

# JavaScript Material

JavaScript is the programming language of the Web.

All modern HTML pages are using JavaScript.

JavaScript is easy to learn.

This tutorial will teach you JavaScript from basic to advanced.

## My First JavaScript



We recommend reading this tutorial, in the sequence listed in the left menu.

## Why Study JavaScript?

JavaScript is one of the **3 languages** all web developers **must** learn:

1. **HTML** to define the content of web pages
2. **CSS** to specify the layout of web pages
3. **JavaScript** to program the behavior of web pages

## JavaScript Introduction

JavaScript is the most popular programming language in the world.

This page contains some examples of what JavaScript can do.

## 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. ECMAScript 5 (JavaScript 1.8.5 - July 2010) is the latest standard.

JavaScript can be placed in the `<body>` and the `<head>` sections of an HTML page.

## The `<script>` Tag

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

### Example

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



Older 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 "asked" for.

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

You will learn much more about functions and events in later chapters.

## 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.



Keeping all code in one place, is always a good habit.

## 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>My 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>My 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>
```



It is a good idea to place scripts at the bottom of the <body> element.  
This can improve page load, because HTML display is not blocked by scripts loading.

## External JavaScript

Scripts can also be placed in external files.

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 the <script> tag:

## Example

```
<!DOCTYPE html>
<html>
<body>
<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 JavaScripts 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.

JavaScript does not have any built-in print or display functions.

# JavaScript Display Possibilities

JavaScript can "display" data in different ways:

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

## 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>
```

## 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 fully loaded, will **delete all existing HTML**:

## Example

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

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

<button onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```



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

## 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>
```



In our examples, we often use the innerHTML method (writing into an HTML element).

## Using console.log()

In your browser, you can use the **console.log()** method to display data.

Activate the browser console with F12, and select "Console" in the menu.



## Example

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

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

<script>
console.log(5 + 6);
</script>

</body>
</html>
```

## JavaScript Syntax

JavaScript **syntax** is the rules, how JavaScript programs are constructed.

## JavaScript Programs

A **computer program** is a list of "instructions" to be "executed" by the computer.

In a programming language, these program instructions are called **statements**.

JavaScript is a **programming language**.

JavaScript statements are separated by **semicolon**:

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



In HTML, JavaScript programs can be executed by the web browser.

## JavaScript Statements

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

## JavaScript Values

The JavaScript syntax defines two types of values: Fixed values and variable values.

Fixed values are called **literals**. Variable values are called **variables**.

## JavaScript Literals

The most important rules for writing fixed values are:

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

10.50

1001

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

"John Doe"

'John Doe'

**Expressions** can also represent fixed values:

$5 + 6$

$5 * 10$

## JavaScript Variables

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

JavaScript uses the **var** keyword to **define** 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:

```
var x;
```

```
x = 6;
```

## JavaScript Operators

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

```
var x = 5;
```

```
var y = 6;
```

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

$(5 + 6) * 10$

## JavaScript Keywords

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

The **var** keyword tells the browser to create a new variable:

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

[Try it Yourself »](#)

## 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:

```
var x = 5; // I will be executed
```

```
// var x = 6; I will NOT be executed
```

[Try it Yourself »](#)

## JavaScript is Case Sensitive

All JavaScript identifiers are **case sensitive**.

The variables **lastName** and **lastname**, are two different variables.

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

JavaScript does not interpret **VAR** or **Var** as the keyword **var**.



It is common, in JavaScript, to use camelCase names.

You will often see names written like lastName (instead of lastname).

## JavaScript Character Set

JavaScript uses the **Unicode** character set.

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

## JavaScript Statements

In HTML, JavaScript statements are "instructions" to be "executed" by the web browser.

### JavaScript Statements

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

### Example

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

## JavaScript Programs

Most JavaScript programs contain many JavaScript statements.

The statements are executed, one by one, in the same order as they are written.

In this example, x, y, and z is given values, and finally z is displayed:

## Example

```
var x = 5;  
var y = 6;  
var z = x + y;  
document.getElementById("demo").innerHTML = z;
```



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

---

## Semicolons ;

Semicolons separate JavaScript statements.

Add a semicolon at the end of each executable statement:

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

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

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



On the web, you might see examples without semicolons.

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

## JavaScript White Space

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

The following lines are equivalent:

```
var person = "Hege";  
var person="Hege";
```

## JavaScript Line Length and Line Breaks

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

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

### Example

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

## JavaScript Code Blocks

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

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

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

### Example

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

```
document.getElementById("myDIV").innerHTML = "How are you?";  
}
```



In this tutorial we use 4 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.

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

Keyword	Description
<b>break</b>	Terminates a switch or a loop
<b>continue</b>	Jumps out of a loop and starts at the top
<b>debugger</b>	Stops the execution of JavaScript, and calls (if available) the debugging function
<b>do ... while</b>	Executes a block of statements, and repeats the block, while a condition is true
<b>for</b>	Marks a block of statements to be executed, as long as a condition is true
<b>function</b>	Declares a function
<b>if ... else</b>	Marks a block of statements to be executed, depending on a condition
<b>return</b>	Exits a function
<b>switch</b>	Marks a block of statements to be executed, depending on different cases
<b>try ... catch</b>	Implements error handling to a block of statements
<b>var</b>	Declares a variable



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

## 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 a line, will be ignored by JavaScript (will not be executed).

This example uses a single line comment before each line, to explain the code:

### Example

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

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

### Example

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

## Multi-line Comments

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

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

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

### Example

```
/*  
The code below will change  
the heading with id = "myH"
```

and the paragraph with id = "myP"

in my web page:

```
*/
```

```
document.getElementById("myH").innerHTML = "My First Page";
```

```
document.getElementById("myP").innerHTML = "My first paragraph.";
```



It is most common to use single line comments.

Block comments are often used for formal documentation.

## Using Comments to Prevent Execution

Using comments to prevent execution of code, are 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

```
//document.getElementById("myH").innerHTML = "My First Page";
```

```
document.getElementById("myP").innerHTML = "My first paragraph.";
```

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

### Example

```
/*
```

```
document.getElementById("myH").innerHTML = "My First Page";
```

```
document.getElementById("myP").innerHTML = "My first paragraph.";
```

```
*/
```

## JavaScript Variables

JavaScript variables are containers for storing data values.

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

## Example

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

From the example above, you can expect:

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

---

## Much Like Algebra

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

## Example

```
var price1 = 5;  
var price2 = 6;  
var total = price1 + price2;
```

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 11.00



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 quotes around a number, it will be treated as a text string.

## Example

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

## 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 is empty (it has no value).

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

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

```
<script>
```

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



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**:

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

A declaration can span multiple lines:

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

## Value = undefined

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

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

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

## Example

```
var carName;
```

## 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

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

## JavaScript Arithmetic

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

### Example

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

You can also add strings, but strings will be concatenated (added end-to-end):

### Example

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

Also try this:

### Example

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



If you add a number to a string, the number will be treated as string, and concatenated.

# JavaScript Operators

## Example

Assign values to variables and add them together:

```
var x = 5;    // assign the value 5 to x
var y = 2;    // assign the value 2 to y
var z = x + y; // assign the value 7 to z (x + y)
```

## JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic between variables and/or values.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

The **addition** operator (+) adds a value:

## Adding

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

The **subtraction** operator (-) subtracts a value.



## Subtracting

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

The **multiplication** operator (\*) multiplies a value.

## Multiplying

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

The **division** operator (/) divides a value.

## Dividing

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

The **modular** operator (%) returns division remainder.

## Modulus

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

The **increment** operator (++) increments a value.

## Incrementing

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

The **decrement** operator (--) decrements a value.

## Decrementing

```
var x = 5;  
x--;  
var z = x;
```

## JavaScript Assignment Operators

Assignment operators are used to 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

The = assignment operator assigns a value to a variable.

## Assignment

```
var x = 10;
```

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

## Assignment

```
var x = 10;  
x += 5;
```

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

## Assignment

```
var x = 10;  
x -= 5;
```

The `*=` assignment operator multiplies a variable.

## Assignment

```
var x = 10;  
x *= 5;
```

The `/=` assignment divides a variable.

## Assignment

```
var x = 10;  
x /= 5;
```

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

## Assignment

```
var x = 10;  
x %= 5;
```

## JavaScript String Operators

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



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

## Example

To add two or more string variables together, use the + operator.

```
txt1 = "What a very";  
txt2 = "nice day";  
txt3 = txt1 + txt2;
```

The result of **txt3** will be:

What a verynice day

To add a space between the two strings, insert a space into one of the strings:

## Example

```
txt1 = "What a very ";  
txt2 = "nice day";  
txt3 = txt1 + txt2;
```

The result of **txt3** will be:

What a very nice day

or insert a space into the expression:

## Example

```
txt1 = "What a very";  
txt2 = "nice day";  
txt3 = txt1 + " " + txt2;
```

The result of **txt3** will be:

What a very nice day

The += operator can also be used to concatenate strings:

## Example

```
txt1 = "What a very ";  
txt1 += "nice day";
```

The result of **txt1** will be:

What a very nice day

## Adding Strings and Numbers

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

### Example

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

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

```
10  
55  
Hello5
```

The rule is: **If you add a number and a string, the result will be a string!**

## JavaScript Data Types

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 produce an error or a result?

JavaScript will treat it as:

### Example

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



If one operand is a string, JavaScript will treat all operands 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
typeof {name:'John', age:34} // Returns object
```



In JavaScript, an array is a special type of object. Therefore `typeof [1,2,3,4]` returns `object`.

## Undefined

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

### Example

```
var person;           // The value is undefined, the typeof is undefined
```

[Try it Yourself »](#)

You will learn more about undefined later in this tutorial.

## 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
```

## 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: **{}**

```
functionName(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 are used 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(32);
```

## 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 Variables

In JavaScript, functions can be used as variables:

## Example

Instead of:

```
temp = toCelsius(32);  
text = "The temperature is " + temp + " Centigrade";
```

You can use:

```
text = "The temperature is " + toCelsius(32) + " Centigrade";
```




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

# 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
	<code>car.name = Fiat</code> <code>car.model = 500</code> <code>car.weight = 850kg</code> <code>car.color = white</code>	<code>car.start()</code> <code>car.drive()</code> <code>car.brake()</code> <code>car.stop()</code>

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:

```
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
<b>firstName</b>	John
<b>lastName</b>	Doe
<b>age</b>	50
<b>eyeColor</b>	blue

## Object Methods

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

Methods are stored in properties as **function definitions**.

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



JavaScript objects are containers for named values, called properties and 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]*

### Example1

```
person.lastName;
```



## Example2

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

## Accessing Object Methods

You access an object method with the following syntax:

```
objectName.methodName()
```

## Example

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

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

## Example

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

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

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

```
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 Scope

Scope is the set of variables you have access to.

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 carName as a global variable, even if it is executed inside a function.

### Example

```
// code here can use carName

function myFunction() {
  carName = "Volvo";

  // code here can use carName
}
```

## 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.

## 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

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

function myFunction() {
```

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

## Did You Know?



Your global variables, or functions, can overwrite window variables or functions.  
Anyone, including the window object, can overwrite your global variables or 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='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 it's own element (using **this.innerHTML**):

## Example

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



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

## Example

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

## Common HTML Events

Here is a list of some common HTML events:

Event	Description
<b>onchange</b>	An HTML element has been changed
<b>onclick</b>	The user clicks an HTML element
<b>onmouseover</b>	The user moves the mouse over an HTML element
<b>onmouseout</b>	The user moves the mouse away from an HTML element
<b>onkeydown</b>	The user pushes a keyboard key
<b>onload</b>	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 input 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 Strings

JavaScript strings are used for storing and manipulating text.

## 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 lists of special characters that can be added to a text string with the backslash sign:

Code	Outputs
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\\</code>	backslash
<code>\n</code>	new line
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\b</code>	backspace
<code>\f</code>	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!";
```

However, 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");
```

```
// type of x will return String  
// type of y will return Object
```



Don't create strings as objects. It slows down execution speed, and produces nasty side effects:

When using the `==` equality operator, equal strings looks 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:

## Example

```
var x = "John";  
var y = new String("John");  
  
// (x === y) is false because x and y have different types
```

The `===` operator expects equality in both type and value.

## String Properties and Methods

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 Properties

Property	Description
<b>constructor</b>	Returns the function that created the String object's prototype
<b>length</b>	Returns the length of a string
<b>prototype</b>	Allows you to add properties and methods to an object

## String Methods

Method	Description
<b>charAt()</b>	Returns the character at the specified index (position)
<b>charCodeAt()</b>	Returns the Unicode of the character at the specified index
<b>concat()</b>	Joins two or more strings, and returns a copy of the joined strings
<b>fromCharCode()</b>	Converts Unicode values to characters
<b>indexOf()</b>	Returns the position of the first found occurrence of a specified value in a string
<b>lastIndexOf()</b>	Returns the position of the last found occurrence of a specified value in a string
<b>localeCompare()</b>	Compares two strings in the current locale
<b>match()</b>	Searches a string for a match against a regular expression, and returns the matches
<b>replace()</b>	Searches a string for a value and returns a new string with the value replaced
<b>search()</b>	Searches a string for a value and returns the position of the match
<b>slice()</b>	Extracts a part of a string and returns a new string
<b>split()</b>	Splits a string into an array of substrings
<b>substr()</b>	Extracts a part of a string from a start position through a number of characters
<b>substring()</b>	Extracts a part of a string between two specified positions
<b>toLocaleLowerCase()</b>	Converts a string to lowercase letters, according to the host's locale
<b>toLocaleUpperCase()</b>	Converts a string to uppercase letters, according to the host's locale
<b>toLowerCase()</b>	Converts a string to lowercase letters
<b>toString()</b>	Returns the value of a String object
<b>toUpperCase()</b>	Converts a string to uppercase letters
<b>trim()</b>	Removes whitespace from both ends of a string
<b>valueOf()</b>	Returns the primitive value of a String object

# JavaScript String Methods

String methods help you to work with strings.

## 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 does 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 cannot 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.

## 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
```

# JavaScript Numbers

JavaScript has only one type of number.

Numbers can be written with, or without, decimals.

## JavaScript Numbers

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 value is not a 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
```

NaN is a number: `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
```



Don't create Number objects. They slow down execution speed, and produce nasty side effects:

## Example

```
var x = 123;  
var y = new Number(123);  
(x === y) // is false because x is a number and y is an object.
```

## Number Properties and Methods

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.

## Number Properties

Property	Description
MAX_VALUE	Returns the largest number possible in JavaScript
MIN_VALUE	Returns the smallest number possible in JavaScript
NEGATIVE_INFINITY	Represents negative infinity (returned on overflow)
NaN	Represents a "Not-a-Number" value
POSITIVE_INFINITY	Represents infinity (returned on overflow)

## Example

```
var x = Number.MAX_VALUE;
```

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

```
var x = 6;  
var y = x.MAX_VALUE; // y becomes undefined
```



Number methods are covered in the next chapter

# JavaScript Number Methods

Number methods help you to work with numbers.

## Global Methods

JavaScript global functions 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

## Number Methods

JavaScript number methods are methods that can be used on numbers:

Method	Description
toString()	Returns a number as a string
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
valueOf()	Returns a number as a number



All number methods return a new variable. They do not change the original variable.

## The `toString()` Method

**`toString()`** returns a number as a string.

All number methods can be used on any type of numbers (literals, variables, or expressions):

### Example

```
var x = 123;  
x.toString();      // returns 123 from variable x  
(123).toString();  // returns 123 from literal 123  
(100 + 23).toString(); // returns 123 from expression 100 + 23
```

## The `toExponential()` Method

**`toExponential()`** returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of character behind the decimal point:

### Example

```
var x = 9.656;  
x.toExponential(2); // returns 9.66e+0  
x.toExponential(4); // returns 9.6560e+0  
x.toExponential(6); // returns 9.656000e+0
```

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

```
var x = 9.656;  
x.toFixed(0);    // returns 10  
x.toFixed(2);    // returns 9.66  
x.toFixed(4);    // returns 9.6560  
x.toFixed(6);    // returns 9.656000
```



toFixed(2) is perfect for working with money.

## The toPrecision() Method

**toPrecision()** returns a string, with a number written with a specified length:

### Example

```
var x = 9.656;  
x.toPrecision(); // returns 9.656  
x.toPrecision(2); // returns 9.7  
x.toPrecision(4); // returns 9.656  
x.toPrecision(6); // returns 9.65600
```

## Converting Variables to Numbers

There are 3 JavaScript functions 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.

## The Number() Method

**Number()**, can be used to convert JavaScript variables to numbers:

### Example

```
x = true;
Number(x);    // returns 1
x = false;
Number(x);    // returns 0
x = new Date();
Number(x);    // returns 1404568027739
x = "10"
Number(x);    // returns 10
x = "10 20"
Number(x);    // returns NaN
```

## The parseInt() Method

**parseInt()** parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

### Example

```
parseInt("10");    // returns 10
parseInt("10.33"); // returns 10
parseInt("10 20 30"); // returns 10
parseInt("10 years"); // returns 10
parseInt("years 10"); // returns NaN
```

If the number cannot be converted, NaN (Not a Number) is returned.

## The parseFloat() Method

**parseFloat()** parses a string and returns a number. Spaces are allowed. Only the first number is returned:

### Example

```
parseFloat("10");    // returns 10
parseFloat("10.33"); // returns 10.33
parseFloat("10 20 30"); // returns 10
parseFloat("10 years"); // returns 10
parseFloat("years 10"); // returns NaN
```

If the number cannot be converted, NaN (Not a Number) is returned.

## The valueOf() Method

**valueOf()** returns a number as a number.

### Example

```
var x = 123;
x.valueOf();    // returns 123 from variable x
(123).valueOf(); // returns 123 from literal 123
(100 + 23).valueOf(); // returns 123 from expression 100 + 23
```

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.



In JavaScript, all data types have a `valueOf()` and a `toString()` method.

## Complete JavaScript Number Reference

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

# 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);    // returns -8
```

### Example

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

## Math.random()

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

### Example

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

## 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
<b>abs(x)</b>	Returns the absolute value of x
<b>acos(x)</b>	Returns the arccosine of x, in radians
<b>asin(x)</b>	Returns the arcsine of x, in radians
<b>atan(x)</b>	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
<b>atan2(y,x)</b>	Returns the arctangent of the quotient of its arguments
<b>ceil(x)</b>	Returns x, rounded upwards to the nearest integer
<b>cos(x)</b>	Returns the cosine of x (x is in radians)
<b>exp(x)</b>	Returns the value of E <sup>x</sup>
<b>floor(x)</b>	Returns x, rounded downwards to the nearest integer
<b>log(x)</b>	Returns the natural logarithm (base E) of x
<b>max(x,y,z,...,n)</b>	Returns the number with the highest value
<b>min(x,y,z,...,n)</b>	Returns the number with the lowest value
<b>pow(x,y)</b>	Returns the value of x to the power of y
<b>random()</b>	Returns a random number between 0 and 1
<b>round(x)</b>	Rounds x to the nearest integer
<b>sin(x)</b>	Returns the sine of x (x is in radians)
<b>sqrt(x)</b>	Returns the square root of x
<b>tan(x)</b>	Returns the tangent of an angle

# JavaScript Dates

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

## 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, a minute, a second, and a millisecond.

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>
```

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.

The **next chapter**, of this tutorial, covers the date object's methods.

## 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, not dynamic. The computer time is ticking, but date objects, once created, are not.

# JavaScript Date Methods

Date methods let you get and set date values (years, months, days, 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
<b>getDate()</b>	Get the day as a number (1-31)
<b>getDay()</b>	Get the weekday as a number (0-6)
<b>getFullYear()</b>	Get the four digit year (yyyy)
<b>getHours()</b>	Get the hour (0-23)



<b>getMilliseconds()</b>	Get the milliseconds (0-999)
<b>getMinutes()</b>	Get the minutes (0-59)
<b>getMonth()</b>	Get the month (0-11)
<b>getSeconds()</b>	Get the seconds (0-59)
<b>getTime()</b>	Get the time (milliseconds since January 1, 1970)

## The getTime() Method

**getTime()** returns the the number of milliseconds since 01.01.1970:

### 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>
```

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
<b><code>setDate()</code></b>	Set the day as a number (1-31)
<b><code>setFullYear()</code></b>	Set the year (optionally month and day yyyy.mm.dd)
<b><code>setHours()</code></b>	Set the hour (0-23)
<b><code>setMilliseconds()</code></b>	Set the milliseconds (0-999)
<b><code>setMinutes()</code></b>	Set the minutes (0-59)
<b><code>setMonth()</code></b>	Set the month (0-11)
<b><code>setSeconds()</code></b>	Set the seconds (0-59)
<b><code>setTime()</code></b>	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 an input value (or any 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.

## JavaScript Arrays

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

### Displaying Arrays

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

## Example

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

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

The first line (in the script) creates an array named cars.

The second line "finds" the element with id="demo", and "displays" the array in the "innerHTML" of it.

## Try it Yourself

Create an array, and assign values to it:

## Example

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

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.

## 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, ...];
```

Example:

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

## 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 access the value of the first element in `myCars`:

```
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.

## You Can Have Different Objects in One Array

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;
```



## 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];
```

[Try it Yourself »](#)

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

### Object:

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

## 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 in cars
var y = cars.sort();    // The sort() method sort cars in alphabetical order
```

Array methods are covered in the next chapter.

## 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 to use 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 standard for loop:

## Example

```
var index;
var fruits = ["Banana", "Orange", "Apple", "Mango"];
for (index = 0; index < fruits.length; index++) {
    text += fruits[index];
}
```

## 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** 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"
```

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object, and all array methods and properties will produce undefined or incorrect results.

## Example:

```
var person = [];
person["firstName"] = "John";
```

```
person["lastName"] = "Doe";  
person["age"] = 46;  
var x = person.length;      // person.length will return 0  
var y = person[0];          // person[0] will return undefined
```

## The Difference Between Arrays and Objects?

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.



Arrays are a special kind of objects, with numbered indexes.

## When to Use Arrays? When to use Objects?

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

## 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 complicates your code and produces nasty side effects:

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

The typeof operator returns object because a JavaScript array is an object.

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

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

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

Or more precisely: it returns true if the object prototype of the argument is "[object array]".

# JavaScript Array Methods

The strength of JavaScript arrays lies in the array methods.

## Converting Arrays to Strings

In JavaScript, all objects have the `valueOf()` and `toString()` methods.

The **`valueOf()`** method is the default behavior for an array. It returns an array as a string:

### Example

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

For JavaScript arrays, `valueOf()` and **`toString()`** are equal.

### Example

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

The **`join()`** method also joins all array elements into a string.

It behaves just like `toString()`, but you can specify the separator:

### Example

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

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
</script>
```

## 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.

The **pop()** method removes the last element from an array:

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();           // Removes the last element ("Mango") from fruits
```

The **push()** method adds a new element to an array (at the end):



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

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");    // Adds a new element ("Kiwi") to fruits
```

The `pop()` method returns the string that was "popped out".

The `push()` method returns the new array length.

## Shifting Elements

Shifting is equivalent to popping, working on the first element instead of the last.

The **shift()** method removes the first element of an array, and "shifts" all other elements one place down.

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();      // Removes the first element "Banana" from fruits
```

The **unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements:

### Example

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

The shift() method returns the string that was "shifted out".

The unshift() method returns the new array length.

## Changing Elements

Array elements are accessed using their **index number**:

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[0] = "Kiwi";    // Changes the first element of fruits to "Kiwi"
```



The length property provides an easy way to append a new element to an array:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Kiwi";    // Appends "Kiwi" to fruit
```

## Deleting Elements

Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator **delete**:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];    // Changes the first element in fruits to undefined
```



Using **delete** on array elements leaves undefined holes in the array. Use `pop()` or `splice()` instead.

---

## Splicing an Array

The **splice()** method can be used to add new items to an array:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

---

## Using splice() to Remove Elements

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0,1);    // Removes the first element of fruits
```

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.

---

## Sorting an Array

The **sort()** method sorts an array alphabetically:

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();      // Sorts the elements of fruits
```

---

## Reversing an Array

The **reverse()** method reverses the elements in an array.

You can use it to sort an array in descending order:

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();      // Sorts the elements of fruits
fruits.reverse();   // Reverses the order of the elements
```

---

## 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**:

### Example

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a-b});
```

Use the same trick to sort an array descending:

## Example

```
var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b-a});
```

---

## The Compare Function

The purpose of the compare function is to define an alternative sort order.

The compare function should return a negative, zero, or positive value, depending on the arguments:

```
function(a, b){return a-b}
```

When the `sort()` function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

### Example:

When comparing 40 and 100, the `sort()` method calls the `compare function(40,100)`.

The function calculates  $40-100$ , and returns  $-60$  (a negative value).

The sort function will sort 40 as a value lower than 100.

---

## Find the Highest (or Lowest) Value

How to find the highest value in an array?

### Example

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b-a});
// now points[0] contains the highest value
```

And the lowest:

## Example

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a-b});
// now points[0] contains the lowest value
```

---

## Joining Arrays

The **concat()** method creates a new array by concatenating two arrays:

### Example

```
var myGirls = ["Cecilie", "Lone"];
var myBoys = ["Emil", "Tobias", "Linus"];
var myChildren = myGirls.concat(myBoys); // Concatenates (joins) myGirls and myBoys
```

The `concat()` method can take any number of array arguments:

### Example

```
var arr1 = ["Cecilie", "Lone"];
var arr2 = ["Emil", "Tobias", "Linus"];
var arr3 = ["Robin", "Morgan"];
var myChildren = arr1.concat(arr2, arr3); // Concatenates arr1 with arr2 and arr3
```

---

## Slicing an Array

The **slice()** method slices out a piece of an array:

### Example

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1,3);
```

## JavaScript Booleans

A JavaScript Boolean represents one of two values: **true** or **false**.

### Boolean Values

Very often, in programming, you will need a data type that can only have one of two values, like

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, JavaScript has a **Boolean** data type. It can only take the values **true** or **false**.

---

### The Boolean() Function

You can use the Boolean() function to find out if an expression (or a variable) is true:

### Example

```
Boolean(10 > 9)    // returns true
```

[Try it Yourself »](#)

Or even easier:

## Example

```
(10 > 9)      // also returns true
10 > 9        // also returns true
```

---

## 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 fundament for JavaScript comparisons and conditions.

---

## Everything With a Real Value is True

### Examples

100

3.14

-15

"Hello"

"false"

7 + 1 + 3.14

5 < 6

---

## Everything Without a Real Value is False

The Boolean value of **0** (zero) is **false**:

```
var x = 0;  
Boolean(x);    // returns false
```

The Boolean value of **-0** (minus zero) is **false**:

```
var x = -0;  
Boolean(x);    // returns false
```

The Boolean value of **""** (empty string) is **false**:

```
var x = "";  
Boolean(x);    // returns false
```

The Boolean value of **undefined** is **false**:

```
var x;  
Boolean(x);    // returns false
```



The Boolean value of **null** is **false**:

```
var x = null;  
Boolean(x);    // returns false
```

The Boolean value of **false** is (you guessed it) **false**:

```
var x = false;  
Boolean(x);    // returns false
```

The Boolean value of **NaN** is **false**:

```
var x = 10 / "H";  
Boolean(x);    // returns false
```

---

## Boolean Properties and Methods

Primitive values, like true and false, 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.

# 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	Try it
==	equal to	x == 8	false	<a href="#">Try it »</a>
		x == 5	true	<a href="#">Try it »</a>
===	equal value and equal type	x === "5"	false	<a href="#">Try it »</a>
		x === 5	true	<a href="#">Try it »</a>
!=	not equal	x != 8	true	<a href="#">Try it »</a>
!==	not equal value or not equal type	x !== "5"	true	<a href="#">Try it »</a>
		x !== 5	false	<a href="#">Try it »</a>
>	greater than	x > 8	false	<a href="#">Try it »</a>
<	less than	x < 8	true	<a href="#">Try it »</a>
>=	greater than or equal to	x >= 8	false	<a href="#">Try it »</a>
<=	less than or equal to	x <= 8	true	<a href="#">Try it »</a>

---

## 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";
```

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
<b>&amp;&amp;</b>	and	(x < 10 && y > 1) is true
<b>  </b>	or	(x == 5    y == 5) is false
<b>!</b>	not	!(x == y) is true

---

## Conditional Operator

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

### Syntax

*variablename* = (condition) ? value1:value2

### Example

```
voteable = (age < 18) ? "Too young":"Old enough";
```

If the variable age is a value below 18, the value of the variable voteable will be "Too young", otherwise the value of voteable will be "Old enough".

---

## JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number.

The result is converted back to a JavaScript number.

## Example

```
x = 5 & 1;
```

The result in x:

1

Operator	Description	Example	Same as	Result	Decimal
&	AND	x = 5 & 1	0101 & 0001	0001	1
	OR	x = 5   1	0101   0001	0101	5
~	NOT	x = ~ 5	~0101	1010	10
^	XOR	x = 5 ^ 1	0101 ^ 0001	0100	4
<<	Left shift	x = 5 << 1	0101 << 1	1010	10
>>	Right shift	x = 5 >> 1	0101 >> 1	0010	2

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



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

[illegible]

# JavaScript If...Else Statements

Conditional statements are used to perform different actions based on different conditions.

## Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
  - Use **else** to specify a block of code to be executed, if the same condition is false
  - Use **else if** to specify a new condition to test, if the first condition is false
  - Use **switch** to specify many alternative blocks of code to be executed
- 

## The if Statement

Use the **if** statement to specify a block of JavaScript code to be executed if a condition is true.

### Syntax

```
if (condition) {  
    block of code to be executed if the condition is true  
}
```



Note that **if** is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

## Example

Make a "Good day" greeting if the time is less than 20:00:

```
if (time < 20) {  
    greeting = "Good day";  
}
```

The result of greeting will be:

Good day

---

## The else Statement

Use the **else** statement to specify a block of code to be executed if the condition is false.

```
if (condition) {  
    block of code to be executed if the condition is true  
} else {  
    block of code to be executed if the condition is false  
}
```

## Example

If the time is less than 20:00, create a "Good day" greeting, otherwise "Good evening":

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

The result of greeting will be:

Good day

---

## The else if Statement

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

### Syntax

```
if (condition1) {  
    block of code to be executed if condition1 is true
```

```
} else if (condition2) {  
    block of code to be executed if the condition1 is false and condition2 is true  
} else {  
    block of code to be executed if the condition1 is false and condition2 is false  
}
```

## Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

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

The result of greeting will be:

Good day

# JavaScript Switch Statement

The switch statement is used to perform different action based on different conditions.

## The JavaScript Switch Statement

Use the switch statement to select one of many blocks of code to be executed.

## Syntax

```
switch(expression) {  
  case n:  
    code block  
    break;  
  case n:  
    code block  
    break;  
  default:  
    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.

## Example

Use today's weekday number to calculate weekday name: (Sunday=0, Monday=1, Tuesday=2, ...)

```
switch (new Date().getDay()) {  
  case 0:  
    day = "Sunday";  
    break;  
  case 1:  
    day = "Monday";  
    break;  
  case 2:  
    day = "Tuesday";  
    break;  
  case 3:  
    day = "Wednesday";  
    break;  
  case 4:  
    day = "Thursday";  
    break;
```



```
case 5:
    day = "Friday";
    break;
case 6:
    day = "Saturday";
    break;
}
```

The result of day will be:

Monday

---

## The break Keyword

When the JavaScript code interpreter reaches a **break** keyword, it breaks out of the switch block.

This will stop the execution of more execution of code and/or case testing inside the block.



When a match is found, and the job is done, it's time for a break.  
There is no need for more testing.

---

## The default Keyword

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

### Example

If today is neither Saturday nor Sunday, write a default message:

```
switch (new Date().getDay()) {
    case 6:
```

```
        text = "Today is Saturday";
        break;
    case 0:
        text = "Today is Sunday";
        break;
    default:
        text = "Looking forward to the Weekend";
}
```

The result of text will be:

Looking forward to the Weekend

---

## Common Code and Fall-Through

Sometimes, in a switch block, you will want different cases to use the same code, or fall-through to a common default.

Note from the next example, that cases can share the same code block, and that the default case does not have to be the last case in a switch block:

### Example

```
switch (new Date().getDay()) {
    case 1:
    case 2:
    case 3:
    default:
        text = "Looking forward to the Weekend";
        break;
    case 4:
    case 5:
        text = "Soon it is Weekend";
        break;
    case 0:
    case 6:
```

```
    text = "It is Weekend";  
}
```

# JavaScript For Loop

Loops can execute a block of code a number of times.

## JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

### Instead of writing:

```
text += cars[0] + "<br>";  
text += cars[1] + "<br>";  
text += cars[2] + "<br>";  
text += cars[3] + "<br>";  
text += cars[4] + "<br>";  
text += cars[5] + "<br>";
```

### You can write:

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

---

# Different Kinds of Loops

JavaScript supports different kinds of loops:

- **for** - loops through a block of code a number of times
  - **for/in** - loops through the properties of an object
  - **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 is often the tool you will use when you want to create a loop.

The for loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {  
    code block to be executed  
}
```

**Statement 1** is executed before the loop (the code block) starts.

**Statement 2** defines the condition for running the loop (the code block).

**Statement 3** is executed each time after the loop (the code block) has been executed.

## Example

```
for (i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}
```

From the example above, you can read:

Statement 1 sets a variable before the loop starts (var 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 initiate the variable used in the loop (var 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:

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

And you can omit statement 1 (like when your values are set before the loop starts):

### Example:

```
var i = 2;  
var len = cars.length;  
var text = "";  
for (; i < len; i++) {  
    text += cars[i] + "<br>";  
}
```

---

## 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 increases 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--), or larger increment (i = i + 15), or anything else.

Statement 3 can also be omitted (like when you increment your values inside the loop):

### Example:

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

---

## The For/In Loop

The JavaScript for/in statement loops through the properties of an object:

### Example

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

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

# JavaScript While Loop

Loops can execute a block of code as long as a specified condition is true.

---

## The While Loop

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

### Syntax

```
while (condition) {  
    code block to be executed  
}
```

### Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

## Example

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



If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

---

## The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

### Syntax

```
do {  
    code block to be executed  
}  
while (condition);
```

### Example

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

## Example

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

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

---



## Comparing For and While

If you have read the previous chapter, about the for loop, you will discover that a while loop is much the same as a for loop, with statement 1 and statement 3 omitted.

The loop in this example uses a **for loop** to collect the car names from the cars array:

### Example

```
cars = ["BMW","Volvo","Saab","Ford"];
var i = 0;
var text = "";

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

The loop in this example uses a **while loop** to collect the car names from the cars array:

### Example

```
cars = ["BMW","Volvo","Saab","Ford"];
var i = 0;
var text = "";

while (cars[i]) {
    text += cars[i] + "<br>";
    i++;
}
```

## 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.

The **break statement** breaks the loop and continues executing the code after the loop (if any):

### Example

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

Since the if statement has only one single line of code, the braces can be omitted:

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

---

## The Continue Statement

The **continue statement** breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 3:

## Example

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

---

## JavaScript Labels

As you have already seen, in the chapter about the switch statement, JavaScript statements can be labeled.

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 inside a loop.

The break statement, without a label reference, can only be used inside a loop or a switch.

With a label reference, it can be used to "jump out of" any JavaScript code block:

## Example

```
cars = ["BMW", "Volvo", "Saab", "Ford"];
list: {
  text += cars[0] + "<br>";
  text += cars[1] + "<br>";
  text += cars[2] + "<br>";
  text += cars[3] + "<br>";
  break list;
  text += cars[4] + "<br>";
  text += cars[5] + "<br>";
}
```

# JavaScript typeof, null, and undefined

JavaScript typeof, null, undefined, valueOf().

---

## 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
typeof {name:'John', age:34} // Returns object
```



In JavaScript, an array is a special type of object. Therefore typeof [1,2,3,4] returns object.

---

# 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
```

---

# Undefined

In JavaScript, **undefined** is a variable with **no value**.

The **typeof** a variable with no value 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
```

---

## Difference Between Undefined and Null

```
typeof undefined    // undefined
typeof null         // object
null === undefined  // false
null == undefined   // true
```

## JavaScript Type Conversion

Number() converts to a Number, String() converts to a String, Boolean() converts to a Boolean.

---

## JavaScript Data Types

In JavaScript there are 5 different data types that can contain values:

- string
- number
- boolean
- object
- function

There are 3 types of objects:

- Object
- Date
- Array

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

```
typeof "John"           // Returns string
typeof 3.14              // Returns number
typeof NaN               // Returns number
typeof false             // Returns boolean
typeof [1,2,3,4]         // Returns object
typeof {name:'John', age:34} // Returns object
typeof new Date()         // Returns object
typeof function () {}    // Returns function
typeof myCar              // Returns undefined (if myCar is not declared)
typeof null              // Returns object
```

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

You cannot use **typeof** to define if an object is an JavaScript Array or a JavaScript Date.

---

## The constructor Property

The **constructor** property returns the constructor function for all JavaScript variables.

## Example

```
"John".constructor      // Returns function String() { [native code] }
(3.14).constructor      // Returns function Number() { [native code] }
false.constructor       // Returns function Boolean() { [native code] }
[1,2,3,4].constructor   // Returns function Array() { [native code] }
{name:'John', age:34}.constructor // Returns function Object() { [native code] }
new Date().constructor   // Returns function Date() { [native code] }
function () {}.constructor // Returns function Function(){ [native code] }
```

You can check the constructor property to find out if an object is an Array (contains the word "Array"):

## Example

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

You can check the constructor property to find out if an object is a Date (contains the word "Date"):

## Example

```
function isDate(myDate) {
    return myDate.constructor.toString().indexOf("Date") > -1;
}
```

---



# 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 Numbers to Strings

The global method **String()** can convert numbers to strings.

It can be used on any type of numbers, literals, variables, or expressions:

### Example

```
String(x)      // returns a string from a number variable x
String(123)    // returns a string from a number literal 123
String(100 + 23) // returns a string from a number from an expression
```

The Number method **toString()** does the same.

### Example

```
x.toString()
(123).toString()
(100 + 23).toString()
```

Method	Description
<b>toExponential()</b>	Returns a string, with a number rounded and written using exponential notation.
<b>toFixed()</b>	Returns a string, with a number rounded and written with a specified number of decimals.
<b>toPrecision()</b>	Returns a string, with a number written with a specified length

---

## 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"
```

---

## 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)
```

Method	Description
<b>getDate()</b>	Get the day as a number (1-31)
<b>getDay()</b>	Get the weekday a number (0-6)
<b>getFullYear()</b>	Get the four digit year (yyyy)
<b>getHours()</b>	Get the hour (0-23)
<b>getMilliseconds()</b>	Get the milliseconds (0-999)
<b>getMinutes()</b>	Get the minutes (0-59)

<b>getMonth()</b>	Get the month (0-11)
<b>getSeconds()</b>	Get the seconds (0-59)
<b>getTime()</b>	Get the time (milliseconds since January 1, 1970)

---

## 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
```

Method	Description
<b>parseFloat()</b>	Parses a string and returns a floating point number
<b>parseInt()</b>	Parses a string and returns an integer

---

## The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

### Example

```
var y = "5"; // y is a string
var x = + y; // x is a number
```

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a number):

## Example

```
var y = "John"; // y is a string
var x = + y;    // x is a number (NaN)
```

---

## Converting Booleans to Numbers

The global method **Number()** can also convert booleans to numbers.

```
Number(false) // returns 0
Number(true)  // returns 1
```

---

## 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
```

---

## 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" + 1 // returns "51"    because 1 is converted to "1"
"5" - 1 // returns 4       because "5" is converted to 5
```

---

## 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 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:

```
var patt = /w3schools/i
```

Example explained:

**/w3schools/i** is a regular expression.

**w3schools** is a pattern (to be used in a search).

**i** is a modifier (modifies the search to be case-insensitive).

---

## Using String Methods

In JavaScript, regular expressions are often used with the two **string methods**: `search()` and `replace()`.

**The `search()` method** uses an expression to search for a match, and returns the position of the match.

**The `replace()` method** returns a modified string where the pattern is replaced.

---

## Using String `search()` With a Regular Expression

## Example

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

```
var str = "Visit W3Schools";  
var n = str.search(/w3schools/i);
```

The result in n will be:

6

## Using String search() With String

The search method will also accept a string as search argument. The string argument will be converted to a regular expression:

## Example

Use a string to do a search for "W3schools" in a string:

```
var str = "Visit W3Schools!";  
var n = str.search("W3Schools");
```

## Use String replace() With a Regular Expression

## Example

Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

```
var str = "Visit Microsoft!";  
var res = str.replace(/microsoft/i, "W3Schools");
```

The result in res will be:

Visit W3Schools!

## Using String replace() With a String

The replace() method will also accept a string as search argument:

```
var str = "Visit Microsoft!";  
var res = str.replace("Microsoft", "W3Schools");
```

---

## Did You Notice?



Regular expression arguments (instead of string arguments) can be used in the methods above.

Regular expressions can make your search much more powerful (case insensitive for example).

---

## Regular Expression Modifiers

**Modifiers** can be used to perform case-insensitive more global searches:

Modifier	Description
<b>i</b>	Perform case-insensitive matching
<b>g</b>	Perform a global match (find all matches rather than stopping after the first match)
<b>m</b>	Perform multiline matching



# Regular Expression Patterns

**Brackets** are used to find a range of characters:

Expression	Description
<b>[abc]</b>	Find any of the characters between the brackets
<b>[0-9]</b>	Find any of the digits between the brackets
<b>(x y)</b>	Find any of the alternatives separated with

**Metacharacters** are characters with a special meaning:

Metacharacter	Description
<b>\d</b>	Find a digit
<b>\s</b>	Find a whitespace character
<b>\b</b>	Find a match at the beginning or at the end of a word
<b>\uxxxx</b>	Find the Unicode character specified by the hexadecimal number xxxx

**Quantifiers** define quantities:

Quantifier	Description
<b>n+</b>	Matches any string that contains at least one <i>n</i>
<b>n*</b>	Matches any string that contains zero or more occurrences of <i>n</i>
<b>n?</b>	Matches any string that contains zero or one occurrences of <i>n</i>

---

## Using the RegExp Object

In JavaScript, the RegExp object is a regular expression object with predefined properties and methods.

---

### Using test()

The test() method is a RegExp expression method.

It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":

## Example

```
var patt = /e/;  
patt.test("The best things in life are free!");
```

Since there is an "e" in the string, the output of the code above will be:

```
true
```

You don't have to put the regular expression in a variable first. The two lines above can be shortened to one:

```
/e/.test("The best things in life are free!")
```

---

## Using exec()

The `exec()` method is a RegExp expression method.

It searches a string for a specified pattern, and returns the found text.

If no match is found, it returns *null*.

The following example searches a string for the character "e":

### Example 1

```
/e/.exec("The best things in life are free!");
```

Since there is an "e" in the string, the output of the code above will be:

```
e
```

# 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

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

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

<script>
try {
    adddler("Welcome guest!");
}
catch(err) {
    document.getElementById("demo").innerHTML = err.message;
}
</script>

</body>
</html>
```

In the example above we have made a typo in the code (in the **try block**).

The **catch block** catches the error, and executes 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 error.

---

## The throw Statement

The **throw** statement allows you to create a custom error.

The technical term for this is: **throw an exception**.

The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big"; // throw a text
throw 500;       // throw a number
```

If you use **throw** together with **try** and **catch**, you can control program flow and generate custom error messages.

---

## Input Validation Example

This example examines input. If the value is wrong, an exception (err) is thrown.

The exception (err) is caught by the catch statement and a custom error message is displayed:

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

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="message"></p>

<script>
function myFunction() {
  var message, x;
  message = document.getElementById("message");
  message.innerHTML = "";
  x = document.getElementById("demo").value;
  try {
    x = Number(x);
    if(x == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
```

```
    }  
    catch(err) {  
        message.innerHTML = "Input " + err;  
    }  
}  
</script>  
  
</body>  
</html>
```

---

## The finally Statement

The **finally** statement lets you execute code, after try and catch, regardless of the result:

```
try {  
    Block of code to try  
}  
catch(err) {  
    Block of code to handle errors  
}  
finally {  
    Block of code to be executed regardless of the try / catch result  
}
```

## Example

```
function myFunction() {  
    var message, x;  
    message = document.getElementById("message");  
    message.innerHTML = "";  
    x = document.getElementById("demo").value;  
    try {  
        x = Number(x);  
        if(x == "") throw "is empty";  
        if(isNaN(x)) throw "is not a number";  
        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 = "";
    }
}
```

# JavaScript Debugging

You will soon get lost, writing JavaScript code without a debugger.

---

## JavaScript Debugging

It is difficult to write JavaScript code without a debugger.

Your code might contain syntax errors, or logical errors, that are difficult to diagnose.

Often, when JavaScript code contains errors, nothing will happen. There are no error messages, and you will get no indications where to search for errors.



Normally, errors will happen, every time you try to write some new JavaScript code.

---

## JavaScript Debuggers

Searching for errors in programming code is called code debugging.

Debugging is not easy. But fortunately, all modern browsers have a built-in debugger.

Built-in debuggers can be turned on and off, forcing errors to be reported to the user.

With a debugger, you can also set breakpoints (places where code execution can be stopped), and examine variables while the code is executing.

Normally, otherwise follow the steps at the bottom of this page, you activate debugging in your browser with the F12 key, and select "Console" in the debugger menu.

---

## The console.log() Method

If your browser supports debugging, you can use console.log() to display JavaScript values in the debugger window:

### Example

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

<h1>My First Web Page</h1>

<script>
a = 5;
b = 6;
c = a + b;
console.log(c);
</script>

</body>
</html>
```

---

## Setting Breakpoints

In the debugger window, you can set breakpoints in the JavaScript code.

At each breakpoint, JavaScript will stop executing, and let you examine JavaScript values.



After examining values, you can resume the execution of code (typically with a play button).

---

## The debugger Keyword

The **debugger** keyword stops the execution of JavaScript, and calls (if available) the debugging function.

This has the same function as setting a breakpoint in the debugger.

If no debugging is available, the debugger statement has no effect.

With the debugger turned on, this code will stop executing before it executes the third line.

## Example

```
var x = 15 * 5;  
debugger;  
document.getElementById("demo").innerHTML = x;
```

---

## Major Browsers' Debugging Tools

Normally, you activate debugging in your browser with F12, and select "Console" in the debugger menu.

Otherwise follow these steps:

### Chrome

- Open the browser.
- From the menu, select tools.
- From tools, choose developer tools.
- Finally, select Console.

## Firefox Firebug

- Open the browser.
- Go to the web page:  
<http://www.getfirebug.com>.
- Follow the instructions how to:  
install Firebug

## Internet Explorer

- Open the browser.
- From the menu, select tools.
- From tools, choose developer tools.
- Finally, select Console.

## Opera

- Open the browser.
- Go to the webpage:  
<http://dev.opera.com/articles/view/opera-developer-tools>.
- Follow the instructions how to:  
add a Developer Console button to your toolbar.

## Safari Firebug

- Open the browser.
- Go to the webpage:  
<http://extensions.apple.com>.
- Follow the instructions how to:  
install Firebug Lite.

## Safari Develop Menu

- Go to Safari, Preferences, Advanced in the main menu.
  - Check "Enable Show Develop menu in menu bar".
  - When the new option "Develop" appears in the menu:  
Choose "Show Error Console".
-

## Did You Know?



Debugging is the process of testing, finding, and reducing bugs (errors) in computer programs.

The first known computer bug was a real bug (an insect), stuck in the electronics.

# JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top.

---

## JavaScript Declarations are Hoisted

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

**Example 1** gives the same result as **Example 2**:

### Example 1

```
x = 5; // Assign 5 to x
```

```
elem = document.getElementById("demo"); // Find an element  
elem.innerHTML = x;                      // Display x in the element
```

```
var x; // Declare x
```

### Example 2

```
var x; // Declare x  
x = 5; // Assign 5 to x
```

```
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                      // Display x in the element
```

To understand this, you have to understand the term "hoisting".

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

---

## JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

**Example 1** does **not** give the same result as **Example 2**:

### Example 1

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y
```

### Example 2

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

var y = 7; // Initialize y
```

Does it make sense that y is undefined in the last example?

This is because only the declaration (var y), not the initialization (=7) is hoisted to the top.

Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

Example 2 is the same as writing:

## Example

```
var x = 5; // Initialize x
var y;    // Declare y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

y = 7; // Assign 7 to y
```

---

## Declare Your Variables At the Top !

Hoisting is (to many developers) an unknown or overlooked behavior of JavaScript.

If a developer doesn't understand hoisting, programs may contain bugs (errors).

To avoid bugs, always declare all variables at the beginning of every scope.

Since this is how JavaScript interprets the code, it is always a good rule.



JavaScript in strict mode does not allow variables to be used if they are not declared. Study "using strict"; in the next chapter.

## JavaScript Use Strict

**"use strict";** Defines that JavaScript code should be executed in "strict mode".

---

# The "use strict" Directive

The "use strict" directive is new in JavaScript 1.8.5 (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.



Strict mode is supported in:

Internet Explorer from version 10. Firefox from version 4.

Chrome from version 13. Safari from version 5.1. Opera from version 12.

---

## Declaring Strict Mode

Strict mode is declared by adding "use strict"; to the beginning of a JavaScript file, or a JavaScript function.

Declared at the beginning of a JavaScript file, it has global scope (all code will execute in strict mode).

Declared inside a function, it has local scope (only the code inside the function is in strict mode).

Global declaration:

```
"use strict";  
x = 3.14;    // This will cause an error  
myFunction(); // This will also cause an error
```

```
function myFunction() {  
    x = 3.14;  
}
```

Local declaration:

```
x = 3.14;    // This will not cause an error.  
myFunction(); // This will cause an error
```

```
function myFunction() {  
    "use strict";  
    x = 3.14;  
}
```

---

## 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 (property or object) without declaring it, is not allowed:

```
"use strict";  
x = 3.14;           // This will cause an error (if x has not been declared)
```

Deleting a variable, a function, or an argument, is not allowed.

```
"use strict";  
x = 3.14;  
delete x;           // This will cause an error
```

Defining a property more than once, is not allowed:

```
"use strict";  
var x = {p1:10, p1:20}; // This will cause an error
```

Duplicating a parameter name is not allowed:

```
"use strict";  
function x(p1, p1) {}; // This will cause an error
```

Octal numeric literals and escape characters are not allowed:

```
"use strict";  
var x = 010;         // This will cause an error  
var y = \010;        // This will cause an error
```

Writing to a read-only property is not allowed:

```
"use strict";  
var obj = {};  
obj.defineProperty(obj, "x", {value:0, writable:false});
```

```
obj.x = 3.14;        // This will cause an error
```

Writing to a get-only property is not allowed:



```
"use strict";  
var obj = {get x() {return 0} };
```

```
obj.x = 3.14;          // This will cause an error
```

Deleting an undeletable property is not allowed:

```
"use strict";  
delete Object.prototype; // This will cause an error
```

The string "eval" cannot be used as a variable:

```
"use strict";  
var eval = 3.14;      // This will cause an error
```

The string "arguments" cannot be used as a variable:

```
"use strict";  
var arguments = 3.14; // This will cause an error
```

The with statement is not allowed:

```
"use strict";  
with (Math){x = cos(2)}; // This will cause an error
```

For security reasons, eval() are not allowed to create variables in the scope from which it was called:

```
"use strict";  
eval ("var x = 2");  
alert (x)          // This will cause an error
```

In function calls like f(), the this value was the global object. In strict mode, it is now undefined.

Future reserved keywords are not allowed. These are:

- implements
- interface
- package
- private
- protected

- public
  - static
  - yield
- 

## Watch Out!



The "use strict" directive is only recognized at the **beginning** of a script or a function.

# JavaScript Style Guide and Coding Conventions

Always use the same coding conventions for all your JavaScript projects.

---

## JavaScript Coding Conventions

Coding conventions are **style guidelines for programming**. They typically cover:

- Naming and declaration rules for variables and functions.
- Rules for the use of white space, indentation, and comments.
- Programming practices and principles

Coding conventions **secure quality**:

- Improves code readability
- Make code maintenance easier

Coding conventions can be documented rules for teams to follow, or just be your individual coding practice.



This page describes the general JavaScript code conventions used by W3Schools. You should also read the next chapter "Best Practices", and learn how to avoid coding pitfalls.

---

## Variable Names

At W3schools we use **camelCase** for identifier names (variables and functions).

All names start with a **letter**.

At the bottom of this page, you will find a wider discussion about naming rules.

```
firstName = "John";  
lastName = "Doe";
```

```
price = 19.90;  
discount = 0.10;
```

```
fullPrice = price * 100 / discount;
```

---

## Spaces Around Operators

Always put spaces around operators ( = + / \* ), and after commas:

### Examples:

```
var x = y + z;  
var values = ["Volvo", "Saab", "Fiat"];
```

---

## Code Indentation

Always use 4 spaces for indentation of code blocks:

## Functions:

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



Do not use tabs (tabulators) for indentation. Text editors interpret tabs differently.

---

## Statement Rules

General rules for simple statements:

- Always end simple statement with a semicolon.

## Examples:

```
var values = ["Volvo", "Saab", "Fiat"];
```

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

General rules for complex (compound) statements:

- Put the opening bracket at the end of the first line.
- Use one space before the opening bracket.
- Put the closing bracket on a new line, without leading spaces.
- Do not end complex statement with a semicolon.

## Functions:

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

## Loops:

```
for (i = 0; i < 5; i++) {  
    x += i;  
}
```

## Conditionals:

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

---

## Object Rules

General rules for object definitions:

- Place the opening bracket on the same line as the object name.
- Use colon plus one space between each property and its value.
- Use quotes around string values, not around numeric values.
- Do not add a comma after the last property-value pair.
- Place the closing bracket, on a new line, without leading spaces.
- Always end an object definition with a semicolon.

## Example:

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

Short objects can be written compressed, on one line, like this:

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

---

## Line Length < 80

For readability, avoid lines longer than 80 characters.

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

## Example

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

---

## Naming Conventions

Always use the same naming convention for all your code. For example:

- Variable and function names written as **camelCase**
- Global variable written in **UPPERCASE**
- Constants (like PI) written in **UPPERCASE**

Should you use **hyp-hens**, **camelCase**, or **under\_scores** in variable names?

This is a question programmers often discuss. The answer depends on who you ask:

### Hyphens in HTML and CSS:

HTML5 attributes can start with data- (data-quantity, data-price).

CSS uses hyphens in property-names (font-size).



Hyphens can be mistaken as subtraction attempts. Hyphens are not allowed in JavaScript names.

### Underscores:

Many programmers prefer to use underscores (date\_of\_birth), especially in SQL databases.

Underscores are often used in PHP documentation.

### **CamelCase:**

CamelCase is often preferred by C programmers.

### **camelCase:**

camelCase is used by JavaScript itself, by jQuery, and other JavaScript libraries.



Don't start names with a \$ sign. It will put you in conflict with many JavaScript library names.

---

## **Loading JavaScript in HTML**

Use simple syntax for loading external scripts (the type attribute is not necessary):

```
<script src="myscript.js">
```

---

## **Accessing HTML Elements**

A consequence of using "untidy" HTML styles, might result in JavaScript errors.

These two JavaScript statements will produce different results:

```
var obj = getElementById("Demo")
```

```
var obj = getElementById("demo")
```

---

## **File Extensions**

HTML files should have a **.html** extension (not **.htm**).

CSS files should have a **.css** extension.

JavaScript files should have a **.js** extension.

---

## Use Lower Case File Names

Most web servers (Apache, Unix) are case sensitive about file names:

london.jpg cannot be accessed as London.jpg.

Other web servers (Microsoft, IIS) are not case sensitive:

london.jpg can be accessed as London.jpg or london.jpg.

If you use a mix of upper and lower case, you have to be extremely consistent.

If you move from a case insensitive, to a case sensitive server, even small errors will break your web.

To avoid these problems, always use lower case file names (if possible).

---

## Performance

Coding conventions are not used by computers. Most rules have little impact on the execution of programs.

Indentation and extra spaces are not significant in small scripts.

For code in development, readability should be preferred. Larger production scripts should be minified.



# JavaScript Best Practices

Avoid global variables, avoid new, avoid ==, avoid eval()

---

## Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead, and learn how to use closures.

---

## Always Declare Local Variables

All variables used in a function should be declared as **local** variables.

Local variables **must** be declared with the **var** keyword, otherwise they will become global variables.



Strict mode does not allow undeclared variables.

---

## Declarations on Top

It is good coding practice, to put all declarations at the top of each script or function. This makes it easier to avoid unwanted (implied) global variables. It also gives cleaner code, and reduces the possibility of unwanted re-declarations.

```
var firstName, lastName;  
var price, discount, fullPrice;
```

```
firstName = "John";  
lastName = "Doe";
```

```
price = 19.90;  
discount = 0.10;
```

```
fullPrice = price * 100 / discount;
```

Variable declarations should always be the first statements in scripts and functions.

This also goes for variables in loops:

```
var i;  
for (i = 0; i < 5; i++)
```



Since JavaScript moves the declarations to the top anyway (JavaScript hoisting), it is always a good rule.

---

## Never Declare Numbers, Strings, or Booleans as Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring numbers, strings, or booleans as objects, slows down execution speed, and produces nasty side effects:

### Example

```
var x = "John";  
var y = new String("John");  
(x === y) // is false because x is a string and y is an object.
```

---

## Don't Use new Object()

- Use {} instead of new Object()

- Use "" instead of new String()
- Use 0 instead of new Number()
- Use false instead of new Boolean()
- Use [] instead of new Array()
- Use /(\/)/ instead of new RegExp()
- Use function (){} instead of new function()

## Example

```
var x1 = {};      // new object
var x2 = "";      // new primitive string
var x3 = 0;       // new primitive number
var x4 = false;   // new primitive boolean
var x5 = [];      // new array object
var x6 = /(\/)/;  // new regexp object
var x7 = function(){}; // new function object
```

---

## Beware of Automatic Type Conversions

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

## Example

```
var x = "Hello"; // typeof x is a string
x = 5;           // changes typeof x to a number
```

When doing mathematical operations, JavaScript can convert numbers to strings:

## Example

```
var x = 5 + 7;    // x.valueOf() is 12, typeof x is a number
var x = 5 + "7";  // x.valueOf() is 57, typeof x is a string
```

```
var x = "5" + 7;    // x.valueOf() is 57, typeof x is a string
var x = 5 - 7;      // x.valueOf() is -2, typeof x is a number
var x = 5 - "7";    // x.valueOf() is -2, typeof x is a number
var x = "5" - 7;    // x.valueOf() is -2, typeof x is a number
var x = 5 - "x";    // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

## Example

```
"Hello" - "Dolly"    // returns NaN
```

---

## Use === Comparison

The == comparison operator always converts (to matching types) before comparison.

The === operator forces comparison of values and type:

## Example

```
0 == "";    // true
1 == "1";   // true
1 == true;  // true

0 === "";   // false
1 === "1";  // false
1 === true; // false
```

---

## Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to **undefined**.

Undefined values can break your code. It is a good habit to assign default values to arguments.

### Example

```
function myFunction(x, y) {  
  if (y === undefined) {  
    y = 0;  
  }  
}
```

Or, even simpler:

```
function myFunction(x, y) {  
  y = y || 0;  
}
```

---

## Avoid Using eval()

The `eval()` function is used to run text as code. In almost all cases, it should not be necessary to use it.

Because it allows arbitrary code to be run, it also represents a security problem.

## JavaScript Common Mistakes

---

This chapter points at some common JavaScript mistakes.

---

## Accidentally Using the Assignment Operator

This statement **only** returns true, if x1 equals 10:

```
if (x1 == 10)
```

This statement **always** returns true, because the assignment is always true:

```
if (x1 = 10)
```

---

## Confusing Addition & Concatenation

**Addition** is about adding **numbers**.

**Concatenation** is about adding **strings**.

In JavaScript both operations use the same + operator.

Because of this, when adding a number as a number, will produce a different result from adding a number as a string:

```
var x = 10 + 5;    // the result in x is 15
var x = 10 + "5";  // the result in x is "105"
```

When adding two variables, it can be difficult to anticipate the result:

```
var x = 10;
var y = 5;
```

```
var z = x + y;      // the result in z is 15
```

```
var x = 10;  
var y = "5";  
var z = x + y;      // the result in z is "105"
```

## Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits **Floating point numbers** (Floats).

All programming languages, including JavaScript, have difficulties with precise floating point values:

```
var x = 0.1;  
var y = 0.2;  
var z = x + y      // the result in z will not be 0.3  
if (z == 0.3)      // this if test will fail
```

## Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

### Example 1

```
var x =  
"Hello World!";
```

But, breaking a statement in the middle of a string will not work:

### Example 2

```
var x = "Hello  
World!";
```

You must use a "backslash" if you must break a statement in a string:

## Example 3

```
var x = "Hello \  
World!";
```

---

## Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

```
if (x == 19);  
{  
    // code block  
}
```

---

## Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line.

Because of this, these two examples will return the same result:

## Example 1

```
function myFunction(a) {  
    var power = 10  
    return a * power  
}
```



## Example 2

```
function myFunction(a) {  
  var power = 10;  
  return a * power;  
}
```

JavaScript will also allow you to break a statement into two lines.

Because of this, example 3 will also return the same result:

## Example 3

```
function myFunction(a) {  
  var  
  power = 10;  
  return a * power;  
}
```

But, what will happen if you break the return statement in two lines like this:

## Example 4

```
function myFunction(a) {  
  var  
  power = 10;  
  return  
  a * power;  
}
```

The function will return undefined!

Why? Because JavaScript thinks you meant:

## Example 5

```
function myFunction(a) {  
  var  
  power = 10;  
  return;  
  a * power;  
}
```

---

## Explanation

If a statement is incomplete like:

```
var
```

JavaScript will try to complete the statement by reading the next line:

```
power = 10;
```

But since this statement is complete:

```
return
```

JavaScript will automatically close it like this:

```
return;
```

This happens because closing (ending) statements with semicolon is optional in JavaScript.

JavaScript will close the return statement at the end of the line, because it is a complete statement.



Never break a return statement.

---

## Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** use **numbered indexes**:

### Example:

```
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"
```

In JavaScript, **objects** use **named indexes**.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

### Example:

```
var person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;
```

```
var x = person.length;    // person.length will return 0
var y = person[0];        // person[0] will return undefined
```

---

## Ending an Array Definition with a Comma

### Incorrect:

```
points = [40, 100, 1, 5, 25, 10,];
```

Some JSON and JavaScript engines will fail, or behave unexpectedly.

### Correct:

```
points = [40, 100, 1, 5, 25, 10];
```

---

## Ending an Object Definition with a Comma

### Incorrect:

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

Some JSON and JavaScript engines will fail, or behave unexpectedly.

### Correct:

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

---

## Undefined is Not Null

With JavaScript, **null** is for objects, **undefined** is for variables, properties, and methods.

To be null, an object has to be defined, otherwise it will be undefined.

If you want to test if an object exists, this will throw an error if the object is undefined:

## Incorrect:

```
if (myObj !== null && typeof myObj !== "undefined")
```

Because of this, you must test `typeof()` first:

## Correct:

```
if (typeof myObj !== "undefined" && myObj !== null)
```

---

## Expecting Block Level Scope

JavaScript **does not** create a new scope for each code block.

It is true in many programming languages, but **not true** in JavaScript.

It is a common mistake, among new JavaScript developers, to believe that this code returns undefined:

## Example:

```
for (var i = 0; i < 10; i++) {  
  // come code  
}  
return i;
```

# JavaScript Performance

How to speed up your JavaScript code.

---

## Reduce Activity in Loops

Loops are often used in programming.

Every statement inside a loop will be executed for each iteration of the loop.

Search for statements or assignments that can be placed outside the loop.

---

## Reduce DOM Access

Accessing the HTML DOM is very slow, compared to other JavaScript statements.

If you expect to access a DOM element several times, access it once, and use it as a local variable:

## Example

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

---

## Reduce DOM Size

Keep the number of elements in the HTML DOM small.

This will always improve page loading, and speed up rendering (page display), especially on smaller devices.

Every attempt to search the DOM (like `getElementsByName`) is will benefit from a smaller DOM.

---

## Avoid Unnecessary Variables

Don't create new variables if you don't plan to save values.

Often you can replace code like this:

```
var fullName = firstName + " " + lastName;  
document.getElementById("demo").innerHTML = fullName;
```

With this:

```
document.getElementById("demo").innerHTML = firstName + " " + lastName
```

---

## Delay JavaScript Loading

Putting your scripts at the bottom of the page body, lets the browser load the page first.

While a script is downloading, the browser will not start any other downloads. In addition all parsing and rendering activity might be blocked.



The HTTP specification defines that browsers should not download more than two components in parallel.

An alternative is to use **defer="true"** in the script tag. The defer attribute specifies that the script should be executed before the page has finished parsing, but it only works for external scripts.

If possible, you can add your script to the page by code, after the page has loaded:

## Example

```
<script>  
window.onload = downScripts;
```

```
function downScripts() {
```

```
var element = document.createElement("script");
element.src = "myScript.js";
document.body.appendChild(element);
}
</script>
```

---

## Avoid Using with

Avoid using the **with keyword**. It has a negative effect on speed. It also clutters up JavaScript scopes.

The with keyword is **not allowed** in strict mode.

## JavaScript Reserved Words

In JavaScript, some identifiers are reserved words and cannot be used as variables or function names.

---

## JavaScript Standards

All modern browsers fully support ECMAScript 3 (ES3, the third edition of JavaScript from 1999).

ECMAScript 4 (ES4) was never adopted.

ECMAScript 5 (ES5, released in 2009) is the latest official version of JavaScript.

Time passes, and we are now beginning to see complete support for ES5 in all modern browsers.

---



## JavaScript Reserved Words

In JavaScript you cannot use these reserved words as variables, labels, or function names:

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

Words marked with\* are new in ECMAScript5

---

## JavaScript Objects, Properties, and Methods

You should also avoid using the name of JavaScript built-in objects, properties, and methods:

Array	Date	eval	function	hasOwnProperty
Infinity	isFinite	isNaN	isPrototypeOf	length
Math	NaN	name	Number	Object
prototype	String	toString	undefined	valueOf

---

## Java Reserved Words

JavaScript is often used together with Java. You should avoid using some Java objects and properties as JavaScript identifiers:

getClass	java	JSONArray	javaClass	JavaObject	JavaPackage
----------	------	-----------	-----------	------------	-------------

---

## Windows Reserved Words

JavaScript can be used outside HTML. It can be used as the programming language in many other applications.

In HTML you must (for portability you should) avoid using the name of HTML and Windows objects and properties:

alert	all	anchor	anchors	area
assign	blur	button	checkbox	clearInterval
clearTimeout	clientInformation	close	closed	confirm
constructor	crypto	decodeURI	decodeURIComponent	defaultStatus
document	element	elements	embed	embeds
encodeURIComponent	encodeURIComponent	escape	event	fileUpload
focus	form	forms	frame	innerHeight
innerWidth	layer	layers	link	location
mimeType	navigate	navigator	frames	frameRate
hidden	history	image	images	offscreenBuffering
open	opener	option	outerHeight	outerWidth
packages	pageXOffset	pageYOffset	parent	parseFloat
parseInt	password	pkcs11	plugin	prompt
propertyIsEnum	radio	reset	screenX	screenY
scroll	secure	select	self	setInterval
setTimeout	status	submit	taint	text
textarea	top	unescape	untaint	window

---

## HTML Event Handlers

In addition you should avoid using the name of all HTML event handlers.

Examples:

onblur	onclick	onerror	onfocus
onkeydown	onkeypress	onkeyup	onmouseover
onload	onmouseup	onmousedown	onsubmit

---

## Nonstandard JavaScript

In addition to reserved words, there are also some nonstandard keywords used in some JavaScript implementations.

One example is the **const** keyword used to define variables. Some JavaScript engines will treat const as a synonym to var. Other engines will treat const as a definition for read-only variables.

Const is an extension to JavaScript. It is supported by the JavaScript engine used in Firefox and Chrome. But it is not a part of the JavaScript standards ES3 or ES5. **Do not use it.**

## JavaScript HTML DOM

With the HTML DOM, JavaScript can access and change all the elements of an HTML document.

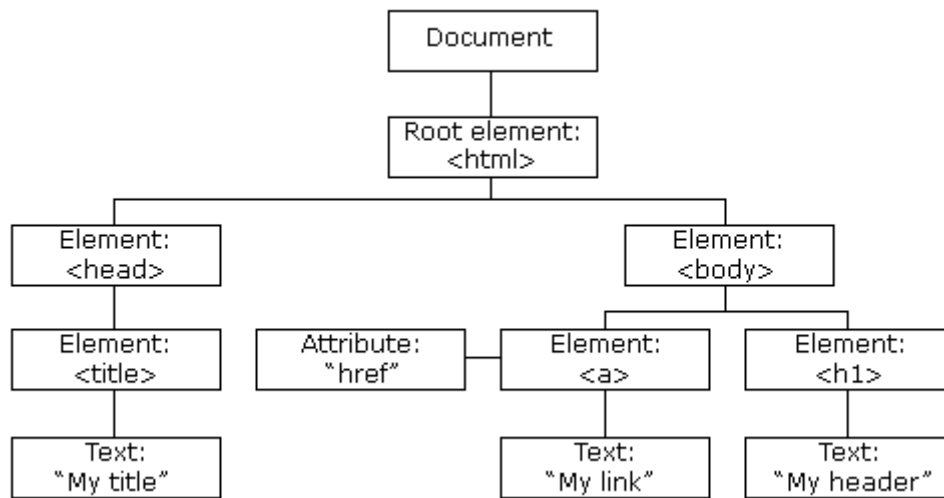
---

### The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a **Document Object Model** of the page.

The **HTML DOM** model is constructed as a tree of **Objects**:

# The HTML DOM Tree of Objects



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

---

## What You Will Learn

In the next chapters of this tutorial you will learn:

- How to change the content of HTML elements
  - How to change the style (CSS) of HTML elements
  - How to react to HTML DOM events
  - How to add and delete HTML elements
-

# What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents:

*"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
  - XML DOM - standard model for XML documents
  - HTML DOM - standard model for HTML documents
- 

## What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

## JavaScript - HTML DOM Methods

---

HTML DOM methods are **actions** you can perform (on HTML Elements)

HTML DOM properties are **values** (of HTML Elements) that you can set or change

---

# The DOM Programming Interface

The HTML DOM can be accessed with JavaScript (and with other programming languages).

In the DOM, all HTML elements are defined as **objects**.

The programming interface is the properties and methods of each object.

A **property** is a value that you can get or set (like changing the content of an HTML element).

A **method** is an action you can do (like add or deleting an HTML element).

---

## Example

The following example changes the content (the innerHTML) of the <p> element with id="demo":

## Example

```
<html>
<body>

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

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

</body>
</html>
```

In the example above, getElementById is a **method**, while innerHTML is a **property**.

---

## The getElementById Method

The most common way to access an HTML element is to use the id of the element.

In the example above the getElementById method used id="demo" to find the element.

---

## The innerHTML Property

The easiest way to get the content of an element is by using the **innerHTML** property.

The innerHTML property is useful for getting or replacing the content of HTML elements.



The innerHTML property can be used to get or change any HTML element, including `<html>` and `<body>`.

# JavaScript HTML DOM Document

With the HTML DOM, the document object is your web page.

---

## The HTML DOM Document

In the HTML DOM object model, the document object represents your web page.

The document object is the owner of all other objects in your web page.

If you want to access objects in an HTML page, you always start with accessing the document object.

Below are some examples of how you can use the document object to access and manipulate HTML.

The next chapters demonstrate the methods.

---

## Finding HTML Elements

Method	Description
<code>document.getElementById()</code>	Find an element by element id
<code>document.getElementsByTagName()</code>	Find elements by tag name
<code>document.getElementsByClassName()</code>	Find elements by class name

---

## Changing HTML Elements

Method	Description
<code>element.innerHTML=</code>	Change the inner HTML of an element
<code>element.attribute=</code>	Change the attribute of an HTML element
<code>element.setAttribute(attribute,value)</code>	Change the attribute of an HTML element
<code>element.style.property=</code>	Change the style of an HTML element

---

## Adding and Deleting Elements

Method	Description
<code>document.createElement()</code>	Create an HTML element
<code>document.removeChild()</code>	Remove an HTML element
<code>document.appendChild()</code>	Add an HTML element
<code>document.replaceChild()</code>	Replace an HTML element
<code>document.write(text)</code>	Write into the HTML output stream

---

## Adding Events Handlers

Method	Description
<code>document.getElementById(id).onclick=function(){code}</code>	Adding event handler code to an onclick event



---

## Finding HTML Objects

The first HTML DOM Level 1 (1998), defined 11 HTML objects, object collections, and properties. These are still valid in HTML5.

Later, in HTML DOM Level 3, more objects, collections, and properties were added.

Property	Description	DOM
document.anchors	Returns all <a> elements that have a name attribute	1
document.applets	Returns all <applet> elements (Deprecated in HTML5)	1
document.baseURI	Returns the absolute base URI of the document	3
document.body	Returns the <body> element	1
document.cookie	Returns the document's cookie	1
document.doctype	Returns the document's doctype	3
document.documentElement	Returns the <html> element	3
document.documentMode	Returns the mode used by the browser	3
document.documentURI	Returns the URI of the document	3
document.domain	Returns the domain name of the document server	1
document.domConfig	Obsolete. Returns the DOM configuration	3
document.embeds	Returns all <embed> elements	3
document.forms	Returns all <form> elements	1
document.head	Returns the <head> element	3
document.images	Returns all <img> elements	1
document.implementation	Returns the DOM implementation	3
document.inputEncoding	Returns the document's encoding (character set)	3
document.lastModified	Returns the date and time the document was updated	3
document.links	Returns all <area> and <a> elements that have a href attribute	1
document.readyState	Returns the (loading) status of the document	3
document.referrer	Returns the URI of the referrer (the linking document)	1
document.scripts	Returns all <script> elements	3
document.strictErrorChecking	Returns if error checking is enforced	3
document.title	Returns the <title> element	1
document.URL	Returns the complete URL of the document	1

# JavaScript HTML DOM Elements

This page teaches you how to find and access HTML elements in an HTML page.

---

## Finding HTML Elements

Often, with JavaScript, you want to manipulate HTML elements.

To do so, you have to find the elements first. There are a couple of ways to do this:

- Finding HTML elements by id
  - Finding HTML elements by tag name
  - Finding HTML elements by class name
  - Finding HTML elements by HTML object collections
- 

## Finding HTML Elements by Id

The easiest way to find HTML elements in the DOM, is by using the element id.

This example finds the element with id="intro":

### Example

```
var x = document.getElementById("intro");
```

If the element is found, the method will return the element as an object (in x).

If the element is not found, x will contain null.

---

## Finding HTML Elements by Tag Name

This example finds the element with id="main", and then finds all <p> elements inside "main":

### Example

```
var x = document.getElementById("main");  
var y = x.getElementsByTagName("p");
```

---

## Finding HTML Elements by Class Name

If you want to find all HTML elements with the same class name. Use this method:

```
document.getElementsByClassName("intro");
```

The example above returns a list of all elements with class="intro".



Finding elements by class name does not work in Internet Explorer 5,6,7, and 8.

---

## Finding HTML Elements by HTML Object Collections

This example finds the form element with id="frm1", in the forms collection, and displays all element values:

### Example

```
var x = document.getElementById("frm1");  
var text = "";  
var i;  
for (i = 0; i < x.length; i++) {  
    text += x.elements[i].value + "<br>";  
}
```

```
}  
document.getElementById("demo").innerHTML = text;
```

The following HTML objects (and object collections) are also accessible:

- [document.anchors](#)
- [document.body](#)
- [document.documentElement](#)
- [document.embeds](#)
- [document.forms](#)
- [document.head](#)
- [document.images](#)
- [document.links](#)
- [document.scripts](#)
- [document.title](#)

# JavaScript HTML DOM - Changing HTML

The HTML DOM allows JavaScript to change the content of HTML elements.

---

## Changing the HTML Output Stream

JavaScript can create dynamic HTML content:

**Date: Mon Jan 05 2015 16:39:07 GMT+0530 (India Standard Time)**

In JavaScript, `document.write()` can be used to write directly to the HTML output stream:

## Example

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

```
<script>
document.write(Date());
</script>
```

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



Never use `document.write()` after the document is loaded. It will overwrite the document.

---

## Changing HTML Content

The easiest way to modify the content of an HTML element is by using the **innerHTML** property.

To change the content of an HTML element, use this syntax:

```
document.getElementById(id).innerHTML = new HTML
```

This example changes the content of a `<p>` element:

### Example

```
<html>
<body>

<p id="p1">Hello World!</p>

<script>
document.getElementById("p1").innerHTML = "New text!";
</script>

</body>
</html>
```

This example changes the content of an `<h1>` element:

## Example

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

<h1 id="header">Old Header</h1>

<script>
var element = document.getElementById("header");
element.innerHTML = "New Header";
</script>

</body>
</html>
```

Example explained:

- The HTML document above contains an `<h1>` element with `id="header"`
- We use the HTML DOM to get the element with `id="header"`
- A JavaScript changes the content (`innerHTML`) of that element

---

## Changing the Value of an Attribute

To change the value of an HTML attribute, use this syntax:

```
document.getElementById(id).attribute=new value
```

This example changes the value of the `src` attribute of an `<img>` element:

## Example

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

```
<body>



<script>
document.getElementById("myImage").src = "landscape.jpg";
</script>

</body>
</html>
```

Example explained:

- The HTML document above contains an `<img>` element with `id="myImage"`
- We use the HTML DOM to get the element with `id="myImage"`
- A JavaScript changes the `src` attribute of that element from `"smiley.gif"` to `"landscape.jpg"`

## JavaScript HTML DOM - Changing CSS

The HTML DOM allows JavaScript to change the style of HTML elements.

---

### Changing HTML Style

To change the style of an HTML element, use this syntax:

```
document.getElementById(id).style.property=new style
```

The following example changes the style of a `<p>` element:

### Example

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

```
<p id="p2">Hello World!</p>
```

```
<script>  
document.getElementById("p2").style.color = "blue";  
</script>
```

```
<p>The paragraph above was changed by a script.</p>
```

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

---

## Using Events

The HTML DOM allows you to execute code when an event occurs.

Events are generated by the browser when "things happen" to HTML elements:

- An element is clicked on
- The page has loaded
- Input fields are changed

You will learn more about events in the next chapter of this tutorial.

This example changes the style of the HTML element with id="id1", when the user clicks a button:

## Example

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

```
<h1 id="id1">My Heading 1</h1>
```

```
<button type="button"  
onclick="document.getElementById('id1').style.color = 'red'">
```



Click Me!</button>

</body>

</html>

# JavaScript HTML DOM Events

HTML DOM allows JavaScript to react to HTML events.

## Example

Mouse Over Me

Click Me

---

## Reacting to Events

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

`onclick=JavaScript`

Examples of HTML events:

- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user strokes a key

In this example, the content of the `<h1>` element is changed when a user clicks on it:

## Example

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

<h1 onclick="this.innerHTML='Oops!'">Click on this text!</h1>

</body>
</html>
```

In this example, a function is called from the event handler:

## Example

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

<h1 onclick="changeText(this)">Click on this text!</h1>

<script>
function changeText(id) {
    id.innerHTML = "Oops!";
}
</script>

</body>
</html>
```

---

## HTML Event Attributes

To assign events to HTML elements you can use event attributes.

## Example

Assign an onclick event to a button element:

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

In the example above, a function named *displayDate* will be executed when the button is clicked.

---

## Assign Events Using the HTML DOM

The HTML DOM allows you to assign events to HTML elements using JavaScript:

### Example

Assign an onclick event to a button element:

```
<script>  
document.getElementById("myBtn").onclick = function(){ displayDate() };  
</script>
```

In the example above, a function named *displayDate* is assigned to an HTML element with the id="myBtn".

The function will be executed when the button is clicked.

---

## The onload and onunload Events

The onload and onunload events are triggered when the user enters or leaves the page.

The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

The onload and onunload events can be used to deal with cookies.

## Example

```
<body onload="checkCookies()">
```

---

## The onchange Event

The onchange event are often used in combination with validation of input fields.

Below is an example of how to use the onchange. The upperCase() function will be called when a user changes the content of an input field.

## Example

```
<input type="text" id="fname" onchange="upperCase()">
```

---

## The onmouseover and onmouseout Events

The onmouseover and onmouseout events can be used to trigger a function when the user mouses over, or out of, an HTML element.

## Example

A simple onmouseover-onmouseout example:

**Mouse Over Me**

---

## The onmousedown, onmouseup and onclick Events

The onmousedown, onmouseup, and onclick events are all parts of a mouse-click. First when a mouse-button is clicked, the onmousedown event is triggered, then, when the mouse-button is released, the onmouseup event is triggered, finally, when the mouse-click is completed, the onclick event is triggered.

### Example

A simple onmousedown-onmouseup example:

Click Me

## JavaScript HTML DOM EventListener

### The addEventListener() method

#### Example

Add an event listener that fires when a user clicks a button:

```
document.getElementById("myBtn").addEventListener("click", displayDate);
```

The addEventListener() method attaches an event handler to the specified element.

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.

You can add many event handlers to one element.

You can add many event handlers of the same type to one element, i.e two "click" events.

You can add event listeners to any DOM object not only HTML elements. i.e the window object.

The `addEventListener()` method makes it easier to control how the event reacts to bubbling.

When using the `addEventListener()` method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

You can easily remove an event listener by using the `removeEventListener()` method.

---

## Syntax

```
element.addEventListener(event, function, useCapture);
```

The first parameter is the type of the event (like "click" or "mousedown").

The second parameter is the function we want to call when the event occurs.

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.



Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".

---

## Add an Event Handler to an Element

### Example

Alert "Hello World!" when the user clicks on an element:

```
element.addEventListener("click", function(){ alert("Hello World!"); });
```

You can also refer to an external "named" function:

## Example

Alert "Hello World!" when the user clicks on an element:

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

```
function myFunction() {  
    alert ("Hello World!");  
}
```

---

## Add Many Event Handlers to the Same Element

The `addEventListener()` method allows you to add many events to the same element, without overwriting existing events:

## Example

```
element.addEventListener("click", myFunction);  
element.addEventListener("click", mySecondFunction);
```

You can add events of different types to the same element:

## Example

```
element.addEventListener("mouseover", myFunction);  
element.addEventListener("click", mySecondFunction);  
element.addEventListener("mouseout", myThirdFunction);
```

---

## Add an Event Handler to the Window Object

The `addEventListener()` method allows you to add event listeners on any HTML DOM object such as HTML elements, the HTML document, the window object, or other objects that supports events, like the `xmlHttpRequest` object.

### Example

Add an event listener that fires when a user resizes the window:

```
window.addEventListener("resize", function(){
    document.getElementById("demo").innerHTML = sometext;
});
```

---

## Passing Parameters

When passing parameter values, use an "anonymous function" that calls the specified function with the parameters:

### Example

```
element.addEventListener("click", function(){ myFunction(p1, p2); });
```

---

## Event Bubbling or Event Capturing?

There are two ways of event propagation in the HTML DOM, bubbling and capturing.

Event propagation is a way of defining the element order when an event occurs. If you have a `<p>` element inside a `<div>` element, and the user clicks on the `<p>` element, which element's "click" event should be handled first?

In *bubbling* the inner most element's event is handled first and then the outer: the `<p>` element's click event is handled first, then the `<div>` element's click event.



In *capturing* the outer most element's event is handled first and then the inner: the <div> element's click event will be handled first, then the <p> element's click event.

With the `addEventListener()` method you can specify the propagation type by using the "useCapture" parameter:

```
addEventListener(event, function, useCapture);
```

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

## Example

```
document.getElementById("myP").addEventListener("click", myFunction, true);  
document.getElementById("myDiv").addEventListener("click", myFunction, true);
```

---

## The `removeEventListener()` method

The `removeEventListener()` method removes event handlers that have been attached with the `addEventListener()` method:

## Example

```
element.removeEventListener("mousemove", myFunction);
```

---

## Browser Support

The numbers in the table specifies the first browser version that fully supports these methods.

Method					
<b><code>addEventListener()</code></b>	1.0	9.0	1.0	1.0	7.0
<b><code>removeEventListener()</code></b>	1.0	9.0	1.0	1.0	7.0

**Note:** The `addEventListener()` and `removeEventListener()` methods are not supported in IE 8 and earlier versions and Opera 7.0 and earlier versions. However, for these specific browser versions, you can use the `attachEvent()` method to attach an event handlers to the element, and the `detachEvent()` method to remove it:

```
element.attachEvent(event, function);  
element.detachEvent(event, function);
```

## Example

Cross-browser solution:

```
var x = document.getElementById("myBtn");  
if (x.addEventListener) {           // For all major browsers, except IE 8 and earlier  
    x.addEventListener("click", myFunction);  
} else if (x.attachEvent) {         // For IE 8 and earlier versions  
    x.attachEvent("onclick", myFunction);  
}
```

# JavaScript HTML DOM Navigation

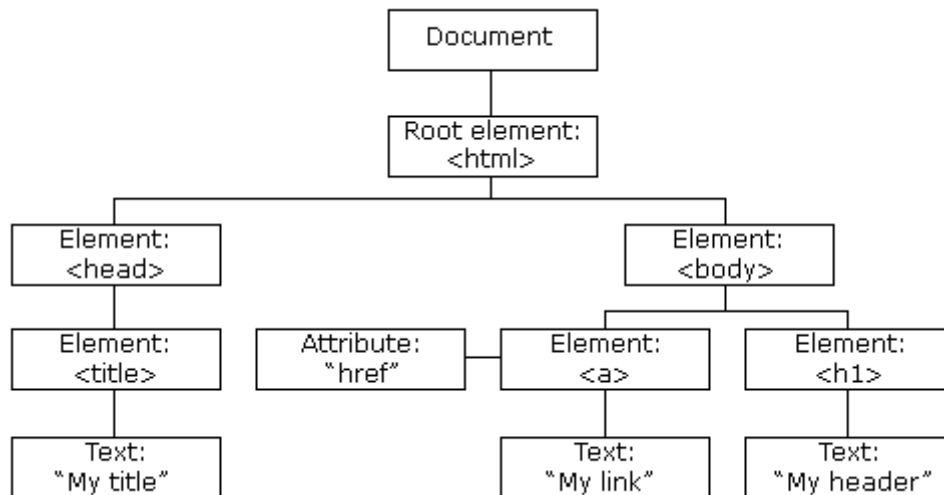
With the HTML DOM, you can navigate the node tree using node relationships.

---

## DOM Nodes

According to the W3C HTML DOM standard, everything in an HTML document is a node:

- The entire document is a document node
- Every HTML element is an element node
- The text inside HTML elements are text nodes
- Every HTML attribute is an attribute node
- All comments are comment nodes



With the HTML DOM, all nodes in the node tree can be accessed by JavaScript.

New nodes can be created, and all nodes can be modified or deleted.

---

## Node Relationships

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

- In a node tree, the top node is called the root (or root node)
- Every node has exactly one parent, except the root (which has no parent)
- A node can have a number of children
- Siblings (brothers or sisters) are nodes with the same parent

```
<html>
```

```
<head>
```

```
<title>DOM Tutorial</title>
```

```
</head>
```

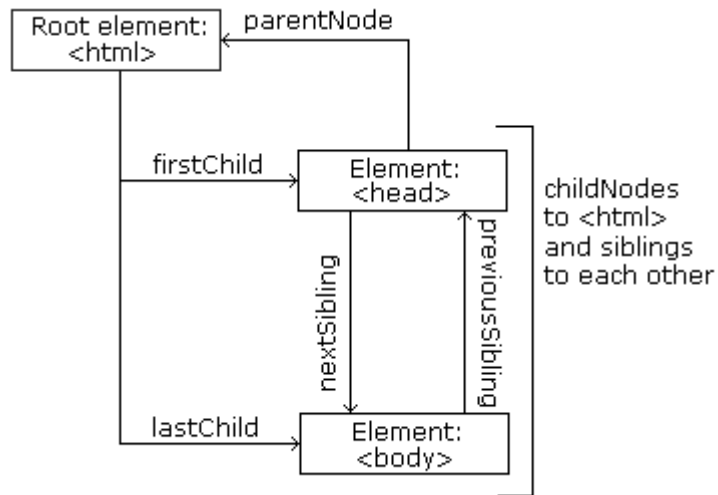
```
<body>
```

```
<h1>DOM Lesson one</h1>
```

```
<p>Hello world!</p>
```

</body>

</html>



From the HTML above you can read:

- <html> is the root node
- <html> has no parents
- <html> is the parent of <head> and <body>
- <head> is the first child of <html>
- <body> is the last child of <html>

and:

- <head> has one child: <title>
- <title> has one child (a text node): "DOM Tutorial"
- <body> has two children: <h1> and <p>
- <h1> has one child: "DOM Lesson one"
- <p> has one child: "Hello world!"
- <h1> and <p> are siblings

---

## Navigating Between Nodes

You can use the following node properties to navigate between nodes with JavaScript:

- parentNode

- `childNodes[nodenumbers]`
  - `firstChild`
  - `lastChild`
  - `nextSibling`
  - `previousSibling`
- 

## Warning !

A common error in DOM processing is to expect an element node to contain text.

In this example: **<title>DOM Tutorial</title>**, the element node `<title>` does not contain text. It contains a **text node** with the value "DOM Tutorial".

The value of the text node can be accessed by the node's **innerHTML** property, or the **nodeValue**.

---

## Child Nodes and Node Values

In addition to the `innerHTML` property, you can also use the `childNodes` and `nodeValue` properties to get the content of an element.

The following example collects the node value of an `<h1>` element and copies it into a `<p>` element:

### Example

```
<html>
<body>

<h1 id="intro">My First Page</h1>

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

<script>
var myText = document.getElementById("intro").childNodes[0].nodeValue;
document.getElementById("demo").innerHTML = myText;
```

```
</script>
```

```
</body>
```

```
</html>
```

In the example above, `getElementById` is a method, while `childNodes` and `nodeValue` are properties.

In this tutorial we use the `innerHTML` property. However, learning the method above is useful for understanding the tree structure and the navigation of the DOM.

Using the `firstChild` property is the same as using `childNodes[0]`:

## Example

```
<html>
```

```
<body>
```

```
<h1 id="intro">My First Page</h1>
```

```
<p id="demo">Hello World!</p>
```

```
<script>
```

```
myText = document.getElementById("intro").firstChild.nodeValue;
```

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

```
</script>
```

```
</body>
```

```
</html>
```

---

## DOM Root Nodes

There are two special properties that allow access to the full document:

- document.body - The body of the document
- document.documentElement - The full document

## Example

```
<html>
<body>

<p>Hello World!</p>
<div>
<p>The DOM is very useful!</p>
<p>This example demonstrates the <b>document.body</b> property.</p>
</div>

<script>
alert(document.body.innerHTML);
</script>

</body>
</html>
```

## Example

```
<html>
<body>

<p>Hello World!</p>
<div>
<p>The DOM is very useful!</p>
<p>This example demonstrates the <b>document.documentElement</b>
property.</p>
</div>

<script>
alert(document.documentElement.innerHTML);
</script>
```

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

---

## The nodeName Property

The nodeName property specifies the name of a node.

- nodeName is read-only
- nodeName of an element node is the same as the tag name
- nodeName of an attribute node is the attribute name
- nodeName of a text node is always #text
- nodeName of the document node is always #document

**Note:** nodeName always contains the uppercase tag name of an HTML element.

---

## The nodeValue Property

The nodeValue property specifies the value of a node.

- nodeValue for element nodes is undefined
  - nodeValue for text nodes is the text itself
  - nodeValue for attribute nodes is the attribute value
- 

## The.nodeType Property

The nodeType property returns the type of node. nodeType is read only.

The most important node types are:

### Element type NodeType

Element	1
Attribute	2



Text	3
Comment	8
Document	9

# JavaScript HTML DOM Elements (Nodes)

Adding and Removing Nodes (HTML Elements)

---

## Creating New HTML Elements (Nodes)

To add a new element to the HTML DOM, you must create the element (element node) first, and then append it to an existing element.

### Example

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var para = document.createElement("p");
var node = document.createTextNode("This is new.");
para.appendChild(node);

var element = document.getElementById("div1");
element.appendChild(para);
</script>
```

---

## Example Explained

This code creates a new `<p>` element:

```
var para = document.createElement("p");
```

To add text to the `<p>` element, you must create a text node first. This code creates a text node:

```
var node = document.createTextNode("This is a new paragraph.");
```

Then you must append the text node to the `<p>` element:

```
para.appendChild(node);
```

Finally you must append the new element to an existing element.

This code finds an existing element:

```
var element = document.getElementById("div1");
```

This code appends the new element to the existing element:

```
element.appendChild(para);
```

---

## Creating new HTML Elements - insertBefore()

The `appendChild()` method in the previous example, appended the new element as the last child of the parent.

If you don't want that you can use the `insertBefore()` method:

### Example

```
<div id="div1">  
<p id="p1">This is a paragraph.</p>  
<p id="p2">This is another paragraph.</p>  
</div>
```

```
<script>
var para = document.createElement("p");
var node = document.createTextNode("This is new.");
para.appendChild(node);

var element = document.getElementById("div1");
var child = document.getElementById("p1");
element.insertBefore(para,child);
</script>
```

---

## Removing Existing HTML Elements

To remove an HTML element, you must know the parent of the element:

### Example

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var parent = document.getElementById("div1");
var child = document.getElementById("p1");
parent.removeChild(child);
</script>
```

---

### Example Explained

This HTML document contains a `<div>` element with two child nodes (two `<p>` elements):

```
<div id="div1">  
<p id="p1">This is a paragraph.</p>  
<p id="p2">This is another paragraph.</p>  
</div>
```

Find the element with id="div1":

```
var parent = document.getElementById("div1");
```

Find the <p> element with id="p1":

```
var child = document.getElementById("p1");
```

Remove the child from the parent:

```
parent.removeChild(child);
```



It would be nice to be able to remove an element without referring to the parent.  
But sorry. The DOM needs to know both the element you want to remove, and its parent.

Here is a common workaround: Find the child you want to remove, and use its parentNode property to find the parent:

```
var child = document.getElementById("p1");  
child.parentNode.removeChild(child);
```

---

## Replacing HTML Elements

To replace an element to the HTML DOM, use the replaceChild() method:

### Example

```
<div id="div1">  
<p id="p1">This is a paragraph.</p>  
<p id="p2">This is another paragraph.</p>  
</div>
```

```
<script>
var para = document.createElement("p");
var node = document.createTextNode("This is new.");
para.appendChild(node);

var parent = document.getElementById("div1");
var child = document.getElementById("p1");
parent.replaceChild(para,child);
</script>
```

# JavaScript HTML DOM Node List

A node list is a collection of nodes

---

## HTML DOM Node List

The `getElementsByTagName()` method returns a **node list**. A node list is an array-like collection of nodes.

The following code selects all `<p>` nodes in a document:

### Example

```
var x = document.getElementsByTagName("p");
```

The nodes can be accessed by an index number. To access the second `<p>` node you can write:

```
y = x[1];
```

**Note:** The index starts at 0.

---

## HTML DOM Node List Length

The length property defines the number of nodes in a node list:

### Example

```
var myNodelist = document.getElementsByTagName("p");  
document.getElementById("demo").innerHTML = myNodelist.length;
```

Example explained:

1. Get all <p> elements in a node list
2. Display the length of the node list

The length property is useful when you want to loop through the nodes in a node list:

### Example

Change the background color of all <p> elements in a node list:

```
var myNodelist = document.getElementsByTagName("p");  
var i;  
for (i = 0; i < myNodelist.length; i++) {  
    myNodelist[i].style.backgroundColor = "red";  
}
```

#### **A node list is not an array!**



A node list may look like an array, but it is not. You can loop through the node list and refer to its nodes like an array. However, you cannot use Array Methods, like `valueOf()` or `join()` on the node list.

# JavaScript Window - The Browser Object Model

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser.

---

## The Browser Object Model (BOM)

There are no official standards for the **B**rowser **O**bject **M**odel (BOM).

Since modern browsers have implemented (almost) the same methods and properties for JavaScript interactivity, it is often referred to, as methods and properties of the BOM.

---

## The Window Object

The **window** object is supported by all browsers. It represent the browser's window.

All global JavaScript objects, functions, and variables automatically become members of the window object.

Global variables are properties of the window object.

Global functions are methods of the window object.

Even the document object (of the HTML DOM) is a property of the window object:

```
window.document.getElementById("header");
```

is the same as:

```
document.getElementById("header");
```

---

# Window Size

Three different properties can be used to determine the size of the browser window (the browser viewport, NOT including toolbars and scrollbars).

For Internet Explorer, Chrome, Firefox, Opera, and Safari:

- `window.innerHeight` - the inner height of the browser window
- `window.innerWidth` - the inner width of the browser window

For Internet Explorer 8, 7, 6, 5:

- `document.documentElement.clientHeight`
- `document.documentElement.clientWidth`
- or
- `document.body.clientHeight`
- `document.body.clientWidth`

A practical JavaScript solution (covering all browsers):

## Example

```
var w = window.innerWidth
|| document.documentElement.clientWidth
|| document.body.clientWidth;

var h = window.innerHeight
|| document.documentElement.clientHeight
|| document.body.clientHeight;
```

The example displays the browser window's height and width: (NOT including toolbars/scrollbars)

---

## Other Window Methods

Some other methods:



- `window.open()` - open a new window
- `window.close()` - close the current window
- `window.moveTo()` -move the current window
- `window.resizeTo()` -resize the current window

# JavaScript Window Screen

The `window.screen` object contains information about the user's screen.

---

## Window Screen

The **`window.screen`** object can be written without the window prefix.

Properties:

- `screen.width`
  - `screen.height`
  - `screen.availWidth`
  - `screen.availHeight`
  - `screen.colorDepth`
  - `screen.pixelDepth`
- 

## Window Screen Width

The `screen.width` property returns the width of the visitor's screen in pixels.

## Example

Display the width of the screen in pixels:

```
"Screen Width: " + screen.width
```

Result will be:

Screen Width: 1366

## Window Screen Height

The screen.height property returns the height of the visitor's screen in pixels.

### Example

Display the height of the screen in pixels:

```
"Screen Height: " + screen.height
```

Result will be:

Screen Height: 768

## Window Screen Available Width

The screen.availWidth property returns the width of the visitor's screen, in pixels, minus interface features like the Windows Taskbar.

### Example

Display the available width of the screen in pixels:

```
"Available Screen Width: " + screen.availWidth
```

Result will be:

Available Screen Width: 1366

## Window Screen Available Height

The `screen.availHeight` property returns the height of the visitor's screen, in pixels, minus interface features like the Windows Taskbar.

### Example

Display the available height of the screen in pixels:

```
"Available Screen Height: " + screen.availHeight
```

Result will be:

Available Screen Height: 720

## Window Screen Color Depth

The `screen.colorDepth` property returns the number of bits used to display one color.

All modern computers use 24 or 32 bits hardware to display 16,777,216 different colors ("True Colors").

Older computers used 16 bits, which gives a maximum of 65,536 different colors ("High Colors")

Very old computers, and old cell phones used 8 bits ("VGA colors").

### Example

Display the color depth of the screen in bits:

```
"Screen Color Depth: " + screen.colorDepth
```

Result will be:

Screen Color Depth: 24



Some computers report 32. Most computers report 24. Both display "True Colors" (16,777,216 different colors).

---

## Window Screen Pixel Depth

The `screen.pixelDepth` property returns the pixel depth of the screen.

### Example

Display the pixel depth of the screen in bits:

```
"Screen Pixel Depth: " + screen.pixelDepth
```

Result will be:

Screen Pixel Depth: 24



For modern computers, Color Depth and Pixel Depth are equal.

## JavaScript Window Location

The `window.location` object can be used to get the current page address (URL) and to redirect the browser to a new page.

---

### Window Location

The **`window.location`** object can be written without the window prefix.

Some examples:

- `window.location.href` returns the href (URL) of the current page
  - `window.location.hostname` returns the domain name of the web host
  - `window.location.pathname` returns the path and filename of the current page
  - `window.location.protocol` returns the web protocol used (`http://` or `https://`)
  - `window.location.assign` loads a new document
- 

## Window Location Href

The **`window.location.href`** property returns the URL of the current page.

### Example

Display the href (URL) of the current page:

```
"Page location is " + window.location.href;
```

Result is:

Page location is `http://www.w3schools.com/js/js_window_location.asp`

---

## Window Location Hostname

The **`window.location.hostname`** property returns the name of the internet host (of the current page).

### Example

Display the name of the host:

```
"Page host is " + window.location.hostname;
```

Result is:

Page hostname is `www.w3schools.com`

---

## Window Location Pathname

The **window.location.pathname** property returns the path name of page.

### Example

Display the path name of the current URL:

```
"Page path is " + window.location.pathname;
```

Result is:

```
/js/js_window_location.asp
```

---

## Window Location Protocol

The **window.location.protocol** property returns the web protocol of the page.

### Example

Display the web protocol:

```
"Page protocol is " + window.location.protocol;
```

Result is:

Page protocol is http:

---

# Window Location Assign

The **window.location.assign()** method loads a new document.

## Example

Load a new document:

```
<html>
<head>
<script>
function newDoc() {
    window.location.assign("http://www.w3schools.com")
}
</script>
</head>
<body>

<input type="button" value="Load new document" onclick="newDoc()">

</body>
</html>
```

# JavaScript Window History

The window.history object contains the browsers history.

---

## Window History

The **window.history** object can be written without the window prefix.

To protect the privacy of the users, there are limitations to how JavaScript can access this object.

Some methods:

- `history.back()` - same as clicking back in the browser
  - `history.forward()` - same as clicking forward in the browser
- 

## Window History Back

The `history.back()` method loads the previous URL in the history list.

This is the same as clicking the Back button in the browser.

### Example

Create a back button on a page:

```
<html>
<head>
<script>
function goBack() {
    window.history.back()
}
</script>
</head>
<body>

<input type="button" value="Back" onclick="goBack()">

</body>
</html>
```

The output of the code above will be:

---

## Window History Forward

The `history.forward()` method loads the next URL in the history list.



This is the same as clicking the Forward button in the browser.

## Example

Create a forward button on a page:

```
<html>
<head>
<script>
function goForward() {
    window.history.forward()
}
</script>
</head>
<body>

<input type="button" value="Forward" onclick="goForward()">

</body>
</html>
```

The output of the code above will be:

# JavaScript Window Navigator

The window.navigator object contains information about the visitor's browser.

---

## Window Navigator

The **window.navigator** object can be written without the window prefix.

Some examples:

- navigator.appName
- navigator.appCodeName
- navigator.platform

---

## Navigator Cookie Enabled

The property `cookieEnabled` returns true if cookies are enabled, otherwise false:

### Example

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

<script>
document.getElementById("demo").innerHTML =
"Cookies Enabled is " + navigator.cookieEnabled;
</script>
```

## The Browser Names

The properties **`appName`** and **`appCodeName`** return the name of the browser:

### Example

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

<script>
document.getElementById("demo").innerHTML =
"Name is " + navigator.appName + ". Code name is " + navigator.appCodeName;
</script>
```

Did you know?



IE11, Chrome, Firefox, and Safari return `appName` "Netscape".

Chrome, Firefox, IE, Safari, and Opera all return `appCodeName` "Mozilla".

---

## The Browser Engine

The property **product** returns the engine name of the browser:

### Example

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

```
<script>  
document.getElementById("demo").innerHTML = navigator.product;  
</script>
```

---

## The Browser Version I

The property **appVersion** returns version information about the browser:

### Example

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

```
<script>  
document.getElementById("demo").innerHTML = navigator.appVersion;  
</script>
```

---

## The Browser Version II

The property **userAgent** also returns version information about the browser:

### Example

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

```
<script>  
document.getElementById("demo").innerHTML = navigator.userAgent;  
</script>
```

---

## Warning !!!

The information from the navigator object can often be misleading, and should not be used to detect browser versions because:

- Different browsers can use the same name
  - The navigator data can be changed by the browser owner
  - Some browsers misidentify themselves to bypass site tests
  - Browsers cannot report new operating systems, released later than the browser
- 

## The Browser Platform

The property **platform** returns the browser platform (operating system):

### Example

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

```
<script>  
document.getElementById("demo").innerHTML = navigator.platform;  
</script>
```

---

# The Browser Language

The property **language** returns the browser's language:

## Example

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

```
<script>  
document.getElementById("demo").innerHTML = navigator.language;  
</script>
```

---

## Is Java Enabled?

The method **javaEnabled()** returns true if Java is enabled:

## Example

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

```
<script>  
document.getElementById("demo").innerHTML = navigator.javaEnabled();  
</script>
```

# JavaScript Popup Boxes

JavaScript has three kind of popup boxes: Alert box, Confirm box, and Prompt box.

---

## Alert Box

An alert box is often used if you want to make sure information comes through to the user.

When an alert box pops up, the user will have to click "OK" to proceed.

### Syntax

```
window.alert("sometext");
```

The **window.alert** method can be written without the window prefix.

## Example

```
alert("I am an alert box!");
```

---

## Confirm Box

A confirm box is often used if you want the user to verify or accept something.

When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed.

If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

### Syntax

```
window.confirm("sometext");
```

The **window.confirm()** method can be written without the window prefix.

## Example

```
var r = confirm("Press a button");
if (r == true) {
    x = "You pressed OK!";
} else {
    x = "You pressed Cancel!";
}
```

---

## Prompt Box

A prompt box is often used if you want the user to input a value before entering a page.

When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.

If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

### Syntax

```
window.prompt("sometext","defaultText");
```

The **window.prompt()** method can be written without the window prefix.

## Example

```
var person = prompt("Please enter your name", "Harry Potter");
if (person != null) {
    document.getElementById("demo").innerHTML =
    "Hello " + person + "! How are you today?";
}
```

---

## Line Breaks

To display line breaks inside a popup box, use a back-slash followed by the character n.

### Example

```
alert("Hello\nHow are you?");
```

## JavaScript Timing Events

1

2

3

4

5

6 JavaScript can be executed in time-intervals.

7 This is called timing events.

8

9

10

11

12

---



# JavaScript Timing Events

With JavaScript, it is possible to execute some code at specified time-intervals. This is called timing events.

It's very easy to time events in JavaScript. The two key methods that are used are:

- `setInterval()` - executes a function, over and over again, at specified time intervals
- `setTimeout()` - executes a function, once, after waiting a specified number of milliseconds

**Note:** The `setInterval()` and `setTimeout()` are both methods of the HTML DOM Window object.

---

## The `setInterval()` Method

The `setInterval()` method will wait a specified number of milliseconds, and then execute a specified function, and it will continue to execute the function, once at every given time-interval.

### Syntax

```
window.setInterval("javascript function", milliseconds);
```

The **`window.setInterval()`** method can be written without the window prefix.

The first parameter of `setInterval()` should be a function.

The second parameter indicates the length of the time-intervals between each execution.

**Note:** There are 1000 milliseconds in one second.

## Example

Alert "hello" every 3 seconds:

```
setInterval(function () {alert("Hello")}, 3000);
```

The example show you how the setInterval() method works, but it is not very likely that you want to alert a message every 3 seconds.

Below is an example that will display the current time. The setInterval() method is used to execute the function once every 1 second, just like a digital watch.

## Example

Display the current time:

```
var myVar=setInterval(function () {myTimer()}, 1000);

function myTimer() {
    var d = new Date();
    document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
```

---

## How to Stop the Execution?

The clearInterval() method is used to stop further executions of the function specified in the setInterval() method.

### Syntax

```
window.clearInterval(intervalVariable)
```

The **window.clearInterval()** method can be written without the window prefix.

To be able to use the clearInterval() method, you must use a global variable when creating the interval method:

```
myVar=setInterval("javascript function", milliseconds);
```

Then you will be able to stop the execution by calling the clearInterval() method.

## Example

Same example as above, but we have added a "Stop time" button:

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

<button onclick="clearInterval(myVar)">Stop time</button>

<script>
var myVar = setInterval(function () {myTimer()}, 1000);
function myTimer() {
    var d = new Date();
    document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
</script>
```

---

## The setTimeout() Method

### Syntax

```
window.setTimeout("javascript function", milliseconds);
```

The **window.setTimeout()** method can be written without the window prefix.

The setTimeout() method will wait the specified number of milliseconds, and then execute the specified function.

The first parameter of setTimeout() should be a function.

The second parameter indicates how many milliseconds, from now, you want to execute the first parameter.

## Example

Click a button. Wait 3 seconds. The page will alert "Hello":

```
<button onclick = "setTimeout(function(){alert('Hello')},3000)">Try it</button>
```

---

## How to Stop the Execution?

The `clearTimeout()` method is used to stop the execution of the function specified in the `setTimeout()` method.

### Syntax

```
window.clearTimeout(timeoutVariable)
```

The **`window.clearTimeout()`** method can be written without the window prefix.

To be able to use the `clearTimeout()` method, you must use a global variable when creating the timeout method:

```
myVar=setTimeout("javascript function", milliseconds);
```

Then, if the function has not already been executed, you will be able to stop the execution by calling the `clearTimeout()` method.

## Example

Same example as above, but with an added "Stop" button:

```
<button onclick="myVar=setTimeout(function(){alert('Hello')},3000)">Try it</button>
```

```
<button onclick="clearTimeout(myVar)">Try it</button>
```

## JavaScript Cookies

Cookies let you store user information in web pages.

---

## What are Cookies?

Cookies are data, stored in small text files, on your computer.

When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.

Cookies were invented to solve the problem "how to remember information about the user":

- When a user visits a web page, his name can be stored in a cookie.
- Next time the user visits the page, the cookie "remembers" his name.

Cookies are saved in name-value pairs like:

```
username=John Doe
```

When a browser request a web page from a server, cookies belonging to the page is added to the request. This way the server gets the necessary data to "remember" information about users.

---

## Create a Cookie with JavaScript

JavaScript can create, read, and delete cookies with the **document.cookie** property.

With JavaScript, a cookie can be created like this:

```
document.cookie="username=John Doe";
```

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

```
document.cookie="username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
```

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

```
document.cookie="username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC;  
path=/";
```

---

## Read a Cookie with JavaScript

With JavaScript, cookies can be read like this:

```
var x = document.cookie;
```



document.cookie will return all cookies in one string much like: cookie1=value; cookie2=value; cookie3=value;

---

## Change a Cookie with JavaScript

With JavaScript, you can change a cookie the same way as you create it:

```
document.cookie="username=John Smith; expires=Thu, 18 Dec 2013 12:00:00 UTC;  
path="/;
```

The old cookie is overwritten.

---

## Delete a Cookie with JavaScript

Deleting a cookie is very simple. Just set the expires parameter to a passed date:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Note that you don't have to specify a cookie value when you delete a cookie.

---

## The Cookie String

The document.cookie property looks like a normal text string. But it is not.

Even if you write a whole cookie string to document.cookie, when you read it out again, you can only see the name-value pair of it.

If you set a new cookie, older cookies are not overwritten. The new cookie is added to `document.cookie`, so if you read `document.cookie` again you will get something like:

```
cookie1=value; cookie2=value;
```

If you want to find the value of one specified cookie, you must write a JavaScript function that searches for the cookie value in the cookie string.

---

## JavaScript Cookie Example

In the example to follow, we will create a cookie that stores the name of a visitor.

The first time a visitor arrives to the web page, he will be asked to fill in his name. The name is then stored in a cookie.

The next time the visitor arrives at the same page, he will get a welcome message.

For the example we will create 3 JavaScript functions:

1. A function to set a cookie value
  2. A function to get a cookie value
  3. A function to check a cookie value
- 

## A Function to Set a Cookie

First, we create a function that stores the name of the visitor in a cookie variable:

### Example

```
function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime() + (exdays*24*60*60*1000));  
    var expires = "expires="+d.toUTCString();
```

```
document.cookie = cname + "=" + cvalue + "; " + expires;
}
```

### **Example explained:**

The parameters of the function above are the name of the cookie (cname), the value of the cookie (cvalue), and the number of days until the cookie should expire (exdays).

The function sets a cookie by adding together the cookiename, the cookie value, and the expires string.

---

## **A Function to Get a Cookie**

Then, we create a function that returns the value of a specified cookie:

### **Example**

```
function getCookie(cname) {
    var name = cname + "=";
    var ca = document.cookie.split(';');
    for(var i=0; i<ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0)==' ') c = c.substring(1);
        if (c.indexOf(name) == 0) return c.substring(name.length,c.length);
    }
    return "";
}
```

### **Function explained:**

Take the cookiename as parameter (cname).

Create a variable (name) with the text to search for (cname + "=").

Split document.cookie on semicolons into an array called ca (ca = document.cookie.split(';')).

Loop through the ca array (i=0;i<ca.length;i++), and read out each value c=ca[i]).



If the cookie is found (c.indexOf(name) == 0), return the value of the cookie (c.substring(name.length,c.length).

If the cookie is not found, return "".

---

## A Function to Check a Cookie

Last, we create the function that checks if a cookie is set.

If the cookie is set it will display a greeting.

If the cookie is not set, it will display a prompt box, asking for the name of the user, and stores the username cookie for 365 days, by calling the setCookie function:

### Example

```
function checkCookie() {  
    var username=getCookie("username");  
    if (username!="") {  
        alert("Welcome again " + username);  
    }else{  
        username = prompt("Please enter your name:", "");  
        if (username != "" && username != null) {  
            setCookie("username", username, 365);  
        }  
    }  
}
```

---

## All Together Now

### Example

```
function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime() + (exdays*24*60*60*1000));
```

```

    var expires = "expires="+d.toUTCString();
    document.cookie = cname + "=" + cvalue + "; " + expires;
}

function getCookie(cname) {
    var name = cname + "=";
    var ca = document.cookie.split(';');
    for(var i=0; i<ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0)==' ') c = c.substring(1);
        if (c.indexOf(name) == 0) return c.substring(name.length, c.length);
    }
    return "";
}

function checkCookie() {
    var user = getCookie("username");
    if (user != "") {
        alert("Welcome again " + user);
    } else {
        user = prompt("Please enter your name:", "");
        if (user != "" && user != null) {
            setCookie("username", user, 365);
        }
    }
}

```

The example above runs the checkCookie() function when the page loads.