

JavaScript Data

String, Number, Boolean, Array, Object.

JavaScript Data Types

JavaScript variables can hold many **data types**: numbers, strings, arrays, objects and more:

```
var length = 16;           // Number
var lastName = "Johnson"; // String
var cars = ["Saab", "Volvo", "BMW"]; // Array
var 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:

```
var 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:

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

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

Example

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

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 Has Dynamic Types

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

Example

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

JavaScript Strings

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

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

Example

```
var carName = "Volvo XC60"; // Using double quotes
```

```
var carName = 'Volvo XC60'; // Using single quotes
```

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

Example

```
var answer = "It's alright"; // Single quote inside double quotes
```

```
var answer = "He is called 'Johnny'"; // Single quotes inside double quotes
```

```
var answer = 'He is called "Johnny"'; // Double quotes inside single quotes
```

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

```
var x1 = 34.00; // Written with decimals
```

```
var x2 = 34; // Written without decimals
```

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

Example

```
var y = 123e5; // 123000000
```

```
var z = 123e-5; // 0.00123
```

You will learn more about numbers later in this tutorial.

JavaScript Booleans

Booleans can only have two values: true or false.

Example

```
var x = true;
```

```
var y = false;
```

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

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

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

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

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:

Example

```
typeof "John"           // Returns "string"  
typeof 3.14             // Returns "number"  
typeof false            // Returns "boolean"  
typeof [1,2,3,4]        // Returns "object" (not "array", see note below)  
typeof {name:'John', age:34} // Returns "object"
```

The typeof operator returns "object" for arrays because in JavaScript arrays are objects.

Undefined

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

Example

```
var person;           // Value is undefined, type is undefined
```

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

Example

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

Empty Values

An empty value has nothing to do with undefined.

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

Example

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

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

```
var person = null;      // Value is null, but type is still an object
```

You can also empty an object by setting it to undefined:

Example

```
var person = undefined; // Value is undefined, type is undefined
```

Difference Between Undefined and Null

```
typeof undefined      // undefined
```

```
typeof null           // object
```

```
null === undefined    // false
```

```
null == undefined     // true
```

JavaScript Arrays

JavaScript arrays are used to store multiple values in a single variable.

Example

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

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:

```
var car1 = "Saab";
```

```
var car2 = "Volvo";
```

```
var 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:

```
var array-name = [item1, item2, ...];
```

Spaces and line breaks are not important. A declaration can span multiple lines:

Example

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

Never put a comma after the last element (like "BMW",).

The effect is inconsistent across browsers.

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example

```
var cars = new Array("Saab", "Volvo", "BMW");
```

The two examples above do exactly the same. There is no need to use `new Array()`.

For simplicity, readability and execution speed, use the first one (the array literal method).

Access the Elements of an Array

You refer to an array element by referring to the **index number**.

This statement accesses the value of the first element in cars:

```
var name = cars[0];
```

This statement modifies the first element in cars:

```
cars[0] = "Opel";
```

[0] is the first element in an array. [1] is the second. Array indexes start with 0.

Access the Full Array

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

Example

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

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

Arrays are Objects

Arrays are a special type of objects. The **typeof** operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

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

Array:

```
var person = ["John", "Doe", 46];
```

Objects use **names** to access its "members". In this example, **person.firstName** returns John:

Object:

```
var person = {firstName:"John", lastName:"Doe", age:46};
```

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.

Because of this, you can have variables of different types in the same Array.

You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;  
myArray[1] = myFunction;  
myArray[2] = myCars;
```

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

Examples

```
var x = cars.length; // The length property returns the number of elements  
var y = cars.sort(); // The sort() method sorts arrays  
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

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.length; // the length of fruits is 4  
The length property is always one more than the highest array index.
```

Adding Array Elements

The easiest way to add a new element to an array is using the push method:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Lemon"); // adds a new element (Lemon) to fruits  
New element can also be added to an array using the length property:
```

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Lemon"; // adds a new element (Lemon) to fruits  
Adding elements with high indexes can create undefined "holes" in an array:
```

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[10] = "Lemon";           // adds a new element (Lemon) to fruits
```

Looping Array Elements

The best way to loop through an array, is using a "for" loop:

Example

```
var fruits, text, fLen, i;

fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;
text = "<ul>";
for (i = 0; i < fLen; i++) {
    text += "<li>" + fruits[i] + "</li>";
}
```

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

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;    // person.length will return 3
var y = person[0];        // person[0] will return "John"
```

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**.

Avoid new Array()

There is no need to use the JavaScript's built-in array constructor **new** Array().

Use [] instead.

These two different statements both create a new empty array named points:

```
var points = new Array();    // Bad
var points = [];            // Good
```

These two different statements both create a new array containing 6 numbers:

```
var points = new Array(40, 100, 1, 5, 25, 10); // Bad
var points = [40, 100, 1, 5, 25, 10];         // Good
```

The **new** keyword only complicates the code. It can also produce some unexpected results:

```
var points = new Array(40, 100); // Creates an array with two elements (40 and 100)
What if I remove one of the elements?
```

```
var points = new Array(40);    // Creates an array with 40 undefined elements !!!!!
```

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":

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
typeof fruits;    // returns object
```

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()**:

```
Array.isArray(fruits);    // returns true
```

The problem with this solution is that ECMAScript 5 is **not supported in older browsers**.

Solution 2:

To solve this problem you can create your own **isArray()** function:

```
function isArray(x) {
    return x.constructor.toString().indexOf("Array") > -1;
}
```

The function above always returns true if the argument is an array.

Or more precisely: it returns true if the object prototype contains the word "Array".

Solution 3:

The **instanceof** operator returns true if an object is created by a given constructor:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
fruits instanceof Array    // returns true
```

JavaScript Functions

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

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

Example

```
function myFunction(p1, p2) {  
    return p1 * p2;           // The function returns the product of p1 and p2  
}
```

JavaScript Function Syntax

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

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

The parentheses may include parameter names separated by commas:

(parameter1, parameter2, ...)

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

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

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **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:

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

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

The result in x will be:

12

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:

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

The () Operator Invokes the Function

Using the example above, toCelsius refers to the function object, and toCelsius() refers to the function result.

Example

Accessing a function without () will return the function definition:

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

Functions Used as Variable Values

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

Example

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

```
var x = toCelsius(77);
```

```
var text = "The temperature is " + x + " Celsius";
```

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

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

JavaScript Scope

In JavaScript, objects and functions are also variables.

In JavaScript, scope is the set of variables, objects, and functions you have access to.

JavaScript has function scope: The scope changes inside functions.

Local JavaScript Variables

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

Local variables have **local scope**: They can only be accessed within the function.

Example

```
// code here can not use carName
```

```
function myFunction() {  
    var carName = "Volvo";
```

```
    // code here can use carName
```

```
}
```

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

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

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

A global variable has **global scope**: All scripts and functions on a web page can access it.

Example

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

```
// code here can use carName
```

```
function myFunction() {
```

```
    // code here can use carName
```

```
}
```

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable **carName**, even if the value is assigned inside a function.

Example

```
myFunction();
```

```
// code here can use carName
```

```
function myFunction() {  
    carName = "Volvo";  
}
```

Do NOT create global variables unless you intend to.

Global Variables in HTML

With JavaScript, the global scope is the complete JavaScript environment.

In HTML, the global scope is the window object. All global variables belong to the window object.

Example

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

```
// code here can use window.carName
```

Did You Know?

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.

Local variables are deleted when the function is completed.

Global variables are deleted when you close the page.

Function Arguments

Function arguments (parameters) work as local variables inside functions.

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:

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

With double quotes:

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

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

Example

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

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

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

Example

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

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 ...


You will learn a lot more about events and event handlers in the HTML DOM chapters.

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.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

All cars have the same **properties**, but the property values differ from car to car.

All cars have the same **methods**, but the methods are performed at different times.

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

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

```
var car = "Fiat";
```

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

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

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

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

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

Object Properties

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

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

Property	Property Value
----------	----------------

firstName	John
lastName	Doe
age	50
eyeColor	blue

Object 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;}

JavaScript objects are containers for named values called properties or methods.

Object Definition

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

Example

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

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

Example

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

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName

or

objectName["propertyName"]

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

Example

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

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

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:

```
var x = new String();    // Declares x as a String object
```

```
var y = new Number();    // Declares y as a Number object
```

```
var z = new Boolean();    // Declares z as a Boolean object
```

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

You will learn more about objects later in this tutorial.

JavaScript Dates

The Date object lets you work with dates (years, months, days, hours, minutes, seconds, and milliseconds)

JavaScript Date Formats

A JavaScript date can be written as a string:

Thu Sep 08 2016 16:27:48 GMT+0530 (India Standard Time)

or as a number:

1473332268224

Dates written as numbers, specifies the number of milliseconds since January 1, 1970, 00:00:00.

Displaying Dates

In this tutorial we use a script to display dates inside a `<p>` element with `id="demo"`:

Example

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

```
<script>
```

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

```
</script>
```

The script above says: assign the value of `Date()` to the content (innerHTML) of the element with `id="demo"`.

You will learn how to display a date, in a more readable format, at the bottom of this page.

Creating Date Objects

The Date object lets us work with dates.

A date consists of a year, a month, a day, an hour, a minute, a second, and milliseconds.

Date objects are created with the **new Date()** constructor.

There are **4 ways** of initiating a date:

```
new Date()
```

```
new Date(milliseconds)
```

```
new Date(dateString)
```

```
new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

Using `new Date()`, creates a new date object with the **current date and time**:

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d;
</script>
```

Using `new Date(date string)`, creates a new date object from the **specified date and time**:

Example

```
<script>
var d = new Date("October 13, 2014 11:13:00");
document.getElementById("demo").innerHTML = d;
</script>
```

Valid date strings (date formats) are described in the next chapter.

Using `new Date(number)`, creates a new date object as **zero time plus the number**.

Zero time is 01 January 1970 00:00:00 UTC. The number is specified in milliseconds:

Example

```
<script>
var d = new Date(86400000);
document.getElementById("demo").innerHTML = d;
</script>
```

JavaScript dates are calculated in milliseconds from 01 January, 1970 00:00:00 Universal Time (UTC). One day contains 86,400,000 millisecond.

Using `new Date(7 numbers)`, creates a new date object with the **specified date and time**:

The 7 numbers specify the year, month, day, hour, minute, second, and millisecond, in that order:

Example

```
<script>
var d = new Date(99,5,24,11,33,30,0);
document.getElementById("demo").innerHTML = d;
</script>
```

Variants of the example above let us omit any of the last 4 parameters:

Example

```
<script>
var d = new Date(99,5,24);
document.getElementById("demo").innerHTML = d;
</script>
```

JavaScript counts months from 0 to 11. January is 0. December is 11.

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 objects, using either local time or UTC (universal, or GMT) time.

Date methods are covered in a later chapter.

Displaying Dates

When you display a date object in HTML, it is automatically converted to a string, with the **toString()** method.

Example

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

```
<script>
d = new Date();
document.getElementById("demo").innerHTML = d;
</script>
```

Is the same as:

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

```
<script>
d = new Date();
document.getElementById("demo").innerHTML = d.toString();
</script>
```

The **toUTCString()** method converts a date to a UTC string (a date display standard).

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.toUTCString();
</script>
```

The **toDateString()** method converts a date to a more readable format:

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.toDateString();
</script>
```

Date objects are static. The computer time is ticking, but date objects, once created, are not.

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.

Read more about time zones in the next chapters.

JavaScript Date Methods

Date methods let you get and set date values (years, months, days, hours, minutes, seconds, milliseconds)

Date Get Methods

Get methods are used for getting a part of a date. Here are the most common (alphabetically):

Method	Description
<code>getDate()</code>	Get the day as a number (1-31)
<code>getDay()</code>	Get the weekday as a number (0-6)
<code>getFullYear()</code>	Get the four digit year (yyyy)
<code>getHours()</code>	Get the hour (0-23)
<code>getMilliseconds()</code>	Get the milliseconds (0-999)
<code>getMinutes()</code>	Get the minutes (0-59)
<code>getMonth()</code>	Get the month (0-11)
<code>getSeconds()</code>	Get the seconds (0-59)
<code>getTime()</code>	Get the time (milliseconds since January 1, 1970)

The `getTime()` Method

`getTime()` returns the number of milliseconds since January 1, 1970:

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.getTime();
</script>
```

The `getFullYear()` Method

`getFullYear()` returns the year of a date as a four digit number:

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.getFullYear();
```

</script>

The `getDay()` Method

`getDay()` returns the weekday as a number (0-6):

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.getDay();
</script>
```

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

```
<script>
var d = new Date();
var days = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"];
document.getElementById("demo").innerHTML = days[d.getDay()];
</script>
```

Date Set Methods

Set methods are used for setting a part of a date. Here are the most common (alphabetically):

Method	Description
<code>setDate()</code>	Set the day as a number (1-31)
<code>setFullYear()</code>	Set the year (optionally month and day)
<code>setHours()</code>	Set the hour (0-23)
<code>setMilliseconds()</code>	Set the milliseconds (0-999)
<code>setMinutes()</code>	Set the minutes (0-59)
<code>setMonth()</code>	Set the month (0-11)
<code>setSeconds()</code>	Set the seconds (0-59)
<code>setTime()</code>	Set the time (milliseconds since January 1, 1970)

The `setFullYear()` Method

setFullYear() sets a date object to a specific date. In this example, to January 14, 2020:

Example

```
<script>
var d = new Date();
d.setFullYear(2020, 0, 14);
document.getElementById("demo").innerHTML = d;
</script>
```

The setDate() Method

setDate() sets the day of the month (1-31):

Example

```
<script>
var d = new Date();
d.setDate(20);
document.getElementById("demo").innerHTML = d;
</script>
```

The setDate() method can also be used to **add days** to a date:

Example

```
<script>
var d = new Date();
d.setDate(d.getDate() + 50);
document.getElementById("demo").innerHTML = d;
</script>
```

If adding days, shifts the month or year, the changes are handled automatically by the Date object.

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

```
<script>
var msec = Date.parse("March 21, 2012");
document.getElementById("demo").innerHTML = msec;
</script>
```

You can then use the number of milliseconds to **convert it to a date** object:

Example

```
<script>
var msec = Date.parse("March 21, 2012");
var d = new Date(msec);
document.getElementById("demo").innerHTML = d;
</script>
```

Compare Dates

Dates can easily be compared.

The following example compares today's date with January 14, 2100:

Example

```
var today, someday, text;
today = new Date();
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;
JavaScript counts months from 0 to 11. January is 0. December is 11.
```

UTC Date Methods

UTC date methods are used for working 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

Complete JavaScript Date Reference

For a complete reference, go to our [Complete JavaScript Date Reference](#).

The reference contains descriptions and examples of all Date properties and methods.

JavaScript Numbers

JavaScript has only one type of number.

Numbers can be written with, or without, decimals.

JavaScript numbers can be written with, or without decimals:

Example

```
var x = 34.00; // A number with decimals
var y = 34;    // A number without decimals
```

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

Example

```
var x = 123e5; // 12300000
var y = 123e-5; // 0.00123
```

JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:

Value (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

Precision

Integers (numbers without a period or exponent notation) are considered accurate up to 15 digits.

Example

```
var x = 9999999999999999; // x will be 9999999999999999
var y = 9999999999999999; // y will be 10000000000000000
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

Example

```
var x = 0.2 + 0.1; // x will be 0.30000000000000004
```

To solve the problem above, it helps to multiply and divide:

Example

```
var x = (0.2 * 10 + 0.1 * 10) / 10;    // x will be 0.3
```

Hexadecimal

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

Example

```
var x = 0xFF;           // x will be 255
```

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 as base 16 (hex), base 8 (octal), or base 2 (binary).

Example

```
var myNumber = 128;
myNumber.toString(16);    // returns 80
myNumber.toString(8);     // returns 200
myNumber.toString(2);     // returns 10000000
```

Infinity

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

Example

```
var myNumber = 2;
while (myNumber != Infinity) {    // Execute until Infinity
    myNumber = myNumber * myNumber;
}
```

Division by 0 (zero) also generates Infinity:

Example

```
var x = 2 / 0;           // x will be Infinity
var y = -2 / 0;          // y will be -Infinity
```

Infinity is a number: `typeof Infinity` returns number.

Example


```
typeof Infinity;    // returns "number"
```

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

```
var x = 100 / "Apple"; // x will be NaN (Not a Number)
```

However, if the string contains a numeric value , the result will be a number:

Example

```
var x = 100 / "10";    // x will be 10
```

You can use the global JavaScript function `isNaN()` to find out if a value is a number.

Example

```
var x = 100 / "Apple";  
isNaN(x);           // returns true because x is Not a Number
```

Watch out for NaN. If you use NaN in a mathematical operation, the result will also be NaN:

Example

```
var x = NaN;  
var y = 5;  
var z = x + y;      // z will be NaN  
Or the result might be a concatenation:
```

Example

```
var x = NaN;  
var y = "5";  
var z = x + y;      // z will be NaN5  
NaN is a number, and typeof NaN returns number:
```

Example

```
typeof NaN;         // returns "number"
```

Numbers Can be Objects

Normally JavaScript numbers are primitive values created from literals: **var x = 123**

But numbers can also be defined as objects with the keyword `new`: **var y = new Number(123)**

Example

```
var x = 123;  
var y = new Number(123);
```

```
// typeof x returns number
```

```
// typeof y returns object
```

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 == equality operator, equal numbers looks equal:

Example

```
var x = 500;  
var y = new Number(500);
```

```
// (x == y) is true because x and y have equal values
```

When using the === equality operator, equal numbers are not equal, because the === operator expects equality in both type and value.

Example

```
var x = 500;  
var y = new Number(500);
```

```
// (x === y) is false because x and y have different types
```

Or even worse. Objects cannot be compared:

Example

```
var x = new Number(500);  
var y = new Number(500);
```

```
// (x == y) is false because objects cannot be compared
```

JavaScript Math Object

The Math object allows you to perform mathematical tasks on numbers.

The Math Object

The Math object allows you to perform mathematical tasks.

The Math object includes several mathematical methods.

One common use of the Math object is to create a random number:

Example

```
Math.random();    // returns a random number
```

Math has no constructor. No methods have to create a Math object first.

Math.min() and Math.max()

Math.min() and Math.max() can be used to find the lowest or highest value in a list of arguments:

Example

```
Math.min(0, 150, 30, 20, -8, -200);    // returns -200
```

Math.random()

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

Example

```
Math.random();    // returns a random number
```

Math.random() always returns a number lower than 1.

Math.round()

Math.round() rounds a number to the nearest integer:

Example

```
Math.round(4.7);    // returns 5
```

```
Math.round(4.4);    // returns 4
```

Math.ceil()

Math.ceil() rounds a number **up** to the nearest integer:

Example

```
Math.ceil(4.4);      // returns 5
```

Math.floor()

Math.floor() rounds a number **down** to the nearest integer:

Example

```
Math.floor(4.7);      // returns 4
```

Math.floor() and Math.random() can be used together to return a random number between 0 and 10:

Example

```
Math.floor(Math.random() * 11); // returns a random number between 0 and 10
```

Math Constants

JavaScript provides 8 mathematical constants that can be accessed with the Math object:

Example

```
Math.E      // returns Euler's number
Math.PI     // returns PI
Math.SQRT2   // returns the square root of 2
Math.SQRT1_2 // returns the square root of 1/2
Math.LN2     // returns the natural logarithm of 2
Math.LN10    // returns the natural logarithm of 10
Math.LOG2E   // returns base 2 logarithm of E
Math.LOG10E  // returns base 10 logarithm of E
```

Math Object Methods

Method	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
asin(x)	Returns the arcsine of x, in radians
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
ceil(x)	Returns x, rounded upwards to the nearest integer
cos(x)	Returns the cosine of x (x is in radians)

<code>exp(x)</code>	Returns the value of E^x
<code>floor(x)</code>	Returns x, rounded downwards to the nearest integer
<code>log(x)</code>	Returns the natural logarithm (base E) of x
<code>max(x,y,z,...,n)</code>	Returns the number with the highest value
<code>min(x,y,z,...,n)</code>	Returns the number with the lowest value
<code>pow(x,y)</code>	Returns the value of x to the power of y
<code>random()</code>	Returns a random number between 0 and 1
<code>round(x)</code>	Rounds x to the nearest integer
<code>sin(x)</code>	Returns the sine of x (x is in radians)
<code>sqrt(x)</code>	Returns the square root of x
<code>tan(x)</code>	Returns the tangent of an angle

Complete Math Reference

For a complete reference, go to our [complete Math object reference](#).

The reference contains descriptions and examples of all Math properties and methods.

JavaScript Strings

A JavaScript string simply stores a series of characters like "John Doe".

A string can be any text inside quotes. You can use single or double quotes:

Example

```
var carname = "Volvo XC60";  
var carname = 'Volvo XC60';
```

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

Example

```
var answer = "It's alright";  
var answer = "He is called 'Johnny'";  
var answer = 'He is called "Johnny"';
```

String Length

The length of a string is found in the built in property **length**:

Example

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
var sln = txt.length;
```

Special Characters

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

```
var y = "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 **\ escape character**.

The backslash escape character turns special characters into string characters:

Example

```
var x = 'It\'s alright';  
var y = "We are the so-called \"Vikings\" from the north."
```

The escape character (\) can also be used to insert other special characters in a string.

This is the list of special characters that can be added to a text string with the backslash sign:

Code

Outputs

\'	single quote
\"	double quote
\\	backslash
\n	new line
\r	carriage return
\t	tab
\b	backspace
\f	form feed

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

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

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

Example

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

The \ method is not a ECMAScript (JavaScript) standard.
Some browsers do not allow spaces behind the \ character.

The safest (but a little slower) way to break a long string is to use string addition:

Example

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

You cannot break up a code line with a backslash:

Example

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

Strings Can be Objects

Normally, JavaScript strings are primitive values, created from literals: **var firstName = "John"**

But strings can also be defined as objects with the keyword new: **var firstName = new String("John")**

Example

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

```
// typeof x will return string  
// typeof y will return object
```

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 == equality operator, equal strings look equal:

Example

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

```
// (x == y) is true because x and y have equal values
```

When using the === equality operator, equal strings are not equal, because the === operator expects equality in both type and value.

Example

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

```
// (x === y) is false because x and y have different types (string and object)
```

Or even worse. Objects cannot be compared:

Example

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

```
// (x == y) is false because x and y are different objects
```

```
// (x == x) is true because both are the same object
```


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

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
var sln = txt.length;
```

Finding a String in a String

The **indexOf()** method returns the index of (the position of) the **first** occurrence of a specified text in a string:

Example

```
var str = "Please locate where 'locate' occurs!";  
var pos = str.indexOf("locate");
```

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

Example

```
var str = "Please locate where 'locate' occurs!";  
var pos = str.lastIndexOf("locate");
```

Both the **indexOf()**, and the **lastIndexOf()** methods return -1 if the text is not found.

JavaScript counts positions from zero.

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

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

Searching for a String in a String

The **search()** method searches a string for a specified value and returns the position of the match:

Example

```
var str = "Please locate where 'locate' occurs!";  
var pos = str.search("locate");
```

Did You Notice?

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

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

The two methods are equal, but the `search()` method can take much more powerful search values.

You will learn more about powerful search values in the chapter about regular expressions.

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 starting index (position), and the ending index (position).

This example slices out a portion of a string from position 7 to position 13:

Example

```
var str = "Apple, Banana, Kiwi";  
var res = str.slice(7,13);
```

The result of `res` will be:

Banana

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

```
var str = "Apple, Banana, Kiwi";  
var res = str.slice(-12,-6);
```

The result of `res` will be:

Banana

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

Example

```
var res = str.slice(7);
```

or, counting from the end:

Example

```
var res = str.slice(-12);
```

Negative positions do not work in Internet Explorer 8 and earlier.

The substring() Method

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

The difference is that **substring()** cannot accept negative indexes.

Example

```
var str = "Apple, Banana, Kiwi";  
var res = str.substring(7,13);
```

The result of *res* will be:

Banana

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

```
var str = "Apple, Banana, Kiwi";  
var res = str.substr(7,6);
```

The result of *res* will be:

Banana

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

The second parameter can not be negative, because it defines the length.

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

Replacing String Content

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

Example

```
str = "Please visit Microsoft!";  
var n = str.replace("Microsoft","W3Schools");
```

The `replace()` method can also take a regular expression as the search value.

By default, the `replace()` function replaces only the first match. To replace all matches, use a regular expression with a `g` flag (for global match):

Example

```
str = "Please visit Microsoft!";  
var n = str.replace(/Microsoft/g,"W3Schools");
```

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

Converting to Upper and Lower Case

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

Example

```
var text1 = "Hello World!";    // String  
var text2 = text1.toUpperCase(); // text2 is text1 converted to upper
```

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

Example

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

The `concat()` Method

`concat()` joins two or more strings:

Example

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

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

Example

```
var text = "Hello" + " " + "World!";  
var 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.

Extracting String Characters

There are 2 **safe** methods for extracting string characters:

- `charAt(position)`
- `charCodeAt(position)`

The `charAt()` Method

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

Example

```
var str = "HELLO WORLD";  
str.charAt(0);           // returns H
```

The `charCodeAt()` Method

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

Example

```
var str = "HELLO WORLD";  
  
str.charCodeAt(0);       // returns 72
```

Accessing a String as an Array is Unsafe

You might have seen code like this, accessing a string as an array:

```
var str = "HELLO WORLD";  
  
str[0];                  // returns H
```

This is **unsafe** and **unpredictable**:

- It does not work in all browsers (not in IE5, IE6, IE7)
- It makes strings look like arrays (but they are not)
- `str[0] = "H"` does not give an error (but does not work)

If you want to read a string as an array, convert it to an array first.

Converting a String to an Array

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

Example

```
var txt = "a,b,c,d,e"; // String
txt.split(",");        // Split on commas
txt.split(" ");        // Split on spaces
txt.split("|");        // Split on pipe
```

If the separator is omitted, the returned array will contain the whole string in index [0].

If the separator is "", the returned array will be an array of single characters:

Example

```
var txt = "Hello";    // String
txt.split("");        // Split in characters
```

Complete String Reference

For a complete reference, go to our [Complete JavaScript String Reference](#).

The reference contains descriptions and examples of all string properties and methods.