and produces a new number.</p>

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

<script>
let x = 100 + 50;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

or variables:
Example:

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

<h2>JavaScript Arithmetic</h2>
<p>A typical arithmetic operation takes two numbers (or variables) and produces a new number.</p>

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

<script>
let a = 100;
let b = 50;
let x = a + b;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

or expressions:

Example:

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

<h2>JavaScript Arithmetic</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>
```

Operators and Operands

The numbers (in an arithmetic operation) are called **operands**.
The operation (to be performed between the two operands) is defined by an **operator**.

Operand	Operator	Operand
100	+	50

Adding

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

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

<h2>JavaScript Arithmetic</h2>
<h3>The + Operator.</h3>

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

```
<script>
let x = 5;
let y = 2;
let z = x + y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

Subtracting

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

Example:

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

<h2>JavaScript Arithmetic</h2>
<h3>The - Operator.</h3>

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

<script>
let x = 5;
let y = 2;
let z = x - y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

Multiplying

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

Example:

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

```
<h2>JavaScript Arithmetic</h2>
<h3>The * Operator.</h3>

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

<script>
let x = 5;
let y = 2;
let z = x * y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

Dividing

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

Example:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arithmetic</h2>
<h3>The / Operator.</h3>

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

<script>
let x = 5;
let y = 2;
let z = x / y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

Remainder

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

Example:

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

<h2>JavaScript Arithmetic</h2>
<h3>The % Operator.</h3>

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

<script>
let x = 5;
let y = 2;
let z = x % y;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Arithmetic</h2>
<h3>The ++ Operator.</h3>

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

<script>
```

```
let x = 5;
x++;
let z = x;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

Decrementing

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

Example:

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

<h2>JavaScript Arithmetic</h2>
<h3>The - - Operator.</h3>

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

<script>
let x = 5;
x - - ;
let z = x;
document.getElementById("demo").innerHTML = z;
</script>

</body>
</html>
```

Exponentiation

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

Example:

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

```
<h2>JavaScript Arithmetic</h2>
<h3>The ** Operator.</h3>

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

<script>
let x = 5;
document.getElementById("demo").innerHTML = x ** 2;
</script>

</body>
</html>
```

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

Example:

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

<h2>JavaScript Arithmetic</h2>
<h3>Math.pow()</h3>

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

<script>
let x = 5;
document.getElementById("demo").innerHTML = Math.pow(x,2);
</script>

</body>
</html>
```


Operator Precedence

Operator precedence describes the order in which operations are performed in an arithmetic expression.

Example:

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

```
<h2>JavaScript Arithmetic</h2>
<p>Multiplication has precedence over addition.</p>

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

<script>
document.getElementById("demo").innerHTML = 100 + 50 * 3;
</script>

</body>
</html>
```

Is the result of example above the same as 150 * 3, or is it the same as 100 + 150?
Is the addition or the multiplication done first?
As in traditional school mathematics, the multiplication is done first.
Multiplication (*) and division (/) have higher **precedence** than addition (+) and subtraction (-).
And (as in school mathematics) the precedence can be changed by using parentheses:
Example:

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

<h2>JavaScript Arithmetic</h2>
<p>Multiplication has precedence over addition.</p>
<p>But parenthesis has precedence over multiplication.</p>

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

<script>
document.getElementById("demo").innerHTML = (100 + 50) * 3;
</script>

</body>
</html>
```

When using parentheses, the operations inside the parentheses are computed first.
When many operations have the same precedence (like addition and subtraction), they are computed from left to right:
Example:

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

<h2>JavaScript Arithmetic</h2>
<p>When many operations has the same precedence, they are computed from left to right.</p>

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

<script>
document.getElementById("demo").innerHTML = 100 + 50 - 3;
</script>

</body>
</html>
```

JavaScript Operator Precedence Values

Pale red entries indicates ECMAScript 2015 (ES6) or higher.

Value	Operator	Description	Example
21	( )	Expression grouping	(3 + 4)
20	.	Member	person.name
20	[]	Member	person["name"]
20	()	Function call	myFunction()
20	new	Create	new Date()
18	++	Postfix Increment	i++
18	--	Postfix Decrement	i--
17	++	Prefix Increment	++i
17	--	Prefix Decrement	--i
17	!	Logical not	!(x==y)
17	typeof	Type	typeof x

16	**	Exponentiation (ES2016)	10 ** 2
15	*	Multiplication	10 * 5
15	/	Division	10 / 5
15	%	Division Remainder	10 % 5
14	+	Addition	10 + 5
14	-	Subtraction	10 - 5
13	<<	Shift left	x << 2
13	>>	Shift right	x >> 2
13	>>>	Shift right (unsigned)	x >>> 2
12	<	Less than	x < y
12	<=	Less than or equal	x <= y
12	>	Greater than	x > y
12	>=	Greater than or equal	x >= y
12	in	Property in Object	"PI" in Math
12	instanceof	Instance of Object	instanceof Array
11	==	Equal	x == y
11	===	Strict equal	x === y
11	!=	Unequal	x != y
11	!==	Strict unequal	x !== y
10	&	Bitwise AND	x & y
9	^	Bitwise XOR	x ^ y
8	|	Bitwise OR	x | y
7	&&	Logical AND	x && y
6	||	Logical OR	x || y
5	??	Nullish Coalescing	x ?? y
4	? :	Condition	? "Yes" : "No"
3	+=	Assignment	x += y
3	/=	Assignment	x /= y
3	-=	Assignment	x -= y
3	*=	Assignment	x *= y
3	%=	Assignment	x %= y

3	<<=	Assignment	x <<= y
3	>>=	Assignment	x >>= y
3	>>>=	Assignment	x >>>= y
3	&=	Assignment	x &= y
3	^=	Assignment	x ^= y
3	|=	Assignment	x |= y
2	yield	Pause Function	yield x
1	,	Comma	5 , 6

Expressions in parentheses are fully computed before the value is used in the rest of the expression.

JAVASCRIPT ASSIGNMENT

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

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
>>=	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

The **= operator is a part of ECMAScript 2016.

Assignment Examples

The = assignment operator assigns a value to a variable.

Assignment:

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

<h2>JavaScript Assignments</h2>
<h3>The = Operator</h3>

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

<script>
let x = 10;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

The += assignment operator adds a value to a variable.
Assignment:

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

<h2>JavaScript Assignments</h2>
<h3>The += Operator</h3>

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

<script>
let x = 10;
x += 5;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

The -= assignment operator subtracts a value from a variable.
Assignment:

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

<h2>JavaScript Assignments</h2>
<h3>The - = Operator</h3>

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

<script>
let x = 10;
x - = 5;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

The *= assignment operator multiplies a variable.
Assignment:

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

<h2>JavaScript Assignments</h2>
<h3>The *= Operator</h3>

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

<script>
let x = 10;
x *= 5;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

The /= assignment divides a variable.

Assignment:

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

<h2>JavaScript Assignments</h2>
<h3>The /= Operator</h3>

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

<script>
let x = 10;
x /= 5;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

The %= assignment operator assigns a remainder to a variable.

Assignment:

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

<h2>JavaScript Assignments</h2>
<h3>The %= Operator</h3>

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

<script>
let x = 10;
x %= 5;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

JAVASCRIPT DATA TYPES

JavaScript variables can hold different data types: numbers, strings, objects and more:

```
let length = 16;                               // Number
let lastName = "Johnson";                      // String
let x = {firstName:"John", lastName:"Doe"};    // Object
```

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";
```

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

Example:

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

<h2>JavaScript</h2>
<p>When adding a number and a string, JavaScript will treat the number as a string.</p>

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

<script>
let x = 16 + "Volvo";
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

Example:

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

<h2>JavaScript</h2>
<p>When adding a number and a string, JavaScript will treat the number as a string.</p>

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

<script>
let x = "Volvo" + 16;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

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

JavaScript:

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

<h2>JavaScript</h2>
<p>JavaScript evaluates expressions from left to right. Different sequences can produce different results:</p>

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

<script>
let x = 16 + 4 + "Volvo";
document.getElementById("demo").innerHTML = x;
</script>

</body>
```

```
</html>
```

JavaScript:

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

<h2>JavaScript</h2>
<p>JavaScript evaluates expressions from left to right. Different sequences can produce different results:</p>

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

<script>
let x = "Volvo" + 16 + 4;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Data Types</h2>
<p>JavaScript has dynamic types. This means that the same variable can be used to hold different data types:</p>

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

```
<script>
let x;              // Now x is undefined
x = 5;              // Now x is a Number
x = "John";         // Now x is a String
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Strings</h2>
<p>Strings are written with quotes. You can use single or double quotes:</p>

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

<script>
let carName1 = "Volvo XC60";
let carName2 = 'Volvo XC60';
document.getElementById("demo").innerHTML =
carName1 + "<br>" +
carName2;
</script>

</body>
</html>
```

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

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript Strings</h2>
<p>Strings are written with quotes. You can use single or double quotes:</p>

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

<script>
let answer1 = "It's alright";
let answer2 = "He is called 'Johnny'";
let answer3 = 'He is called "Johnny"';
document.getElementById("demo").innerHTML =
answer1 + "<br>" +
answer2 + "<br>" +
answer3;
</script>

</body>
</html>
```

You will learn more about strings later in this tutorial.

JavaScript Numbers

JavaScript has only one type of numbers.
Numbers can be written with, or without decimals:
Example:

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

<h2>JavaScript Numbers</h2>
<p>Numbers can be written with, or without decimals:</p>

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

<script>
let x1 = 34.00;
let x2 = 34;
```

```
let x3 = 3.14;
document.getElementById("demo").innerHTML =
x1 + "<br>" + x2 + "<br>" + x3;
</script>

</body>
</html>
```

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

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

<h2>JavaScript Numbers</h2>
<p> Extra large or extra small numbers can be written with scientific (exponential) notation:</p>

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

<script>
let y = 123e5;
let z = 123e-5;
document.getElementById("demo").innerHTML =
y + "<br>" + z;
</script>

</body>
</html>
```


You will learn more about numbers later in this tutorial.

JavaScript Booleans

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

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

```
<h2>JavaScript Booleans </h2>
<p> Booleans can only have two values: true or false.</p>

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

<script>
let x = 5;
let y = 5;
let z = 6;
document.getElementById("demo").innerHTML =
(x == y) + "<br>" + (x == z);
</script>

</body>
</html>
```

Booleans are often used in conditional testing.
You will learn more about conditional testing later in this tutorial.

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:

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

<h2>JavaScript Arrays </h2>
<p> Array indexes are zero-based, which means the first item is [0].</p>

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

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = car[0]
</script>
```

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

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.
You will learn more about **arrays** later in this tutorial.

JavaScript Objects

JavaScript objects are written with curly braces {}.
Object properties are written as name:value pairs, separated by commas.
Example:

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

<h2>JavaScript Objects </h2>
<p id="demo"></p>

<script>
const person =
    { firstName:"John
    ",lastName:"Doe",
    age:50,
    eyeColor:"blue"
};
document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```


The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.
You will learn more about **objects** later in this tutorial.

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:

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

<h2>JavaScript Typeof </h2>
<p>The typeof operator returns the type of a variable or an expression.</p>

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

<script>
document.getElementById("demo").innerHTML =
typeof "" + ” <br>” +
typeof "John" + ” <br>” +
typeof "John Doe"
</script>

</body>
</html>
```

Example:

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

<h2>JavaScript Typeof </h2>
<p>The typeof operator returns the type of a variable or an expression.</p>

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

<script>
document.getElementById("demo").innerHTML =
typeof 0  +” <br>” +
typeof 314 +” <br>” +
typeof 3.14 +” <br>” +
typeof (3) +” <br>” +
typeof (3 + 4) ;
</script>
```

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

You will learn more about **typeof** later in this tutorial.

Undefined

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

Example:

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

<h2>JavaScript </h2>
<p>he value (and the data type) of a variable with no value is
<b>undefined</b>.</p>

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

<script>
let car;
document.getElementById("demo").innerHTML =
car + "<br>" + typeof car;
</script>

</body>
</html>
```


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

Example:

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

<h2>JavaScript </h2>
<p>Variables can be emptied if you set the value to <b>undefined</b>.</p>

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

```
<script>
let car= “Volvo;
car = undefined;
document.getElementById("demo").innerHTML =
car + "<br>" + typeof car;
</script>

</body>
</html>
```

Empty Values

An empty value has nothing to do with undefined.

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

Example:

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

<h2>JavaScript </h2>
<p>An empty string has both a legal value and a type:.</p>

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

<script>
let car = " ";
document.getElementById("demo").innerHTML =
"The value is: " +
car + "<br>" +
"The type is: " + typeof car;
</script>

</body>
</html>
```


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:

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

<h2>JavaScript Functions </h2>
<p>This example calls a function which performs a calculation, and returns the result:</p>

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

<script>
function myFunction(p1, p2)
 {return p1 * p2;
}
document.getElementById("demo").innerHTML = myFunction(4,3);
</script>

</body>
</html>
```

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.

A Function is much the same as a Procedure or a Subroutine, in other programming languages.

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)

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

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:

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

<h2>JavaScript Functions </h2>
<p>This example calls a function which performs a calculation, and returns the result:</p>

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

<script>
var x = myFunction(4, 3);
document.getElementById("demo").innerHTML = x;
function myFunction(a, b) {
  return a * b;
 }
</script>

</body>
```

```
</html>
```

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.
Example:
Convert Fahrenheit to Celsius:

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

<h2>JavaScript Functions </h2>
<p>This example calls a function to convert from Fahrenheit to Celsius:</p>

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

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

</body>
</html>
```

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.
Example:

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

<h2>JavaScript Functions </h2>
```

```
<p>Accessing a function without () will return the function definition instead of the function result:</p>

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

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

</body>
</html>
```

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:

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

<h2>JavaScript Functions </h2>

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

<script>
document.getElementById("demo").innerHTML =
"The temperature is " + toCelsius(77) + " Celsius";
function toCelsius(fahrenheit) {
 return (5/9) * (fahrenheit-32);
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Functions </h2>

<p>Outside myFunction() carName is undefined.</p>

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

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

<script>
myFunction();
function myFunction()
 {let carName = "Volvo";
 document.getElementById("demo").innerHTML =
typeof carName + " " + carName;
}

document.getElementById("demo2").innerHTML =
typeof carName;
</script>

</body>
</html>
```

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:

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

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

<h2>JavaScript Variables</h2>

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

<script>
// Create and display a variable:
let car = "Fiat";
document.getElementById("demo2").innerHTML = car;
</script>

</body>
</html>
```

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

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

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

<h2>JavaScript Objects</h2>

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

<script>
// Create an Objects:
const car = {type:"Fiat", model:"500", color:"white"};

// Display some data from the object:
document.getElementById("demo2").innerHTML = "The car type is " +
car.type;
</script>

</body>
</html>
```

The values are written as **name:value** pairs (name and value separated by a colon).

It is a common practice to declare objects with the const keyword.
Learn more about using const with objects in the chapter: JS Const.

Object Definition

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

Example:

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

<h2>JavaScript Objects</h2>

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

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

// Display some data from the object:
document.getElementById("demo2").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```

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

Example:

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

<h2>JavaScript Objects</h2>

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

<script>
// Create an Objects:
const person =
 { firstName:
 "John",lastName:
 "Doe", age: 50,
 eyeColor: "blue"
};

// Display some data from the object:
document.getElementById("demo2").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
```

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

Object Properties

The **name:values** pairs in JavaScript objects are called **properties**:

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName

or

objectName["propertyName"]

Example1:

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

<h2>JavaScript Objects</h2>

<p>There are two different ways to access an object property.</p>

<p>You can use person.property or person["property"].</p>

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

<script>
// Create an Objects:
const person =
{ firstName: "John",
lastName: "Doe",
  id : 5566
};
```

```
// Display some data from the object:
document.getElementById("demo2").innerHTML =
person.firstName + " " + person.lastName;
</script>

</body>
</html>
```

Example2:

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

<h2>JavaScript Objects</h2>

<p>There are two different ways to access an object property.</p>

<p>You can use person.property or person["property"].</p>

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

<script>
// Create an Objects:
const person =
 { firstName:
 "John",lastName:
 "Doe", id : 5566
};

// Display some data from the object:
document.getElementById("demo2").innerHTML =
person[“firstName”] + " " + person[“lastName”]
</script>

</body>
</html>
```

JavaScript objects are containers for **named values** called properties.

Object Methods

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

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

A method is a function stored as a property.

Example:

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

*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**.
Read more about the this keyword at JS this Keyword.

Accessing Object Methods

You access an object method with the following syntax:

objectName.methodName()

Example:

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

```
<h2>JavaScript Objects</h2>

<p>An object method is a function definition, stored as a property value.</p>

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

<script>
// Create an Objects:
const person =
 { firstName:
 "John",lastName:
 "Doe", id : 5566
};
fullName: function() {
   return this.firstName + " " + this.lastName;
 }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>

</body>
</html>
```

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

Example:

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

<h2>JavaScript Objects</h2>

<p>If you access an object method without (), it will return the function definition:</p>

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

<script>
```

```
// Create an Objects:
const person =
 { firstName:
 "John",lastName:
 "Doe", id : 5566
};
fullName: function() {
   return this.firstName + " " + this.lastName;
 }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>

</body>
</html>
```

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
```

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:
Example:

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

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

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

</body>
</html>
```

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:

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

<h2>JavaScript HTML Events</h2>
<button onclick="this.innerHTML = Date()">The time is?</button>

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

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

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

Example:

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

<h2>JavaScript HTML Events</h2>
<p>Click the button to display the date.</p>

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

<script>
function displayDate()
  { document.getElementById("demo").innerHTML = Date();
}
</script>

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

</body>
</html>
```


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

What can JavaScript Do?

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 ...

JAVASCRIPT STRINGS

JavaScript strings are used for storing and manipulating text.

JavaScript Strings

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

Example:

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

<h2>JavaScript strings</h2>

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

<script>
let text = "John Doe";      // String written inside quotes
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

You can use single or double quotes:

Example:

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

<h2>JavaScript strings</h2>
<p>Strings are written inside quotes. You can use single or double quotes:</p>

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

<script>
let carName1 = "Volvo XC60"; // Double quotes
let carName2 = 'Volvo XC60';  // Single quotes

document.getElementById("demo").innerHTML = text;
carName1 + " " + carName2;
</script>

</body>
</html>
```

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

Example:

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

<h2>JavaScript strings</h2>
<p>You can use quotes inside a string, as long as they don't match the quotes surrounding the string.</p>

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

<script>
let answer1 = "It's alright";
let answer2 = "He is called 'Johnny'";
```

```
let answer3 = 'He is called "Johnny"';

document.getElementById("demo").innerHTML = text;
answer1 + "<br>" + answer2 + "<br>" + answer3;
 </script>

</body>
</html>
```

String Length

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

Example:

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

<h2>JavaScript strings properties</h2>
<p>The length property returns the length of a string:</p>

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

<script>
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
document.getElementById("demo").innerHTML = text.lenght;
</script>

</body>
</html>
```


Escape Character

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 ".

The solution to avoid this problem, is to use the **backslash escape character**.

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

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

The sequence \" inserts a double quote in a string:
Example:

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

<h2>JavaScript strings</h2>
<p>The escape sequence \" inserts a double quote in a string.</p>

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

<script>
let text = "We are the so-called \"Vikings\" from the north.";
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

The sequence \' inserts a single quote in a string:
Example:

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

<h2>JavaScript strings</h2>
<p>The escape sequence \" inserts a single quote in a string.</p>

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

<script>
let text= 'It\'s alright.';
document.getElementById("demo").innerHTML = text;
</script>
```

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

The sequence \\ inserts a backslash in a string:
Example:

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

<h2>JavaScript strings</h2>
<p>The escape sequence \\ inserts a backslash in a string.</p>

<p id="demo"></p>
<script>
let text = "The character \\ is called backslash.";
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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

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 Code Lines

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:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript Statments</h2>
<p>The best place to break a code line is after an operator or a comma.</p>

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

<script>
document.getElementById("demo").innerHTML =
"Hello Dolly!";
</script>

</body>
</html>
```

You can also break up a code line **within a text string** with a single backslash:
Example:

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

<h2>JavaScript Strings</h2>
<p>You can break a code line within a text string with a backslash.</p>

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

<script>
document.getElementById("demo").innerHTML = "Hello \
Dolly!";
</script>

</body>
</html>
```

The \ method is not the preferred method. It might not have universal support. Some browsers do not allow spaces behind the \ character.

A safer way to break up a string, is to use string addition:

Example:

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

<h2>JavaScript Strings</h2>
<p>The safest way to break a code line in a string is using string addition.</p>

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

<script>
document.getElementById("demo").innerHTML = "Hello " +
"Dolly!";
</script>

</body>
</html>
```

You cannot break up a code line with a backslash:

Example:

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

<h2>JavaScript Statments</h2>

<p id="demo"> You cannot break a code line with a \ backslash.</p>

<script>
document.getElementById("demo").innerHTML = \
"Hello Dolly!";
</script>

</body>
</html>
```

Strings Can be Objects

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

```
let firstName = "John";
```

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

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

Example:

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

<h2>JavaScript Strings</h2>

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

<script>
let x = "John";             // x is a string
let y = new String("John"); // y is an object

document.getElementById("demo").innerHTML =
typeof x + "<br>" + typeof y;
</script>

</body>
</html>
```

Don't create strings as objects. It slows down execution speed.
The new keyword complicates the code. This can produce some unexpected results:

When using the == operator, equal strings are equal:

Example:

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

<h2>Never Create Strings as Objects</h2>
<p>Strings and objects cannot be safely compared.</p>

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

<script>
```

```
let x = "John";                 // x is a string
let y = new String("John");     // y is an object
document.getElementById("demo").innerHTML = (x==y);
</script>

</body>
</html>
```

When using the === operator, equal values may not be equal, because the === operator expects equality in both data type and value.

Example:

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

<h2>Never Create Strings as Objects</h2>
<p>Strings and objects cannot be safely compared. </p>

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

<script>
let x = "John";                 // x is a string
let y = new String("John");     // y is an object
document.getElementById("demo").innerHTML = (x===y);
</script>

</body>
</html>
```

Or even worse. Objects cannot be compared:

Example:

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

<h2>Never Create Strings as Objects</h2>
<p> JavaScript objects cannot be compared.</p>

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

```
<script>
let x = new String("John");         // x is an object
let y = new String("John");        // y is an object
document.getElementById("demo").innerHTML = (x==y);
</script>

</body>
</html>
```

Example:

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

<h2>Never Create Strings as Objects</h2>
<p> JavaScript objects cannot be compared.</p>

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

<script>
let x = new String("John");         // x is an object
let y = new String("John");        // y is an object
document.getElementById("demo").innerHTML = (x===y);
</script>

</body>
</html>
```

Note the difference between (x==y) and (x===y).
Also note that comparing two JavaScript objects will **always** return false.

JAVASCRIPT STRING METHODS

String methods help you to work with strings.

String Methods and Properties

Primitive values, like "John Doe", cannot have properties or methods (because they are not objects).

But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

String Length

The length property returns the length of a string:

Example:

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

<h2>JavaScript String Properties</h2>
<p>The length property returns the length of a string:</p>

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

<script>
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
document.getElementById("demo").innerHTML = text.length;
</script>

</body>
</html>
```

Extracting String Parts

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

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

The slice() Method

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

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

This example slices out a portion of a string from position 7 to position 12 (13-1):

Example:

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

```
<body>

<h2>JavaScript String Methods</h2>
<p>The slice() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(7, 13);
</script>

</body>
</html>
```

Remember: JavaScript counts positions from zero. First position is 0.

If a parameter is negative, the position is counted from the end of the string. This example slices out a portion of a string from position -12 to position -6:
Example:

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

<h2>JavaScript String Methods</h2>
<p>The slice() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(-12, -6);
</script>

</body>
</html>
```

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

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

<h2>JavaScript String Methods</h2>
<p>The slice() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(7);
</script>

</body>
</html>
```

or, counting from the end:
Example:

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

<h2>JavaScript String Methods</h2>
<p>The slice() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(-12);
</script>
```

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

The substring() Method

substring() is similar to slice().
The difference is that substring() cannot accept negative indexes.
Example:

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

<h2>JavaScript String Methods</h2>
<p>The substring() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(7,13);
</script>

</body>
</html>
```

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

The substr() Method

substr() is similar to slice().
The difference is that the second parameter specifies the **length** of the extracted part.
Example:

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

<h2>JavaScript String Methods</h2>
```

```
<p>The substring() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(7,6);
</script>

</body>
</html>
```

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

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

<h2>JavaScript String Methods</h2>
<p>The substring() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(7);
</script>

</body>
</html>
```

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

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

```
<h2>JavaScript String Methods</h2>
<p>The substring() method extract a part of a string and returns the extracted parts in a new string:</p>

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

<script>
let str = "Apple, Banana, Kiwi";
document.getElementById("demo").innerHTML = str.slice(-4);
</script>

</body>
</html>
```

Replacing String Content

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Replace "Microsoft" with "ditrp" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Please visit Microsoft!</p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text.replace("Microsoft", "Ditrp");
}
</script>

</body>
</html>
```

The replace() method does not change the string it is called on. It returns a new string.

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

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

<h2>JavaScript String Methods</h2>
<p>Replace "Microsoft" with "Ditrp" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Please visit Microsoft and Microsoft! </p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text.replace("Microsoft", "Ditrp");
}
</script>

</body>
</html>
```

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

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

<h2>JavaScript String Methods</h2>
<p>Try to replace "Microsoft" with "ditrp" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Please visit Microsoft </p>
```

```
<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text.replace("Microsoft", "Ditrp");
}
</script>
<p>The replace() method is case sensitive. MICROSOFT (with upper-case) will not be replaced. </p>
</body>
</html>
```

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Replace "Microsoft" with "ditrp" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Please visit Microsoft </p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text.replace(/MICROSOFT/i, "Ditrp");
}
</script>

</body>
</html>
```

Note that regular expressions are written without quotes.

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Replace all occurrences of "Microsoft" with "Ditrp" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Please visit Microsoft and Microsoft! </p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text.replace(/MICROSOFT/g, "Ditrp");
}
</script>

</body>
</html>
```

Converting to Upper and Lower Case

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Convert string to upper case:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Hello World! </p>

<script>
```

```
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text1.toUpperCase();
}
</script>

</body>
</html>
```

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Convert string to Lower case:</p>

<button onclick="myFunction()">Try it</button>

<p id="demo">Hello World! </p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text1.toLowerCase();
}
</script>

</body>
</html>
```

The concat() Method

concat() joins two or more strings:

Example:

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

```
<h2>JavaScript String Methods</h2>
<p>The concat() method joins two or more strings:</p>

<button onclick="myFunction()">Try it</button>

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

<script>
let text1 = "Hello";
let text2 = "World";
let text3 = text1.concat(" ", text2); let text =
  document.getElementById("demo").innerHTML =text3;

</script>

</body>
</html>
```

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!");
```

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.

String.trim()

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

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

<h2>JavaScript String.trim()</h2>
<p>Click the button to alert the string with removed whitespace.</p>
```

```
<button onclick="myFunction()">Try it</button>

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

<script>
function myFunction() {
let text = "       Hello World!
Alert(text.trim() );
}
</script>

</body>
</html>
```

JavaScript String Padding

ECMAScript 2017 added two String methods: padStart and padEnd to support padding at the beginning and at the end of a string.

Example:

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

<h2>JavaScript String Methods</h2>
<p>The padStart() method pads a string with another string:</p>

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

<script>
let text = "5";
document.getElementById("demo").innerHTML = text.padStart(4,0);
</script>

</body>
</html>
```

Example:

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

```
<h2>JavaScript String Methods</h2>
<p>The padEnd() method pads a string with another string:</p>

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

<script>
let text = "5";
document.getElementById("demo").innerHTML = text.padEnd(4,0);
</script>

</body>
</html>
```

String Padding is not supported in Internet Explorer.
Firefox and Safari were the first browsers with support for JavaScript stringpadding:

Chrome 57	Edge 15	Firefox 48	Safari 10	Opera 44
Mar 2017	Apr 2017	Aug 2016	Sep 2016	Mar 2017

Extracting String Characters

There are 3 methods for extracting string characters:

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

The charAt() Method

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

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

<h2>JavaScript String Methods</h2>
<p>The charAt() method returns the character at a given position in a string:</p>
```

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

<script>
let text = "HELLO WORLD";
document.getElementById("demo").innerHTML = text.charAt(0);
</script>

</body>
</html>
```

The charCodeAt() Method

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).

Example:

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

<h2>JavaScript String Methods</h2>
<p>The charCodeAt() method returns the character at a given position in a string:</p>

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

<script>
let text = "HELLO WORLD";
document.getElementById("demo").innerHTML = text.charCodeAt(0);
</script>

</body>
</html>
```

Property Access

ECMAScript 5 (2009) allows property access [ ] on strings:

Example:

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

```
<h2>JavaScript String Methods</h2>
<p>ECMAScript 5 allows property access on strings:</p>

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

<script>
let text = "HELLO WORLD";
document.getElementById("demo").innerHTML = str(0);
</script>

</body>
</html>
```

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:

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

<h2>JavaScript String Methods</h2>
<p>ES5 (2009) allows property acces on strings. But read only:</p>

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

<script>
let text = "HELLO WORLD";
text[0] = "A"  // does not work

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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

Converting a String to an Array

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Display the first array element, after a string split:</p>
<p id="demo"></p>

<script>
let text = "a,b,c,d,e,f";
const myArray = text.split(",")
document.getElementById("demo").innerHTML =  myArray[0];
</script>

</body>
</html>
```

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:

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

<h2>JavaScript String Methods</h2>
<p>Using String.split():</p>

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

<script>
let text = "Hello";
const myArray = text.split(",")

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

```
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

JAVASCRIPT STRING SEARCH

JavaScript methods for searching strings:

- String.indexOf()
- String.lastIndexOf()
- String.startsWith()
- String.endsWith()

String.indexOf()

The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string:

Example:

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

<h2>JavaScript String Methods</h2>
<p>The indexOf() method returns the position of the first occurrence of a specified text:</p>

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

<script>
let str = "Please locate where 'locate' occurs!";
document.getElementById("demo").innerHTML = str.indexOf("locate");
</script>

</body>
</html>
```

JavaScript counts positions from zero.
0 is the first position in a string, 1 is the second, 2 is the third ...

String.lastIndexOf()

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>The lastIndexOf() method returns the position of the last occurrence of a specified text:</p>

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

<script>
let str = "Please locate where 'locate' occurs!";
document.getElementById("demo").innerHTML = str.lastIndexOf("locate");
</script>

</body>
</html>
```

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>Both indexOf(), and lastIndexOf() return -1 if the text is not found:</p>

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

<script>
let str = "Please locate where 'locate' occurs!";
document.getElementById("demo").innerHTML = str.IndexOf("John ");
</script>
```

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

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

Example:

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

<h2>JavaScript String Methods</h2>
<p>The indexOf() method accepts a second parameter as the starting position for the search:</p>

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

<script>
let str = "Please locate where 'locate' occurs!";
document.getElementById("demo").innerHTML = str.indexOf("locate", 15);
</script>

</body>
</html>
```

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:

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

<h2>JavaScript String Methods</h2>

<p>The lastIndexOf() method accepts a second parameter as the starting position for the search.</p>
<p>Remember that the lastIndexOf() method searches backwards, so position 15 means start the search at position 15, and search to the beginning.</p>
<p>Position 15 is position 15 from the beginning.</p>
```

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

<script>
let str = "Please locate where 'locate' occurs!";
document.getElementById("demo").innerHTML = str.lastIndexOf
("locate",15);
</script>

</body>
</html>
```

String.search()

The search() method searches a string for a specified value and returns the position of the match:

Example:

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

<h2>JavaScript String Methods</h2>

<p>The search() method returns the position of the first occurrence of a specified text in a string:</p>

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

<script>
let str = "Please locate where 'locate' occurs!";
document.getElementById("demo").innerHTML = str.search("locate");
</script>

</body>
</html>
```

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.

String.match()

The match() method searches a string for a match against a regular expression, and returns the matches, as an Array object.

Example 1:

Search a string for "ain":

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

<h2>JavaScript String Search</h2>

<p>Search a string for "ain":</p>

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

<script>
let text = "The rain in SPAIN stays mainly in the plain";
document.getElementById("demo").innerHTML = text.match(/ain/g);
</script>

</body>
</html>
```


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

Syntax:

string.match(*regexp*)

regexp	Required. The value to search for, as a regular expression.
Returns:	An Array, containing the matches, one item for each match, or *null* if no match is found

Example 2

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

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

<h2>JavaScript String Search</h2>

<p>Perform a global, case-insensitive search for "ain":</p>

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

<script>
let text = "The rain in SPAIN stays mainly in the plain";
document.getElementById("demo").innerHTML = text.match(/ain/g);
</script>

</body>
</html>
```

String.includes()

The includes() method returns true if a string contains a specified value.

Example:

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

<h2>JavaScript String Search</h2>

<p>Check if a string includes "world":</p>

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

<p>The includes() method is not supported in Internet Explorer.</p>

<script>
let text = "Hello world, welcome to the universe.";
document.getElementById("demo").innerHTML = text.includes("world");
</script>

</body>
```

```
</html>
```

Browser Support

String.includes() is not supported in Internet Explorer.

				
Chrome 41	Edge 12	Firefox 40	Safari 9	Opera 28
Mar 2015	Jul 2015	Aug 2015	Oct 2015	Mar 2015

Syntax:

string.includes(*searchvalue*, *start*)

searchvalue	Required. The string to search for
start	Optional. Default 0. Position to start the search
Returns:	Returns true if the string contains the value, otherwise false
JS Version:	ES6 (2015)

Check if a string includes "world", starting the search at position 12:

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

<h2>JavaScript String Search</h2>

<p>Check if a string includes "world":</p>

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

<p>The includes() method is not supported in Internet Explorer.</p>

<script>
let text = "Hello world, welcome to the universe.";
document.getElementById("demo").innerHTML = text.includes("world",12);
</script>

</body>
</html>
```

String.startsWith()

The startsWith() method returns true if a string begins with a specified value,

otherwise false:
Example:

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

<h2>JavaScript Strings</h2>

<p>Check if a string starts with "Hello":</p>

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

<p> The startsWith() method is not supported in Internet Explorer.</p>

<script>
let text = "Hello world, welcome to the universe.";
document.getElementById("demo").innerHTML = text.startsWith("Hello");
</script>

</body>
</html>
```

Syntax:

string.startsWith(*searchvalue*, *start*)

Parameter Values

Parameter	Description
searchvalue	Required. The value to search for.
start	Optional. Default 0. The position to start the search.

Examples:

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

<h2>JavaScript Strings</h2>

<p>The startsWith() method.</p>

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

```
<p>The startsWith() method is not supported in Internet Explorer.</p>

<script>
let text = "Hello world, welcome to the universe.";
document.getElementById("demo").innerHTML = text.startsWith("world", 6);
</script>

</body>
</html>
```

Note: The startsWith() method is case sensitive.

The startsWith() method is not supported in Internet Explorer.
The first browser versions with full support was:

				
Chrome 41	Edge 12	Firefox 17	Safari 9	Opera 28
Mar 2015	Jul 2015	Aug 2015	Oct 2015	Mar 2015

String.endsWith()

The endsWith() method returns true if a string ends with a specified value, otherwise false:

Example

Check if a string ends with "Doe":

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

<h2>JavaScript Strings</h2>

<p>Check if a string ends with "Doe":</p>

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

<p>The endsWith() method is not supported in Internet Explorer.</p>

<script>
let text = "John Doe";
document.getElementById("demo").innerHTML = text.endsWith("Doe");
```

```
</script>

</body>
</html>
```

Syntax:

string.endswith(*searchvalue*, *length*)

Parameter Values

Parameter	Description
searchvalue	Required. The value to search for.
length	Optional. The length to search.

Check in the 11 first characters of a string ends with "world":

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

<h2>JavaScript Strings</h2>

<p>Check in the 11 first characters of a string ends with "world":</p>

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

<p>The endsWith() method is not supported in Internet Explorer.</p>

<script>
let text = "Hello world, welcome to the universe.";
document.getElementById("demo").innerHTML = text.endsWith("world", 11);
</script>

</body>
</html>
```

Note: The endsWith() method is case sensitive.

The endsWith() method is not supported in Internet Explorer.
The first browser versions with full support was:

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

JAVASCRIPT TEMPLATE LITERALS

Synonyms:

- Template Literals
- Template Strings
- String Templates
- Back-Tics Syntax

Back-Tics Syntax

Template Literals use back-ticks (``) rather than the quotes ("") to define a string:

Example:

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

<h2>JavaScript Template Literals</h2>

<p>Template literals use back-ticks (``) to define a string:</p>

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

<p>Template literals are not supported in Internet Explorer.</p>

<script>
let text = `Hello World!`;
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Quotes Inside Strings

With **template literals**, you can use both single and double quotes inside a

string:
Example:

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

<h2>JavaScript Template Literals</h2>

<p>Template literals use back-ticks (``) to define a string:</p>

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

<p>With back-ticks, you can use both single and double quotes inside a string:</p>

<script>
let text = `He's often called "Johnny"`;
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Multiline Strings

Template literals allows multiline strings:

Example:

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

<h2>JavaScript Template Literals</h2>

<p>Template literals allows multiline strings</p>

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

<p>Template literals are not supported in Internet Explorer.</p>

<script>
```

```
let text =`The quick brown fox jumps over the lazy dog`;
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Interpolation

Template literals provide an easy way to interpolate variables and expressions into strings.

The method is called string interpolation.

The syntax is:

${...}

Variable Substitutions

Template literals allow variables in strings:

Example:

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

<h2>JavaScript Template Literals</h2>

<p>Template literals allows variables in strings:</p>

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

<p>Template literals are not supported in Internet Explorer.</p>

<script>
let firstName = "John";
let lastName = "Doe";

let text = `Welcome ${firstName}, ${lastName}!`;
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Automatic replacing of variables with real values is called **string interpolation**.

Expression Substitution

Template literals allow expressions in strings:

Example:

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

<h2>JavaScript Template Literals</h2>

<p>Template literals allows variables in strings:</p>

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

<p>Template literals are not supported in Internet Explorer.</p>

<script>
let price = 10;
let VAT = 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```


Automatic replacing of expressions with real values is called **string interpolation**.

HTML Templates

Example:

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

<h2>JavaScript Template Literals</h2>
```

```
<p>Template literals allows variables in strings:</p>

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

<p>Template literals are not supported in Internet Explorer.</p>

<script>
let header = "Templates Literals";
let tags = ["template literals", "javascript", "es6"];

let html = `<h2>${header}</h2><ul>`;
for (const x of tags) {
  html += `<li>${x}</li>`;
}

html += `</ul>`;
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Browser Support

Template Literals are not supported in Internet Explorer.

The first browser versions with full support was:

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

JAVASCRIPT NUMBERS

JavaScript has only one type of number. Numbers can be written with or without decimals.

Example:

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

```
<body>

<h2>JavaScript Numbers</h2>

<p>Numbers can be written with or without decimals:</p>

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

<script>
let x = 3.14;
let y = 3;
document.getElementById("demo").innerHTML = x +  “<br>” + y;
</script>

</body>
</html>
```

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

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

<h2>JavaScript Numbers</h2>

<p>Extra large or extra small numbers can be written with scientific (exponent) notation:</p>

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

<script>
let x = 123e5;
let y = 123e-5;
document.getElementById("demo").innerHTML = x +  “<br>” + y;
</script>

</body>
</html>
```

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)

Precision

Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

Example:

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

<h2>JavaScript Numbers</h2>

<p>Integers (numbers without a period or exponent notation) are accurate up to 15 digits:</p>

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

<script>
let x = 999999999999999;
let y = 9999999999999999;
document.getElementById("demo").innerHTML = x + "<br>" + y;
</script>

</body>
</html>
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

Example:

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

```
<body>

<h2>JavaScript Numbers</h2>

<p>Floating point arithmetic is not always 100% accurate.</p>

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

<script>
let x = 0.2 + 0.1;
document.getElementById("demo").innerHTML = "0.2 + 0.1 = " + x;
</script>

</body>
</html>
```

To solve the problem above, it helps to multiply and divide:
Example:

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

<h2>JavaScript Numbers</h2>

<p>Floating point arithmetic is not always 100% accurate.</p>
<p id="demo"></p>

<p>But it helps to multiply and divide:</p>

<script>
let x = 0.2 + 0.1;
document.getElementById("demo1").innerHTML = "0.2 + 0.1 = " + x;
let x = (0.2 * 10 + 0.1 * 10) / 10;
document.getElementById("demo2").innerHTML = "0.2 + 0.1 = " + y;
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Numbers</h2>

<p>If you add two numbers, the result will be a number:</p>
<p id="demo"></p>

<script>
let x = 10;
let y = 20;
let z = x + y;
document.getElementById("demo").innerHTML =z;
 </script>

</body>
</html>
```

If you add two strings, the result will be a string concatenation:

Example:

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

<h2>JavaScript Numbers</h2>

<p> If you add two numeric strings, the result will be a concatenated string:</p>
<p id="demo"></p>

<script>
```

```
let x = "10";
let y = "20";
let z = x + y;
document.getElementById("demo").innerHTML =z;
 </script>

</body>
</html>
```

If you add a number and a string, the result will be a string concatenation:
Example:

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

<h2>JavaScript Numbers</h2>

<p> If you add a number and a numeric string, the result will be a concatenated string:</p>
<p id="demo"></p>

<script>
let x = 10;
let y = "20";
let z = x + y;
document.getElementById("demo").innerHTML =z;
 </script>

</body>
</html>
```

If you add a string and a number, the result will be a string concatenation:
Example:

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

<h2>JavaScript Numbers</h2>
```

```
<p> If you add a numeric string and a number, the result will be a concatenated string:</p>

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

<script>
let x = "10";
let y = 20;
let z = x + y;
document.getElementById("demo").innerHTML = "The result is: " + x + y;
 </script>

</body>
</html>
```

A common mistake is to expect this result to be 30:
Example:

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

<h2>JavaScript Numbers</h2>

<p> A common mistake is to expect this result to be 30:</p>

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

<script>
var x = 10;
var y = 20;
document.getElementById("demo").innerHTML = "The result is: " + x + y;
 </script>

</body>
</html>
```

A common mistake is to expect this result to be 102030:
Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript Numbers</h2>

<p> A common mistake is to expect this result to be 102030:</p>

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

<script>
let x = 10;
let y = 20;
let z = "30";
let result = x + y + z;
document.getElementById("demo").innerHTML = result ;
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Numbers</h2>
```

```
<p> JavaScript will try to convert strings to numbers when dividing::</p>

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

<script>
Let x = "100";
let y = "10";
let z = x / y;
document.getElementById("demo").innerHTML = z ;
 </script>

</body>
</html>
```

This will also work:

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

<h2>JavaScript Numbers</h2>

<p> JavaScript will try to convert strings to numbers when multiplying:</p>

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

<script>
Let x = "100";
let y = "10";
let z = x *y;
document.getElementById("demo").innerHTML = z ;
 </script>

</body>
</html>
```

And this will work:

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

```
<h2>JavaScript Numbers</h2>

<p> JavaScript will try to convert strings to numbers when subtracting::</p>

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

<script>
Let x = "100";
let y = "10";
let z = x - y;
document.getElementById("demo").innerHTML = z ;
 </script>

</body>
</html>
```

But this will not work:

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

<h2>JavaScript Numbers</h2>

<p> JavaScript will NOT convert strings to numbers when adding:</p>

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

<script>
Let x = "100";
let y = "10";
let z = x + y;
document.getElementById("demo").innerHTML = z ;
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Numbers</h2>

<p> A number divided by a non-numeric string becomes NaN (Not a Number):</p>

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

<script>
document.getElementById("demo").innerHTML = 100 / "Apple" ;
 </script>

</body>
</html>
```


However, if the string contains a numeric value , the result will be a number:

Example:

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

<h2>JavaScript Numbers</h2>

<p> A number divided by a numeric string becomes a number:</p>

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

<script>
document.getElementById("demo").innerHTML = 100 / "10";
```

```
</script>

</body>
</html>
```

You can use the global JavaScript function isNaN() to find out if a value is a not a number:
Example:

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

<h2>JavaScript Numbers</h2>

<p> You can use the global JavaScript function isNaN() to find out if a value is not a number:</p>

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

<script>
let x = 100 / "Apple";
document.getElementById("demo").innerHTML = isNaN(x);
</script>

</body>
</html>
```

Watch out for NaN. If you use NaN in a mathematical operation, the result will also be NaN:
Example:

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

<h2>JavaScript Numbers</h2>

<p> If you use NaN in a mathematical operation, the result will also be NaN:</p>
```

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

<script>
let x = NaN;
let y = 5;
document.getElementById("demo").innerHTML = x + y;
 </script>

</body>
</html>
```

Or the result might be a concatenation:
Example:

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

<h2>JavaScript Numbers</h2>

<p> If you use NaN in a mathematical operation, the result can be a concatenation:</p>

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

<script>
let x = NaN;
let y = "5";
document.getElementById("demo").innerHTML = x + y;
 </script>

</body>
</html>
```

NaN is a number: typeof NaN returns number:
Example:

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

```
<h2>JavaScript Numbers</h2>

<p> The typeof NaN is number:</p>

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

<script>
let x = NaN;
let y = "5";
document.getElementById("demo").innerHTML= typeof x;
 </script>

</body>
</html>
```

Infinity

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

Example:

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

<h2>JavaScript Numbers</h2>

<p>Infinity is returned if you calculate a number outside the largest possible number:</p>

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

<script>
let myNumber = 2;
let txt = " ";
while (myNumber != Infinity)
{ myNumber = myNumber *
myNumber;txt = txt + myNumber +
"<br>";
}
document.getElementById("demo").innerHTML= txt;
 </script>
```

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

Division by 0 (zero) also generates Infinity:
Example:

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

<h2>JavaScript Numbers</h2>

<p>Division by zero generates Infinity:</p>

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

<script>
let x =  2 / 0;
let y = -2 / 0;
document.getElementById("demo").innerHTML= x + “<br>” + y;
 </script>

</body>
</html>
```

Infinity is a number: typeof Infinity returns number.
Example:

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

<h2>JavaScript Numbers</h2>

<p>Infinity is a number:</p>

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

<script>
let x =  2 / 0;
let y = -2 / 0;
```

```
document.getElementById("demo").innerHTML= typeof Infinity;
 </script>

</body>
</html>
```

Hexadecimal

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

Example:

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

<h2>JavaScript Numbers</h2>

<p>Numeric constants, preceded by 0x, are interpreted as hexadecimal:</p>

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

<script>
let x = 0xFF;
document.getElementById("demo").innerHTML= “0xFF=” + x;
 </script>

</body>
</html>
```


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:

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

```
<h2>JavaScript Numbers</h2>

<p>The toString() method can output numbers from base 2 to 36:</p>

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

<script>
let myNumber = 32;
document.getElementById("demo").innerHTML= "32 = " + "<br>" +
" Decimal "  + myNumber.toString(10); + "<br>" +
" Hexadecimal " myNumber.toString(16); + "<br>" +
" Octal " myNumber.toString(8); + "<br>" +
" Binary " myNumber.toString(2);

 </script>

</body>
</html>
```

Numbers Can be 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:

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

<h2>JavaScript Numbers</h2>

<p>A number can be an object, but there is no need to create a number as an object.</p>

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

<script>
```

```
let x = 123;
let y = new Number(123);
document.getElementById("demo").innerHTML= typeof x + "<br>" + typeof y;
 </script>

</body>
</html>
```

Do not create Number objects. It slows down execution speed.
The new keyword complicates the code. This can produce some unexpected results:

When using the == operator, equal numbers are equal:
Example:

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

<h2>JavaScript Numbers</h2>

<p>Never create numbers as objects.</p>
<p>Numbers and objects cannot be safely compared.</p>

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

<script>
let x = 500;      // x is a number
let y = new Number(500);      // y is an object
document.getElementById("demo").innerHTML= (x == y);
 </script>

</body>
</html>
```

When using the === operator, equal numbers are not equal, because the === operator expects equality in both type and value.
Example:

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

```
<body>

<h2>JavaScript Numbers</h2>

<p>Never create numbers as objects.</p>
<p>Numbers and objects cannot be safely compared.</p>

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

<script>
let x = 500;       // x is a number
let y = new Number(500);          // y is an object
document.getElementById("demo").innerHTML= (x === y);
 </script>

</body>
</html>
```

Or even worse. Objects cannot be compared:

Example:

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

<h2>JavaScript Numbers</h2>

<p>Never create numbers as objects.</p>
<p>Numbers and objects cannot be compared.</p>

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

<script>
let x = new Number(500);     // x is an object
let y = new Number(500);     // y is an object
document.getElementById("demo").innerHTML= (x == y);
 </script>

</body>
</html>
```

Note the difference between (x==y) and (x===y).
Comparing two JavaScript objects will always return false.

JAVASCRIPT NUMBER METHODS

Number methods help you work with numbers.

Number Methods and Properties

- Primitive values (like 3.14 or 2014), cannot have properties and methods (because they are not objects).
- But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

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:

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

<h2>JavaScript Numbers Methods</h2>

<p>The toString() method converts a number to a string.</p>

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

<script>
let x = 123;
document.getElementById("demo").innerHTML=
  x.toString() + “<br>” +
   (123).toString() + “<br>” +
   (100 + 23).toString();
 </script>

</body>
```

```
</html>
```

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:

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

<h2>JavaScript Numbers Methods</h2>

<p>The toExponential() method returns a string, with the number rounded and written using exponential notation.</p>

<p>An optional parameter defines the number of digits behind the decimal point.</p>

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

<script>
let x = 9.656;
document.getElementById("demo").innerHTML=
x.toExponential( ) + “<br>” +
  x.toExponential(2) + “<br>” +
  x.toExponential(4) + “<br>” +
  x.toExponential(6);
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Numbers Methods</h2>

<p>The toFixed() method rounds a number to a given number of digits.</p>
<p>For working with money, toFixed(2) is perfect.</p>

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

<script>
let x = 9.656;
document.getElementById("demo").innerHTML=
x.toFixed(0) + “<br>” +
  x.toFixed(2) + “<br>” +
  x.toFixed(4) + “<br>” +
  x.toFixed(6);
 </script>

</body>
</html>
```

toFixed(2) is perfect for working with money.

The toPrecision() Method

toPrecision() returns a string, with a number written with a specified length:

Example:

Example:

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

<h2>JavaScript Numbers Methods</h2>

<p>The toPrecision() method returns a string, with a number written with a
specified length:</p>
```

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

<script>
let x = 9.656;
document.getElementById("demo").innerHTML=
x.toPrecision() + “<br>” +
  x.toPrecision(2) + “<br>” +
  x.toPrecision(4) + “<br>” +
  x.toPrecision(6);
 </script>

</body>
</html>
```

The valueOf() Method

valueOf() returns a number as a number.

Example:

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

<h2>JavaScript Numbers Methods</h2>

<p>The valueOf() method returns a number as a number:</p>

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

<script>
let x = 123;
document.getElementById("demo").innerHTML=
  x.valueOf() + “<br>” +
  (123).valueOf() + “<br>” +
  (100 + 23).valueOf();
 </script>

</body>
</html>
```

- 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 variables to numbers:

- The Number() method
- The parseInt() method
- The parseFloat() method

These methods are not **number** methods, but **global** JavaScript methods.

Global JavaScript Methods

JavaScript global methods can be used on all JavaScript data types.
These are the most relevant methods, when working with numbers:

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 an integer

The Number() Method

Number() can be used to convert JavaScript variables to numbers:
Example:

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

<h2>JavaScript Global Methods</h2>

<p>The Number() method converts variables to numbers:</p>

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

<script>
document.getElementById("demo").innerHTML=
```

```
 Number(true) + “<br>” +
 Number(false) + “<br>” +
 Number("10") + “<br>” +
 Number(" 10") + “<br>” +
 Number("10 ") + “<br>” +
 Number(" 10 ") + “<br>” +
 Number("10.33") + “<br>” +
 Number("10,33") + “<br>” +
 Number("10 33") + “<br>” +
 Number("John");
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Global Methods</h2>

<p>The Number() method can convert a date to a number:</p>

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

<script>
Let x= new Date("1970-01-01");
document.getElementById("demo").innerHTML= Number(x);
</script>

</body>
</html>
```

The Number() 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:

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

<h2>JavaScript Global Methods</h2>

<p>The Number() method can convert a date to a number:</p>

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

<script>
Let x= new Date("1970-01-02");
document.getElementById("demo").innerHTML= Number(x);
 </script>

</body>
</html>
```

Example:

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

<h2>JavaScript Global Methods</h2>

<p>The Number() method can convert a date to a number:</p>

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

<script>
Let x= new Date("2017-09-30");
document.getElementById("demo").innerHTML= Number(x);
 </script>

</body>
</html>
```

The parseInt() Method

parseInt() parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

Example:

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

<h2>JavaScript Global Methods</h2>
<h2>parseInt()</h2>
<p>The global JavaScript function parseInt() converts strings to numbers:</p>

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

<script>
document.getElementById("demo").innerHTML=
  parseInt("-10") + “<br>” +
  parseInt("-10.33") + “<br>” +
  parseInt("10") + “<br>” +
  parseInt("10.33") + “<br>” +
  parseInt("10 20 30") + “<br>” +
  parseInt("10 years") + “<br>” +
  parseInt("years 10");
 </script>

</body>
</html>
```


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:

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

<h2>JavaScript Global Methods</h2>
```

```
<p>The parseFloat() method converts strings to numbers:</p>

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

<script>
document.getElementById("demo").innerHTML=
  parseFloat("10") + “<br>” +
  parseFloat("10.33") + “<br>” +
  parseFloat("10 6") + “<br>” +
  parseFloat("10 years") + “<br>” +
  parseFloat("years 10");
 </script>

</body>
</html>
```

If the number cannot be converted, NaN (Not a Number) is returned.

Number Properties

Property	Description
MAX_VALUE	Returns the largest number possible in JavaScript
MIN_VALUE	Returns the smallest number possible in JavaScript
POSITIVE_INFINITY	Represents infinity (returned on overflow)
NEGATIVE_INFINITY	Represents negative infinity (returned on overflow)
NaN	Represents a "Not-a-Number" value

JavaScript MIN_VALUE and MAX_VALUE

MAX_VALUE returns the largest possible number in JavaScript.

Example:

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

<h2>JavaScript Number Properties</h2>

<p>MAX_VALUE returns the largest possible number in JavaScript.</p>

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

```
<script>
let x = Number.MAX_VALUE;
document.getElementById("demo").innerHTML= x;
 </script>

</body>
</html>
```

MIN_VALUE returns the lowest possible number in JavaScript.
Example:

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

<h2>JavaScript Number Properties</h2>

<p>MAX_VALUE returns the smallest number possible in JavaScript.</p>

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

<script>
let x = Number.MIN_VALUE;
document.getElementById("demo").innerHTML= x;
 </script>

</body>
</html>
```

JavaScript POSITIVE_INFINITY
Example:

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

<h2>JavaScript Number Properties</h2>

<p>POSITIVE_INFINITY</p>

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

```
<script>
let x = Number.POSITIVE_INFINITY;
document.getElementById("demo").innerHTML= x;
 </script>

</body>
</html>
```

POSITIVE_INFINITY is returned on overflow:
Example:

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

<h2>JavaScript Number Properties</h2>

<p> POSITIVE_INFINITY is returned on overflow:</p>

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

<script>
let x = 1 / 0;
document.getElementById("demo").innerHTML= x;
 </script>

</body>
</html>
```

JavaScript NEGATIVE_INFINITY
Example:

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

<h2>JavaScript Number Properties</h2>

<p> NEGATIVE_INFINITY</p>

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

```
<script>
let x = Number.NEGATIVE_INFINITY;
document.getElementById("demo").innerHTML= x;
 </script>

</body>
</html>
```

NEGATIVE_INFINITY is returned on overflow:
Example:

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

<h2>JavaScript Number Properties</h2>

<p> NEGATIVE_INFINITY</p>

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

<script>
let x = -1 / 0;
document.getElementById("demo").innerHTML= x;
 </script>

</body>
</html>
```

JavaScript NaN - Not a Number
Example:

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

<h2>JavaScript Number Properties</h2>

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

<script>
```

```
document.getElementById("demo").innerHTML= Number.NaN;
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Numbers</h2>

<p>A number divided by a non-numeric string becomes NaN (Not a Number):
</p>

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

<script>
document.getElementById("demo").innerHTML= 100 / "Apple";
 </script>

</body>
</html>
```


Number Properties Cannot be Used on Variables

- Number properties belongs to the JavaScript's number object wrapper called **Number**.
- These properties can only be accessed as Number.MAX_VALUE.
- Using *myNumber*.MAX_VALUE, where *myNumber* is a variable, expression, or value, will return undefined:

Example:

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

```
<h2>JavaScript Numbers Properties</h2>

<p>Using a Number property on a variable, expression, or value, will return undefined:</p>

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

<script>
let x = 6;
document.getElementById("demo").innerHTML= x.MAX_VALUE;
 </script>

</body>
</html>
```

JAVASCRIPT ARRAYS

JavaScript arrays are used to store multiple values in a single variable.

Example

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

<h2>JavaScript Arrays</h2>

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

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML= cars;
 </script>

</body>
</html>
```

It is a common practice to declare arrays with the const keyword.
Learn more about using const with arrays in the chapter: JS Const.

What is an Array?

An array is a special variable, which can hold more than one value at a time.
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, ...];
```

Example:

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

<h2>JavaScript Arrays</h2>

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

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML= cars;
 </script>

</body>
</html>
```

Spaces and line breaks are not important. A declaration can span multiple lines:

Example:

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

<h2>JavaScript Arrays</h2>

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

<script>
const cars =
["Saab",
 "Volvo",
 "BMW"
];
document.getElementById("demo").innerHTML= cars;
 </script>

</body>
</html>
```

You can also create an array, and then provide the elements:

Example:

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

<h2>JavaScript Arrays</h2>

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

<script>
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
document.getElementById("demo").innerHTML= cars;
 </script>
```

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

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example:

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

<h2>JavaScript Arrays</h2>

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

<script>
const cars = new Array("Saab", "Volvo", "BMW");
document.getElementById("demo").innerHTML= cars;
 </script>

</body>
</html>
```

The two examples above do exactly the same.
There is no need to use new Array().
For simplicity, readability and execution speed, use the first one (the array literal method).

Accessing Array Elements

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

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

<h2>JavaScript Arrays</h2>

<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>

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

```
<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML= cars[0];
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Arrays</h2>

<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>

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

<script>
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
document.getElementById("demo").innerHTML= cars;
 </script>

</body>
</html>
```

Access the Full Array

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

Example:

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

<h2>JavaScript Arrays</h2>

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

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Arrays</h2>

<p>Arrays use numbers to access its elements.</p>

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

<script>
const person = ["John", "Doe", 46];
document.getElementById("demo").innerHTML = person[0] ;
</script>
```

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

Objects use **names** to access its "members". In this example, person.firstName returns John:
Object:

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

<h2>JavaScript Objects</h2>
<p>JavaScript uses names to access object properties.</p>
<p id="demo"></p>

<script>
const person = {firstName:"John", lastName:"Doe", age:46};
document.getElementById("demo").innerHTML = person. firstName;
 </script>

</body>
</html>
```

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:

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

<h2>JavaScript Arrays</h2>
<p>The length property returns the length of an array.</p>

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

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.length;
 </script>

</body>
</html>
```

The length property is always one more than the highest array index.

Accessing the First Array Element

Example:

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

<h2>JavaScript Arrays</h2>

<p>JavaScript array elements are accesses using numeric indexes (starting from 0).</p>

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

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits[0];
```

```
</script>

</body>
</html>
```

Accessing the Last Array Element

Example:

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

<h2>JavaScript Arrays</h2>

<p>JavaScript array elements are accesses using numeric indexes (starting from 0).</p>

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

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits[fruits.length - 1];
</script>

</body>
</html>
```

Looping Array Elements

The safest way to loop through an array, is using a for loop:

Example:

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

<h2>JavaScript Arrays</h2>

<p>The best way to loop through an array is using a standard for loop:</p>

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

```
<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;

text = "<ul>";
for (let i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

You can also use the Array.forEach() function:
Example:

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

<h2>JavaScript Arrays</h2>

<p>Array.forEach() calls a function for each array element.</p>

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

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

let text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";

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

function myFunction(value) {
```

```
  text += "<li>" + value + "</li>";
}
</script>

</body>
</html>
```

Adding Array Elements

The easiest way to add a new element to an array is using the push() method:

Example:

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

<h2>JavaScript Arrays</h2>

<p>The push method appends a new element to an array.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
const fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = fruits;

function myFunction( )
  {fruits.push("Lemon");
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

New element can also be added to an array using the length property:

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript Arrays</h2>

<p>The length property provides an easy way to append new elements to an array without using the push() method.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
const fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = fruits;

function myFunction( )
  { fruits[fruits.length] =
  "Lemon";
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```


WARNING !
Adding elements with high indexes can create undefined "holes" in an array:

Example:

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

<h2>JavaScript Arrays</h2>

<p>Adding elements with high indexes can create undefined "holes" in an array.</p>

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

```
<script>
const fruits = ["Banana", "Orange", "Apple"];
fruits[6] = "Lemon";

let fLen = fruits.length;
let text = " ";
for (i = 0; i < fLen; i++)
  { text += fruits[i] +
  "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Arrays</h2>

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

<script>
const person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
document.getElementById("demo").innerHTML =
person[0] + " " + person.length;
</script>
```

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

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:

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

<h2>JavaScript Arrays</h2>

<p>If you use a named index when accessing an array, JavaScript will redefine the array to a standard object, and some array methods and properties will produce undefined or incorrect results.</p>

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

<script>
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
document.getElementById("demo").innerHTML =
person[0] + " " + person.length;
</script>

</body>
</html>
```


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:

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

<h2>JavaScript Arrays</h2>

<p>Avoid using new Array(). Use [] instead.</p>

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

<script>
// const points = new Array(40, 100, 1, 5, 25, 10);
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points[0];
</script>

</body>
</html>
```

The new keyword can produce some unexpected results:

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

```
<h2>JavaScript Arrays</h2>

<p>Create an Array with three elements.</p>

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

<script>
var points = new Array(40, 100, 1);
document.getElementById("demo").innerHTML = points;
</script>

</body>
</html>
```

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

<h2>JavaScript Arrays</h2>

<p>Create an Array with two elements.</p>

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

<script>
var points = new Array(40, 100);
document.getElementById("demo").innerHTML = points;
</script>

</body>
</html>
```

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

<h2>JavaScript Arrays</h2>

<p>Avoid using new Array().</p>
```

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

<script>
var points = new Array(40);
document.getElementById("demo").innerHTML = points;
</script>

</body>
</html>
```

A Common Error

```
const points = [40];
```

is not the same as:

```
const points = New Array(40);

<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Arrays</h2>

<p>Avoid using new element.</p>

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

<script>
var points = (40);
document.getElementById("demo").innerHTML = points;
</script>

</body>
</html>

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

```
<h2>JavaScript Arrays</h2>

<p>Avoid using new Array().</p>

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

<script>
var points = new Array(40);
document.getElementById("demo").innerHTML = points(0);
</script>

</body>
</html>
```

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":

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

<h2>JavaScript Arrays</h2>

<p>The typeof operator, when used on an array, returns object:</p>

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

<script>
const fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = typeof fruits;
</script>

</body>
</html>
```

The typeof operator returns object because a JavaScript array is an object.

Solution 1:

To solve this problem ECMAScript 5 defines a new method Array.isArray():

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

<h2>JavaScript Arrays</h2>

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

<script>
const fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = Array.isArray(fruits);
</script>

</body>
</html>
```

Solution 2:

The instanceof operator returns true if an object is created by a given constructor:

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

<h2>JavaScript Arrays</h2>

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

<script>
var fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = fruits instanceof Array;
</script>

</body>
</html>
```


JAVASCRIPT ARRAY METHODS

Converting Arrays to Strings

The JavaScript method toString() converts an array to a string of (comma separated) array values.
Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>toString()</h2>
<p>The toString() method returns an array as a comma separated string:</p>

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

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

</body>
</html>
```

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:

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

<h2>JavaScript Arrays Methods</h2>
<h2>join</h2>
<p>The join() method joins array elements into a string.:</p>
<p>It this example we have used " * " as a separator between the elements:
</p>

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

<script>
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
</script>

</body>
</html>
```

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.

Popping

The pop() method removes the last element from an array:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>pop</h2>
<p>The pop() method removes the last element from an array.</p>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
fruits.pop();
document.getElementById("demo2").innerHTML = fruits;
</script>

</body>
</html>
```

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

Example;

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

<h2>JavaScript Arrays Methods</h2>
<h2>pop()</h2>
<p>The pop() method removes the last element from an array.</p>
<p>The return value of the pop() method is the removed item.</p>

<p id="demo1"></p>
<p id="demo2"></p>
<p id="demo3"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
document.getElementById("demo2").innerHTML = fruits.pop();
document.getElementById("demo3").innerHTML = fruits;
</script>

</body>
</html>
```

Pushing

The push() method adds a new element to an array (at the end):

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>push()</h2>
<p>The push() method appends a new element to an array.</p>

<button onclick="myFunction()">Try it</button>
<p id="demo"></p>

<script>
```

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

function myFunction()
 {fruits.push("Kiwi");
 document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

The push() method returns the new array length:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>push()</h2>
<p>The push() method returns the new array length.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;

function myFunction()
 { document.getElementById("demo1").innerHTML =
 fruits.push("Kiwi");document.getElementById("demo2").innerHTML =
 fruits;
}
</script>
</body>
```

```
</html>
```

Shifting Elements

Shifting is equivalent to popping, working on the first element instead of the last.

The shift() method removes the first array element and "shifts" all other elements to a lower index.

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>shift()</h2>
<p>The shift() method removes the first element of an array (and "shifts" all other elements to the left):</p>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
fruits.shift();
document.getElementById("demo2").innerHTML = fruits;
</script>

</body>
</html>
```


The shift() method returns the value that was "shifted out":

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>shift()</h2>
<p>The shift() method returns the element that was shifted out.</p>
```

```
<p id="demo1"></p>
<p id="demo2"></p>
<p id="demo3"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
document.getElementById("demo2").innerHTML = fruits.shift();
document.getElementById("demo2").innerHTML = fruits;
</script>

</body>
</html>
```

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>unshift()</h2>
<p>The unshift() method adds new elements to the beginning of an array.</p>

<button onclick="myFunction()">Try it</button>

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

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

function myFunction()
   { fruits.unshift("Lemon");
  document.getElementById("demo").innerHTML = fruits;
}
</script>
```

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

The unshift() method returns the new array length.
Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>unshift()</h2>
<p>The unshift() method returns the length of the new array:.</p>

<p id="demo1"></p>
<p id="demo2"></p>
<p id="demo3"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
document.getElementById("demo2").innerHTML = fruits.unshift("Lemon");
document.getElementById("demo3").innerHTML = fruits;
</script>

</body>
</html>
```


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:

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

```
<h2>JavaScript Arrays Methods</h2>
<h2>unshift()</h2>
<p>Array elements are accessed using their index number.</p>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
fruits[0] = "Kiwi";
document.getElementById("demo2").innerHTML = fruits;
</script>

</body>
</html>
```

The length property provides an easy way to append a new element to an array:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<p>The length property provides an easy way to append new elements to an array without using the push() method.</p>

<button onclick="myFunction()">Try it</button>
<p id="demo"></p>

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

function myFunction()
   { fruits[fruits.length] =
   "Kiwi";
  document.getElementById("demo").innerHTML = fruits;
}
```

```
</script>

</body>
</html>
```

Deleting Elements

Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator delete:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<p>Deleting elements leaves undefined holes in an array.</p>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML =
"The first fruit is: " + fruits[0];

delete fruits[0];

document.getElementById("demo2").innerHTML =
"The first fruit is: " + fruits[0];

</script>

</body>
</html>
```

Using **delete** may leave undefined holes in the array. Use pop() or shift() instead.

Splicing an Array

The splice() method can be used to add new items to an array:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>splice()</h2>
<p>The splice() method adds new elements to an array..</p>

<button onclick="myFunction()">Try it</button>
<p id="demo1"></p>
<p id="demo2"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango" ];
document.getElementById("demo1").innerHTML = "Original Array:<br>" +
fruits;

function myFunction()
   { fruits.splice(2, 0, "Lemon",
   "Kiwi");
  document.getElementById("demo").innerHTML = "New Array:<br>" + fruits;
}

</script>

</body>
</html>
```


- 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:

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

```
<h2>JavaScript Arrays Methods</h2>
<h2>splice()</h2>
<p>The splice() method adds new elements to an array, and returns an array with the deleted elements (if any).</p>

<button onclick="myFunction()">Try it</button>

<p id="demo1"></p>
<p id="demo2"></p>
<p id="demo3"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango" ];
document.getElementById("demo1").innerHTML = "Original Array:<br>" +
fruits;

function myFunction() {
  Let Removed = fruits.splice(2, 2, "Lemon", "Kiwi");
  document.getElementById("demo2").innerHTML = "New Array:<br>" + fruits;
  document.getElementById("demo2").innerHTML = "Remove Itme:<br>" +
Removed;
}
</script>

</body>
</html>
```


Using splice() to Remove Elements

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>splice()</h2>
<p>The splice() methods can be used to remove array elements.</p>
```

```
<button onclick="myFunction()">Try it</button>

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

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango" ];
document.getElementById("demo").innerHTML = fruits;
function myFunction() {
  fruits.splice(0,                                    1);
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

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.

Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays:
Example (Merging Two Arrays):

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

<h2>JavaScript Arrays Methods</h2>
<h2>concat()</h2>
<p>The concat() method is used to merge (concatenate) arrays:</p>

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

<script>
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];
const myChildren = myGirls.concat(myBoys);
```

```
document.getElementById("demo").innerHTML = myChildern;
</script>

</body>
</html>
```

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):

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

<h2>JavaScript Arrays Methods</h2>
<h2>concat()</h2>
<p>The concat() method is used to merge (concatenate) arrays:</p>

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

<script>
const array1 = ["Cecilie", "Lone"];
const array2 = ["Emil", "Tobias", "Linus"];
const array3 = ["Robin", "Morgan"];
const myChildren = array1.concat(array2, array3);
document.getElementById("demo").innerHTML = myChildern;
</script>

</body>
</html>
```

The concat() method can also take strings as arguments:

Example (Merging an Array with Values):

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

<h2>JavaScript Arrays Methods</h2>
```

```
<h2>concat()</h2>
<p>The concat() method can also merge string values to arrays:</p>

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

<script>
const array1 = ["Emil", "Tobias", "Linus"];
const myChildren = array1.concat("Peter");
document.getElementById("demo").innerHTML = myChildern;
</script>

</body>
</html>
```

Slicing an Array

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:

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

<h2>JavaScript Arrays Methods</h2>
<h2>slice()</h2>
<p>This example slices out a part of an array starting from array element 1 ("Orange"):</p>

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

<script>
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1);
document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;
</script>

</body>
</html>
```

The slice() method creates a new array. It does not remove any elements from the source array.

This example slices out a part of an array starting from array element 3 ("Apple"):
Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>slice()</h2>
<p>This example slices out a part of an array starting from array element 3 ("Apple")</p>

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

<script>
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(3);
document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;
</script>

</body>
</html>
```


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:

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

<h2>JavaScript Arrays Methods</h2>
<h2>slice()</h2>
<p>When the slice() method is given two arguments, it selects array elements from the start argument, and up to (but not included) the end argument:</p>

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

```
<script>
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3);
document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;
</script>

</body>
</html>
```

If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

Example:

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

<h2>JavaScript Arrays Methods</h2>
<h2>slice()</h2>
<p>This example slices out a part of an array starting from array element 2 ("Lemon"):</p>

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

<script>
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(2);
document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript Arrays Methods</h2>
<h2>toString()</h2>
<p>The toString() method returns an array as a comma separated string:</p>

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

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

</body>
</html>
```

Example:

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

<h2>JavaScript Arrays Methods</h2>
<p>JavaScript automatically converts an array to a comma separated string
when a simple value is expected:</p>

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

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
</script>

</body>
</html>
```

All JavaScript objects have a toString() method.

JAVASCRIPT SORTING ARRAYS

Sorting an Array

The sort() method sorts an array alphabetically:

Example:

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

<h2>JavaScript Array Sort</h2>
<p>The sort() method sorts an array alphabetically.</p>

<button onclick="myFunction()">Try it</button>

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

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

function myFunction()
   {fruits.sort();
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

Reversing an Array

The reverse() method reverses the elements in an array.

You can use it to sort an array in descending order:

Example:

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

<h2>JavaScript Array Sort Reverse</h2>
```

```
<p>The reverse() method reverses the elements in an array.</p>
<p>By combining sort() and reverse() you can sort an array in descending order.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
// Create and display an array:
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;

function myFunction() {
  // First sort the array
   fruits.sort();
   // Then reverse it:
  fruits.reverse();
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```


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**:

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

<h2>JavaScript Array Sort </h2>
```

```
<p> Click the button to sort the array in ascending order.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction()
{ points.sort(function(a, b){return a - b});
  document.getElementById("demo").innerHTML = points
}
</script>

</body>
</html>
```

Use the same trick to sort an array descending:

Example:

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

<h2>JavaScript Array Sort</h2>
<p> Click the button to sort the array in descending order.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction()
  { points.sort(function(a, b){return b - a});
  document.getElementById("demo").innerHTML = points;
```

```
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Array Sort</h2>
<p> Click the buttons to sort the array alphabetically or numerically.</p>

<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>

</body>
</html>
```

Sorting an Array in Random Order

Example:

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

<h2>JavaScript Array Sort</h2>
<p> Click the button (again and again) to sort the array in random order.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction() {
  points.sort(function(a, b){return 0.5 - Math.random()});
  document.getElementById("demo").innerHTML = points;
```

```
}
</script>

</body>
</html>
```

The Fisher Yates Method

- The above example, *array*.sort(), is not accurate, it will favor some numbers over the 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:

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

<h2>JavaScript Array Sort</h2>
<h3>The Fisher Yates Method</h3>
<p> Click the button (again and again) to sort the array in random order.</p>

<button onclick="myFunction()">Try it</button>
<p id="demo"></p>

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction() {
  for (let i = points.length -1; i > 0; i--)
 {let j = Math.floor(Math.random() * i)
 let k = points[i]
 points[i] = points[j]
 points[j] = k
}
  document.getElementById("demo").innerHTML = points;
}
</script>
```

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

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:

Example:

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

<h2>JavaScript Array Sort</h2>
<p> The lowest number is <span id=”demo"></span>.</p>

<script>
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
document.getElementById("demo").innerHTML = points[0];
</script>

</body>
</html>
```


Sorting descending:

Example:

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

<h2>JavaScript Array Sort</h2>
<p> The highest number is <span id=”demo"></span>.</p>

<script>
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
document.getElementById("demo").innerHTML = points[0];
</script>
```

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

Sorting a whole array is a very inefficient method if you only want to find the highest (or lowest) value.

Using Math.max() on an Array

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

Example:

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

<h2>JavaScript Array Sort</h2>
<p> The highest number is <span id=”demo"></span>.</p>

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = myArrayMax(points);

function myArrayMax(arr)
 { return Math.max.apply(null,
 arr);
}
</script>

</body>
</html>
```

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:

Example:

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

<h2>JavaScript Array Sort</h2>
<p> The lowest number is <span id=”demo"></span>.</p>
```

```
<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = myArrayMin(points);

function myArrayMin(arr)
 { return Math.min.apply(null,
 arr);
}
</script>

</body>
</html>
```

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

My Min / Max JavaScript Methods

The fastest solution is to use a "home made" method.

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

Example (Find Max):

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

<h2>JavaScript Array Sort</h2>
<p> The highest number is <span id=”demo"></span>.</p>

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = myArrayMax(points);

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

```
  return max;
}
</script>
</body>
</html>
```

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

Example (Find Min):

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

<h2>JavaScript Array Sort</h2>
<p> The lowest number is <span id=”demo"></span>.</p>

<script>
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = myArrayMin(points);

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

</body>
</html>
```


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:

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

<h2>JavaScript Array Sort</h2>
<p> Click the buttons to sort car objects on age.</p>

<button onclick="myFunction()">Sort</button>
<p id=”demo"></p>

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

displayCars();

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

function displayCars()
{ document.getElementById("demo").innerHTML
=cars[0].type + " " + cars[0].year + "<br>" +
cars[1].type + " " + cars[1].year + "<br>" +
```

```
  cars[2].type + " " + cars[2].year;
}
</script>

</body>
</html>
```

Comparing string properties is a little more complex:
Example:

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

<h2>JavaScript Array Sort</h2>
<p> Click the buttons to sort car objects on type.</p>

<button onclick="myFunction()">Sort</button>
<p id=”demo"></p>

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

displayCars();

function      myfunction()
  {    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;
});
displayCars();
}
```

```
function displayCars()
{ document.getElementById("demo").innerHTML
=cars[0].type + " " + cars[0].year + "<br>" +
cars[1].type + " " + cars[1].year + "<br>" +
cars[2].type + " " + cars[2].year;
}
</script>

</body>
</html>
```

JAVASCRIPT ARRAY ITERATION

Array iteration methods operate on every array item.

Array.forEach()

The forEach() method calls a function (a callback function) once for each array element.

Example:

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

<h2>JavaScript Array.forEach</h2>
<p> Calls a function once for each array element.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);
document.getElementById("demo").innerHTML = txt;

function myFunction(value, index, array)
  {txt += value + "<br>";
```

```
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Array.forEach</h2>
<p> Calls a function once for each array element.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);
document.getElementById("demo").innerHTML = txt;

function myFunction(value)
  {txt += value + "<br>";
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Array.map()</h2>
<p> Creates a new array by performing a function on each array element.</p>

<p id=”demo"></p>

<script>
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

document.getElementById("demo").innerHTML = numbers2;

function myFunction(value, index, array) {
  return value * 2;
}
</script>

</body>
</html>
```


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:

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

```
<body>

<h2>JavaScript Array.map()</h2>
<p> Creates a new array by performing a function on each array element.</p>

<p id=”demo"></p>

<script>
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

document.getElementById("demo").innerHTML = numbers2;

function myFunction(value) {
  return value * 2;
}
</script>

</body>
</html>
```

Array.filter()

The filter() method creates a new array with array elements that passes a test.
This example creates a new array from elements with a value larger than 18:
Example:

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

<h2>JavaScript Array.filter()</h2>
<p> Creates a new array with all array elements that passes a test.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

document.getElementById("demo").innerHTML = over18;
```

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

</body>
</html>
```

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:

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

<h2>JavaScript Array.filter()</h2>
<p> Creates a new array with all array elements that passes a test.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

document.getElementById("demo").innerHTML = over18;

function myFunction(value)
  {return value > 18;
}
</script>

</body>
</html>
```

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 reduce Right().

The reduce() method does not reduce the original array.

This example finds the sum of all numbers in an array:

Example:

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

<h2>JavaScript Array.reduce()</h2>
<p> This example finds the sum of all numbers in an array:</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);

document.getElementById("demo").innerHTML = "The sum is " + sum;

function myFunction(total, value, index, array)
 {return total + value;
}
</script>

</body>
</html>
```


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:

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

<h2>JavaScript Array.reduce()</h2>
<p> This example finds the sum of all numbers in an array:</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);

document.getElementById("demo").innerHTML = "The sum is " + sum;

function myFunction(total, value) {
  return total + value;
}
</script>

</body>
</html>
```

The reduce() method can accept an initial value:

Example:

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

<h2>JavaScript Array.reduce()</h2>
<p> This example finds the sum of all numbers in an array:</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction, 100);

document.getElementById("demo").innerHTML = "The sum is " + sum;
```

```
function myFunction(total, value)
  {return total + value;
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Array.reduceRight()</h2>
<p> This example finds the sum of all numbers in an array:</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
let sum = numbers1.reduceRight(myFunction);

document.getElementById("demo").innerHTML = "The sum is " + sum;

function myFunction(total, value, index, array)
  {return total + value;
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Array.reduceRight()</h2>
<p> This example finds the sum of all numbers in an array:</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
let sum = numbers1.reduceRight(myFunction);

document.getElementById("demo").innerHTML = "The sum is " + sum;

function myFunction(total, value) {
  return total + value;
}
</script>

</body>
</html>
```

Array.every()

The every() method check if all array values pass a test.

This example check if all array values are larger than 18:

Example:

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

<h2>JavaScript Array.every()</h2>
```

```
<p> The every() method checks if all array values pass a test.</p>

<p id=”demo"></p>

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

document.getElementById("demo").innerHTML = " All over 18 is " + allOver18;

function myFunction(value, index, array)
 {return value > 18;
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Array.every()</h2>
<p> The every() method checks if all array values pass a test.</p>

<p id=”demo"></p>

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

document.getElementById("demo").innerHTML = " All over 18 is " + allOver18;
```

```
function myFunction(value)
 {return value > 18;
}
</script>

</body>
</html>
```

Array.some()

The some() method check if some array values pass a test.

This example check if some array values are larger than 18:

Example:

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

<h2>JavaScript Array.some()</h2>
<p> The some() method checks if some array values pass a test.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];
let someOver18 = numbers.some(myFunction);

document.getElementById("demo").innerHTML = " Some over 18 is" +someOver18;

function myFunction(value, index, array)
 {return value > 18;
}
</script>

</body>
</html>
```

Note that the function takes 3 arguments:

- The item value
- The item index

- The array itself

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":

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

<h2>JavaScript Array.indexOf()</h2>

<p id=”demo"></p>

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

document.getElementById("demo").innerHTML = "Apple is found in position "+
position;
</script>

</body>
</html>
```

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.

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":

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

<h2>JavaScript Array.lastIndexOf()</h2>

<p id=”demo"></p>

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

document.getElementById("demo").innerHTML = "Apple is found in position "+
position;
</script>

</body>
</html>
```

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

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:

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

<h2>Array includes()</h2>
```

```
<p>Check if the fruit array contains "Mango":</p>

<p id=”demo"></p>

<p><strong>Note:</strong> The includes method is not supported in Edge 13 (and earlier versions).</p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.includes("Mango");
</script>

</body>
</html>
```

Syntax:
array.includes(*search-item*)

Array.includes() allows to check for NaN values. Unlike Array.indexOf().
Array.includes() is not supported in Internet Explorer and Edge 12/13.
The first browser versions with full support are:

				
Chrome 47	Edge 14	Firefox 43	Safari 9	Opera 34
Des 2015	Aug 2016	Des 2015	Oct 2015	Des 2015

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:

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

<h2>JavaScript Array.find()</h2>
<p id=”demo"></p>

<script>
const numbers = [4, 9, 16, 25, 29];
```

```
let first = numbers.find(myFunction);

document.getElementById("demo").innerHTML = "First number over 18 is " +
first;

function myFunction(value, index, array)
  {return value > 18;
}
</script>

</body>
</html>
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.find() is not supported in older browsers. The first browser versions with full support are:

				
Chrome 47	Edge 14	Firefox 43	Safari 9	Opera 34
Des 2015	Aug 2016	Des 2015	Oct 2015	Des 2015

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:

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

<h2>JavaScript Array.findIndex()</h2>
<p id=”demo"></p>

<script>
const numbers = [4, 9, 16, 25, 29];
```

```
document.getElementById("demo").innerHTML = "First number over 18 has Index " + numbers.findIndex(myFunction);

function myFunction(value, index, array)
 {return value > 18;
}
</script>

</body>
</html>
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.findIndex() is not supported in older browsers. The first browser versions with full support are:

Chrome 47	Edge 14	Firefox 43	Safari 9	Opera 34
Des 2015	Aug 2016	Des 2015	Oct 2015	Des 2015

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:

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

<h2>JavaScript Arrays</h2>
<p>The Array.from() method returns an Array object from any object with a length property or any iterable object.</p>

<p id=”demo"></p>

<script>
const myArr = Array.from("ABCDEFG");
document.getElementById("demo").innerHTML = myarr;
```

```
</script>

<p>The Array.from() method is not supported in Internet Explorer.</p>

</body>
</html>
```

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:

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

<h2>JavaScript Arrays</h2>
<p>The Array.keys() method returns an Array Iterator object with the keys of the array.</p>

<p id=”demo"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const keys = fruits.keys();

let texr =” “;
for (let x of keys)
  { text += x + "<br>";
}

document.getElementById("demo").innerHTML = text;
</script>

<p> Array.keys() is not supported in Internet Explorer.</p>

</body>
</html>
```

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:

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

<h2>JavaScript const</h2>
<p id=”demo"></p>

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>

</body>
</html>
```

Cannot be Reassigned

An array declared with const cannot be reassigned:

Example:

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

<h2>JavaScript const</h2>

<p>You can NOT reassign a constant array:</p>

<p id=”demo"></p>

<script>
Try {
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"];
}
```

```
catch (err)
{ document.getElementById("demo").innerHTML =
err;
}
</script>

</body>
</html>
```

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:

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

<h2>JavaScript const</h2>

<p>Declaring a constant array does NOT make the elements unchangeble:</p>

<p id=”demo"></p>

<script>
// Create an array:
const cars = ["Saab", "Volvo", "BMW"];

// Change an element:
cars[0] = "Toyota";

// Add an element:
cars.push("Audi");

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

```
</script>

</body>
</html>
```

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 arrays 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:

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

<h2>JavaScript Hoisting</h2>

<p id=”demo"></p>

<script>
cars = ["Saab", "Volvo", "BMW"];
var cars;
document.getElementById("demo").innerHTML = cars[0};
```

```
</script>

</body>
</html>
```

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:

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

<h2>Declaring an Array Using const</h2>

<p id=”demo"></p>

<script>
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"
document.getElementById("demo").innerHTML = cars[0};
</script>

</body>
</html>
```


An array declared with var does not have block scope:

Example;

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

<h2>Declaring an Array Using const</h2>
```

```
<p id=”demo"></p>

<script>
var cars = ["Saab", "Volvo", "BMW"];
// Here cars[0] is "Tesla"
{
  var cars = ["Toyota", "Volvo", "BMW"];
  // Here cars[0] is "Toyota"
}
// Here cars[0] is "Toyota"
document.getElementById("demo").innerHTML = cars[0};
</script>

</body>
</html>
```

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
}
```

JAVASCRIPT DATE OBJECTS

JavaScript **Date Object** lets us work with dates:

Thu Sep 16 2021 15:44:47 GMT+0530 (India Standard Time)

Year: 2021 Month: 9 Day: 16 Hours: 15 Minutes: 44 Seconds: 47

Example:

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

<h2>JavaScript new Date</h2>

<p id=”demo"></p>
```

```
<script>
const d = new Date();
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

JavaScript Date Output

By default, JavaScript will use the browser's time zone and display a date as a full text string:

Thu Sep 16 2021 15:44:47 GMT+0530 (India Standard Time)

You will learn much more about how to display dates, later in this tutorial.

Creating Date Objects

Date objects are created with the new Date() constructor.

There are **4 ways** to create a new date object:

```
new Date()
new Date(year, month, day, hours, minutes, seconds, milliseconds)
new Date(milliseconds)
new Date(date string)
```

new Date()

new Date() creates a new date object with the **current date and time**:

Example:

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

<h2>JavaScript new Date()</h2>
<p>Using new Date(), creates a new date object with the current date and time:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d;
</script>
```

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

Date objects are static. The computer time is ticking, but date objects are not.

new Date(*year, month, ...*)
new Date(*year, month, ...*) creates a new date object with a **specified date and time**.
7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):
Example:

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

<h2>JavaScript new Date()</h2>

<p>Using new Date(7 numbers), creates a new date object with the specified date and time:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11, 24, 10, 33, 30, 0);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

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:

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

```
<body>

<h2>JavaScript new Date()</h2>

<p>JavaScript counts months from 0 to 11.</p>
<p>Specifying a month higher than 11, will not result in an error but add the overflow to the next year:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11, 24, 10, 33, 30, 0);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Specifying a day higher than max, will not result in an error but add the overflow to the next month:
Specifying:

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

<h2>JavaScript new Date()</h2>

<p>JavaScript counts months from 0 to 11.</p>
<p>Specifying a day higher than max, will not result in an error but add the overflow to the next month:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 05, 35, 10, 33, 30, 0);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Using 6, 4, 3, or 2 Numbers

6 numbers specify year, month, day, hour, minute, second:

Example:

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

<h2>JavaScript new Date()</h2>
<p>6 numbers specify year, month, day, hour, minute and second:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11, 24, 10, 33, 30);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

5 numbers specify year, month, day, hour, and minute:

Example:

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

<h2>JavaScript new Date()</h2>
<p>5 numbers specify year, month, day, hour, and minute:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11, 24, 10, 33);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

4 numbers specify year, month, day, and hour:

Example:

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

<h2>JavaScript new Date()</h2>
<p>4 numbers specify year, month, day, and hour:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11, 24, 10);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

3 numbers specify year, month, and day:

Example:

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

<h2>JavaScript new Date()</h2>
<p>3 numbers specify year, month, day:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11, 24);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

2 numbers specify year and month:

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript new Date()</h2>
<p>2 numbers specify year, month:</p>

<p id=”demo"></p>

<script>
const d = new Date(2018, 11);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

Example:

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

<h2>JavaScript new Date()</h2>
<p> One parameter will be interpreted as new Date(milliseconds).</p>

<p id=”demo"></p>

<script>
const d = new Date(2018);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Previous Century

One and two digit years will be interpreted as 19xx:

Example:

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

<h2>JavaScript new Date()</h2>
<p> Two digit years will be interpreted as 19xx:</p>

<p id=”demo"></p>

<script>
const d = new Date(99, 11, 24);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Example:

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

<h2>JavaScript new Date()</h2>
<p> One digit years will be interpreted as 19xx:</p>

<p id=”demo"></p>

<script>
const d = new Date(9, 11, 24);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

new Date(dateString)

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

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript new Date()</h2>
<p> A Date object can be created with a specified date and time:</p>

<p id=”demo"></p>

<script>
const d = new Date("October 13, 2014 11:13:00");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Date strings are described in the next chapter.

JavaScript Stores Dates as Milliseconds

- JavaScript stores dates as number of milliseconds since January 01, 1970, 00:00:00 UTC (Universal Time Coordinated).
- Zero time is January 01, 1970 00:00:00 UTC.
- Now the time is: **1631787287940** milliseconds past January 01, 1970

*new Date(*milliseconds*)*

new Date(*milliseconds*) creates a new date object as **zero time plus milliseconds**:

Example:

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

<h2>JavaScript new Date()</h2>
<p>Using new Date(milliseconds), creates a new date object as January 1, 1970, 00:00:00 Universal Time (UTC) plus the milliseconds:</p>

<p id=”demo"></p>

<script>
```

```
const d = new Date(0);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

01 January 1970 **plus** 100 000 000 000 milliseconds is approximately 03 March 1973:

Example:

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

<h2>JavaScript new Date()</h2>
<p> 100000000000 milliseconds from Jan 1, 1970, is approximately Mar 3, 1973:</p>

<p id=”demo"></p>

<script>
const d = new Date(100000000000);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

January 01 1970 **minus** 100 000 000 000 milliseconds is approximately October 31 1966:

Example:

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

<h2>JavaScript new Date()</h2>
<p> 100000000000 milliseconds from Jan 1, 1970, is approximately Oct 31, 1966:</p>
```

```
<p id=”demo"></p>

<script>
const d = new Date(-100000000000);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Example:

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

<h2>JavaScript new Date()</h2>
<p> Using new Date(milliseconds), creates a new date object as January 1, 1970, 00:00:00 Universal Time (UTC) plus the milliseconds:</p>

<p id=”demo"></p>

<script>
const d = new Date(86400000);
document.getElementById("demo").innerHTML = d;
</script>

<p>One day (24 hours) is 86,400,000 milliseconds.</p>

</body>
</html>
```

One day (24 hours) is 86 400 000 milliseconds.

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.

Date methods and time zones are covered in the next chapters.

Displaying Dates

JavaScript will (by default) output dates in full text string format:

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

<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date( );
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

When you display a date object in HTML, it is automatically converted to a string, with the toString() method.

Example:

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

<h2>JavaScript toString()</h2>
<p>The toString() method converts a date to a string:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.toString();
</script>

</body>
</html>
```

The toUTCString() method converts a date to a UTC string (a date display standard).
Example:

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

<h2>JavaScript Date()</h2>
<p>The toUTCString() method converts a date to a UTC string (a date display standard):</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.toUTCString();
</script>

</body>
</html>
```

The toDateString() method converts a date to a more readable format:
Example:

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

<h2>JavaScript toDateString()</h2>
<p> The toDateString() method converts a date to a date string:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.toDateString();
</script>

</body>
</html>
```

The toISOString() method converts a Date object to a string, using the ISO standard format:
Example:

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

<h2>JavaScript toISOString()</h2>
<p>The toISOString() method converts a date to a date string, using the ISO standard format:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.toISOString();
</script>

</body>
</html>
```

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:
Wed Mar 25 2015 05:30:00 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)

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

<h2>JavaScript ISO Dates</h2>

<p id=”demo"></p>

<script>
const d = new Date("2015-03-25");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript ISO Dates</h2>

<p id=”demo"></p>

<script>
const d = new Date("2015-03");
document.getElementById("demo").innerHTML = d;
```

```
</script>

</body>
</html>
```

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:

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

<h2>JavaScript ISO Dates</h2>

<p id=”demo"></p>

<script>
const d = new Date("2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript ISO Dates</h2>
<p>Separate date and time with a capital T.</p>
```

```
<p>Indicate UTC time with a capital Z.</p>

<p id=”demo"></p>

<script>
const d = new Date("2015-03-25T12:00:00Z");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript ISO Dates</h2>
<p>Modify the time relative to UTC by adding +HH:MM or subtraction -HH:MM to the time.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
new Date("2015-03-25T12:00:00-06:30");
</script>

</body>
</html>
```

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:

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

<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date("03/25/2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```


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:

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

<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date("Mar 25 2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Month and day can be in any order:

Example:

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

<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date("25 Mar 2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

And, month can be written in full (January), or abbreviated (Jan):
Example:

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

<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date("January 25 2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Example:

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

<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date("Jan 25 2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Commas are ignored. Names are case insensitive:
Example:

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

```
<h2>JavaScript new Date()</h2>

<p id=”demo"></p>

<script>
const d = new Date("JANUARY, 25, 2015");
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript Date.parse()</h2>
<p>Date.parse() returns the number of milliseconds between the date and January 1, 1970:</p>

<p id=”demo"></p>

<script>
let msec = Date.parse("March 21, 2012");
document.getElementById("demo").innerHTML = msec;
</script>

</body>
</html>
```

You can then use the number of milliseconds to **convert it to a date** object:

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript Date.parse()</h2>
<p>Date.parse(string) returns milliseconds.</p>
<p>You can use the return value to convert the string to a date object:</p>

<p id=”demo"></p>

<script>
let msec = Date.parse("March 21, 2012");
const d = new Date(msec);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

JAVASCRIPT GET DATE METHODS

These methods can be used for getting information from a date object:

Method	Description
getFullYear()	Get the **year** as a four digit number (yyyy)
getMonth()	Get the **month** as a number (0-11)
getDate()	Get the **day** as a number (1-31)
getHours()	Get the **hour** (0-23)
getMinutes()	Get the **minute** (0-59)
getSeconds()	Get the **second** (0-59)
getMilliseconds()	Get the **millisecond** (0-999)
getTime()	Get the time (milliseconds since January 1, 1970)
getDay()	Get the weekday as a number (0-6)
Date.now()	Get the time. ECMAScript 5.

The getTime() Method

The getTime() method returns the number of milliseconds since January 1, 1970:

Example:

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

<h2>JavaScript getTime()</h2>
<p>The internal clock in JavaScript counts from midnight January 1, 1970.</p>
<p>The getTime() function returns the number of milliseconds since then:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getTime();
</script>

</body>
</html>
```

The getFullYear() Method

The getFullYear() method returns the year of a date as a four digit number:

Example:

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

<h2>JavaScript getFullYear()</h2>
<p>The getFullYear() method returns the full year of a date:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getFullYear();
</script>

</body>
</html>
```

The getMonth() Method

The getMonth() method returns the month of a date as a number (0-11):

Example:

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

<h2>JavaScript getMonth()</h2>
<p>The getMonth() method returns the month of a date as a number from 0 to 11.</p>
<p>To get the correct month, you must add 1:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getMonth() + 1;
</script>

</body>
</html>
```

In JavaScript, the first month (January) is month number 0, so December returns month number 11.

You can use an array of names, and getMonth() to return the month as a name:

Example:

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

<h2>JavaScript getMonth()</h2>
<p>The getMonth() method returns the month as a number:</p>
<p>You can use an array to display the name of the month:</p>

<p id=”demo"></p>

<script>
const d = new Date();
const months =
```

```
["January", "February", "March", "April", "May", "June", "July", "August", "Sept
ember", "October", "November", "December"];
document.getElementById("demo").innerHTML = months[d.getMonth()];
</script>

</body>
</html>
```

The getDate() Method

The getDate() method returns the day of a date as a number (1-31):

Example:

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

<h2>JavaScript getDate()</h2>
<p>The getDate() method returns the day of a date as a number (1-31):</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getDate();
</script>

</body>
</html>
```

The getHours() Method

The getHours() method returns the hours of a date as a number (0-23):

Example:

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

<h2>JavaScript getHours()</h2>
<p>The getHours() method returns the hours of a date as a number (0-
23):</p>
```

```
<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getHours();
</script>

</body>
</html>
```

The getMinutes() Method

The getMinutes() method returns the minutes of a date as a number (0-59):

Example:

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

<h2>JavaScript getMinutes()</h2>
<p>The getMinutes() method returns the minutes of a date as a number (0-59):</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getMinutes();
</script>

</body>
</html>
```

The getSeconds() Method

The getSeconds() method returns the seconds of a date as a number (0-59):

Example:

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

<h2>JavaScript getSeconds()</h2>
```

```
<p>The getSeconds() method returns the seconds of a date as a number (0-59):</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getSeconds();
</script>

</body>
</html>
```

The getMilliseconds() Method

The getMilliseconds() method returns the milliseconds of a date as a number (0-999):

Example:

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

<h2> JavaScript getMilliseconds()</h2>
<p>The getMilliseconds() method returns the milliseconds of a date as a number (0-999):</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getMilliseconds();
</script>

</body>
</html>
```

The getDay() Method

The getDay() method returns the weekday of a date as a number (0-6):

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2> JavaScript getDay</h2>
<p>The getDay() method returns the weekday as a number:</p>

<p id=”demo"></p>

<script>
const d = new Date();
document.getElementById("demo").innerHTML = d.getDay();
</script>

</body>
</html>
```

In JavaScript, the first day of the week (0) means "Sunday", even if some countries in the world consider the first day of the week to be "Monday"

You can use an array of names, and getDay() to return the weekday as a name:
Example:

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

<h2> JavaScript getDay</h2>
<p>The getDay() method returns the weekday as a number:</p>
<p>You can use an array to display the name of the weekday:</p>

<p id=”demo"></p>

<script>
const d = new Date();
const days =
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturda
y"];
document.getElementById("demo").innerHTML = days[d.getDay()];
</script>
```

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

UTC Date Methods

UTC date methods are used for working with UTC dates (Universal Time Zone dates):

Method	Description
getUTCDate()	Same as getDate(), but returns the UTC date
getUTCDay()	Same as getDay(), but returns the UTC day
getUTCFullYear()	Same as getFullYear(), but returns the UTC year
getUTCHours()	Same as getHours(), but returns the UTC hour
getUTCMilliseconds()	Same as getMilliseconds(), but returns the UTC milliseconds
getUTCMinutes()	Same as getMinutes(), but returns the UTC minutes
getUTCMonth()	Same as getMonth(), but returns the UTC month
getUTCSeconds()	Same as getSeconds(), but returns the UTC seconds

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:

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

<h2>JavaScript setFullYear()</h2>
<p>The setFullYear() method sets the year of a date object:</p>

<p id=”demo"></p>

<script>
const d = new Date();
d.setFullYear(2020);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

The setFullYear() method can **optionally** set month and day:
Example:

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

<h2>JavaScript setFullYear()</h2>
<p>The setFullYear() method can optionally set month and day.</p>
<p>Please note that month counts from 0. December is month 11:</p>

<p id=”demo"></p>

<script>
const d = new Date();
d.setFullYear(2020, 11, 3);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

The setMonth() Method

The setMonth() method sets the month of a date object (0-11):

Example:

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

<h2>JavaScript setMonth()</h2>
<p>The setMonth() method sets the mont of a date object.</p>
<p>Note that months count from 0. December is month 11:</p>

<p id=”demo"></p>

<script>
const d = new Date();
d.setMonth(11);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

The setDate() Method

The setDate() method sets the day of a date object (1-31):

Example:

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

<h2>JavaScript setDate()</h2>
<p>The setDate() method sets the day of a date object:</p>

<p id=”demo"></p>

<script>
const d = new Date();
d.setDate(15);
document.getElementById("demo").innerHTML = d;
</script>
```

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

The setDate() method can also be used to **add days** to a date:
Example:

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

<h2>JavaScript setDate()</h2>
<p>The setDate() method can be used to add days to a date.</p>

<p id=”demo"></p>

<script>
const d = new Date();
d.setDate(d.getDate() + 50);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

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:

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

<h2>JavaScript setHours()</h2>
<p>The setHours() method sets the hours of a date objecte.</p>

<p id=”demo"></p>

<script>
const d = new Date();
```

```
d.setHours(22);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

The setMinutes() Method

The setMinutes() method sets the minutes of a date object (0-59):

Example:

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

<h2>JavaScript setMinutes()</h2>
<p>The setMinutes() method sets the minutes of a date object (0-59):</p>

<p id=”demo"></p>

<script>
const d = new Date();
d.setMinutes(30);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

The setSeconds() Method

The setSeconds() method sets the seconds of a date object (0-59):

Example:

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

<h2>JavaScript setSeconds()</h2>
<p>The setSeconds() method sets the seconds of a date object (0-59):</p>

<p id=”demo"></p>
```

```
<script>
const d = new Date();
d.setSeconds(30);
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Compare Dates

Dates can easily be compared.

The following example compares today's date with January 14, 2100:

Example:

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

<p id=”demo"></p>

<script>
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.";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

JavaScript counts months from 0 to 11. January is 0. December is 11.

JAVASCRIPT MATH OBJECT

The JavaScript Math object allows you to perform mathematical tasks on numbers.

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>

<p id=”demo"></p>

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

</body>
</html>
```

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:

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

<h2>JavaScript Math Constants</h2>
```

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

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
"<p><b>Math.E:</b> " + Math.E + "</p>" +
"<p><b>Math.PI:</b> " + Math.PI + "</p>" +
"<p><b>Math.SQRT2:</b> " + Math.SQRT2 + "</p>" +
"<p><b>Math.SQRT1_2:</b> " + Math.SQRT1_2 + "</p>" +
"<p><b>Math.LN2:</b> " + Math.LN2 + "</p>" +
"<p><b>Math.LN10:</b> " + Math.LN10 + "</p>" +
"<p><b>Math.LOG2E:</b> " + Math.LOG2E + "</p>" +
"<p><b>Math.Log10E:</b> " + Math.LOG10E + "</p>";
</script>

</body>
</html>
```

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:

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 (new in ES6)

Math.round()

Math.round(x) returns the nearest integer:

Example:

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

<h2>JavaScript Math.round()</h2>

<p>Math.round(x) returns the value of x rounded to its nearest integer:</p>
```

```
<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = Math.round(4.4);
</script>

</body>
</html>
```

Math.ceil()

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

Example:

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

<h2>JavaScript Math.cel()</h2>

<p>Math.ceil() rounds a number <strong>up</strong> to its nearest integer:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.ceil(4.4);
</script>

</body>
</html>
```

Math.floor()

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

Example:

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

<h2>JavaScript Math.floor()</h2>
```

```
<p>Math.floor(x) returns the value of x rounded <strong>down</strong> to its nearest integer:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.floor(4.7);
</script>

</body>
</html>
```

Math.trunc()

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

Example:

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

<h2>JavaScript Math.trunc()</h2>

<p>Math.trunc(x) returns the integer part of x:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.trunc(4.7);
</script>

</body>
</html>
```

Math.sign()

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

Example:

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

<h2>JavaScript Math.sing()</h2>
```

```
<p>Math.sign(x) returns if x is negative, null or positive:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.sign(4);
</script>

</body>
</html>
```

Math.pow()

Math.pow(x, y) returns the value of x to the power of y:

Example:

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

<h2>JavaScript Math.pow()</h2>

<p>Math.pow(x,y) returns the value of x to the power of y:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.pow(8, 2);
</script>

</body>
</html>
```

Math.sqrt()

Math.sqrt(x) returns the square root of x:Example:

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

<h2>JavaScript Math.sqrt()</h2>
```

```
<p>Math.sqrt(x) returns the square root of x:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.sqrt(64);
</script>

</body>
</html>
```

Math.abs()

Math.abs(x) returns the absolute (positive) value of x:

Example:

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

<h2>JavaScript Math.abs()</h2>

<p>Math.abs(x) returns the absolute (positive) value of x:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.abs(-4.4);
</script>

</body>
</html>
```


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:

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

```
<body>

<h2>JavaScript Math.sin()</h2>

<p>Math.sin(x) returns the sin of x (given in radians):</p>
<p>Angle in radians = (angle in degrees) * PI / 180.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
"The sine value of 90 degrees is " + Math.sin(90 * Math.PI / 180);
</script>

</body>
</html>
```

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 x PI / 180.

Example:

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

<h2>JavaScript Math.cos()</h2>

<p>Math.cos(x) returns the cosine of x (given in radians):</p>
<p>Angle in radians = (angle in degrees) * PI / 180.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
"The cosine value of 0 degrees is " + Math.cos(0 * Math.PI / 180);
</script>
```

```
</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.min()</h2>

<p>Math.min() returns the lowest value in a list of arguments:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.min(0, 150, 30, 20, -8, -200);
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.max()</h2>

<p>Math.max() returns the highest value in a list of arguments.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.max(0, 150, 30, 20, -8, -200);
</script>
```

```
</body>
</html>
```

Math.random()

Math.random() returns a random number between 0 (inclusive), and 1 (exclusive):

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.random()</h2>

<p>Math.random() returns a random number between 0 and 1:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  Math.random();
</script>

</body>
</html>
```

You will learn more about Math.random() in the next chapter of this tutorial.

The Math.log() Method

Math.log(x) returns the natural logarithm of x:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.log()</h2>

<p>Math.log() returns the natural logarithm of a number:</p>

<p id=”demo"></p>

<script>
```

```
document.getElementById("demo").innerHTML = Math.log(1);
</script>

</body>
</html>
```

The natural logarithm returns the time needed to reach a certain level of growth.
Math.E and Math.log() are twins.
How many times must we multiply Math.E to get 10?
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.log()</h2>

<p>How many times must we multiply Math.E to get 10?</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = Math.log(10);
</script>

</body>
</html>
```


The Math.log2() Method

Math.log2(x) returns the base 2 logarithm of x.
How many times must we multiply 2 to get 8?
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.log()</h2>

<p>Math.log2() returns the base 2 logarithm of a number.</p>
<p>How many times must we multiply 2 to get 8?</p>
```

```
<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = Math.log2(8);
</script>

</body>
</html>
```

The Math.log10() Method

Math.log10(x) returns the base 10 logarithm of x.
How many times must we multiply 10 to get 1000?
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.log10()</h2>

<p>Math.log10() returns the base 10 logarithm of a number.</p>
<p>How many times must we multiply 10 to get 1000?</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = Math.log10(1000);
</script>

</body>
</html>
```

Math Object Methods

Method	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
acosh(x)	Returns the hyperbolic arccosine of x
asin(x)	Returns the arcsine of x, in radians
asinh(x)	Returns the hyperbolic arcsine of x

atan(x)	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 Ex
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)

JAVASCRIPT RANDOM

Math.random()

Math.random() returns a random number between 0 (inclusive), and 1 (exclusive):

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.random()</h2>

<p>Math.random() returns a random number between 0 (included) and 1 (excluded):</p>
```

```
<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = Math.random();
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() * 10) returns a random integer between 0 and 9 (both included):</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.floor(Math.random() * 10);
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
```

```
<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() * 11) returns a random integer between 0 and 10 (both included):</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.floor(Math.random() * 11);
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() * 100)) returns a random integer between 0 and 99 (both included):</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.floor(Math.random() * 100);
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript Math</h2>

<p>Math.floor() used with Math.random() * 101 returns a random integer between 0 and 100 (both included):</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.floor(Math.random() * 101);
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() * 10) + 1) returns a random integer between 1 and 10 (both included):</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.floor(Math.random() * 10) + 1;
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math</h2>
```

```
<p>Math.floor(Math.random() * 100) + 1) returns a random integer between 1 and 100 (both included):</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
Math.floor(Math.random() * 100) + 1;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.random()</h2>

<p>Every time you click the button, getRndInteger(min, max) returns a random number between 0 and 9 (both included):</p>

<button onclick="document.getElementById('demo').innerHTML =
getRndInteger(0,10)"> Click Me </button>
<p id=”demo"></p>

<script>
function getRndInteger(min, max) {
  return Math.floor(Math.random() * (max - min) ) + min;
}
</script>

</body>
</html>
```

This JavaScript function always returns a random number between min and max (both included):
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.random()</h2>

<p>Every time you click the button, getRndInteger(min, max) returns a random number between 1 and 10 (both included):</p>

<button onclick="document.getElementById('demo').innerHTML = getRndInteger(1,10)"> Click Me </button>

<p id=”demo"></p>

<script>
function getRndInteger(min, max) {
  return Math.floor(Math.random() * (max - min + 1) ) + min;
}
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
<p>Display the value of Boolean(10 > 9):</p>

 <p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = Boolean(10 > 9);
</script>

</body>
</html>
```

Or even easier:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
<p>Display the value of 10 > 9:</p>

 <p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = 10 > 9;
</script>

</body>
</html>
```

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)

The Boolean value of an expression is the basis for all JavaScript comparisons and conditions.

Everything With a "Value" is True

Examples:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
"100 is " + Boolean(100) + "<br>" +
"3.14 is " + Boolean(3.14) + "<br>" +
"-15 is " + Boolean(-15) + "<br>" +
"Any (not empty) string is " + Boolean("Hello") + "<br>" +
"Even the string 'false' is " + Boolean('false') + "<br>" +
"Any expression (except zero) is " + Boolean(1 + 7 + 3.14);
</script>

</body>
</html>
```

Everything Without a "Value" is False

The Boolean value of **0** (zero) is **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
```

```
 <p>Display the Boolean value of 0:</p>

<p id=”demo"></p>

<script>
let x = 0;
document.getElementById("demo").innerHTML = Boolean(x);
</script>

</body>
</html>
```

The Boolean value of **-0** (minus zero) is **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Display the Boolean value of -0:</p>

<p id=”demo"></p>

<script>
let x = -0;
document.getElementById("demo").innerHTML = Boolean(x);
</script>

</body>
</html>
```

The Boolean value of **""** (empty string) is **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Display the Boolean value of “”:</p>

<p id=”demo"></p>
```

```
<script>
let x = “”;
document.getElementById("demo").innerHTML = Boolean(x);
</script>

</body>
</html>
```

The Boolean value of **undefined** is **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Display the Boolean value of undefined:</p>

<p id=”demo"></p>

<script>
let x =;
document.getElementById("demo").innerHTML = Boolean(x);
</script>

</body>
</html>
```


The Boolean value of **null** is **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Display the Boolean value of null:</p>

<p id=”demo"></p>

<script>
let x = null;
document.getElementById("demo").innerHTML = Boolean(x);
</script>
```

```
</body>
</html>
```

The Boolean value of **false** is (you guessed it) **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Display the Boolean value of false:</p>

<p id=”demo"></p>

<script>
let x = false;
document.getElementById("demo").innerHTML = Boolean(x);
</script>

</body>
</html>
```

The Boolean value of **NaN** is **false**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Display the Boolean value of NaN:</p>

<p id=”demo"></p>

<script>
let x = 10 / "Hallo";
document.getElementById("demo").innerHTML = Boolean(x);
</script>

</body>
</html>
```

Booleans Can be 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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Never create booleans as objects.</p>
<p>Booleans and objects cannot be safely compared.</p>

<p id=”demo"></p>

<script>
let x = false;          // x is a boolean
let y = new Boolean(false);      // y is an object
document.getElementById("demo").innerHTML = typeof x + "<br>" + typeof y;
</script>

</body>
</html>
```

Do not create Boolean objects. It slows down execution speed.

The new keyword complicates the code. This can produce some unexpected results:

When using the == operator, equal booleans are equal:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Never create booleans as objects.</p>
<p>Booleans and objects cannot be safely compared.</p>

<p id=”demo"></p>
```

```
<script>
let x = false;          // x is a boolean
let y = new Boolean(false);       // y is an object
document.getElementById("demo").innerHTML = (x==y);
</script>

</body>
</html>
```

When using the === operator, equal booleans are not equal, because the === operator expects equality in both type and value.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Never create booleans as objects.</p>
<p>Booleans and objects cannot be safely compared.</p>

<p id=”demo"></p>

<script>
let x = false;          // x is a boolean
let y = new Boolean(false);       // y is an object
document.getElementById("demo").innerHTML = (x===y);
</script>

</body>
</html>
```

Or even worse. Objects cannot be compared:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Booleans</h2>
 <p>Never create booleans as objects.</p>
<p>Booleans and objects cannot be safely compared.</p>
```

```
<p id=”demo"></p>

<script>
let x = new Boolean(false);         // x is an object
let y = new Boolean(false);         // y is an object
document.getElementById("demo").innerHTML = (x==y);
</script>

</body>
</html>
```

Note the difference between (x==y) and (x===y).
Comparing two JavaScript objects will always return false.

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
==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	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	(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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Comparison</h2>

 <p>Input your age and click the button.</p>

<button onclick="myFunction()">Try it</button>

<input id=”age" value = “18” />

<p id=”demo"></p>

<script>
Function My Function( ) {
```

```
  let age = document.getElementById("age").value;
  let voteable = (age < 18) ? "Too young":"Old enough";
  document.getElementById("demo").innerHTML = voteable + " to vote.";
}
</script>

</body>
</html>
```

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
2 < 12	true
2 < "12"	true
2 < "John"	false
2 > "John"	false
2 == "John"	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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Comparison</h2>

 <p>Input your age and click the button.</p>
```

```
<input id=”age" value = “18” />

<button onclick="myFunction()">Try it</button>

<p id=”demo"></p>

<script>
Function My Function( ) {
    let voteable;
    let age = Number(document.getElementById("age").value);
    if (isNaN(age)) {
      voteable = "Input is not a number";
   } else {
      voteable = (age < 18) ? "Too young" : "Old enough";
    }
    document.getElementById("demo").innerHTML = voteable + " to vote.";
}
</script>

</body>
</html>
```

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 switch statement is described in the next chapter.

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript If</h2>

 <p>Display "Good day!" if the hour is less than 18:00:</p>

<p id=”demo">Good Evening!</p>

<script>
   if (new Date().getHours() < 18)
   { document.getElementById("demo").innerHTML = "Good day!";
}
</script>

</body>
</html>
```


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":

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript If . . else</h2>

 <p>A time-based greeting:</p>

<p id=”demo"></p>

<script>
Const hour = new Date().getHours();
Let greeting;
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
document.getElementById("demo").innerHTML = "Good day!";
</script>

</body>
</html>
```


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":

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript If . . else</h2>

 <p>A time-based greeting:</p>

<p id=”demo"></p>

<script>
Const time = new Date().getHours();
Let greeting;
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20)
  { greeting = "Good
  day";
} else {
  greeting = "Good evening";
}
document.getElementById("demo").innerHTML = greeting;
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript switch</h2>

<p id=”demo"></p>

<script>
Let day;
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";
}
document.getElementById("demo").innerHTML = “Today is” + day;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript switch</h2>

<p id=”demo"></p>

<script>
Let text;
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";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

The default case does not have to be the last case in a switch block:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript switch</h2>

<p id=”demo"></p>

<script>
Let text;
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";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript switch</h2>

<p id=”demo"></p>

<script>
Let text;
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";
}
```

```
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript switch</h2>

<p id=”demo"></p>

<script>
let x = "0";

switch (x)
 {case 0:
   text = "Off";
   break;
 case 1:
   text = "On";
 break;
 default:
   text = "No value found";
}
```

```
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford", "Fiat", "Audi"];

let text= ” “;
for (let i = 0; i < cars.length; i++)
  {text += cars[i] + "<br>";
}
```

```
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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 loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.
Statement 2 defines the condition for executing the code block.
Statement 3 is executed (every time) after the code block has been executed.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

<p id=”demo"></p>

<script>
let text= ” “;

for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
document.getElementById("demo").innerHTML = text;
```

```
</script>

</body>
</html>
```

From the example above, you can read:

Statement 1 sets a variable before the loop starts (let i = 0).
Statement 2 defines the condition for the loop to run (i must be less than 5).
Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Statement 1

Normally you will use statement 1 to initialize the variable used in the loop (let i = 0).
This is not always the case, JavaScript doesn't care. Statement 1 is optional.
You can initiate many values in statement 1 (separated by comma):
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford", "Fiat", "Audi"];

let I, len, text;
for (let i = 0, len = cars.length, text = ""; i < len; i++)
  {text += cars[i] + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

And you can omit statement 1 (like when your values are set before the loop starts):

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford"];
let i = 2;
let len = cars.length;
let text = "";

for (; i < len; i++) {
  text += cars[i] + "<br>";
}

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```


Statement 2

- Often statement 2 is used to evaluate the condition of the initial variable.
- This is not always the case, JavaScript doesn't care. Statement 2 is also optional.
- If statement 2 returns true, the loop will start over again, if it returns false, the loop will end.

If you omit statement 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.

Statement 3

- Often statement 3 increments the value of the initial variable.

- This is not always the case, JavaScript doesn't care, and statement 3 is optional.
- Statement 3 can do anything like negative increment (i--), positive increment (i = i + 15), or anything else.
- Statement 3 can also be omitted (like when you increment your values inside the loop):

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford"];
let i = 0;
let len = cars.length;
let text = "";

for (; i < len; ) {
  text += cars[i] + "<br>";
  i++;
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```


Loop Scope

Using var in a loop:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript let</h2>
```

```
<p id=”demo"></p>

<script>
var i = 5;
for (var i = 0; i < 10; i++) {
  // some code
}
document.getElementById("demo").innerHTML = i;
</script>

</body>
</html>
```

Using let in a loop:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript let</h2>

<p id=”demo"></p>

<script>
let i = 5;
for (let i = 0; i < 10; i++) {
  // some code
}
document.getElementById("demo").innerHTML = i;
</script>

</body>
</html>
```


- 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.

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For In Loop</h2>
<p>The for in statement loops through the properties of an object:</p>

<p id=”demo"></p>

<script>
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person)
  { text += person[x] +
  “ ”;
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For In</h2>
<p>The for in statement can loops over array values:</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x] + “<br>”;
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```


- 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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array.forEach()</h2>
<p>Calls a function once for each array element.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);
document.getElementById("demo").innerHTML = txt;

function myFunction(value, index, array)
 {txt += value + “<br>”;
}
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array.forEach()</h2>
```

```
<p>Calls a function once for each array element.</p>

<p id=”demo"></p>

<script>
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);
document.getElementById("demo").innerHTML = txt;

function myFunction(value)
  {txt += value + “<br>”;
}
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Of Loop</h2>
<p>The for of statement loops through the values of any iterable object:</p>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars)
  { text += x + “<br>”;
}

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Looping over a String

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Of Loop</h2>
<p>The for of statement loops through the values of an iterable object.</p>
```

```
<p id=”demo"></p>

<script>
let language = "JavaScript";

let text = "";
for (let x of language)
{text += x + “<br>”;
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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
}
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript While Loop</h2>

<p id=”demo"></p>
```

```
<script>
let text = "";
while (i < 10) {
  text += "The number is " + i;
  i++;
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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);
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Do While Loop</h2>

<p id=”demo"></p>

<script>
let text = ""
```

```
Let I = 0;

do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford"];

let i = 0;
let text = "";
for (;cars[i];) {
  text += cars[i] +  “<br>”;
  i++;
}
```

```
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

The loop in this example uses a while loop to collect the car names from the cars array:

Example:

```
<!DOCTYPE html>
<html>
<body>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford"];

let i = 0;
let text = "";
while (cars[i];) {
  text += cars[i] +  “<br>”;
  i++;
}

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Loops</h2>
<p>A loop with a <b>break</b> statement.</p>

<p id=”demo"></p>

<script>
let text = "";
for (let i = 0; i < 10; i++)
  {if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Loops</h2>

<p>A loop with a <b>continue</b> statement.</p>
<p>A loop which will skip the step where i = 3.</p>
```

```
<p id=”demo"></p>

<script>
let text = "";
for (let i = 0; i < 10; i++)
  {if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript break</h2>

<p id=”demo"></p>

<script>
const cars = ["BMW", "Volvo", "Saab", "Ford"];
let text = "";

list: {
  text += cars[0] + "<br>";
  text += cars[1] + "<br>";
  break list;
  text += cars[2] + "<br>";
  text += cars[3] + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript iterables</h2>
<p>Iterate over a String:</p>

<p id=”demo"></p>

<script>
// Create a String
const name = "Ditrp";

// List all Elements
Let text= ” ”
for (const x of name)
  {text += x + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Iterating Over an Array

You can use a for..of loop to iterate over the elements of an Array:
Example:

```
<!DOCTYPE html>
<html>
```

```
<body>

<h2>JavaScript iterables</h2>
<p>Iterate over an Array:</p>

<p id=”demo"></p>

<script>
// Create a Array
const letters = ["a","b","c"];

// List all Elements
Let text= ” ”
for (const x of letters)
  {text += x + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Iterating Over a Set

You can use a for..of loop to iterate over the elements of a Set:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript iterables</h2>
<p>Iterate over a Set</p>

<p id=”demo"></p>

<script>
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Elements
```

```
Let text= ” ”
for (const x of letters)
  {text += x + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Iterating Over a Map

You can use a for..of loop to iterate over the elements of a Map:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript iterables</h2>
<p>Iterate over a Map</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map([["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);

// List all entries
Let text= ” ”
for (const x of fruits)
  {text += x + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
```

```
</body>
</html>
```

JAVASCRIPT SETS

A JavaScript Set is a collection of unique values.
Each value can only occur once in a Set.

Essential Set Methods

Method	**Description**
new Set()	Creates a new Set
add()	Adds a new element to the Set
delete()	Removes an element from a Set
has()	Returns true if a value exists in the Set
forEach()	Invokes a callback for each element in the Set
values()	Returns an iterator with all the values in a Set
Property	**Description**
size	Returns the number elements in a Set

How to Create a Set

You can create a JavaScript Set by:

- Passing an Array to new Set()
- Create a new Set and use add() to add values
- Create a new Set and use add() to add variables

The new Set() Method

Pass an Array to the new Set() constructor:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>Create a Set from an Array:</p>

<p id=”demo"></p>

<script>
```

```
// Create a Set
const letters = new Set(["a","b","c"]);

// Display set.size
document.getElementById("demo").innerHTML = letter.size;
</script>

</body>
</html>
```

Create a Set and add values:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>Add values to a Set:</p>

<p id=”demo"></p>

<script>
// Create a Set
const letters = new Set();

// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");

// Display set.size
document.getElementById("demo").innerHTML = letter.size;
</script>

</body>
</html>
```

Create a Set and add variables:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>Add variables to a Set:</p>

<p id=”demo"></p>

<script>
// 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);

// Display set.size
document.getElementById("demo").innerHTML = letter.size;
</script>

</body>
</html>
```

The add() Method

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>Adding new elements to a Set:</p>
```

```
<p id=”demo"></p>

<script>
// Create a new Set
const letters = new Set(["a","b","c"]);

// Add a new Element
letters.add("d");
letters.add("e");

// Display set.size
document.getElementById("demo").innerHTML = letter.size;
</script>

</body>
</html>
```

If you add equal elements, only the first will be saved:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>Adding equal elements to a Set:</p>

<p id=”demo"></p>

<script>
// Create a new Set
const letters = new Set( );

// Add values to the set
letters.add("a");
letters.add("b");
letters.add("c");
letters.add("c");
letters.add("c");
letters.add("c");
```

```
letters.add("c");
letters.add("c");

// Display set.size
document.getElementById("demo").innerHTML = letter.size;
</script>

</body>
</html>
```

The forEach() Method

The forEach() method invokes (calls) a function for each Set element:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>forEach() calls a function for each element:</p>

<p id=”demo"></p>

<script>
// Create a Set
const letters = new Set(["a","b","c"]);

// List all Elements
let text = "";
letters.forEach (function(value)
  {text += value;
})
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

The values() Method

The values() method returns a new iterator object containing all the values in a Set:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>forEach() calls a function for each element:</p>

<p id=”demo"></p>

<script>
// Create a Set
const letters = new Set(["a","b","c"]);

// Display set.size
document.getElementById("demo").innerHTML = letters.values( );
</script>

</body>
</html>
```

Now you can use the Iterator object to access the elements:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sets</h2>
<p>forEach() calls a function for each element:</p>

<p id=”demo"></p>

<script>
// Create a Set
const letters = new Set(["a","b","c"]);
```

```
// List all Elements
let text = "";
for (const x of letters.values())
  {text += x;
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

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.

Essensial Map Methods

Method	**Description**
new Map()	Creates a new Map
set()	Sets the value for a key in a Map
get()	Gets the value for a key in a Map
delete()	Removes a Map element specified by the key
has()	Returns true if a key exists in a Map
forEach()	Calls a function for each key/value pair in a Map
entries()	Returns an iterator with the [key, value] pairs in a Map
Property	**Description**
size	Returns the number of elements in Map

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:

```
<!DOCTYPE html>
<html>
```

```
<body>

<h2>JavaScript Map Objects</h2>
<p>Creating a Map from an Array:</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map([["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
document.getElementById("demo").innerHTML = fruits.get("apples");
</script>

</body>
</html>
```

The set() Method

You can add elements to a Map with the set() method:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Map Objects</h2>
<p>Using Map.set( )</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new Map();

// Set Map Values
fruits.set("apples", 500);
```

```
fruits.set("bananas", 300);
fruits.set("oranges", 200);

document.getElementById("demo").innerHTML = fruits.get("apples");
</script>

</body>
</html>
```

The set() method can also be used to change existing Map values:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Map Objects</h2>
<p>Using Map.set( )</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
fruits.set("apples", 500);
document.getElementById("demo").innerHTML = fruits.get("apples");
</script>

</body>
</html>
```

The get() Method

The get() method gets the value of a key in a Map:
Example:

```
<!DOCTYPE html>
<html>
```

```
<body>

<h2>JavaScript Map Objects</h2>
<p>Using Map.get( ):</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);

document.getElementById("demo").innerHTML = fruits.get("apples");
</script>

</body>
</html>
```

The size Property

The size property returns the number of elements in a Map:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Map Objects</h2>
<p>Using Map.size:</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
```

```
]);

document.getElementById("demo").innerHTML = fruits.size;
</script>

</body>
</html>
```

The delete() Method

The delete() method removes a Map element:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Maps</h2>
<p>Deleting Map elements:</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);

// Delete an Element
fruits.delete("apples");

document.getElementById("demo").innerHTML = fruits.size;
</script>

</body>
</html>
```

The has() Method

The has() method returns true if a key exists in a Map:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Maps</h2>
<p>Using Map.has():</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
document.getElementById("demo").innerHTML = fruits.has("apples");
</script>

</body>
</html>
```

Try This:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Maps</h2>
<p>Using Map.has():</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
```

```
  ["bananas", 300],
  ["oranges", 200]
]);

// Delete an Element
fruits.delete("apples");
document.getElementById("demo").innerHTML = fruits.has("apples");
</script>

</body>
</html>
```

JavaScript Objects vs Maps

Differences between JavaScript Objects and Maps:

	Object	**Map**
Iterable	Not directly iterable	Directly iterable
Size	Do not have a size property	Have a size property
Key Types	Keys must be Strings (or Symbols)	Keys can be any datatype
Key Order	Keys are not well ordered	Keys are ordered by insertion
Defaults	Have default keys	Do not have default keys

The forEach() Method

The forEach() method calls a function for each key/value pair in a Map:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Map Objects</h2>
<p>Using Map.forEach():</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
```

```
  ["oranges", 200]
]);

let text = "";
fruits.forEach (function(value, key)
  {text += key + ' = ' + value + “<br>”
})
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

The entries() Method

The entries() method returns an iterator object with the [key, values] in a Map:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Map Objects</h2>
<p>Using Map.entries():</p>

<p id=”demo"></p>

<script>
// Create a Map
const fruits = new
  Map( [["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);

let text = "";
for (const x of fruits.entries())
  {text + = x + “<br>”;
}
document.getElementById("demo").innerHTML = text;
```

```
</script>

</body>
</html>
```

Browser Support

JavaScript Maps are supported in all browsers, except Internet Explorer:

				
Chrome	Edge	Firefox	Safari	Opera

JAVASCRIPT TYPEOF

In JavaScript there are 5 different data types that can contain values:

- string
- number
- boolean
- object
- function

There are 6 types of objects:

- Object
- Date
- Array
- String
- Number
- Boolean

And 2 data types that cannot contain values:

- null
- undefined

The typeof Operator

You can use the typeof operator to find the data type of a JavaScript variable.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2> The JavaScript typeof Operator</h2>
```

```
<p> The typeof operator returns the type of a variable, object, function or expression.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
typeof "John" + ”<br>” +
typeof 3.14 + ”<br>” +
typeof NaN    + ”<br>” +
typeof false + ”<br>” +
typeof [1,2,3,4]  + ”<br>” +
typeof {name:'John', age:34} + ”<br>” +
typeof new Date( )  + ”<br>” +
typeof function ( ) { }    + ”<br>” +
typeof myCar  + ”<br>” +
typeof null;
</script>

</body>
</html>
```

Please observe:

- The data type of NaN is number
- The data type of an array is object
- The data type of a date is object
- The data type of null is object
- The data type of an undefined variable is **undefined** *
- The data type of a variable that has not been assigned a value is also **undefined** *

You cannot use typeof to determine if a JavaScript object is an array (or a date).

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The typeof operator can return one of these primitive types:

- string
- number

- boolean
- undefined

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript typeof</h2>

<p>The typeof operator returns the type of a variable or an expression.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
typeof "John" + ”<br>” +
typeof  3.14  +  ”<br>”  +
typeof  true  +  ”<br>”  +
typeof  false  +  ”<br>”  +
typeof x;
</script>

</body>
</html>
```


Complex Data

The typeof operator can return one of two complex types:

- function
- object

The typeof operator returns "object" for objects, arrays, and null.
The typeof operator does not return "object" for functions.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript typeof</h2>
```

```
<p>The typeof operator returns object for both objects, arrays, and null.</p>
<p>The typeof operator does not return object for functions.</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
typeof {name:'John', age:34} + ”<br>” +
typeof [1,2,3,4] + ”<br>” +
typeof null    + ”<br>” +
typeof function myFunc( ){ };
</script>

</body>
</html>
```

The typeof operator returns "object" for arrays because in JavaScript arrays are objects.

The Data Type of typeof

The typeofoperator is not a variable. It is an operator. Operators (+ - * /) do not have any data type.
But, the typeof operator always **returns a string** (containing the type of the operand).

The constructor Property

The constructor property returns the constructor function for all JavaScript variables.
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript constructor Property</h2>

<p>The constructor property returns the constructor function for a variable or an object.</p>

<p id=”demo"></p>
```

```
<script>
document.getElementById("demo").innerHTML =
"John".constructor    + ”<br>” +
(3.14).constructor    + ”<br>” +
false.constructor    + ”<br>” +
[1,2,3,4].constructor    + ”<br>” +
{name:'John',age:34}.constructor + ”<br>” +
new Date().constructor      + ”<br>” +
function () {}.constructor;
</script>

</body>
</html>
```

You can check the constructor property to find out if an object is an Array (contains the word "Array"):

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Arrays</h2>

<p>This "home made" isArray() function returns true when used on an array:</p>

<p id=”demo"></p>

<script>
const fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = isArray(fruits);

function isArray(myArray) {
  return myArray.constructor.toString().indexOf("Array") > -1;
}
</script>

</body>
</html>
```

Or even simpler, you can check if the object is an **Array function**:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Object</h2>

<p>This "home made" isArray() function returns true when used on an array:</p>

<p id=”demo"></p>

<script>
const fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = isArray(fruits);

function isArray(myArray) {
  return myArray.constructor === Array;
}
</script>

</body>
</html>
```

You can check the constructor property to find out if an object is a Date (contains the word "Date"):
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Date Object</h2>

<p>This "home made" isDate() function returns true when used on an date:</p>

<p id=”demo"></p>

<script>
```

```
const myDate = new Date();
document.getElementById("demo").innerHTML = isDate(myDate);

function isDate(myDate) {
  return myDate.constructor.toString().indexOf("Date") > -1;
}
</script>

</body>
</html>
```

Or even simpler, you can check if the object is a **Date function**:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Date Object</h2>

<p>This "home made" isDate() function returns true when used on an date:</p>

<p id=”demo"></p>

<script>
const myDate = new Date( );
document.getElementById("demo").innerHTML = isDate(myDate);

function isDate(myDate) {
  return myDate.constructor === Date;
}
</script>

</body>
</html>
```

Undefined

In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript</h2>

<p>The value (and the data type) of a variable with no value is
<b>undefined</b>.</p>

<p id=”demo"></p>

<script>
let car;
document.getElementById("demo").innerHTML =
car + "<br>" + typeof car;
</script>

</body>
</html>
```

Any variable can be emptied, by setting the value to undefined. The type will also be undefined.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript</h2>

<p>Variables can be emptied if you set the value to<b>undefined</b>.</p>

<p id=”demo"></p>

<script>
let car= “Volvo”;
car = undefined;
document.getElementById("demo").innerHTML = car + "<br>" + typeof car;
</script>
```

```
</body>
</html>
```

Empty Values

An empty value has nothing to do with undefined.

An empty string has both a legal value and a type.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript</h2>

<p>An empty string has both a legal value and a type:</p>

<p id=”demo"></p>

<script>
let car= “ ”;
document.getElementById("demo").innerHTML =
“The value is”: +
car + "<br>" +
“The type is:” + typeof car;
</script>

</body>
</html>
```

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 consider it a bug in JavaScript that typeof null is an object. It should be null.

You can empty an object by setting it to null:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript</h2>
```

```
<p>Objects can be emptied by setting the value to <b>null</b>.</p>

<p id=”demo"></p>

<script>
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;
document.getElementById("demo").innerHTML =typeof person;
</script>

</body>
</html>
```

You can also empty an object by setting it to undefined:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript</h2>

<p>Objects can be emptied by setting the value to <b>undefined</b>.</p>

<p id=”demo"></p>

<script>
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined;
document.getElementById("demo").innerHTML =person;
</script>

</body>
</html>
```


Difference Between Undefined and Null

undefined and null are equal in value but different in type:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript</h2>

<p>Undefined and null are equal in value but different in type:</p>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =
typeof undefined  + “<br>” +
typeof null  + “<br>” +
(null === undefined) + “<br>” +
(null == undefined);
 </script>

</body>
</html>
```

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() can convert strings to numbers.
Strings containing numbers (like "3.14") convert to numbers (like 3.14).
Empty strings convert to 0.
Anything else converts to NaN (Not a Number).

```
Number("3.14")    // returns 3.14
Number(" ")       // returns 0
```

```
Number("")       // returns 0
Number("99 88")  // returns NaN
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript typeof Operator</h2>

<p>The typeof operator returns the type of a variable or expression:</p>

<p id=”demo"></p>

<script>
let y = "5";
let x = + y;
document.getElementById("demo").innerHTML =  typeof y + “<br>” + typeof x;
 </script>

</body>
</html>
```

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a Number):

Example:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>The JavaScript typeof Operator</h2>

<p>The typeof operator returns the type of a variable or expression:</p>

<p id=”demo"></p>

<script>
let y = "John";
let x = + y;
document.getElementById("demo").innerHTML =  typeof y + “<br>” + typeof x;
 </script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript String() Method</h2>

<p>The String() method can convert a number to a string.</p>

<p id=”demo"></p>

<script>
let x = 123;
document.getElementById("demo").innerHTML =
String(x) + “<br>” +
String(123)    + “<br>” +
 String(100 + 23);
 </script>

</body>
</html>
```

The Number method toString() does the same.
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Number Methods</h2>

<p>The toString() method converts a number to a string.</p>

<p id=”demo"></p>

<script>
let x = 123;
document.getElementById("demo").innerHTML =
x.toString()
(123).toString()
(100 + 23).toString()
 </script>

</body>
</html>
```

More Methods

In the chapter Number Methods, you will find more methods that can be used to convert numbers to strings:

Method	Description
toExponential()	Returns a string, with a number rounded and written using exponential notation.
toFixed()	Returns a string, with a number rounded and written with a specified number of decimals.
toPrecision()	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
getDate()	Get the day as a number (1-31)
getDay()	Get the weekday a number (0-6)
getFullYear()	Get the four digit year (yyyy)
getHours()	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(){}	NaN	"function(){}"	true
{ }	NaN	"[object Object]"	true
null	**0**	"null"	false
undefined	NaN	"undefined"	false

Values in quotes indicate string values.
Red values indicate values (some) programmers might not expect.

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 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 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.

00000000000000000000000000000101 (5)

11111111111111111111111111111010 (~5 = -6)

A signed integer uses the leftmost bit as the minus sign.

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

Bitwise OR

When a bitwise OR is performed on a pair of bits, it returns 1 if one of the bits are 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

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 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

4 bits example:

Operation	Result
1111 ^ 0000	1111
1111 ^ 0001	1110
1111 ^ 0010	1101
1111 ^ 0100	1011

JavaScript Bitwise AND (&)

Bitwise AND returns 1 only if both bits are 1:

Decimal	Binary
5	00000000000000000000000000000101
1	00000000000000000000000000000001
5 & 1	00000000000000000000000000000001 (1)

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Bitwise AND</h2>

<p id=”demo">My First Paragraph.</p>

<script>
document.getElementById("demo").innerHTML =  5 & 1;
</script>
```

```
</body>
</html>
```

JavaScript Bitwise OR (|)

Bitwise OR returns 1 if one of the bits are 1:

Decimal	Binary
5	00000000000000000000000000000101
1	00000000000000000000000000000001
5 | 1	00000000000000000000000000000101 (5)

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Bitwise OR</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  5 | 1;
</script>

</body>
</html>
```

JavaScript Bitwise XOR (^)

Bitwise XOR returns 1 if the bits are different:

Decimal	Binary
5	00000000000000000000000000000101
1	00000000000000000000000000000001
5 ^ 1	00000000000000000000000000000100 (4)

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Bitwise XOR</h2>

<p id=”demo"></p>
```

```
<script>
document.getElementById("demo").innerHTML =  5 ^ 1;
</script>

</body>
</html>
```

JavaScript Bitwise NOT (~)

Decimal	Binary
5	00000000000000000000000000000101
~5	11111111111111111111111111111010 (-6)

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Bitwise NOT</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  ~5;
</script>

</body>
</html>
```

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:

Decimal	Binary
5	00000000000000000000000000000101
5 << 1	00000000000000000000000000001010 (10)

Example:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript Bitwise Left</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  5 << 1;
</script>

</body>
</html>
```

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:

Decimal	Binary
-5	11111111111111111111111111111011
-5 >> 1	11111111111111111111111111111101 (-3)

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Sign Preserving Bitwise Right.</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  -5 >> 1;
</script>

</body>
</html>
```

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:

Decimal	Binary
5	00000000000000000000000000000101

5 >>> 1	00000000000000000000000000000010 (2)

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Bitwise Right</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  5 >>> 1;
</script>

</body>
</html>
```

Binary Numbers

Binary numbers with only one bit set is easy to understand:

Binary Representation	Decimal value
00000000000000000000000000000001	1
00000000000000000000000000000010	2
00000000000000000000000000000100	4
00000000000000000000000000001000	8
00000000000000000000000000010000	16
00000000000000000000000000100000	32
00000000000000000000000001000000	64

Setting a few more bits reveals the binary pattern:

Binary Representation	Decimal value
00000000000000000000000000000101	5 (4 + 1)
00000000000000000000000000001101	13 (8 + 4 + 1)
00000000000000000000000000101101	45 (32 + 8 + 4 + 1)

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:

Binary Representation	Decimal value
00000000000000000000000000000101	5

11111111111111111111111111111011	-5
00000000000000000000000000000110	6
11111111111111111111111111111010	-6
00000000000000000000000000101000	40
11111111111111111111111111011000	-40

Converting Decimal to Binary

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Convert Decimal to Binary</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML = dec2bin(- 5);
function dec2bin(dec){
  return (dec >>> 0).toString(2);
}
</script>

</body>
</html>
```

Converting Binary to Decimal

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Convert Binary to Decimal</h2>

<p id=”demo"></p>

<script>
document.getElementById("demo").innerHTML =  bin2dec(101);
```

```
function bin2dec(bin){
  return parseInt(bin, 2).toString(10);
}
</script>

</body>
</html>
```

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:

/Ditrp/i;

Example explained:
/Ditrp/i is a regular expression.
Ditrp 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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript String Methods</h2>
<p>Search a string for "DITRP", and display the position of the match:</p>

<p id=”demo"></p>

<script>
let text = "Visit DITRP!";
let n = text.search("DITRP");
document.getElementById("demo").innerHTML =  n;
</script>

</body>
</html>
```

Using String search() With a Regular Expression

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Regular Expressions</h2>
<p>Search a string for "DITRP", and display the position of the match:</p>

<p id=”demo"></p>

<script>
```

```
let text = "Visit DITRP!";
let n = text.search("DITRP"/i);
document.getElementById("demo").innerHTML =  n;
</script>

</body>
</html>
```

Using String replace() With a String

The replace() method replaces a specified value with another value in a string:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript String Methods</h2>

<p>Replace "Microsoft" with "DITRP" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>
<p id=”demo">Please visit Microsoft!</p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
text.replace("Microsoft", "DITRP");
}
</script>

</body>
</html>
```

Use String replace() With a Regular Expression

Example:

Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

```
<!DOCTYPE html>
<html>
```

```
<body>

<h2>JavaScript String Methods</h2>

<p>Replace "Microsoft" with "DITRP" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>

<p id=”demo">Please visit Microsoft!</p>

<script>
function myFunction() {
  let text = document.getElementById("demo").innerHTML;
  document.getElementById("demo").innerHTML =
  text.replace("/Microsoft/i", "DITRP");
}
</script>

</body>
</html>
```

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 matches rather than stopping after the first match)
m	Perform multiline matching

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 *n*
n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p>Search for an "e" in the next paragraph:</p>

<p id=”p01">The best things in life are free!</p>
```

```
<p id=”demo">Please visit Microsoft!</p>

<script>
 let text = document.getElementById("p01").innerHTML;
const pattern = /e/;
document.getElementById("demo").innerHTML =  pattern.test(test);
</script>

</body>
</html>
```

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;

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p id=”demo"></p>

<script>
 const obj = /e/.exec("The best things in life are free!");
document.getElementById("demo").innerHTML =
"Found " + obj[0] + " in position " + obj.index + " in the text: " + obj.input;
</script>

</body>
</html>
```

JAVASCRIPT ERRORS - THROW AND TRY TO CATCH

- The try statement lets you test a block of code for errors.
- The catch statement lets you handle the error.
- The throw statement lets you create custom errors.
- The finally statement lets you execute code, after try and catch, regardless of the result.

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 "adddlert" to deliberately produce an error:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Error Handling</h2>

<p>This example demonstrates how to use <b>catch</b> to diplay an error.</p>

<p id="demo"></p>

<script>
try {
  adddlert("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script>
```

```
</body>
</html>
```

JavaScript catches **adddlert** 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>

<h2>JavaScript try catch</h2>

<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 == "") 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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript try catch</h2>

<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 == "") 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 =
    "";
  }
}
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Errors</h2>

<p>You cannot set the number of significant digits of a number to 500:</p>

<p id="demo">

<script>
let num = 1;
try {
  num.toPrecision(500);
 }
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
</script>

</body>
</html>
```

Reference Error

A ReferenceError is thrown if you use (reference) a variable that has not been declared:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Errors</h2>

<p>You cannot use the value of a non-existing variable:</p>

<p id="demo"></p>

<script>
let x = 5;
try {
  x = y + 1;
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
</script>

</body>
</html>
```

Syntax Error

A SyntaxError is thrown if you try to evaluate code with a syntax error.

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Errors</h2>

<p>You cannot evaluate code that contains a syntax error:</p>

<p id="demo"></p>

<script>
```

```
try
  { eval("alert('Hello)");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
</script>

</body>
</html>
```

Type Error

A TypeError is thrown if you use a value that is outside the range of expected types:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Errors</h2>

<p>You cannot convert a number to upper case:</p>

<p id="demo"></p>

<script>
let num = 1;
try {
  num.toUpperCase();
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
</script>

</body>
</html>
```

URI (Uniform Resource Identifier) Error

A URIError is thrown if you use illegal characters in a URI function:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Errors</h2>

<p>Some characters cannot be decoded with decodeURI():</p>

<p id="demo"></p>

<script>
try {
  decodeURI("%%%");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
</script>

</body>
</html>
```


Non-Standard Error Object Properties

Mozilla and Microsoft defines 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.

JAVASCRIPT SCOPE

Scope determines the accessibility (visibility) of variables.
JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

Block Scope

- Before ES6 (2015), JavaScript 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, become **LOCAL** to the function.
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Scope</h2>

<p><b>carName</b>is undefined outside myFunction():</p>

<p id="demo"></p>
```

```
<p id="demo"></p>

<script>
function myFunction()
 {let carName = "Volvo";
 document.getElementById("demo").innerHTML = typeof carName + " " +
carName;
}
document.getElementById("demo").innerHTML = typeof carName;
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Scope</h2>

<p>A GLOBAL variable can be accessed from any script or function.</p>

<p id="demo"></p>

<script>
  let carName = "Volvo";
myFunction();
  function myFunction() {
document.getElementById("demo").innerHTML = “I can Display” + carName;
}
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Global Variables</h2>

<p>If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable:</p>

<p id="demo"></p>

<script>
  myFunction();
// code here can use carName as a global variable
document.getElementById("demo").innerHTML = “I can Display” + carName;

function myFunction()
  {carName = "Volvo";
}
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Scope</h2>

<p>In HTML, global variables defined with <b>var</b>, will become window variables.</p>

<p id="demo"></p>
<script>
var carName = "Volvo";

// code here can use window.carName
document.getElementById("demo").innerHTML = “I can Display” +window.carName;
</script>

</body>
</html>
```

Global variables defined with the let keyword do not belong to the window object:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Global Variables</h2>
```

```
<p> In HTML, global variables defined with <b>let</b>, will not become window variables.</p>

<p id="demo"></p>

<script>
let carName = "Volvo";
// code here can not use window.carName
document.getElementById("demo").innerHTML = “I can Display” +window.carName;
</script>

</body>
</html>
```

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.

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:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
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
</script>

</body>
</html>
```

Example 2:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
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
</script>

</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Hoisting</h2>
<p>With <b>let</b>, you cannot use a variable before it is declared.</p>

<p id="demo"></p>
<script>
Try {
  carName = “saab”;
  let carName = "Volvo";
}
Catch(err)
  { document.getElementById("demo").innerHTML = err;
}
</script>

</body>
</html>
```

Using a const variable before it is declared, is a syntax errror, so the code will simply not run.
Example:
This code will not run.

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Hoisting</h2>
<p>With <b>let</b>, you cannot use a variable before it is declared.</p>
<p>Try to remove the //.</p>

<p id="demo"></p>

<script>
carName = "Volvo";
//const carName;
document.getElementById("demo").innerHTML = carName;
</script>

</body>
</html>
```

JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

Example 1 does **not** give the same result as **Example 2**:

Example 1:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
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
</script>

</body>
</html>
```

Example 2:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = "x is " + x + " and y is " + y;  // Display x and y

var y = 7; // Initialize y
</script>

</body>
</html>
```

- 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:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
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
</script>

</body>
</html>
```

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.

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					
"use strict"	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:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Using a variable without declaring it, is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
x = 3.14;       // This will cause an error (x is not defined).
</script>
</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>Global "use strict" declaration.</h2>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
myFunction();

function myFunction() {
```

```
  y = 3.14;  // This will cause an error (y is not defined)
}
</script>

</body>
</html>
```

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

```
<!DOCTYPE html>
<html>
<body>

<p>"use strict" in a function will only cause errors in that function.</p>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
x = 3.14;      // This will not cause an error.
myFunction();

function myFunction()
  {"use strict";
  y = 3.14;  // This will cause an error (y is not defined).
}
</script>

</body>
</html>
```


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:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Using a variable without declaring it, is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
x = 3.14;          // This will cause an error (x is not defined).
</script>

</body>
</html>
```

Objects are variables too.

Using an object, without declaring it, is not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Using an object without declaring it, is not allowed.</h3>
```

```
<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
x = {p1:10, p2:20}; // This will cause an error (x is not defined).
</script>

</body>
</html>
```

Deleting a variable (or object) is not allowed.

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Deleting a variable (or object) is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
let x = 3.14;
delete x;      // This will cause an error
</script>

</body>
</html>
```

Deleting a function is not allowed.

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Deleting a function is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
```

```
"use  strict";
function x(p1, p2) {};
delete x;              // This will cause an error
</script>

</body>
</html>
```

Duplicating a parameter name is not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Duplicating a parameter name is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
function x(p1, p1) {};  // This will cause an error
</script>

</body>
</html>
```

Octal numeric literals are not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Octal numeric literals are not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
let x = 010;           // This will cause an error
```

```
</script>

</body>
</html>
```

Octal escape characters are not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Octal escape characters are not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
let x = "\010";          // This will cause an error
</script>

</body>
</html>
```

Writing to a read-only property is not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Writing to a read-only property is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
const obj = {};
Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14;          // This will cause an error
```

```
</script>

</body>
</html>
```

Writing to a get-only property is not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Writing to a get-only property is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
const obj = {get x() {return 0} };

obj.x = 3.14;          // This will cause an error
</script>
</body>
</html>
```

Deleting an undeletable property is not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Deleting an udeletable property is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
delete Object.prototype; // This will cause an error
</script>
```

```
</body>
</html>
```

The word eval cannot be used as a variable:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>The string "eval" cannot be used as a variable.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
let eval = 3.14;       // This will cause an error
</script>

</body>
</html>
```

The word arguments cannot be used as a variable:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>The string "arguments" cannot be used as a variable.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
let arguments = 3.14;    // This will cause an error
</script>

</body>
</html>
```

The with statement is not allowed:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>The with statement is not allowed.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
with (Math){x = cos(2)}; // This will cause an error
</script>

</body>
</html>
```

For security reasons, eval() is not allowed to create variables in the scope from which it was called:

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>For security reasons, eval() is not allowed to create variables in the scope from which it was called.</h3>

<p>Activate debugging in your browser (F12) to see the error report.</p>

<script>
"use strict";
eval ("let x = 2");
alert (x);          // This will cause an error
</script>

</body>
</html>
```

- 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):

```
<!DOCTYPE html>
<html>
<body>

<h2>With "use strict":</h2>
<h3>Inside functions, the "this" keyword is no longer the global object if not specified:.</h3>

<script>
"use strict";
function myFunction() {
  alert(this); // will alert "undefined"
}
myFunction();
</script>

</body>
</html>
```

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

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>With "use strict":</h2>
<h3>Future reserved keywords are not allowed in strict mode.</h3>

<p>Activate debugging in your browser (F12) to see the error report..</p>

<script>
"use strict";
let public = 1500;      // This will cause an error
</script>

</body>
</html>
```

Watch Out!
The "use strict" directive is only recognized at the **beginning** of a script or a function.

The Javascript **This** Keyword

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>
<p>In this example, <b>this</b>represents the <b>person</b> object. </p>
<p>Because the person object "owns" the fullName method.</p>

<script>
// Create an object:
const person =
{ firstName: "John",
lastName : "Doe",
id : 5566,
```

```
  fullName : function( ) {
    return this.firstName + " " + this.lastName;
  }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>

</body>
</html>
```

*What is **this**?*

The JavaScript this keyword refers to the object it belongs to.

It has different values depending on where it is used:

- In a method, this refers to the **owner 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(), and apply() can refer this to **any object**.

***this** in a Method*

In an object method, this refers to the "**owner**" of the method.

In the example on the top of this page, this refers to the **person** object.

The **person** object is the **owner** of the **fullName** method.

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>
<p>In this example, <b>this</b>represents the <b>person</b>object. </p>
<p>Because the person object "owns" the fullName method.</p>

<p id="demo"> </p>

<script>
// Create an object:
const person =
{ firstName: "John",
```

```
  lastName : "Doe",
  id : 5566,
  fullName : function( ) {
    return this.firstName + " " + this.lastName;
  }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>

</body>
</html>
```

this Alone

When used alone, the **owner** is the Global object, so this refers to the Global object.

In a browser window the Global object is [object Window]:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>
<p> In this example, <b>this</b>refers to the window Object:</p>

<p id="demo"> </p>

<script>
let x = this;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

In **strict mode**, when used alone, this also refers to the Global object [object Window]:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>
<p> In this example, <b>this</b>refers to the window Object:</p>

<p id="demo"> </p>

<script>
"use strict";
let x = this;
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

***this** in a Function (Default)*

In a JavaScript function, the owner of the function is the **default** binding for this.

So, in a function, this refers to the Global object [object Window].

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>
<p> In this example, <b>this</b> represents the object that "owns" myFunction:</p>

<p id="demo"> </p>

<script>
document.getElementById("demo").innerHTML = myFunction( );
function myFunction() {
  return this;
}
</script>
```

```
</body>
</html>
```

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:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>

<p>In a function, by default, <b>this</b>  refers to the Global object.</p>
<p> In strict mode, <b>this</b> is <b>undefined</b>, because strict mode does not allow default binding:</p>

<p id="demo"> </p>

<script>

"use strict";
document.getElementById("demo").innerHTML = myFunction( );
function myFunction() {
  return this;
}
</script>

</body>
</html>
```

this in Event Handlers

In HTML event handlers, this refers to the HTML element that received the event:
Example:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>The JavaScript <i>this</i>Keyword</h2>

<button onclick="this.style.display='none'">Click to Remove Me!</button>

</body>
</html>
```

Object Method Binding

In these examples, this is the **person** object (The person object is the "owner" of the function):

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>

<p>In this example, <b>this</b> represents the person object that "owns" the myFunction method. </p>

<p id="demo"> </p>

<script>
// Create an object:
const person =
{ firstName : "John",
lastName  : "Doe",
id        : 5566,
  myFunction : function()
   {return this;
  }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.myFunction();
</script>

</body>
</html>
```

Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>

<p>In this example, <b>this</b> represents the <b>person</b> object</p>
<p>Because the person object "owns" the fullName method</p>

<script>
// Create an object:
const person =
 { firstName:
 "John",lastName :
 "Doe",id     :
 5566,
 fullName : function() {
  return this.firstName + " " + this.lastName;
 }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>

</body>
</html>
```

In other words: **this.firstName** means the **firstName** property of **this** (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.
You can read more about call() and apply() later in this tutorial.
In the example below, when calling person1.fullName with person2 as argument, this will refer to person2, even if it is a method of person1:
Example:

```
<!DOCTYPE html>
<html>
<body>

<h2>The JavaScript <i>this</i>Keyword</h2>

<p>In this example, <strong>this</strong> refers to person2, even if it is a method of person1:</p>
<p>Because the person object "owns" the fullName method</p>

<p id="demo"></p>

<script>
const person1 =
 { fullName: function( )
 {
   return this.firstName + " " + this.lastName;
 }
}
const person2 =
 { firstName:"John
 ",lastName:
 "Doe",
}
let x =person1.fullName.call(person2);
document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```


VIDYAPITH ACADEMY

A unit of **AITDC (OPC) PVT. LTD**.

IAF Accredited An ISO 9001:2015 Certified Institute.

Registered Under Ministry of Corporate Affairs

(CIN U80904AS2020OPC020468)

Registered Under MSME, Govt. of India. (UAN- AS04D0000207).

Registered Under MHRD (CR act) Govt. of India.