

JavaScript Tutorial

JavaScript is the programming language of HTML and the Web.

JavaScript is easy to learn.

This tutorial will teach you JavaScript from basic to advanced.

Examples in Each Chapter

With our "Try it Yourself" editor, you can change all examples and view the results.

Example

My First JavaScript

Click me to display Date and Time

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

Learn by Examples

Examples are better than 1000 words. Examples are often easier to understand than text explanations.

This tutorial supplements all explanations with clarifying "Try it Yourself" examples.

[JavaScript Examples](#)

If you try all the examples, you will learn a lot about JavaScript, in a very short time!

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

Web pages are not the only place where JavaScript is used. Many desktop and server programs use JavaScript. Node.js is the best known. Some databases, like MongoDB and CouchDB, also use JavaScript as their programming language.

Did You Know?

JavaScript and Java are completely different languages, both in concept and design. JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997. ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

You can read more about the different JavaScript versions in the chapter [JS Versions](#).

Learning Speed

In this tutorial, the learning speed is your choice.

Everything is up to you.

If you are struggling, take a break, or reread the material.

Always make sure you understand **all** the "Try-it-Yourself" examples.

JavaScript References

W3Schools maintains a complete JavaScript reference, including all HTML and browser objects. The reference contains examples for all properties, methods and events, and is continuously updated according to the latest web standards.

[Complete JavaScript Reference](#)

JavaScript Exercises and Quiz Test

Test your JavaScript skills at W3Schools!

[Start JavaScript Exercises!](#)

[Start JavaScript Quiz!](#)

JavaScript Introduction

This page contains some examples of what JavaScript can do.

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is **getElementById()**.

This example uses the method to "find" an HTML element (with id="demo") and changes the element content (**innerHTML**) to "Hello JavaScript":

Example

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

JavaScript accepts both double and single quotes:

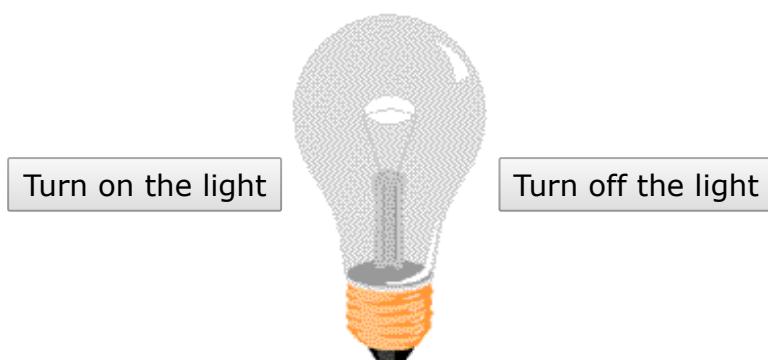
Example

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

JavaScript Can Change HTML Attribute Values

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

The Light Bulb



JavaScript Can Change HTML Styles (CSS)

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

Example

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

JavaScript Can Hide HTML Elements

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

Example

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

JavaScript Can Show HTML Elements

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

Example

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

JavaScript Where To

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>
```

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

JavaScript Functions and Events

A JavaScript **function** is a block of JavaScript code, that can be executed when "called" for. For example, a function can be called when an **event** occurs, like when the user clicks a button.

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.

JavaScript in <head>

In this example, a JavaScript function is placed in the <head> section of an HTML page. The function is invoked (called) when a button is clicked:

Example

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
    document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body>
<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>
</body>
</html>
```

JavaScript in <body>

In this example, a JavaScript function is placed in the <body> section of an HTML page. The function is invoked (called) when a button is clicked:

Example

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

<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

<script>
function myFunction() {
    document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>

</body>
</html>
```

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

External JavaScript

Scripts can also be placed in external files:

External file: myScript.js

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

External scripts are practical when the same code is used in many different web pages. JavaScript files have the file extension **.js**.

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

Example

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

You can place an external script reference in `<head>` or `<body>` as you like. The script will behave as if it was located exactly where the `<script>` tag is located.

External scripts cannot contain `<script>` tags.

External JavaScript Advantages

Placing scripts in external files has some advantages:

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

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

Example

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

External References

External scripts can be referenced with a full URL or with a path relative to the current web page.

This example uses a full URL to link to a script:

Example

```
<script src="https://www.w3schools.com/js/myScript1.js"></script>
```

This example uses a script located in a specified folder on the current web site:

Example

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

This example links to a script located in the same folder as the current page:

Example

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

You can read more about file paths in the chapter [HTML File Paths](#).

JavaScript Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

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

Using innerHTML

To access an HTML element, JavaScript can use the **document.getElementById(id)** method. The **id** attribute defines the HTML element. The **innerHTML** property defines the HTML content:

Example

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

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

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

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

</body>
</html>
```

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

Using document.write()

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

Example

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

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

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

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

Example

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

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

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

</body>
</html>
```

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

Using window.alert()

You can use an alert box to display data:

Example

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

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

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

</body>
</html>
```

Using console.log()

For debugging purposes, you can use the **console.log()** method to display data.

You will learn more about debugging in a later chapter.

Example

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

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

</body>
</html>
```

JavaScript Statements

Example

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

JavaScript Programs

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

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

JavaScript Statements

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

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

Example

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

Most JavaScript programs contain many JavaScript statements.

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

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

Semicolons ;

Semicolons separate JavaScript statements.

Add a semicolon at the end of each executable statement:

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

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

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

On the web, you might see examples without semicolons. Ending statements with semicolon is not required, but highly recommended.

JavaScript White Space

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

The following lines are equivalent:

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

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

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

JavaScript Line Length and Line Breaks

For best readability, programmers often like to avoid code lines longer than 80 characters. If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

Example

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

JavaScript Code Blocks

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

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

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

Example

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

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

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

JavaScript Syntax

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

```
var x, y;          // How to declare variables x = 5; y = 6;      //
How to assign values z = x + y;           // How to compute values
```

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'

JavaScript Variables

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

JavaScript uses the **var** keyword to **declare** variables.

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

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

```
var x;
x = 6;
```

JavaScript Operators

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

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

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

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

JavaScript Expressions

An expression is a combination of values, variables, and operators, which computes to a value. The computation is called an evaluation.

For example, $5 * 10$ evaluates to 50:

```
5 * 10
```

Expressions can also contain variable values:

```
x * 10
```

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

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

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

JavaScript Keywords

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

The **var** keyword tells the browser to create variables:

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

JavaScript Comments

Not all JavaScript statements are "executed".

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

Comments are ignored, and will not be executed:

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

You will learn more about comments in a later chapter.

JavaScript Identifiers

Identifiers are names.

In JavaScript, identifiers are used to name variables (and keywords, and functions, and labels).

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

In JavaScript, the first character must be a letter, or an underscore (_), or a dollar sign (\$). Subsequent characters may be letters, digits, underscores, or dollar signs.

Numbers are not allowed as the first character. This way JavaScript can easily distinguish identifiers from numbers.

JavaScript is Case Sensitive

All JavaScript identifiers are **case sensitive**.

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

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

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

JavaScript and Camel Case

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

Hyphens:

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

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

Underscore:

first_name, last_name, master_card, inter_city.

Upper Camel Case (Pascal Case):

FirstName, LastName, MasterCard, InterCity.



Lower Camel Case:

JavaScript programmers tend to use camel case that starts with a lowercase letter:
firstName, lastName, masterCard, interCity.

JavaScript Character Set

JavaScript uses the **Unicode** character set.

Unicode covers (almost) all the characters, punctuations, and symbols in the world.
For a closer look, please study our [Complete Unicode Reference](#).

JavaScript Comments

JavaScript comments can be used to explain JavaScript code, and to make it more readable.

JavaScript comments can also be used to prevent execution, when testing alternative code.

Single Line Comments

Single line comments start with //.

Any text between // and the end of the line will be ignored by JavaScript (will not be executed).

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

Example

```
// 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 is suitable for code testing.

Adding `//` in front of a code line changes the code lines from an executable line to a comment.

This example uses `//` to prevent execution of one of the code lines:

Example

```
//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.

JavaScript variables are containers for storing data values.

JavaScript Identifiers

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

These unique names are called **identifiers**.

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

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

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

JavaScript identifiers are case-sensitive.

The Assignment Operator

In JavaScript, the equal sign (=) is an "assignment" operator, not an "equal to" operator. This is different from algebra. The following does not make sense in algebra:

```
x = x + 5
```

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

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

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

JavaScript Data Types

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

In programming, text values are called text strings.

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

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

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

Example

```
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 has no value. (Technically it has the value of **undefined**)

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

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

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

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

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

Example

```
<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:

Example

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

Also try this:

Example

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

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

Now try this:

Example

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

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)
```

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

Assignment

```
var x = 10;
```

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

Adding

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

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

Multiplying

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

JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic on numbers:

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

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

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	$x = y$	$x = y$
$+=$	$x += y$	$x = x + y$
$-=$	$x -= y$	$x = x - y$
$*=$	$x *= y$	$x = x * y$
$/=$	$x /= y$	$x = x / y$
$%=$	$x \%= y$	$x = x \% y$

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

Assignment

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

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

JavaScript String Operators

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

Example

```
var txt1 = "John";  
var txt2 = "Doe";  
var txt3 = txt1 + " " + txt2;
```

The result of txt3 will be:

John Doe

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

Example

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

The result of txt1 will be:

What a very nice day

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

Adding Strings and Numbers

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

Example

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

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

```
10  
55  
Hello5
```

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

JavaScript Comparison Operators

Operator	Description
<code>==</code>	equal to
<code>===</code>	equal value and equal type
<code>!=</code>	not equal
<code>!==</code>	not equal value or not equal type
<code>></code>	greater than
<code><</code>	less than
<code>>=</code>	greater than or equal to
<code><=</code>	less than or equal to
<code>?</code>	ternary operator

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

JavaScript Logical Operators

Operator	Description
&&	logical and
	logical or
!	logical not

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

JavaScript Type Operators

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

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

JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.

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

The examples above uses 4 bits unsigned examples. But JavaScript uses 32-bit signed numbers. Because of this, in JavaScript, `~5` will not return 10. It will return -6.
`~0000000000000000000000000000101` will return
1111111111111111111111111111010
Bitwise operators are fully described in the [**JS Bitwise**](#) chapter.

JavaScript Arithmetic

JavaScript Arithmetic Operators

Arithmetic operators perform arithmetic on numbers (literals or variables).

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

Arithmetic Operations

A typical arithmetic operation operates on two numbers.

The two numbers can be literals:

Example

```
var x = 100 + 50;
```

or variables:

Example

```
var x = a + b;
```

or expressions:

Example

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

Operators and Operands

The numbers (in an arithmetic operation) are called **operands**.

The operation (to be performed between the two operands) is defined by an **operator**.

Operand	Operator	Operand
100	+	50

Adding

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

Example

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

Subtracting

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

Example

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

Multiplying

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

Example

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

Dividing

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

Example

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

Remainder

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

Example

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

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

Incrementing

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

Example

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

Decrementing

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

Example

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

Operator Precedence

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

Example

```
var x = 100 + 50 * 3;
```

Is the result of example above the same as $150 * 3$, or is it the same as $100 + 150$?

Is the addition or the multiplication done first?

As in traditional school mathematics, the multiplication is done first.

Multiplication (*) and division (/) have higher **precedence** than addition (+) and subtraction (-).

And (as in school mathematics) the precedence can be changed by using parentheses:

Example

```
var x = (100 + 50) * 3;
```

When using parentheses, the operations inside the parentheses are computed first.

When many operations have the same precedence (like addition and subtraction), they are computed from left to right:

Example

```
var x = 100 + 50 - 3;
```

JavaScript Operator Precedence Values

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

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

Value	Operator	Description	Example
15	**	Exponentiation (ES7)	10 ** 2
14	*	Multiplication	10 * 5
14	/	Division	10 / 5
14	%	Division Remainder	10 % 5
13	+	Addition	10 + 5
13	-	Subtraction	10 - 5
12	<<	Shift left	x << 2
12	>>	Shift right	x >> 2
12	>>>	Shift right (unsigned)	x >>> 2
11	<	Less than	x < y
11	<=	Less than or equal	x <= y
11	>	Greater than	x > y
11	>=	Greater than or equal	x >= y
11	in	Property in Object	"PI" in Math
11	instanceof	Instance of Object	instanceof Array
10	==	Equal	x == y
10	=====	Strict equal	x ===== y
10	!=	Unequal	x != y
10	!=====	Strict unequal	x !===== y

Value	Operator	Description	Example
9	&	Bitwise AND	x & y
8	^	Bitwise XOR	x ^ y
7		Bitwise OR	x y
6	&&	Logical AND	x && y
5		Logical OR	x y
4	? :	Condition	? "Yes" : "No"
3	+=	Assignment	x += y
3	+=	Assignment	x += y
3	-=	Assignment	x -= y
3	*=	Assignment	x *= y
3	%=	Assignment	x %= y
3	<<=	Assignment	x <<= y
3	>>=	Assignment	x >>= y
3	>>>=	Assignment	x >>>= y
3	&=	Assignment	x &= y
3	^=	Assignment	x ^= y
3	=	Assignment	x = y
2	yield	Pause Function	yield x
1	,	Comma	5 , 6

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

JavaScript Assignment

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
<<=	x <<= y	x = x << y
>>=	x >>= y	x = x >> y
>>>=	x >>>= y	x = x >>> y
&=	x &= y	x = x & y
^=	x ^= y	x = x ^ y
=	x = y	x = x y
**=	x **= y	x = x ** y

The **= operator is an experimental part of the ECMAScript 2016 proposal (ES7). It is not stable across browsers. Do not use it.

Assignment Examples

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 Data Types

JavaScript Data Types

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

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

The Concept of Data Types

In programming, data types is an important concept.

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

Without data types, a computer cannot safely solve this:

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

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

JavaScript will treat the example above as:

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

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

Example

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

Example

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

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

JavaScript:

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

JavaScript:

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

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

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

JavaScript Types are Dynamic

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

Example

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

[Try it yourself »](#)

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
```

[Try it yourself »](#)

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
```

[Try it yourself »](#)

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
```

[Try it yourself »](#)

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

Example

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

[Try it yourself »](#)

You will learn more about numbers later in this tutorial.

JavaScript Booleans

Booleans can only have two values: true or false.

Example

```
var x = 5;  
var y = 5;  
var z = 6;  
(x == y)      // Returns true  
(x == z)      // Returns 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.

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

Example

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

Example

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

Undefined

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

Example

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

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

Example

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

Empty Values

An empty value has nothing to do with undefined.

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

Example

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

Null

In JavaScript null is "nothing". It is supposed to be something that doesn't exist. Unfortunately, in JavaScript, the data type of null is an object.

You can consider it a bug in JavaScript that `typeof null` is an object. It should be null.

You can empty an object by setting it to null:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;           // Now value is null, but type is still an object
```

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

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined;    // Now both value and type is undefined
```

Difference Between Undefined and Null

Undefined and null are equal in value but different in type:

```
typeof undefined          // undefined
typeof null              // object

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

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The **typeof** operator can return one of these primitive types:

- string
- number
- boolean
- undefined

Example

```
typeof "John"           // Returns "string"
typeof 3.14             // Returns "number"
typeof true              // Returns "boolean"
typeof false             // Returns "boolean"
typeof x                 // Returns "undefined" (if x has no value)
```

Complex Data

The **typeof** operator can return one of two complex types:

- function
- object

The **typeof** operator returns object for both objects, arrays, and null.

The **typeof** operator does not return object for functions.

Example

```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4]            // Returns "object" (not "array", see note
                           below)
typeof null                 // Returns "object"
typeof function myFunc(){}  // Returns "function"
```

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

JavaScript Functions

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

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

Example

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

JavaScript Function Syntax

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

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

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

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

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

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

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

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

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

Function Invocation

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

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

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

Function Return

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

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

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

Example

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

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

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

The result in x will be:

12

Why Functions?

You can reuse code: Define the code once, and use it many times.

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

Example

Convert Fahrenheit to Celsius:

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

The () Operator Invokes the Function

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

Accessing a function without () will return the function definition instead of the function result:

Example

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

Functions Used as Variable Values

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

Example

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

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

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

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

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

Local Variables

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

Local variables can only be accessed from within the function.

Example

```
// code here can NOT use carName

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

// code here can NOT use carName
```

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

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

JavaScript Objects

Real Life Objects, Properties, and Methods

In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop:

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

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

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

JavaScript Objects

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

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

```
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** called properties or methods.

Object Definition

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

Example

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

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

Example

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

Object Properties

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

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Accessing Object Properties

You can access object properties in two ways:

```
objectName.propertyName
```

or

```
objectName["propertyName"]
```

Example1

```
person.lastName;
```

Example2

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

Object Methods

Objects can also have **methods**.

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

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

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

A method is a function stored as a property.

Example

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

The **this** Keyword

In a function definition, **this** refers to the "owner" of the function.

In the example above, **this** is the **person object** that "owns" the **fullName** function.

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

Read more about the **this** keyword at [JS this Keyword](#).

Accessing Object Methods

You access an object method with the following syntax:

```
objectName.methodName()
```

Example

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

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

Example

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

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

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

```
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 Events

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

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

HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

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

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

JavaScript lets you execute code when events are detected.

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

With single quotes:

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

With double quotes:

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

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

Example

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

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

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

Example

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

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

Example

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

Common HTML Events

Here is a list of some common HTML events:

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

The list is much longer: [W3Schools JavaScript Reference HTML DOM Events](#).

What can JavaScript Do?

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

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

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

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

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

JavaScript Strings

JavaScript strings are used for storing and manipulating text.

JavaScript Strings

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

Example

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

You can use single or double quotes:

Example

```
var carname = "Volvo XC60"; // Double quotes
var carname = 'Volvo XC60'; // 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";
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 = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
var sln = txt.length;
```

Special Characters

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

```
var x = "We are the so-called "Vikings" from the north.;"
```

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

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

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

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

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

Example

```
var x = "We are the so-called \"Vikings\" from the north.;"
```

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

Example

```
var x = 'It\'s alright.';
```

The sequence \\ inserts a backslash in a string:

Example

```
var x = "The character \\ is called backslash.;"
```

Six other escape sequences are valid in JavaScript:

Code	Result
------	--------

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

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

Breaking Long Code Lines

For best readability, programmers often like to avoid code lines longer than 80 characters. If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

Example

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

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

Example

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

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

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

Example

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

You cannot break up a code line with a backslash:

Example

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

Strings Can be Objects

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

```
var firstName = "John";
```

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

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

Example

```
var x = "John";  
var y = new String("John");  
  
// typeof x will return string  
// typeof y will return object
```

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

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

Example

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

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

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

Example

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

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

Or even worse. Objects cannot be compared:

Example

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

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

Example

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

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

Note the difference between `(x==y)` and `(x===y)`. Comparing two JavaScript objects will **always** return false.

JavaScript String Methods

String methods help you to work with strings.

String Methods and Properties

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

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

String Length

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

Example

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

Finding a String in a String

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

Example

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

JavaScript counts positions from zero. 0 is the first position in a string, 1 is the second, 2 is the third ...

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

Example

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

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

Example

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate", 15);
```

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 return the same value?

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

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

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

Extracting String Parts

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

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

The `slice()` Method

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

The method takes 2 parameters: the starting index (position), and the ending index (position).

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

Example

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

The result of `res` will be:

```
Banana
```

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

Example

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

The result of res will be:

```
Banana
```

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

Example

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

or, counting from the end:

Example

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

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

The substring() Method

substring() is similar to slice().

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

Example

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

The result of *res* will be:

```
Banana
```

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

The substr() Method

substr() is similar to slice().

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

Example

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

The result of *res* will be:

```
Banana
```

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

Example

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

The result of res will be:

```
Banana, Kiwi
```

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

Example

```
var str = "Apple, Banana, Kiwi";
var res = str.substr(-4);
```

The result of res will be:

```
Kiwi
```

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 does not change the string it is called on. It returns a new string.

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

Example

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

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

Example

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

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

Example

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

Note that regular expressions are written without quotes.

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

Example

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

You will learn a lot more about regular expressions in the chapter [JavaScript Regular Expressions](#).

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";
var 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.

String.trim()

String.trim() removes whitespace from both sides of a string.

Example

```
var str = "      Hello World!      ";
alert(str.trim());
```

String.trim() is not supported in Internet Explorer 8 or lower.

If you need to support IE 8, you can use String.replace with a regular expression instead:

Example

```
var str = "      Hello World!      ";
alert(str.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, ''));
```

You can also use the replace solution above to add a trim function to the JavaScript String.prototype:

Example

```
if (!String.prototype.trim) {
  String.prototype.trim = function () {
    return this.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, '');
  };
}
var str = "      Hello World!      ";
alert(str.trim());
```

Extracting String Characters

There are 3 methods for extracting string characters:

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

The charAt() Method

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

Example

```
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:

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

Example

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

Property Access

ECMAScript 5 (2009) allows property access [] on strings:

Example

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

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

- It does not work in Internet Explorer 7 or earlier
- It makes strings look like arrays (but they are not)
- If no character is found, [] returns undefined, while charAt() returns an empty string.
- It is read only. str[0] = "A" gives no error (but does not work!)

Example

```
var str = "HELLO WORLD";
str[0] = "A";           // Gives no error, but does not work
str[0];                // returns H
```

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

Converting a String to an Array

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

Example

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

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

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

Example

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

Complete String Reference

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

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

JavaScript Numbers

JavaScript has only one type of number. Numbers can be written with or without decimals.

Example

```
var x = 3.14;      // A number with decimals  
var y = 3;        // A number without decimals
```

[Try it yourself »](#)

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

Example

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

[Try it yourself »](#)

JavaScript Numbers are Always 64-bit Floating Point

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

JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard. This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:

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

Precision

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

Example

```
var x = 999999999999999; // x will be 999999999999999  
var y = 999999999999999; // y will be 1000000000000000
```

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 0.3000000000000004
```

[Try it yourself »](#)

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
```

Adding Numbers and Strings

WARNING !!

JavaScript uses the + operator for both addition and concatenation.
Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
var x = 10;  
var y = 20;  
var z = x + y; // z will be 30 (a number)
```

If you add two strings, the result will be a string concatenation:

Example

```
var x = "10";
var y = "20";
var z = x + y;           // z will be 1020 (a string)
```

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

Example

```
var x = 10;
var y = "20";
var z = x + y;           // z will be 1020 (a string)
```

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

Example

```
var x = "10";
var y = 20;
var z = x + y;           // z will be 1020 (a string)
```

A common mistake is to expect this result to be 30:

Example

```
var x = 10;
var y = 20;
var z = "The result is: " + x + y;
```

A common mistake is to expect this result to be 102030:

Example

```
var x = 10;  
var y = 20;  
var z = "30";  
var result = x + y + z;
```

The JavaScript compiler works from left to right.

First $10 + 20$ is added because x and y are both numbers.

Then $30 + "30"$ is concatenated because z is a string.

Numeric Strings

JavaScript strings can have numeric content:

```
var x = 100;           // x is a number  
  
var y = "100";         // y is a string
```

JavaScript will try to convert strings to numbers in all numeric operations:

This will work:

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

This will also work:

```
var x = "100";  
var y = "10";  
var z = x * y;        // z will be 1000
```

And this will work:

```
var x = "100";
var y = "10";
var z = x - y;           // z will be 90
```

But this will not work:

```
var x = "100";
var y = "10";
var z = x + y;           // z will not be 110 (It will be 10010)
```

In the last example JavaScript uses the + operator to concatenate the strings.

NaN - Not a Number

NaN is a JavaScript reserved word indicating that a number is not a legal number.
Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

Example

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

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

Example

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

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

Example

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

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

Example

```
var x = NaN;  
var y = 5;  
var z = x + y;           // z will be NaN
```

Or the result might be a concatenation:

Example

```
var x = NaN;  
var y = "5";  
var z = x + y;           // z will be NaN5
```

NaN is a number: typeof NaN returns number:

Example

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

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;  
}
```

[Try it yourself »](#)

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"
```

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 from **base 2** to **base 36**.

Hexadecimal is **base 16**. Decimal is **base 10**. Octal is **base 8**. Binary is **base 2**.

Example

```
var myNumber = 32;  
myNumber.toString(10); // returns 32  
myNumber.toString(32); // returns 10  
myNumber.toString(16); // returns 20  
myNumber.toString(8); // returns 40  
myNumber.toString(2); // returns 100000
```

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
```

[Try it yourself »](#)

Do not create Number objects. It slows down execution speed. The **new** keyword complicates the code. This can produce some unexpected results:

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

Example

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

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

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

Example

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

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

Or even worse. Objects cannot be compared:

Example

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

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

Note the difference between `(x==y)` and `(x==y)`. Comparing two JavaScript objects will always return false.

JavaScript Number Methods

Number methods help you work with numbers.

Number Methods and Properties

Primitive values (like 3.14 or 2014), cannot have properties and methods (because they are not objects).

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

The `toString()` Method

`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 characters 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
```

[Try it yourself »](#)

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
```

[Try it yourself »](#)

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
```

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.

All JavaScript data types have a `valueOf()` and a `toString()` method.

Converting Variables to Numbers

There are 3 JavaScript methods that can be used to convert variables to numbers:

- The `Number()` method
- The `parseInt()` method
- The `parseFloat()` method

These methods are not **number** methods, but **global** JavaScript methods.

Global JavaScript Methods

JavaScript global methods can be used on all JavaScript data types.

These are the most relevant methods, when working with numbers:

Method	Description
<code>Number()</code>	Returns a number, converted from its argument.
<code>parseFloat()</code>	Parses its argument and returns a floating point number
<code>parseInt()</code>	Parses its argument and returns an integer

The Number() Method

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

Example

```
Number(true);           // returns 1
Number(false);          // returns 0
Number("10");           // returns 10
Number(" 10 ");          // returns 10
Number("10  ");          // returns 10
Number(" 10  ");         // returns 10
Number("10.33");         // returns 10.33
Number("10,33");         // returns NaN
Number("10 33");         // returns NaN
Number("John");          // returns NaN
```

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

The Number() Method Used on Dates

Number() can also convert a date to a number:

Example

```
Number(new Date("2017-09-30"));    // returns 1506729600000
```

The Number() method above returns the number of milliseconds since 1.1.1970.

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
```

[Try it yourself »](#)

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
```

[Try it yourself »](#)

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

Number Properties

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

JavaScript MIN_VALUE and MAX_VALUE

Example

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

[Try it yourself »](#)

Example

```
var x = Number.MIN_VALUE;
```

[Try it yourself »](#)

JavaScript POSITIVE_INFINITY

Example

```
var x = Number.POSITIVE_INFINITY;
```

[Try it yourself »](#)

`POSITIVE_INFINITY` is returned on overflow:

Example

```
var x = 1 / 0;
```

[Try it yourself »](#)

JavaScript NEGATIVE_INFINITY

Example

```
var x = Number.NEGATIVE_INFINITY;
```

[Try it yourself »](#)

`NEGATIVE_INFINITY` is returned on overflow:

Example

```
var x = -1 / 0;
```

[Try it yourself »](#)

JavaScript NaN - Not a Number

Example

```
var x = Number.NaN;
```

[Try it yourself »](#)

`NaN` is a JavaScript reserved word indicating that a number is not a legal number.
Trying to do arithmetic with a non-numeric string will result in `NaN` (Not a Number):

Example

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

Number Properties Cannot be Used on Variables

Number properties belongs to the JavaScript's number object wrapper called **Number**.
These properties can only be accessed as **Number.MAX_VALUE**.
Using *myNumber.MAX_VALUE*, where *myNumber* is a variable, expression, or value, will return undefined:

Example

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

[Try it yourself »](#)

Complete JavaScript Number Reference

For a complete reference, go to our [Complete JavaScript Number Reference](#).
The reference contains descriptions and examples of all Number properties and methods.

JavaScript Arrays

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

Example

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

What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
var car1 = "Saab"; var car2 = "Volvo"; var car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Creating an Array

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

Syntax:

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

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"  
];
```

Putting a comma after the last element (like "BMW",) is inconsistent across browsers.
IE 8 and earlier will fail.

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 access an array element by referring to the **index number**.

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

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

Example

```
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
```

Array indexes start with 0.

[0] is the first element. [1] is the second element.

Changing an Array Element

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

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

Example

```
var cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
document.getElementById("demo").innerHTML = cars[0];
```

Access the Full Array

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

Example

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

Arrays are Objects

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

But, JavaScript arrays are best described as arrays.

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

Array:

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

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

Object:

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

Array Elements Can Be Objects

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

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

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

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

Array Properties and Methods

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

Examples

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

Array methods are covered in the next chapters.

The length Property

The **length** property of an array returns the length of an array (the number of array elements).

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.length;                                // the length of fruits is 4
```

The length property is always one more than the highest array index.

Accessing the First Array Element

Example

```
fruits = ["Banana", "Orange", "Apple", "Mango"];  
var first = fruits[0];
```

Accessing the Last Array Element

Example

```
fruits = ["Banana", "Orange", "Apple", "Mango"];
var last = fruits[fruits.length - 1];
```

Looping Array Elements

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

Example

```
var fruits, text, fLen, i;
fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;

text = "<ul>";
for (i = 0; i < fLen; i++) {
    text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

You can also use the `Array.forEach()` function:

Example

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

text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";

function myFunction(value) {
    text += "<li>" + value + "</li>";
}
```

Adding Array Elements

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

Example

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

New element can also be added to an array using the length property:

Example

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

WARNING !

Adding elements with high indexes can create undefined "holes" in an array:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[6] = "Lemon";           // adds a new element (Lemon) to
fruits
```

Associative Arrays

Many programming languages support arrays with named indexes.

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

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

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

Example

```
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"
```

WARNING !! If you use named indexes, JavaScript will redefine the array to a standard object. After that, some array methods and properties will produce **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 only complicates the code. It can also produce some unexpected results:

```
var points = new Array(40, 100); // Creates an array with two elements (40 and 100)
```

What if I remove one of the elements?

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

How to Recognize an Array

A common question is: How do I know if a variable is an array?

The problem is that the JavaScript operator **typeof** returns "object":

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
typeof fruits;           // returns object
```

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

Solution 1:

To solve this problem ECMAScript 5 defines a new method **`Array.isArray()`**:

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

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

Solution 2:

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

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

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

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

Solution 3:

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

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
  
fruits instanceof Array      // returns true
```

JavaScript Array Methods

Converting Arrays to Strings

The JavaScript method **toString()** converts an array to a string of (comma separated) array values.

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 in addition you can specify the separator:

Example

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

Popping and Pushing

When you work with arrays, it is easy to remove elements and add new elements.

This is what popping and pushing is:

Popping items **out** of an array, or pushing items **into** an array.

Popping

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

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.pop();           // the value of x is "Mango"
```

Pushing

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

Example

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

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

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.push("Kiwi");   // the value of x is 5
```

Shifting Elements

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

The `shift()` method removes the first array element and "shifts" all other elements to a lower index.

Example

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

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

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.shift();      // the value of x is "Banana"
```

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 `unshift()` method returns the new array length.

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");    // Returns 5
```

Changing Elements

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

Array **indexes** start with 0. [0] is the first array element, [1] is the second, [2] is the third ...

Example

```
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 fruits
```

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** may leave undefined holes in the array. Use `pop()` or `shift()` instead.

Splicing an Array

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

Example

```
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.

Merging (Concatenating) Arrays

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

Example (Merging Two Arrays)

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

The concat() method does not change the existing arrays. It always returns a new array.

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

Example (Merging Three Arrays)

```
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
```

The concat() method can also take values as arguments:

Example (Merging an Array with Values)

```
var arr1 = ["Cecilie", "Lone"];
var myChildren = arr1.concat(["Emil", "Tobias", "Linus"]);
```

Slicing an Array

The **slice()** method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

Example

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

The slice() method creates a new array. It does not remove any elements from the source array.

This example slices out a part of an array starting from array element 3 ("Apple"):

Example

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

The slice() method can take two arguments like slice(1, 3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

Example

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

If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

Example

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(2);
```

Automatic `toString()`

JavaScript automatically converts an array to a comma separated string when a primitive value is expected.

This is always the case when you try to output an array.

These two examples will produce the same result:

Example

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

Example

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

All JavaScript objects have a `toString()` method.

Finding Max and Min Values in an Array

There are no built-in functions for finding the highest or lowest value in a JavaScript array. You will learn how you solve this problem in the next chapter of this tutorial.

Sorting Arrays

Sorting arrays are covered in the next chapter of this tutorial.

Complete Array Reference

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

JavaScript Sorting Arrays

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();           // First sort the elements of fruits
fruits.reverse();        // Then reverse 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.

You can use this code snippet to experiment with numerically and alphabetically sorting:

```
<button onclick="myFunction1()">Sort Alphabetically</button>
<button onclick="myFunction2()">Sort Numerically</button>

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

<script>
var points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction1() {
    points.sort();
    document.getElementById("demo").innerHTML = points;
}
function myFunction2() {
    points.sort(function(a, b){return a - b});
    document.getElementById("demo").innerHTML = points;
}
</script>
```

Sorting an Array in Random Order

Example

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return 0.5 - Math.random()});
```

Find the Highest (or Lowest) Array Value

There are no built-in functions for finding the max or min value in an array.

However, after you have sorted an array, you can use the index to obtain the highest and lowest values.

Sorting ascending:

Example

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

Sorting descending:

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 points[points.length-1] contains the lowest value
```

Sorting a whole array is a very inefficient method if you only want to find the highest (or lowest) value.

Using Math.max() on an Array

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

Example

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

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

Using Math.min() on an Array

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

Example

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

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

My Min / Max JavaScript Methods

The fastest solution is to use a "home made" method.

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

Example (Find Max)

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

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

Example (Find Min)

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

Sorting Object Arrays

JavaScript arrays often contain objects:

Example

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

Even if objects have properties of different data types, the `sort()` method can be used to sort the array.

The solution is to write a compare function to compare the property values:

Example

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

Comparing string properties is a little more complex:

Example

```
cars.sort(function(a, b){  
  var x = a.type.toLowerCase();  
  var y = b.type.toLowerCase();  
  if (x < y) {return -1;}  
  if (x > y) {return 1;}  
  return 0;  
});
```

JavaScript Array Iteration Methods

Array iteration methods operate on every array item.

Array.forEach()

The forEach() method calls a function (a callback function) once for each array element.

Example

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction);

function myFunction(value, index, array) {
    txt = txt + value + "<br>";
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

The example above uses only the value parameter. The example can be rewritten to:

Example

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction);

function myFunction(value) {
    txt = txt + value + "<br>";
}
```

Array.forEach() is supported in all browsers except Internet Explorer 8 or earlier:

				
Yes	9.0	Yes	Yes	Yes

Array.map()

The map() method creates a new array by performing a function on each array element.
The map() method does not execute the function for array elements without values.
The map() method does not change the original array.
This example multiplies each array value by 2:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction);

function myFunction(value, index, array) {
    return value * 2;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

When a callback function uses only the value parameter, the index and array parameters can be omitted:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction);

function myFunction(value) {
    return value * 2;
}
```

Array.map() is supported in all browsers except Internet Explorer 8 or earlier.

				
Yes	9.0	Yes	Yes	Yes

Array.filter()

The filter() method creates a new array with array elements that passes a test.

This example creates a new array from elements with a value larger than 18:

Example

```
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction);

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

In the example above, the callback function does not use the index and array parameters, so they can be omitted:

Example

```
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction);

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

Array.filter() is supported in all browsers except Internet Explorer 8 or earlier.

				
Yes	9.0	Yes	Yes	Yes

Array.reduce()

The reduce() method runs a function on each array element to produce (reduce it to) a single value.

The reduce() method works from left-to-right in the array. See also reduceRight().

The reduce() method does not reduce the original array.

This example finds the sum of all numbers in an array:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduce(myFunction);

function myFunction(total, value, index, array) {
    return total + value;
}
```

Note that the function takes 4 arguments:

- The total (the initial value / previously returned value)
- The item value
- The item index
- The array itself

The example above does not use the index and array parameters. It can be rewritten to:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduce(myFunction);

function myFunction(total, value) {
    return total + value;
}
```

The reduce() method can accept an initial value:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduce(myFunction, 100);

function myFunction(total, value) {
    return total + value;
}
```

Array.reduce() is supported in all browsers except Internet Explorer 8 or earlier.

				
Yes	9.0	Yes	Yes	Yes

Array.reduceRight()

The reduceRight() method runs a function on each array element to produce (reduce it to) a single value.

The reduceRight() works from right-to-left in the array. See also reduce().

The reduceRight() method does not reduce the original array.

This example finds the sum of all numbers in an array:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduceRight(myFunction);

function myFunction(total, value, index, array) {
    return total + value;
}
```

Note that the function takes 4 arguments:

- The total (the initial value / previously returned value)
- The item value
- The item index
- The array itself

The example above does not use the index and array parameters. It can be rewritten to:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduceRight(myFunction);

function myFunction(total, value) {
    return total + value;
}
```

Array.reduce() is supported in all browsers except Internet Explorer 8 or earlier.

				
Yes	9.0	Yes	Yes	Yes

Array.every()

The every() method check if all array values pass a test.

This example check if all array values are larger than 18:

Example

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

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

When a callback function uses the first parameter only (value), the other parameters can be omitted:

Example

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

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

Array.every() is supported in all browsers except Internet Explorer 8 or earlier.

				
Yes	9.0	Yes	Yes	Yes

Array.some()

The some() method check if some array values pass a test.

This example check if some array values are larger than 18:

Example

```
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.some(myFunction);

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.some() is supported in all browsers except Internet Explorer 8 or earlier.

				
Yes	9.0	Yes	Yes	Yes

Array.indexOf()

Search an array for an element value and returns its position.

Note: The first item has position 0, the second item has position 1, and so on.

Example

Search an array for the item "Apple":

```
var fruits = ["Apple", "Orange", "Apple", "Mango"];
var a = fruits.indexOf("Apple");
```

Array.indexOf() is supported in all browsers except Internet Explorer 8 or earlier.

Method					
indexOf()	Yes	9.0	Yes	Yes	Yes

Syntax

```
array.indexOf(item, start)
```

item Required. The item to search for.

start Optional. Where to start the search. Negative values will start at the given position counting from the end, and search to the end.

Array.indexOf() returns -1 if the item is not found.

If the item is present more than once, it returns the position of the first occurrence.

Array.lastIndexOf()

Array.lastIndexOf() is the same as Array.indexOf(), but searches from the end of the array.

Example

Search an array for the item "Apple":

```
var fruits = ["Apple", "Orange", "Apple", "Mango"];
var a = fruits.lastIndexOf("Apple");
```

`Array.lastIndexOf()` is supported in all browsers except Internet Explorer 8 or earlier.

Method					
<code>lastIndexOf()</code>	Yes	9.0	Yes	Yes	Yes

Syntax

```
array.lastIndexOf(item, start)
```

item Required. The item to search for

start Optional. Where to start the search. Negative values will start at the given position counting from the end, and search to the beginning

Array.find()

The `find()` method returns the value of the first array element that passes a test function. This example finds (returns the value of) the first element that is larger than 18:

Example

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

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

`Array.find()` is not supported in older browsers. The first browser versions with full support is listed below.

45	12	25	8	32

Array.findIndex()

The `findIndex()` method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

Example

```
var numbers = [4, 9, 16, 25, 29];
var first = numbers.findIndex(myFunction);

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

`Array.findIndex()` is not supported in older browsers. The first browser versions with full support is listed below.

				
45	12	25	8	32

JavaScript Date Objects

JavaScript **Date Object** lets us work with dates:

Tue Oct 02 2018 17:41:44 GMT+0900 (日本標準時)

Year: 2018

Month: 10

Day: 2

Hours: 17

Minutes 41

Seconds: 44

Example

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

JavaScript Date Output

By default, JavaScript will use the browser's time zone and display a date as a full text string:

Tue Oct 02 2018 17:41:44 GMT+0900 (日本標準時)

You will learn much more about how to display dates, later in this tutorial.

Creating Date Objects

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

There are **4 ways** to create a new date object:

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

new Date()

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

Example

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

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

`new Date(year, month, ...)`

`new Date(year, month, ...)` creates a new date object with a **specified date and time**. 7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

Example

```
var d = new Date(2018, 11, 24, 10, 33, 30, 0);
```

JavaScript counts months from 0 to 11.

January is 0. December is 11.

6 numbers specify year, month, day, hour, minute, second:

Example

```
var d = new Date(2018, 11, 24, 10, 33, 30);
```

5 numbers specify year, month, day, hour, and minute:

Example

```
var d = new Date(2018, 11, 24, 10, 33);
```

4 numbers specify year, month, day, and hour:

Example

```
var d = new Date(2018, 11, 24, 10);
```

3 numbers specify year, month, and day:

Example

```
var d = new Date(2018, 11, 24);
```

2 numbers specify year and month:

Example

```
var d = new Date(2018, 11);
```

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

Example

```
var d = new Date(2018);
```

Previous Century

One and two digit years will be interpreted as 19xx:

Example

```
var d = new Date(99, 11, 24);
```

Example

```
var d = new Date(9, 11, 24);
```

`new Date(dateString)`

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

Example

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

Date strings are described in the next chapter.

JavaScript Stores Dates as Milliseconds

JavaScript stores dates as number of milliseconds since January 01, 1970, 00:00:00 UTC (Universal Time Coordinated).

Zero time is January 01, 1970 00:00:00 UTC.

Now the time is: **1538469704705** milliseconds past January 01, 1970

`new Date(milliseconds)`

`new Date(milliseconds)` creates a new date object as **zero time plus milliseconds**:

Example

```
var d = new Date(0);
```

01 January 1970 **plus** 100 000 000 000 milliseconds is approximately 03 March 1973:

Example

```
var d = new Date(100000000000);
```

January 01 1970 **minus** 100 000 000 000 milliseconds is approximately October 31 1966:

Example

```
var d = new Date(-100000000000);
```

Example

```
var d = new Date(86400000);
```

One day (24 hours) is 86 400 000 milliseconds.

Date Methods

When a Date object is created, a number of **methods** allow you to operate on it. Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of date objects, using either local time or UTC (universal, or GMT) time.

Date methods and time zones are covered in the next chapters.

Displaying Dates

JavaScript will (by default) output dates in full text string format:

```
Wed Mar 25 2015 09:00:00 GMT+0900 (日本標準時)
```

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

Example

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

Same as:

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

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

Example

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

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

Example

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

JavaScript Date Formats

JavaScript Date Input

There are generally 3 types of JavaScript date input formats:

Type	Example
ISO Date	"2015-03-25" (The International Standard)
Short Date	"03/25/2015"
Long Date	"Mar 25 2015" or "25 Mar 2015"

The ISO format follows a strict standard in JavaScript.

The other formats are not so well defined and might be browser specific.

JavaScript Date Output

Independent of input format, JavaScript will (by default) output dates in full text string format:

```
Wed Mar 25 2015 09:00:00 GMT+0900 (日本標準時)
```

JavaScript ISO Dates

ISO 8601 is the international standard for the representation of dates and times.
The ISO 8601 syntax (YYYY-MM-DD) is also the preferred JavaScript date format:

Example (Complete date)

```
var d = new Date("2015-03-25");
```

The computed date will be relative to your time zone. Depending on your time zone, the result above will vary between March 24 and March 25.

ISO Dates (Year and Month)

ISO dates can be written without specifying the day (YYYY-MM):

Example

```
var d = new Date("2015-03");
```

Time zones will vary the result above between February 28 and March 01.

ISO Dates (Only Year)

ISO dates can be written without month and day (YYYY):

Example

```
var d = new Date("2015");
```

Time zones will vary the result above between December 31 2014 and January 01 2015.

ISO Dates (Date-Time)

ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):

Example

```
var d = new Date("2015-03-25T12:00:00Z");
```

Date and time is separated with a capital T.

UTC time is defined with a capital letter Z.

If you want to modify the time relative to UTC, remove the Z and add +HH:MM or -HH:MM instead:

Example

```
var d = new Date("2015-03-25T12:00:00-06:30");
```

UTC (Universal Time Coordinated) is the same as GMT (Greenwich Mean Time).

Omitting T or Z in a date-time string can give different result in different browser.

Time Zones

When setting a date, without specifying the time zone, JavaScript will use the browser's time zone.

When getting a date, without specifying the time zone, the result is converted to the browser's time zone.

In other words: If a date/time is created in GMT (Greenwich Mean Time), the date/time will be converted to CDT (Central US Daylight Time) if a user browses from central US.

JavaScript Short Dates.

Short dates are written with an "MM/DD/YYYY" syntax like this:

Example

```
var d = new Date("03/25/2015");
```

WARNINGS !

In some browsers, months or days with no leading zeroes may produce an error:

```
var d = new Date("2015-3-25");
```

The behavior of "YYYY/MM/DD" is undefined.

Some browsers will try to guess the format. Some will return NaN.

```
var d = new Date("2015/03/25");
```

The behavior of "DD-MM-YYYY" is also undefined.

Some browsers will try to guess the format. Some will return NaN.

```
var d = new Date("25-03-2015");
```

JavaScript Long Dates.

Long dates are most often written with a "MMM DD YYYY" syntax like this:

Example

```
var d = new Date("Mar 25 2015");
```

Month and day can be in any order:

Example

```
var d = new Date("25 Mar 2015");
```

And, month can be written in full (January), or abbreviated (Jan):

Example

```
var d = new Date("January 25 2015");
```

Example

```
var d = new Date("Jan 25 2015");
```

Commas are ignored. Names are case insensitive:

Example

```
var d = new Date("JANUARY, 25, 2015");
```

Date Input - Parsing Dates

If you have a valid date string, you can use the **Date.parse()** method to convert it to milliseconds.

Date.parse() returns the number of milliseconds between the date and January 1, 1970:

Example

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

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

Example

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

JavaScript Get Date Methods

These methods can be used for getting information from a date object:

Method	Description
getFullYear()	Get the year as a four digit number (yyyy)
getMonth()	Get the month as a number (0-11)
getDate()	Get the day as a number (1-31)
getHours()	Get the hour (0-23)
getMinutes()	Get the minute (0-59)
getSeconds()	Get the second (0-59)
getMilliseconds()	Get the millisecond (0-999)
getTime()	Get the time (milliseconds since January 1, 1970)
getDay()	Get the weekday as a number (0-6)
Date.now()	Get the time. ECMAScript 5.

The `getTime()` Method

The **getTime()** method returns the number of milliseconds since January 1, 1970:

Example

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

The getFullYear() Method

The **getFullYear()** method returns the year of a date as a four digit number:

Example

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

The getMonth() Method

The **getMonth()** method returns the month of a date as a number (0-11):

Example

```
var d = new Date();
document.getElementById("demo").innerHTML = d.getMonth();
```

In JavaScript, the first month (January) is month number 0, so December returns month number 11.

You can use an array of names, and getMonth() to return the month as a name:

Example

```
var d = new Date();
var months = ["January", "February", "March", "April", "May", "June",
"July", "August", "September", "October", "November", "December"];
document.getElementById("demo").innerHTML = months[d.getMonth()];
```

The getDate() Method

The **getDate()** method returns the day of a date as a number (1-31):

Example

```
var d = new Date();
document.getElementById("demo").innerHTML = d.getDate();
```

The getHours() Method

The **getHours()** method returns the hours of a date as a number (0-23):

Example

```
var d = new Date();
document.getElementById("demo").innerHTML = d.getHours();
```

The getMinutes() Method

The **getMinutes()** method returns the minutes of a date as a number (0-59):

Example

```
var d = new Date();
document.getElementById("demo").innerHTML = d.getMinutes();
```

The getSeconds() Method

The **getSeconds()** method returns the seconds of a date as a number (0-59):

Example

```
var d = new Date();
document.getElementById("demo").innerHTML = d.getSeconds();
```

The getMilliseconds() Method

The **getMilliseconds()** method returns the milliseconds of a date as a number (0-999):

Example

```
var d = new Date();
document.getElementById("demo").innerHTML = d.getMilliseconds();
```

The getDay() Method

The **getDay()** method returns the weekday of a date as a number (0-6):

Example

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

In JavaScript, the first day of the week (0) means "Sunday", even if some countries in the world consider the first day of the week to be "Monday"

You can use an array of names, and **getDay()** to return the weekday as a name:

Example

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

UTC Date Methods

UTC date methods are used for working with UTC dates (Universal Time Zone dates):

Method	Description
getUTCDate()	Same as getDate(), but returns the UTC date
getUTCDay()	Same as getDay(), but returns the UTC day
getUTCFullYear()	Same as getFullYear(), but returns the UTC year
getUTCHours()	Same as getHours(), but returns the UTC hour
getUTCMilliseconds()	Same as getMilliseconds(), but returns the UTC milliseconds
getUTCMilliseconds()	Same as getMinutes(), but returns the UTC minutes
getUTCMonth()	Same as getMonth(), but returns the UTC month
getUTCSeconds()	Same as getSeconds(), but returns the UTC seconds

Complete JavaScript Date Reference

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

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

JavaScript Random

Math.random()

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

Example

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

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

JavaScript Random Integers

Math.random() used with Math.floor() can be used to return random integers.

Example

```
Math.floor(Math.random() * 10); // returns a random integer from 0 to 9
```

Example

```
Math.floor(Math.random() * 11); // returns a random integer from 0 to 10
```

Example

```
Math.floor(Math.random() * 100); // returns a random integer from 0 to 99
```

Example

```
Math.floor(Math.random() * 101); // returns a random integer from 0 to 100
```

Example

```
Math.floor(Math.random() * 10) + 1; // returns a random integer from 1 to 10
```

Example

```
Math.floor(Math.random() * 100) + 1; // returns a random integer from 1 to 100
```

A Proper Random Function

As you can see from the examples above, it might be a good idea to create a proper random function to use for all random integer purposes.

This JavaScript function always returns a random number between min (included) and max (excluded):

Example

```
function getRndInteger(min, max) {  
    return Math.floor(Math.random() * (max - min)) + min;  
}
```

This JavaScript function always returns a random number between min and max (both included):

Example

```
function getRndInteger(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

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
```

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
<code>==</code>	equal to	<code>if (day == "Monday")</code>
<code>></code>	greater than	<code>if (salary > 9000)</code>
<code><</code>	less than	<code>if (age < 18)</code>

The Boolean value of an expression is the basis for all JavaScript comparisons and conditions.

Everything With a "Value" is True

Examples

```
100  
3.14  
-15  
"Hello"  
"false"  
7 + 1 + 3.14
```

Everything Without a "Value" is False

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

```
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
```

Booleans Can be Objects

Normally JavaScript booleans are primitive values created from literals:

```
var x = false;
```

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

```
var y = new Boolean(false);
```

Example

```
var x = false;  
var y = new Boolean(false);  
  
// typeof x returns boolean  
// typeof y returns object
```

[Try it yourself »](#)

Do not create Boolean objects. It slows down execution speed. The **new** keyword complicates the code. This can produce some unexpected results:

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

Example

```
var x = false;  
var y = new Boolean(false);  
  
// (x == y) is true because x and y have equal values
```

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

Example

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

Or even worse. Objects cannot be compared:

Example

```
var x = new Boolean(false);
var y = new Boolean(false);

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

Note the difference between `(x==y)` and `(x==y)`. Comparing two JavaScript objects will always return false.

Complete Boolean Reference

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

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

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

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	Try it
&&	and	(x < 10 && y > 1) is true	Try it »
	or	(x == 5 y == 5) is false	Try it »
!	not	!(x == y) is true	Try it »

Conditional (Ternary) Operator

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

Syntax

```
variablename = (condition) ? value1:value2
```

Example

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

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

Comparing Different Types

Comparing data of different types may give unexpected results.

When comparing a string with a number, JavaScript will convert the string to a number when doing the comparison. An empty string converts to 0. A non-numeric string converts to NaN which is always false.

Case	Value	Try
<code>2 < 12</code>	true	Try it »
<code>2 < "12"</code>	true	Try it »
<code>2 < "John"</code>	false	Try it »
<code>2 > "John"</code>	false	Try it »
<code>2 == "John"</code>	false	Try it »
<code>"2" < "12"</code>	false	Try it »
<code>"2" > "12"</code>	true	Try it »
<code>"2" == "12"</code>	false	Try it »

When comparing two strings, "2" will be greater than "12", because (alphabetically) 1 is less than 2.

To secure a proper result, variables should be converted to the proper type before comparison:

```
age = Number(age);
if (isNaN(age)) {
    voteable = "Input is not a number";
} else {
    voteable = (age < 18) ? "Too young" : "Old enough";
}
```

JavaScript if else and else if

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

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

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

The switch statement is described in the next chapter.

The if Statement

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

Syntax

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

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

Example

Make a "Good day" greeting if the hour is less than 18:00:

```
if (hour < 18) {  
    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 hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

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

The result of greeting will be:

```
Good 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
```

More Examples

[Random link](#) This example will write a link to either W3Schools or to the World Wildlife Foundation (WWF). By using a random number, there is a 50% chance for each of the links.

JavaScript Switch Statement

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

The JavaScript Switch Statement

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

Syntax

```
switch(expression) {      case x:          code block      break;      case y:  
    code block        break;      default:         code block }
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

(Sunday=0, Monday=1, Tuesday=2 ..)

This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {  
    case 0:  
        day = "Sunday";  
        break;  
    case 1:  
        day = "Monday";  
        break;  
    case 2:  
        day = "Tuesday";  
        break;  
    case 3:  
        day = "Wednesday";  
        break;  
    case 4:  
        day = "Thursday";  
        break;  
    case 5:  
        day = "Friday";  
        break;  
    case 6:  
        day = "Saturday";  
}  
}
```

The result of `day` will be:

Tuesday

The break Keyword

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

This will stop the execution of more code and 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.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

The default Keyword

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

Example

The `getDay()` method returns the weekday as a number between 0 and 6. If today is neither Saturday (6) nor Sunday (0), write a default message:

```
switch (new Date().getDay()) {  
    case 6:  
        text = "Today is Saturday";  
        break;  
    case 0:  
        text = "Today is Sunday";  
        break;  
    default:  
        text = "Looking forward to the Weekend";  
}
```

The result of `text` will be:

```
Looking forward to the Weekend
```

The **default** case does not have to be the last case in a switch block:

Example

```
switch (new Date().getDay()) {  
    default:  
        text = "Looking forward to the Weekend";  
        break;  
    case 6:  
        text = "Today is Saturday";  
        break;  
    case 0:  
        text = "Today is Sunday";  
}  
}
```

If default is not the last case in the switch block, remember to end the default case with a break.

Common Code Blocks

Sometimes you will want different switch cases to use the same code.

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

Example

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

Switching Details

If multiple cases matches a case value, the **first** case is selected.

If no matching cases are found, the program continues to the **default** label.

If no default label is found, the program continues to the statement(s) **after the switch**.

Strict Comparison

Switch cases use **strict** comparison (==).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

In this example there will be no match for x:

Example

```
var x = "0";
switch (x) {
    case 0:
        text = "Off";
        break;
    case 1:
        text = "On";
        break;
    default:
        text = "No value found";
}
```

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:

```
var i;
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 has the following syntax:

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

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example

```
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 initialize the variable used in the loop (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 increments the value of the initial variable.

This is not always the case, JavaScript doesn't care, and statement 3 is optional.

Statement 3 can do anything like negative increment (`i--`), positive increment (`i = i + 15`), or anything else.

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

Example

```
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];
}
```

The While Loop

The while loop and the do/while loop will be explained in the next chapter.

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

```
var 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

```
var 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>";  
}
```

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

To label JavaScript statements you precede the statements with a label name and a colon:

```
label:  
statements
```

The break and the continue statements are the only JavaScript statements that can "jump out of" a code block.

Syntax:

```
break Labelname;  
  
continue Labelname;
```

The continue statement (with or without a label reference) can only be used to **skip one loop iteration**.

The break statement, without a label reference, can only be used to **jump out of a loop or a switch**.

With a label reference, the break statement can be used to **jump out of any code block**:

Example

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

A code block is a block of code between { and }.

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" *
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** *
- The data type of a variable that has not been assigned a value is also **undefined** *

You cannot use **typeof** to determine if a JavaScript object is an array (or a date).

The Data Type of typeof

The **typeof** operator is not a variable. It is an operator. Operators (+ - * /) do not have any data type.

But, the **typeof** operator always **returns a string** (containing the type of the operand).

The constructor Property

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

Example

```
"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;  
}
```

Or even simpler, you can check if the object is an Array function:

Example

```
function isArray(myArray) {  
    return myArray.constructor === Array;  
}
```

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;  
}
```

Or even simpler, you can check if the object is a Date function:

Example

```
function isDate(myDate) {  
    return myDate.constructor === Date;  
}
```

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()
```

In the chapter [Number Methods](#), you will find more methods that can be used to convert numbers to strings:

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

Converting 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)"
```

In the chapter [Date Methods](#), you will find more methods that can be used to convert dates to strings:

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

Converting 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
```

In the chapter [Number Methods](#), you will find more methods that can be used to convert strings to numbers:

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

The Unary + Operator

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

Example

```
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" + 2       // returns "52"        because 2 is converted to "2"
"5" - 2       // returns 3          because "5" is converted to 5
"5" * "2"     // returns 10         because "5" and "2" are converted to 5 and
2
```

Automatic String Conversion

JavaScript automatically calls the variable's `toString()` function when you try to "output" an object or a variable:

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

// if myVar = {name:"Fjohn"} // toString converts to "[object Object]"
// if myVar = [1,2,3,4]      // toString converts to "1,2,3,4"
// if myVar = new Date()     // toString converts to "Fri Jul 18 2014
09:08:55 GMT+0200"
```

Numbers and booleans are also converted, but this is not very visible:

```
// if myVar = 123            // toString converts to "123"
// if myVar = true           // toString converts to "true"
// if myVar = false          // toString converts to "false"
```

JavaScript Type Conversion Table

This table shows the result of converting different JavaScript values to Number, String, and Boolean:

OriginalValue	Converted to Number	Converted to String	Converted to Boolean	Try it
false	0	"false"	false	Try it »
true	1	"true"	true	Try it »
0	0	"0"	false	Try it »
1	1	"1"	true	Try it »
"0"	0	"0"	true	Try it »
"000"	0	"000"	true	Try it »
"1"	1	"1"	true	Try it »
NaN	NaN	"NaN"	false	Try it »
Infinity	Infinity	"Infinity"	true	Try it »
-Infinity	-Infinity	"-Infinity"	true	Try it »
""	0	""	false	Try it »
"20"	20	"20"	true	Try it »
"twenty"	NaN	"twenty"	true	Try it »
[]	0	""	true	Try it »
[20]	20	"20"	true	Try it »
[10,20]	NaN	"10,20"	true	Try it »

["twenty"]	NaN	"twenty"	true	Try it »
["ten", "twenty"]	NaN	"ten,twenty"	true	Try it »
function(){}()	NaN	"function(){}()"	true	Try it »
{ }	NaN	"[object Object]"	true	Try it »
null	0	"null"	false	Try it »
undefined	NaN	"undefined"	false	Try it »

Values in quotes indicate string values.

Red values indicate values (some) programmers might not expect.

JavaScript Bitwise Operations

JavaScript Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Examples

Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
5 1	5	0101 0001	0101
~ 5	10	~0101	1010
5 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

JavaScript Uses 32 bits Bitwise Operands

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.

Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers.

After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.

The examples above uses 4 bits unsigned binary numbers. Because of this `~5` returns 10.

Since JavaScript uses 32 bits signed integers, it will not return 10. It will return -6.

`0000000000000000000000000000101` (5)

`111111111111111111111111111010` ($\sim 5 = -6$)

A signed integer uses the leftmost bit as the minus sign.

Bitwise AND

When a bitwise AND is performed on a pair of bits, it returns 1 if both bits are 1.

One bit example:

4 bits example:

Operation	Result
<code>0 & 0</code>	0
<code>0 & 1</code>	0
<code>1 & 0</code>	0
<code>1 & 1</code>	1

Operation	Result
<code>1111 & 0000</code>	0000
<code>1111 & 0001</code>	0001
<code>1111 & 0010</code>	0010
<code>1111 & 0100</code>	0100

Bitwise OR

When a bitwise OR is performed on a pair of bits, it returns 1 if one of the bits are 1:

One bit example:

4 bits example:

Operation	Result
<code>0 0</code>	0
<code>0 1</code>	1
<code>1 0</code>	1
<code>1 1</code>	1

Operation	Result
<code>1111 0000</code>	1111
<code>1111 0001</code>	1111
<code>1111 0010</code>	1111
<code>1111 0100</code>	1111

Bitwise XOR

When a bitwise XOR is performed on a pair of bits, it returns 1 if the bits are different:

One bit example:

Operation	Result
$0 \wedge 0$	0
$0 \wedge 1$	1
$1 \wedge 0$	1
$1 \wedge 1$	0

Operation	Result
$1111 \wedge 0000$	1111
$1111 \wedge 0001$	1110
$1111 \wedge 0010$	1101
$1111 \wedge 0100$	1011

JavaScript Bitwise AND (&)

Bitwise AND returns 1 only if both bits are 1:

Decimal	Binary
5	0000000000000000000000000000000101
1	0000000000000000000000000000000001
5 & 1	0000000000000000000000000000000001 (1)

Example

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

JavaScript Bitwise OR (|)

Bitwise or returns 1 if one of the bits are 1:

Decimal	Binary
5	00000000000000000000000000000000101
1	00000000000000000000000000000000001
5 1	00000000000000000000000000000000101 (5)

Example

```
var x = 5 | 1;
```

JavaScript Bitwise XOR (^)

Bitwise XOR returns 1 if the bits are different:

Decimal	Binary
5	00000000000000000000000000000000101
1	00000000000000000000000000000000001
5 ^ 1	00000000000000000000000000000000100 (4)

Example

```
var x = 5 ^ 1;
```

JavaScript Bitwise NOT (\sim)

Decimal	Binary
5	00000000000000000000000000000000101
~ 5	111111111111111111111111111111010 (-6)

Example

```
var x = ~5;
```

JavaScript (Zero Fill) Bitwise Left Shift ($<<$)

This is a zero fill left shift. One or more zero bits are pushed in from the right, and the leftmost bits fall off:

Decimal	Binary
5	00000000000000000000000000000000101
$5 << 1$	000000000000000000000000000000001010 (10)

Example

```
var x = 5 << 1;
```

JavaScript (Sign Preserving) Bitwise Right Shift (>>)

This is a sign preserving right shift. Copies of the leftmost bit are pushed in from the left, and the rightmost bits fall off:

Decimal	Binary
-5	11111111111111111111111111111111011
-5 >> 1	1111111111111111111111111111111101 (-3)

Example

```
var x = -5 >> 1;
```

JavaScript (Zero Fill) Right Shift (>>>)

This is a zero fill right shift. One or more zero bits are pushed in from the left, and the rightmost bits fall off:

Decimal	Binary
5	0000000000000000000000000000000101
5 >>> 1	0000000000000000000000000000000010 (2)

Example

```
var x = 5 >>> 1;
```

Binary Numbers

Binary numbers with only one bit set is easy to understand:

Setting a few more bits reveals the binary pattern:

JavaScript binary numbers are stored in two's complement format.

This means that a negative number is the bitwise NOT of the number plus 1:

Converting Decimal to Binary

Example

```
function dec2bin(dec){  
    return (dec >>> 0).toString(2);  
}
```

Converting Binary to Decimal

Example

```
function bin2dec(bin){  
    return parseInt(bin, 2).toString(10);  
}
```

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 String

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

Example

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

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

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 replace() With a String

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

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

Use String replace() With a Regular Expression

Example

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

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

The result in res will be:

```
Visit W3Schools!
```

Did You Notice?

Regular expression arguments (instead of string arguments) can be used in the methods above. Regular expressions can make your search much more powerful (case insensitive for example).

Regular Expression Modifiers

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

Modifier	Description	Try it
i	Perform case-insensitive matching	Try it »
g	Perform a global match (find all matches rather than stopping after the first match)	Try it »
m	Perform multiline matching	Try it »

Regular Expression Patterns

Brackets are used to find a range of characters:

Expression	Description	Try it
[abc]	Find any of the characters between the brackets	Try it »
[0-9]	Find any of the digits between the brackets	Try it »
(x y)	Find any of the alternatives separated with	Try it »

Metacharacters are characters with a special meaning:

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

Quantifiers define quantities:

Quantifier	Description	Try it
n+	Matches any string that contains at least one n	Try it »
n*	Matches any string that contains zero or more occurrences of n	Try it »
n?	Matches any string that contains zero or one occurrences of n	Try it »

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
```

Complete RegExp Reference

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

The reference contains descriptions and examples of all `RegExp` properties and methods.

JavaScript Errors - Throw and Try to Catch

The **try** statement lets you test a block of code for errors.

The **catch** statement lets you handle the error.

The **throw** statement lets you create custom errors.

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

Errors Will Happen!

When executing JavaScript code, different errors can occur.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

Example

In this example we have written alert as adddlert to deliberately produce an error:

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

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

JavaScript catches **adddlert** as an error, and executes the catch code to handle it.

JavaScript try and catch

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements **try** and **catch** come in pairs:

```
try {  
    Block of code to try  
}  
catch(err) {  
    Block of code to handle errors  
}
```

JavaScript Throws Errors

When an error occurs, JavaScript will normally stop and generate an error message.

The technical term for this is: JavaScript will **throw an exception (throw an error)**.

JavaScript will actually create an **Error object** with two properties: **name** and **message**.

The throw Statement

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

Technically you can **throw an exception (throw an error)**.

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

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

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

Input Validation Example

This example examines input. If the value is wrong, an exception (err) is thrown. The exception (err) is caught by the catch statement and a custom error message is displayed:

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

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

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>

<script>
function myFunction() {
    var message, x;
    message = document.getElementById("p01");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "") throw "empty";
        if(isNaN(x)) throw "not a number";
        x = Number(x);
        if(x < 5) throw "too low";
        if(x > 10) throw "too high";
    }
    catch(err) {
        message.innerHTML = "Input is " + err;
    }
}
</script>

</body>
</html>
```

HTML Validation

The code above is just an example.

Modern browsers will often use a combination of JavaScript and built-in HTML validation, using predefined validation rules defined in HTML attributes:

```
<input id="demo" type="number" min="5" max="10" step="1"
```

You can read more about forms validation in a later chapter of this tutorial.

The finally Statement

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

```
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("p01");  
    message.innerHTML = "";  
    x = document.getElementById("demo").value;  
    try {  
        if(x == "") throw "is empty";  
        if(isNaN(x)) throw "is not a number";  
        x = Number(x);  
        if(x > 10) throw "is too high";  
        if(x < 5) throw "is too low";  
    }  
    catch(err) {  
        message.innerHTML = "Error: " + err + ".";  
    }  
    finally {  
        document.getElementById("demo").value = "";  
    }  
}
```

The Error Object

JavaScript has a built in error object that provides error information when an error occurs.

The error object provides two useful properties: name and message.

Error Object Properties

Property	Description
name	Sets or returns an error name
message	Sets or returns an error message (a string)

Error Name Values

Six different values can be returned by the error name property:

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

The six different values are described below.

Eval Error

An **EvalError** indicates an error in the eval() function.

Newer versions of JavaScript does not throw any EvalError. Use SyntaxError instead.

Range Error

A **RangeError** is thrown if you use a number that is outside the range of legal values.

For example: You cannot set the number of significant digits of a number to 500.

Example

```
var num = 1;
try {
    num.toPrecision(500);    // A number cannot have 500 significant digits
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

Reference Error

A **ReferenceError** is thrown if you use (reference) a variable that has not been declared:

Example

```
var x;
try {
    x = y + 1;    // y cannot be referenced (used)
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

Syntax Error

A **SyntaxError** is thrown if you try to evaluate code with a syntax error.

Example

```
try {
    eval("alert('Hello)"); // Missing ' will produce an error
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

Type Error

A **TypeError** is thrown if you use a value that is outside the range of expected types:

Example

```
var num = 1;
try {
    num.toUpperCase(); // You cannot convert a number to upper case
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

URI (Uniform Resource Identifier) Error

A **URIError** is thrown if you use illegal characters in a URI function:

Example

```
try {
    decodeURI("%%%"); // You cannot URI decode percent signs
}
catch(err) {
    document.getElementById("demo").innerHTML = err.name;
}
```

Non-Standard Error Object Properties

Mozilla and Microsoft defines some non-standard error object properties:

fileName (Mozilla) lineNumber (Mozilla) columnNumber (Mozilla) stack (Mozilla)

description (Microsoft) number (Microsoft)

Do not use these properties in public web sites. They will not work in all browsers.

JavaScript Scope

Scope determines the accessibility (visibility) of variables.

JavaScript Function Scope

In JavaScript there are two types of scope:

- Local scope
- Global scope

JavaScript has function scope: Each function creates a new scope.

Scope determines the accessibility (visibility) of these variables.

Variables defined inside a function are not accessible (visible) from outside the function.

Local JavaScript Variables

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

Local variables have **Function scope**: They can only be accessed from 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 also use carName  
  
}
```

JavaScript Variables

In JavaScript, objects and functions are also variables.

Scope determines the accessibility of variables, objects, and functions from different parts of the code.

Automatically Global

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

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

Example

```
myFunction();  
  
// code here can use carName  
  
function myFunction() {  
    carName = "Volvo";  
}
```

Strict Mode

All modern browsers support running JavaScript in "Strict Mode".

You will learn more about how to use strict mode in a later chapter of this tutorial.

Global variables are not created automatically in "Strict Mode".

Global Variables in HTML

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

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

Example

```
var carName = "Volvo";  
  
// code here can use window.carName
```

Warning

Do NOT create global variables unless you intend to.

Your global variables (or functions) can overwrite window variables (or functions). Any function, including the window object, can overwrite your global variables and functions.

The Lifetime of JavaScript Variables

The lifetime of a JavaScript variable starts when it is declared.

Local variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser window (or tab), but remain available to new pages loaded into the same window.

Function Arguments

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

JavaScript Hoisting

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

JavaScript Declarations are Hoisted

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

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

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

Example 1

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

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

var x; // Declare x
```

Example 2

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

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

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

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

The let and const Keywords

Variables and constants declared with **let** or **const** are not hoisted!

Read more about **let** and **const** in [JS Let / Const](#).

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

JavaScript in strict mode does not allow variables to be used if they are not declared. Study "**use 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 was new in ECMAScript version 5.

It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

With strict mode, you can not, for example, use undeclared variables.

All modern browsers support "use strict" except Internet Explorer 9 and lower:

Directive					
"use strict"	13.0	10.0	4.0	6.0	12.1

The numbers in the table specify the first browser version that fully supports the directive.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string, so IE 9 will not throw an error even if it does not understand it.

Declaring Strict Mode

Strict mode is declared by adding `"use strict";` to the beginning of a script or a function.

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

Example

```
"use strict";
x = 3.14;           // This will cause an error because x is not declared
```

Example

```
"use strict";
myFunction();

function myFunction() {
    y = 3.14; // This will also cause an error because y is not declared
}
```

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

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

function myFunction() {
    "use strict";
    y = 3.14; // This will cause an error
}
```

The "use strict"; Syntax

The syntax, for declaring strict mode, was designed to be compatible with older versions of JavaScript.

Compiling a numeric literal (`4 + 5;`) or a string literal (`"John Doe";`) in a JavaScript program has no side effects. It simply compiles to a non existing variable and dies.

So `"use strict";` only matters to new compilers that "understand" the meaning of it.

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Not Allowed in Strict Mode

Using a variable, without declaring it, is not allowed:

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

Objects are variables too.

Using an object, without declaring it, is not allowed:

```
"use strict";
x = {p1:10, p2:20}; // This will cause an error
```

Deleting a variable (or object) is not allowed.

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

Deleting a function is not allowed.

```
"use strict";
function x(p1, p2) {};
delete x; // This will cause an error
```

Duplicating a parameter name is not allowed:

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

Octal numeric literals are not allowed:

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

Octal escape characters are not allowed:

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

Writing to a read-only property is not allowed:

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

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

Writing to a get-only property is not allowed:

```
"use strict";
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() is 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 Proof!

Keywords reserved for future JavaScript versions can NOT be used as variable names in strict mode.

These are:

- implements
- interface
- let
- package
- private
- protected
- public
- static
- yield

```
"use strict";
var public = 1500;      // This will cause an error
```

Watch Out!

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

The JavaScript **this** Keyword

Example

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

What is "**this**"?

In a function definition, **this** refers to the "**owner**" of the function.

In the example above, **this** refers to the **person** object.

The **person** object "owns" the **fullName** method.

Default Binding

When used alone, **this** refers to the **Global object**.

In a browser the Global object is [**object Window**]:

Example

```
var x = this;
```

When used in a function, **this** refers to the **Global object**.

Example

```
function myFunction() {  
    return this;  
}
```

In strict mode, **this** will be **undefined**, because strict mode does not allow default binding:

Example

```
"use strict";
function myFunction() {
    return this;
}
```

Object Method Binding

In these examples, **this** is the **person** object (The person object is the "owner" of the function):

Example

```
var person = {
    firstName : "John",
    lastName  : "Doe",
    id        : 5566,
    myFunction : function() {
        return this;
    }
};
```

Example

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

In other words: **this.firstName** means the **firstName** property of **this** (person) object.

Explicit Function Binding

The **call()** and **apply()** methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.

You can read more about `call()` and `apply()` later in this tutorial.

In this example, when calling `person1.fullName` with `person2` as argument, **this** will refer to `person2`, even if it is a method of `person1`:

Example

```
var person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
var person2 = {
  firstName: "John",
  lastName: "Doe",
}
person1.fullName.call(person2); // Will return "John Doe"
```

JavaScript Let

ECMAScript 2015

ES2015 introduced two important new JavaScript keywords: **let** and **const**.

These two keywords provide **Block Scope** variables (and constants) in JavaScript.

Before ES2015, JavaScript had only two types of scope: **Global Scope** and **Function Scope**.

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

Example

```
var carName = "Volvo";  
  
// code here can use carName  
  
function myFunction() {  
    // code here can also use carName  
}
```

Global variables can be accessed from anywhere in a JavaScript program.

Function Scope

Variables declared **Locally** (inside a function) have **Function Scope**.

Example

```
// code here can NOT use carName  
  
function myFunction() {  
    var carName = "Volvo";  
    // code here CAN use carName  
}  
  
// code here can NOT use carName
```

Local variables can only be accessed from inside the function where they are declared.

JavaScript Block Scope

Variables declared with the **var keyword** can not have **Block Scope**.

Variables declared inside a block `{}` can be accessed from outside the block.

Example

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

Before ES2015 JavaScript did not have **Block Scope**.

Variables declared with the **let keyword** can have Block Scope.

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

Example

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

Redeclaring Variables

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

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

Example

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

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

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

Example

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

Browser Support

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

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

				
Chrome 49	IE / Edge 12	Firefox 44	Safari 11	Opera 36
Mar, 2016	Jul, 2015	Jan, 2015	Sep, 2017	Mar, 2016

Loop Scope

Using **var** in a loop:

Example

```
var i = 5;  
for (var i = 0; i < 10; i++) {  
    // some statements  
}  
// Here i is 10
```

Using **let** in a loop:

Example

```
let i = 5;
for (let i = 0; i < 10; i++) {
    // some statements
}
// Here i is 5
```

In the first example, using **var**, the variable declared in the loop redeclares the variable outside the loop.

In the second example, using **let**, the variable declared in the loop does not redeclare the variable outside the loop.

When **let** is used to declare the **i** variable in a loop, the **i** variable will only be visible within the loop.

Function Scope

Variables declared with **var** and **let** are quite similar when declared inside a function.

They will both have **Function Scope**:

```
function myFunction() {
    var carName = "Volvo";    // Function Scope
}

function myFunction() {
    let carName = "Volvo";    // Function Scope
}
```

Global Scope

Variables declared with **var** and **let** are quite similar when declared outside a block.

They will both have **Global Scope**:

```
var x = 2;           // Global scope
```

```
let x = 2;           // Global scope
```

Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the **var** keyword belong to the window object:

Example

```
var carName = "Volvo";
// code here can use window.carName
```

Global variables defined with the **let** keyword do not belong to the window object:

Example

```
let carName = "Volvo";
// code here can not use window.carName
```

Redeclaring

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

Example

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

Redeclaring a **var** variable with **let**, in the same scope, or in the same block, is not allowed:

Example

```
var x = 2;          // Allowed  
let x = 3;         // Not allowed  
  
{  
    var x = 4;     // Allowed  
    let x = 5     // Not allowed  
}
```

Redeclaring a **let** variable with **let**, in the same scope, or in the same block, is not allowed:

Example

```
let x = 2;          // Allowed  
let x = 3;         // Not allowed  
  
{  
    let x = 4;     // Allowed  
    let x = 5     // Not allowed  
}
```

Redeclaring a **let** variable with **var**, in the same scope, or in the same block, is not allowed:

Example

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

{
  let x = 4;      // Allowed
  var x = 5;      // Not allowed
}
```

Redeclaring a variable with **let**, in another scope, or in another block, is allowed:

Example

```
let x = 2;          // Allowed

{
  let x = 3;      // Allowed
}

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

Hoisting

Variables defined with **var** are hoisted to the top. ([Js Hoisting](#))

Example

```
// you CAN use carName here
var carName;
```

Variables defined with **let** are not hoisted to the top.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Example

```
// you can NOT use carName here  
let carName;
```

JavaScript Const

ECMAScript 2015

ES2015 introduced two important new JavaScript keywords: **let** and **const**.

Variables defined with **const** behave like **let** variables, except they cannot be reassigned:

Example

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

Block Scope

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

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

Example

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

You can learn more about [Block Scope](#) in the previous chapter: [JavaScript Let](#).

Assigned when Declared

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

Incorrect

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

Correct

```
const PI = 3.14159265359;
```

Not Real Constants

The keyword const is a little misleading.

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

Because of this, we cannot change constant primitive values, but we can change the properties of constant objects.

Primitive Values

If we assign a primitive value to a constant, we cannot change the primitive value:

Example

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

Constant Objects can Change

You can change the properties of a constant object:

Example

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

But you can NOT reassign a constant object:

Example

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

Constant Arrays can Change

You can change the elements of a constant array:

Example

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

But you can NOT reassign a constant array:

Example

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

Browser Support

The **const** keyword is not supported in Internet Explorer 10 or earlier.

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

				
Chrome 49	IE / Edge 11	Firefox 36	Safari 10	Opera 36
Mar, 2016	Oct, 2013	Feb, 2015	Sep, 2016	Mar, 2016

Redeclaring

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

Example

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

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

Example

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

Redeclaring or reassigning an existing **const** variable, in the same scope, or in the same block, is not allowed:

Example

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

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

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

Example

```
const x = 2;          // Allowed

{
  const x = 3;      // Allowed
}

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

Hoisting

Variables defined with **var** are hoisted to the top. ([Js Hoisting](#))

Example

```
carName = "Volvo";      // You CAN use carName here  
var carName;
```

Variables defined with **const** are not hoisted to the top.

Example

```
carName = "Volvo";      // You can NOT use carName here  
const carName = "Volvo";
```

JavaScript Debugging

“

Errors can (will) happen, every time you write some new computer code.

Code Debugging

Programming code might contain syntax errors, or logical errors.

Many of these errors are difficult to diagnose.

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

Searching for (and fixing) errors in programming code is called code debugging.

JavaScript Debuggers

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

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

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

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

The console.log() Method

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

Example

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

<h1>My First Web Page</h1>

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

</body>
</html>
```

Tip: Read more about the `console.log()` method in our [JavaScript Console Reference](#).

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>
- Follow the instructions how to:add a Developer Console button to your toolbar.

Safari Firebug

- Open the browser.
- Go to the webpage:<http://safari-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 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;
tax = 0.20;

fullPrice = price + (price * tax);
```

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. Different editors interpret tabs differently.

Statement Rules

General rules for simple statements:

- Always end a 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 a 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, using spaces only between properties, 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 variables written in **UPPERCASE** (We don't, but it's quite common)
- Constants (like PI) written in **UPPERCASE**

Should you use **hyp-hens**, **camelCase**, or **under_scores** in variable names?

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

Hyphens in HTML and CSS:

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

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

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

Underscores:

Many programmers prefer to use underscores (date_of_birth), especially in SQL databases.

Underscores are often used in PHP documentation.

PascalCase:

PascalCase is often preferred by C programmers.

camelCase:

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

Do not start names with a \$ sign. It will put you in conflict with many JavaScript library names.

Loading JavaScript in HTML

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

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

Accessing HTML Elements

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

These two JavaScript statements will produce different results:

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

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

If possible, use the same naming convention (as JavaScript) in HTML.

[Visit the HTML Style Guide.](#)

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 can break your web site.

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

Performance

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

Indentation and extra spaces are not significant in small scripts.

For code in development, readability should be preferred. Larger production scripts should be 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 a good coding practice to put all declarations at the top of each script or function.

This will:

- Give cleaner code
- Provide a single place to look for local variables
- Make it easier to avoid unwanted (implied) global variables
- Reduce the possibility of unwanted re-declarations

```
// Declare at the beginning
var firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price * 100 / discount;
```

This also goes for loop variables:

```
// Declare at the beginning
var i;

// Use later
for (i = 0; i < 5; i++) {
```

By default, JavaScript moves all declarations to the top ([JavaScript Hoisting](#)).

Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

- Give cleaner code
- Provide a single place to initialize variables
- Avoid undefined values

```
// Declare and initiate at the beginning
var firstName = "",
    lastName = "",
    price = 0,
    discount = 0,
    fullPrice = 0,
    myArray = [],
    myObject = {};
```

Initializing variables provides an idea of the intended use (and intended data type).

Never Declare Number, String, or Boolean Objects

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

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Example

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

Or even worse:

Example

```
var x = new String("John");
var y = new String("John");
(x == y) // is false because you cannot compare objects.
```

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;  
    }  
}
```

ECMAScript 2015 allows default parameters in the function call:

```
function (a=1, b=1) { // funtion code }
```

Read more about function parameters and arguments at [Function Parameters](#)

End Your Switches with Defaults

Always end your switch statements with a default. Even if you think there is no need for it.

Example

```
switch (new Date().getDay()) {  
    case 0:  
        day = "Sunday";  
        break;  
    case 1:  
        day = "Monday";  
        break;  
    case 2:  
        day = "Tuesday";  
        break;  
    case 3:  
        day = "Wednesday";  
        break;  
    case 4:  
        day = "Thursday";  
        break;  
    case 5:  
        day = "Friday";  
        break;  
    case 6:  
        day = "Saturday";  
        break;  
    default:  
        day = "Unknown";  
}
```

Avoid 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 out some common JavaScript mistakes.

Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This **if** statement returns **false** (as expected) because x is not equal to 10:

```
var x = 0;
if (x == 10)
```

This **if** statement returns **true** (maybe not as expected), because 10 is true:

```
var x = 0;
if (x = 10)
```

This **if** statement returns **false** (maybe not as expected), because 0 is false:

```
var x = 0;
if (x = 0)
```

An assignment always returns the value of the assignment.

Expecting Loose Comparison

In regular comparison, data type does not matter. This if statement returns true:

```
var x = 10;
var y = "10";
if (x == y)
```

In strict comparison, data type does matter. This if statement returns false:

```
var x = 10;
var y = "10";
if (x === y)
```

It is a common mistake to forget that switch statements use strict comparison:
This case switch will display an alert:

```
var x = 10;
switch(x) {
  case 10: alert("Hello");
}
```

This case switch will not display an alert:

```
var x = 10;
switch(x) {
  case "10": alert("Hello");
}
```

Confusing Addition & Concatenation

Addition is about adding **numbers**.

Concatenation is about adding **strings**.

In JavaScript both operations use the same + operator.

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

```
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
```

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

Example

```
var z = (x * 10 + y * 10) / 10;           // z will be 0.3
```

Breaking a JavaScript String

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

Example 1

```
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 Definitions with a Comma

Trailing commas in object and array definition are legal in ECMAScript 5.

Object Example:

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

Array Example:

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

WARNING !!

Internet Explorer 8 will crash.

JSON does not allow trailing commas.

JSON:

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

JSON:

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

Undefined is Not Null

JavaScript objects, variables, properties, and methods can be **undefined**.

In addition, empty JavaScript objects can have the value **null**.

This can make it a little bit difficult to test if an object is empty.

You can test if an object exists by testing if the type is **undefined**:

Example:

```
if (typeof myObj === "undefined")
```

But you cannot test if an object is **null**, because this will throw an error if the object is undefined:

Incorrect:

```
if (myObj === null)
```

To solve this problem, you must test if an object is not **null**, and not **undefined**.
But this can still throw an error:

Incorrect:

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

Because of this, you must test for not **undefined** before you can test for not **null**:

Correct:

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

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.
This code will display the value of i (10), even OUTSIDE the for loop block:

Example

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

JavaScript Performance

How to speed up your JavaScript code.

Reduce Activity in Loops

Loops are often used in programming.

Each statement in a loop, including the for statement, is executed for each iteration of the loop.

Statements or assignments that can be placed outside the loop will make the loop run faster.

Bad:

```
var i;  
for (i = 0; i < arr.length; i++) {
```

Better Code:

```
var i;  
var l = arr.length;  
for (i = 0; i < l; i++) {
```

The bad code accesses the length property of an array each time the loop is iterated.

The better code accesses the length property outside the loop and makes the loop run faster.

Reduce DOM Access

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

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

Example

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

Reduce DOM Size

Keep the number of elements in the HTML DOM small.

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

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

Avoid Unnecessary Variables

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

Often you can replace code like this:

```
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 after the page has finished parsing, but it only works for external scripts.

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

Example

```
<script>
window.onload = function() {
    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

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

You can read more about the different JavaScript versions in the chapter [JS Versions](#).

Removed Reserved Words

abstract	boolean	byte	char
double	final	float	goto
int	long	native	short
synchronized	throws	transient	volatile

Do not use these words as variables. ECMAScript 5/6 does not have full support in all browsers.

JavaScript Objects, Properties, and Methods

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

Java Reserved Words

getClass	java	JavaArray	javaClass
JavaObject	JavaPackage		

Other Reserved Words

JavaScript can be used as the programming language in many applications.

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

HTML Event Handlers

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

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

JavaScript Versions

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

ECMAScript is the official name of the language.

From 2015 ECMAScript is named by year (ECMAScript 2015).

ECMAScript Editions

Ver	Official Name	Description
1	ECMAScript 1 (1997)	First Edition.
2	ECMAScript 2 (1998)	Editorial changes only.
3	ECMAScript 3 (1999)	Added Regular Expressions. Added try/catch.
4	ECMAScript 4	Never released.
5	ECMAScript 5 (2009) Read More: JS ES5	Added "strict mode". Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods.
5.1	ECMAScript 5.1 (2011)	Editorial changes.
6	ECMAScript 2015 Read More: JS ES6	Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex().
7	ECMAScript 2016	Added exponential operator (**). Added Array.prototype.includes.
8	ECMAScript 2017	Added string padding. Added new Object properties. Added Async functions. Added Shared Memory.
9	ECMAScript 2018	Added rest / spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp.

ECMAScript is often abbreviated to ES.

Browser Support

ECMAScript 3 is fully supported in all browsers.

ECMAScript 5 is fully supported in all **modern** browsers.

Browser Support for ES5 (2009)

Browser	Version	From Date
Chrome	23	Sep 2012
Firefox	21	Apr 2013
IE	9*	Mar 2011
IE / Edge	10	Sep 2012
Safari	6	Jul 2012
Opera	15	Jul 2013

* Internet Explorer 9 does not support ECMAScript 5 "use strict".

Browser Support for ES6 (ECMAScript 2015)

Browser	Version	Date
Chrome	68	Apr 2017
Firefox	54	Jun 2017
Edge	14	Aug 2016
Safari	10	Sep 2016
Opera	55	Aug 2017

Internet Explorer does not support ECMAScript 2015.

Browser Support for ES7 (ECMAScript 2016)

Browser	Version	Date
Chrome	68	May 2018
Opera	47	Jul 2018

JavaScript / ECMAScript

JavaScript was developed for Netscape. Netscape 2 was the first browser to run JavaScript. After Netscape the Mozilla foundation continued to develop JavaScript for the Firefox browser. The latest JavaScript version was 1.8.5. (Identical to ECMAScript 5).

ECMAScript was developed by ECMA International after the organization adopted JavaScript. The first edition of ECMAScript was released in 1997.

This list compares the version numbers of the different products:

Year	JavaScript	ECMA	Browser
1996	1.0		Netscape 2
1997		ECMAScript 1	IE 4
1998	1.3		Netscape 4
1999		ECMAScript 2	IE 5
2000		ECMAScript 3	IE 5.5
2000	1.5		Netscape 6
2000	1.5		Firefox 1
2011		ECMAScript 5	IE 9 (Except "use strict")
2011	1.8.5		Firefox 4 (Except leading zeroes in parseInt)
2012			IE 10
2012			Chrome 23
2012			Safari 6
2013			Firefox 21
2013			Opera 15
2015		ECMAScript 2015	Partially Supported in all Browser

IE 4 was the first browser to support ECMAScript 1 (1997).

IE 5 was the first browser to support ECMAScript 2 (1999).

IE 5.5 was the first browser to support ECMAScript 3 (2000).

IE 9* was the first browser to support ECMAScript 5 (2011).

Internet Explorer 9 does not support ECMAScript 5 "use strict".

Chrome 23, IE 10, and Safari 6 were the first browsers to **fully** support ECMAScript 5:

				
Chrome 23	IE10 / Edge	Firefox 21	Safari 6	Opera 15
Sep 2012	Sep 2012	Apr 2013	Jul 2012	Jul 2013

ECMAScript 5 - JavaScript 5

What is ECMAScript 5?

ECMAScript 5 is also known as ES5 and ECMAScript 2009

This chapter introduces some of the most important features of ES5.

ECMAScript 5 Features

These were the new features released in 2009:

- The "use strict" Directive
- String.trim()
- Array.isArray()
- Array.forEach()
- Array.map()
- Array.filter()
- Array.reduce()
- Array.reduceRight()
- Array.every()
- Array.some()
- Array.indexOf()
- Array.lastIndexOf()
- JSON.parse()
- JSON.stringify()
- Date.now()
- Property Getters and Setters
- New Object Property Methods

ECMAScript 5 Syntactical Changes

- Property access [] on strings
- Trailing commas in array and object literals
- Multiline string literals
- Reserved words as property names

The "use strict" Directive

"use strict" defines that the JavaScript code should be executed in "strict mode".

With strict mode you can, for example, not use undeclared variables.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string expression. Old browsers will not throw an error if they don't understand it.

Read more in [JS Strict Mode](#).

String.trim()

String.trim() removes whitespace from both sides of a string.

Example

```
var str = "      Hello World!      ";
alert(str.trim());
```

Read more in [JS String Methods](#).

Array.isArray()

Checks whether an object is an array.

Example

```
function myFunction() {
  var fruits = ["Banana", "Orange", "Apple", "Mango"];
  var x = document.getElementById("demo");
  x.innerHTML = Array.isArray(fruits);
}
```

Read more in [JS Arrays](#).

Array.forEach()

The forEach() method calls a function once for each array element.

Example

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction);

function myFunction(value) {
    txt = txt + value + "<br>";
}
```

Learn more in [JS Array Iteration Methods](#).

Array.map()

This example multiplies each array value by 2:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction);

function myFunction(value) {
    return value * 2;
}
```

Learn more in [JS Array Iteration Methods](#).

Array.filter()

This example creates a new array from elements with a value larger than 18:

Example

```
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction);

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

Learn more in [JS Array Iteration Methods](#).

Array.reduce()

This example finds the sum of all numbers in an array:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduce(myFunction);

function myFunction(total, value) {
    return total + value;
}
```

Learn more in [JS Array Iteration Methods](#).

Array.reduceRight()

This example also finds the sum of all numbers in an array:

Example

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduceRight(myFunction);

function myFunction(total, value) {
    return total + value;
}
```

Learn more in [JS Array Iteration Methods](#).

Array.every()

This example checks if all values are over 18:

Example

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

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

Learn more in [JS Array Iteration Methods](#).

Array.some()

This example checks if some values are over 18:

Example

```
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.some(myFunction);

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

Learn more in [JS Array Iteration Methods](#).

Array.indexOf()

Search an array for an element value and returns its position.

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var a = fruits.indexOf("Apple");
```

Learn more in [JS Array Iteration Methods](#).

Array.lastIndexOf()

Array.lastIndexOf() is the same as Array.indexOf(), but searches from the end of the array.

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var a = fruits.lastIndexOf("Apple");
```

Learn more in [JS Array Iteration Methods](#).

JSON.parse()

A common use of JSON is to receive data from a web server.

Imagine you received this text string from a web server:

```
'{"name":"John", "age":30, "city":"New York"}'
```

The JavaScript function `JSON.parse()` is used to convert the text into a JavaScript object:

```
var obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}');
```

Read more in our [JSON Tutorial](#).

JSON.stringify()

A common use of JSON is to send data to a web server.

When sending data to a web server, the data has to be a string.

Imagine we have this object in JavaScript:

```
var obj = {"name":"John", "age":30, "city":"New York"};
```

Use the JavaScript function `JSON.stringify()` to convert it into a string.

```
var myJSON = JSON.stringify(obj);
```

The result will be a string following the JSON notation.

`myJSON` is now a string, and ready to be sent to a server:

Example

```
var obj = {"name":"John", "age":30, "city":"New York"};
var myJSON = JSON.stringify(obj);
document.getElementById("demo").innerHTML = myJSON;
```

Read more in our [JSON Tutorial](#).

Date.now()

`Date.now()` returns the number of milliseconds since zero date (January 1. 1970 00:00:00 UTC).

Example

```
var timInMSs = Date.now();
```

`Date.now()` returns the same as `getTime()` performed on a `Date` object.

Learn more in [JS Dates](#).

Property Getters and Setters

ES5 lets you define object methods with a syntax that looks like getting or setting a property. This example creates a getter for a property called `fullName`:

Example

```
// Create an object:  
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    get fullName() {  
        return this.firstName + " " + this.lastName;  
    }  
};  
// Display data from the object using a getter:  
document.getElementById("demo").innerHTML = person.fullName;
```

This example creates a setter and a getter for the language property:

Example

```
var person = {
    firstName: "John",
    lastName : "Doe",
    language : "NO",
    get lang() {
        return this.language;
    },
    set lang(value) {
        this.language = value;
    }
};
// Set an object property using a setter:
person.lang = "en";
// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.lang;
```

This example uses a setter to secure upper case updates of language:

Example

```
var person = {
    firstName: "John",
    lastName : "Doe",
    language : "NO",
    set lang(value) {
        this.language = value.toUpperCase();
    }
};
// Set an object property using a setter:
person.lang = "en";
// Display data from the object:
document.getElementById("demo").innerHTML = person.language;
```

Learn more about Gettes and Setters in [JS Object Accessors](#)

New Object Property Methods

Object.defineProperty() is a new Object method in ES5.

It lets you define an object property and/or change a property's value and/or metadata.

Example

```
// Create an Object:  
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "NO",  
};  
// Change a Property:  
Object.defineProperty(person, "language", {  
    value: "EN",  
    writable : true,  
    enumerable : true,  
    configurable : true  
});  
// Enumerate Properties  
var txt = "";  
for (var x in person) {  
    txt += person[x] + "<br>";  
}  
document.getElementById("demo").innerHTML = txt;
```

Next example is the same code, except it hides the language property from enumeration:

Example

```
// Create an Object:  
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "NO",  
};  
  
// Change a Property:  
Object.defineProperty(person, "language", {  
    value: "EN",  
    writable : true,  
    enumerable : false,  
    configurable : true  
});  
  
// Enumerate Properties  
var txt = "";  
for (var x in person) {  
    txt += person[x] + "<br>";  
}  
document.getElementById("demo").innerHTML = txt;
```

This example creates a setter and a getter to secure upper case updates of language:

Example

```
/// Create an Object:  
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "NO"  
};  
// Change a Property:  
Object.defineProperty(person, "language", {  
get : function() { return language },  
set : function(value) { language = value.toUpperCase()}  
});  
// Change Language  
person.language = "en";  
// Display Language  
document.getElementById("demo").innerHTML = person.language;
```

ECMAScript 5 added a lot of new Object Methods to JavaScript:

ES5 New Object Methods

```
// Adding or changing an object property
Object.defineProperty(object, property, descriptor)

// Adding or changing many object properties
Object.defineProperties(object, descriptors)

// Accessing Properties
Object.getOwnPropertyDescriptor(object, property)

// Returns all properties as an array
Object.getOwnPropertyNames(object)

// Returns enumerable properties as an array
Object.keys(object)

// Accessing the prototype
Object.getPrototypeOf(object)

// Prevents adding properties to an object
Object.preventExtensions(object)
// Returns true if properties can be added to an object
Object.isExtensible(object)

// Prevents changes of object properties (not values)
Object.seal(object)
// Returns true if object is sealed
Object.isSealed(object)

// Prevents any changes to an object
Object.freeze(object)
// Returns true if object is frozen
Object.isFrozen(object)
```

Learn more in [Object ECMAScript5](#).

Property Access on Strings

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

Example

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

ECMAScript 5 allows property access on strings:

Example

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

Property access on string might be a little unpredictable.

Read more in [JS String Methods](#).

Trailing Commas

ECMAScript 5 allows trailing commas in object and array definitions:

Object Example

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

Array Example

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

WARNING !!!

Internet Explorer 8 will crash.

JSON does not allow trailing commas.

JSON Objects:

Allowed:

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

```
JSON.parse(person)
```

Not allowed:

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

```
JSON.parse(person)
```

JSON Arrays:

Allowed:

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

Not allowed:

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

Strings Over Multiple Lines

Example

```
"Hello \
Dolly!";
```

The \ method might not have universal support. Older browsers might treat the spaces around the backslash differently. Some older browsers do not allow spaces behind the \ character.

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

Example

```
"Hello " +
"Dolly!";
```

Reserved Words as Property Names

ECMAScript 5 allows reserved words as property names:

Object Example

```
var obj = {name: "John", new: "yes"}
```

Browser Support

Chrome 23, IE 10, and Safari 6 were the first browsers to fully support ECMAScript 5:

				
Chrome 23	IE10 / Edge	Firefox 21	Safari 6	Opera 15
Sep 2012	Sep 2012	Apr 2013	Jul 2012	Jul 2013

ECMAScript 6 - ECMAScript 2015

What is ECMAScript 6?

ECMAScript 6 is also known as ES6 and ECMAScript 2015

Some people like to call it JavaScript 6.

This chapter will introduce some of the new features in ES6.

- JavaScript let
- JavaScript const
- JavaScript default parameter values
- Array.find()
- Array.findIndex()

JavaScript let

The **let** statement allows you to declare a variable with block scope.

Example

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

JavaScript const

The **const** statement allows you to declare a constant (a JavaScript variable with a constant value).

Constants are similar to let variables, except that the value cannot be changed.

Example

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

Read more about **let** and **const** in [JS Let / Const](#).

Default Parameter Values

ES6 allows function parameters to have default values.

Example

```
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

Array.find()

The `find()` method returns the value of the first array element that passes a test function. This example finds (returns the value of) the first element that is larger than 18:

Example

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

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

Array.findIndex()

The `findIndex()` method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

Example

```
var numbers = [4, 9, 16, 25, 29];
var first = numbers.findIndex(myFunction);

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

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

New Number Properties

ES6 added the following properties to the Number object:

- EPSILON
- MIN_SAFE_INTEGER
- MAX_SAFE_INTEGER

Example

```
var x = Number.EPSILON;
```

Example

```
var x = Number.MIN_SAFE_INTEGER;
```

Example

```
var x = Number.MAX_SAFE_INTEGER;
```

New Number Methods

ES6 added 2 new methods to the Number object:

- Number.isInteger()
- Number.isSafeInteger()

The Number.isInteger() Method

The **Number.isInteger()** method returns true if the argument is an integer.

Example

```
Number.isInteger(10);          // returns true
Number.isInteger(10.5);        // returns false
```

The Number.isSafeInteger() Method

A safe integer is an integer that can be exactly represented as a double precision number.

The **Number.isSafeInteger()** method returns true if the argument is a safe integer.

Example

```
Number.isSafeInteger(10);      // returns true  
Number.isSafeInteger(12345678901234567890); // returns false
```

Safe integers are all integers from $-(2^{53} - 1)$ to $+(2^{53} - 1)$. This is safe:

9007199254740991. This is not safe: 9007199254740992.

New Global Methods

ES6 also added 2 new global number methods:

- `isFinite()`
- `isNaN()`

The `isFinite()` Method

The global **isFinite()** method returns false if the argument is Infinity or NaN.

Otherwise it returns true:

Example

```
isFinite(10/0);      // returns false  
isFinite(10/1);      // returns true
```

The `isNaN()` Method

The global **isNaN()** method returns true if the argument is NaN. Otherwise it returns false:

Example

```
isNaN("Hello");      // returns true
```

Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function **keyword**, the **return** keyword, and the **curly brackets**.

Example

```
// ES5
var x = function(x, y) {
    return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Arrow functions do not have their own **this**. They are not well suited for defining object methods.

Arrow functions must be defined before they are used. Using **const** is safer than using **var**, because a function expression is a constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement.

It might be a good habit to keep them:

Example

```
const x = (x, y) => { return x * y };
```

JavaScript JSON

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

What is JSON?

- JSON stands for **JavaScripT O**bject **N**otation
- JSON is a lightweight data interchange format
- JSON is language independent *
- JSON is "self-describing" and easy to understand

* The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

JSON Example

This JSON syntax defines an employees object: an array of 3 employee records (objects):

JSON Example

```
{  
  "employees": [  
    {"firstName": "John", "lastName": "Doe"},  
    {"firstName": "Anna", "lastName": "Smith"},  
    {"firstName": "Peter", "lastName": "Jones"}  
  ]  
}
```

The JSON Format Evaluates to JavaScript Objects

The JSON format is syntactically identical to the code for creating JavaScript objects. Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName": "John"
```

JSON names require double quotes. JavaScript names do not.

JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName": "John", "lastName": "Doe"}
```

JSON Arrays

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

```
"employees": [ {"firstName": "John", "lastName": "Doe"},  
 {"firstName": "Anna", "lastName": "Smith"}, {"firstName": "Peter",  
 "lastName": "Jones"} ]
```

In the example above, the object "employees" is an array. It contains three objects.

Each object is a record of a person (with a first name and a last name).

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.

For simplicity, this can be demonstrated using a string as input.

First, create a JavaScript string containing JSON syntax:

```
var text = '{ "employees" : [ ' +  
 ' { "firstName": "John" , "lastName": "Doe" } , ' +  
 ' { "firstName": "Anna" , "lastName": "Smith" } , ' +  
 ' { "firstName": "Peter" , "lastName": "Jones" } ] }';
```

Then, use the JavaScript built-in function `JSON.parse()` to convert the string into a JavaScript object:

```
var obj = JSON.parse(text);
```

Finally, use the new JavaScript object in your page:

Example

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

<script>
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>
```

You can read more about JSON in our [JSON tutorial](#).

JavaScript Forms

JavaScript Form Validation

HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

JavaScript Example

```
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == "") {
        alert("Name must be filled out");
        return false;
    }
}
```

The function can be called when the form is submitted:

HTML Form Example

```
<form name="myForm" action="/action_page.php" onsubmit="return
validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

JavaScript Can Validate Numeric Input

JavaScript is often used to validate numeric input:

Please input a number between 1 and 10

Submit

Automatic HTML Form Validation

HTML form validation can be performed automatically by the browser:

If a form field (fname) is empty, the **required** attribute prevents this form from being submitted:

HTML Form Example

```
<form action="/action_page.php" method="post">
  <input type="text" name="fname" required>
  <input type="submit" value="Submit">
</form>
```

Automatic HTML form validation does not work in Internet Explorer 9 or earlier.

Data Validation

Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

- has the user filled in all required fields?
- has the user entered a valid date?
- has the user entered text in a numeric field?

Most often, the purpose of data validation is to ensure correct user input.

Validation can be defined by many different methods, and deployed in many different ways.

Server side validation is performed by a web server, after input has been sent to the server.

Client side validation is performed by a web browser, before input is sent to a web server.

HTML Constraint Validation

HTML5 introduced a new HTML validation concept called **constraint validation**.

HTML constraint validation is based on:

- Constraint validation **HTML Input Attributes**
- Constraint validation **CSS Pseudo Selectors**
- Constraint validation **DOM Properties and Methods**

Constraint Validation HTML Input Attributes

Attribute	Description
disabled	Specifies that the input element should be disabled
max	Specifies the maximum value of an input element
min	Specifies the minimum value of an input element
pattern	Specifies the value pattern of an input element
required	Specifies that the input field requires an element
type	Specifies the type of an input element

For a full list, go to [HTML Input Attributes](#).

Constraint Validation CSS Pseudo Selectors

Selector	Description
:disabled	Selects input elements with the "disabled" attribute specified
:invalid	Selects input elements with invalid values
:optional	Selects input elements with no "required" attribute specified
:required	Selects input elements with the "required" attribute specified
:valid	Selects input elements with valid values

For a full list, go to [CSS Pseudo Classes](#).

JavaScript Validation API

Constraint Validation DOM Methods

Property	Description
checkValidity()	Returns true if an input element contains valid data.
setCustomValidity()	Sets the validationMessage property of an input element.

If an input field contains invalid data, display a message:

The checkValidity() Method

```
<input id="id1" type="number" min="100" max="300" required>
<button onclick="myFunction()">OK</button>

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

<script>
function myFunction() {
    var inpObj = document.getElementById("id1");
    if (!inpObj.checkValidity()) {
        document.getElementById("demo").innerHTML =
inpObj.validationMessage;
    }
}
</script>
```

Constraint Validation DOM Properties

Property	Description
validity	Contains boolean properties related to the validity of an input element.
validationMessage	Contains the message a browser will display when the validity is false.
willValidate	Indicates if an input element will be validated.

Validity Properties

The **validity property** of an input element contains a number of properties related to the validity of data:

Property	Description
customError	Set to true, if a custom validity message is set.
patternMismatch	Set to true, if an element's value does not match its pattern attribute.
rangeOverflow	Set to true, if an element's value is greater than its max attribute.
rangeUnderflow	Set to true, if an element's value is less than its min attribute.
stepMismatch	Set to true, if an element's value is invalid per its step attribute.
tooLong	Set to true, if an element's value exceeds its maxLength attribute.
typeMismatch	Set to true, if an element's value is invalid per its type attribute.
valueMissing	Set to true, if an element (with a required attribute) has no value.
valid	Set to true, if an element's value is valid.

Examples

If the number in an input field is greater than 100 (the input's max attribute), display a message:

The rangeOverflow Property

```
<input id="id1" type="number" max="100">
<button onclick="myFunction()">OK</button>

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

<script>
function myFunction() {
    var txt = "";
    if (document.getElementById("id1").validity.rangeOverflow) {
        txt = "Value too large";
    }
    document.getElementById("demo").innerHTML = txt;
}
</script>
```

If the number in an input field is less than 100 (the input's min attribute), display a message:

The rangeUnderflow Property

```
<input id="id1" type="number" min="100">
<button onclick="myFunction()">OK</button>

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

<script>
function myFunction() {
    var txt = "";
    if (document.getElementById("id1").validity.rangeUnderflow) {
        txt = "Value too small";
    }
    document.getElementById("demo").innerHTML = txt;
}
</script>
```

JavaScript Objects

In JavaScript, objects are king. If you understand objects, you understand JavaScript.

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the **new** keyword)
- Numbers can be objects (if defined with the **new** keyword)
- Strings can be objects (if defined with the **new** keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

All JavaScript values, except primitives, are objects.

JavaScript Primitives

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

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

JavaScript defines 5 types of primitive data types:

- string
- number
- boolean
- null
- undefined

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

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

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

Objects are Variables

JavaScript variables can contain single values:

Example

```
var person = "John Doe";
```

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

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

Example

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

A JavaScript object is a collection of **named values**

Object Properties

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

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Objects written as name value pairs are similar to:

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

Object Methods

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

Object properties can be both primitive values, other objects, and functions.

An **object method** is an object property containing a **function definition**.

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

JavaScript objects are containers for named values, called properties and methods.

You will learn more about methods in the next chapters.

Creating a JavaScript Object

With JavaScript, you can define and create your own objects.

There are different ways to create new objects:

- Define and create a single object, using an object literal.
- Define and create a single object, with the keyword new.
- Define an object constructor, and then create objects of the constructed type.

In ECMAScript 5, an object can also be created with the function `Object.create()`.

Using an Object Literal

This is the easiest way to create a JavaScript Object.

Using an object literal, you both define and create an object in one statement.

An object literal is a list of name:value pairs (like `age:50`) inside curly braces `{}`.

The following example creates a new JavaScript object with four properties:

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"  
};
```

Using the JavaScript Keyword new

The following example also creates a new JavaScript object with four properties:

Example

```
var person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

The two examples above do exactly the same. There is no need to use `new Object()`. For simplicity, readability and execution speed, use the first one (the object literal method).

JavaScript Objects are Mutable

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

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

```
var x = person; // This will not create a copy of person.
```

The object `x` is **not a copy** of `person`. It **is** `person`. Both `x` and `person` are the same object. Any changes to `x` will also change `person`, because `x` and `person` are the same object.

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}  
  
var x = person;  
x.age = 10;           // This will change both x.age and person.age
```

Note: JavaScript variables are not mutable. Only JavaScript objects.

JavaScript Object Properties

Properties are the most important part of any JavaScript object.

JavaScript Properties

Properties are the values associated with a JavaScript object.

A JavaScript object is a collection of unordered properties.

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

Accessing JavaScript Properties

The syntax for accessing the property of an object is:

```
objectName.property          // person.age
```

or

```
objectName["property"]      // person["age"]
```

or

```
objectName[expression]      // x = "age"; person[x]
```

The expression must evaluate to a property name.

Example 1

```
person.firstname + " is " + person.age + " years old.;"
```

Example 2

```
person["firstname"] + " is " + person["age"] + " years old.;"
```

JavaScript for...in Loop

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

Syntax

```
for (variable in object) {      code to be executed }
```

The block of code inside of the for...in loop will be executed once for each property.
Looping through the properties of an object:

Example

```
var person = {fname:"John", lname:"Doe", age:25};  
  
for (x in person) {  
    txt += person[x];  
}
```

Adding New Properties

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

Assume that the person object already exists - you can then give it new properties:

Example

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

You cannot use reserved words for property (or method) names. JavaScript naming rules apply.

Deleting Properties

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

Example

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

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

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

The delete operator is designed to be used on object properties. It has no effect on variables or functions.

The delete operator should not be used on predefined JavaScript object properties. It can crash your application.

Property Attributes

All properties have a name. In addition they also have a value.

The value is one of the property's attributes.

Other attributes are: enumerable, configurable, and writable.

These attributes define how the property can be accessed (is it readable?, is it writable?)

In JavaScript, all attributes can be read, but only the value attribute can be changed (and only if the property is writable).

(ECMAScript 5 has methods for both getting and setting all property attributes)

Prototype Properties

JavaScript objects inherit the properties of their prototype.

The delete keyword does not delete inherited properties, but if you delete a prototype property, it will affect all objects inherited from the prototype.

JavaScript Object Methods

Example

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

The **this** Keyword

In a function definition, **this** refers to the "owner" of the function.

In the example above, **this** is the **person object** that "owns" the **fullName** function.

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

Read more about the **this** keyword at [JS this Keyword](#).

JavaScript Methods

JavaScript methods are actions that can be performed on objects.

A JavaScript **method** is a property containing a **function definition**.

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

Methods are functions stored as object properties.

Accessing Object Methods

You access an object method with the following syntax:

```
objectName.methodName()
```

You will typically describe `fullName()` as a method of the `person` object, and `fullName` as a property.

The `fullName` property will execute (as a function) when it is invoked with `()`.

This example accesses the `fullName()` **method** of a `person` object:

Example

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

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

Example

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

Using Built-In Methods

This example uses the `toUpperCase()` method of the `String` object, to convert a text to uppercase:

```
var message = "Hello world!"; var x = message.toUpperCase();
```

The value of `x`, after execution of the code above will be:

```
HELLO WORLD!
```

Adding a Method to an Object

Adding a new method to an object is easy:

Example

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

JavaScript Object Accessors

JavaScript Accessors (Getters and Setters)

ECMAScript 5 (2009) introduced Getter and Setters.

Getters and setters allow you to define Object Accessors (Computed Properties).

JavaScript Getter (The get Keyword)

This example uses a **lang** property to **get** the value of the **language** property.

Example

```
// Create an object:  
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "en",  
    get lang() {  
        return this.language;  
    }  
};  
// Display data from the object using a getter:  
document.getElementById("demo").innerHTML = person.lang;
```

JavaScript Setter (The set Keyword)

This example uses a **lang** property to **set** the value of the **language** property.

Example

```
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "",  
    set lang(lang) {  
        this.language = lang;  
    }  
};  
// Set an object property using a setter:  
person.lang = "en";  
// Display data from the object:  
document.getElementById("demo").innerHTML = person.language;
```

JavaScript Function or Getter?

What is the differences between these two examples?

Example 1

```
var person = {
    firstName: "John",
    lastName : "Doe",
    fullName : function() {
        return this.firstName + " " + this.lastName;
    }
};
// Display data from the object using a method:
document.getElementById("demo").innerHTML = person.fullName();
```

Example 2

```
var person = {
    firstName: "John",
    lastName : "Doe",
    get fullName() {
        return this.firstName + " " + this.lastName;
    }
};
// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.fullName;
```

Example 1 access fullName as a function: person.fullName().

Example 2 access fullName as a property: person.fullName.

The second example provides simpler syntax.

Data Quality

JavaScript can secure better data quality when using getters and setters.

Using the **lang** property, in this example, returns the value of the **language** property in upper case:

Example

```
// Create an object:  
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "en",  
    get lang() {  
        return this.language.toUpperCase();  
    }  
};  
// Display data from the object using a getter:  
document.getElementById("demo").innerHTML = person.lang;
```

Using the **lang** property, in this example, stores an upper case value in the **language** property:

Example

```
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    language : "",  
    set lang(lang) {  
        this.language = lang.toUpperCase();  
    }  
};  
// Set an object property using a setter:  
person.lang = "en";  
// Display data from the object:  
document.getElementById("demo").innerHTML = person.language;
```

Why Using Getters and Setters?

- It gives simpler syntax
- It allows equal syntax for properties and methods
- It can secure better data quality
- It is useful for doing things behind-the-scenes

A Counter Example

Example

```
var obj = {
    counter : 0,
    get reset() {
        this.counter = 0;
    },
    get increment() {
        this.counter++;
    },
    get decrement() {
        this.counter--;
    },
    set add(value) {
        this.counter += value;
    },
    set subtract(value) {
        this.counter -= value;
    }
};
// Play with the counter:
obj.reset;
obj.add = 5;
obj.subtract = 1;
obj.increment;
obj.decrement;
```

Object.defineProperty()

The Object.defineProperty() method can also be used to add Getters and Setters:

Example

```
// Define object
var obj = {counter : 0};

// Define setters
Object.defineProperty(obj, "reset", {
    get : function () {this.counter = 0;}
});
Object.defineProperty(obj, "increment", {
    get : function () {this.counter++;}
});
Object.defineProperty(obj, "decrement", {
    get : function () {this.counter--;}
});
Object.defineProperty(obj, "add", {
    set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
    set : function (i) {this.counter += i;}
});

// Play with the counter:
obj.reset;
obj.add = 5;
obj.subtract = 1;
obj.increment;
obj.decrement;
```

Browser Support

Getters and Setters are not supported in Internet Explorer 8 or earlier:

				
Yes	9.0	Yes	Yes	Yes

JavaScript Object Constructors

Example

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

[Try it yourself »](#)

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

Object Types (Blueprints) (Classes)

The examples from the previous chapters are limited. They only create single objects. Sometimes we need a "**blueprint**" for creating many objects of the same "type".

The way to create an "object type", is to use an **object constructor function**.

In the example above, **function Person()** is an object constructor function.

Objects of the same type are created by calling the constructor function with the **new** keyword:

```
var myFather = new Person("John", "Doe", 50, "blue");  
var myMother = new Person("Sally", "Rally", 48, "green");
```

[Try it yourself »](#)

The **this** Keyword

In JavaScript, the thing called **this** is the object that "owns" the code.

The value of **this**, when used in an object, is the object itself.

In a constructor function **this** does not have a value. It is a substitute for the new object. The value of **this** will become the new object when a new object is created.

Note that **this** is not a variable. It is a keyword. You cannot change the value of **this**.

Adding a Property to an Object

Adding a new property to an existing object is easy:

Example

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

The property will be added to myFather. Not to myMother. (Not to any other person objects).

Adding a Method to an Object

Adding a new method to an existing object is easy:

Example

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

The method will be added to myFather. Not to myMother. (Not to any other person objects).

Adding a Property to a Constructor

You cannot add a new property to an object constructor the same way you add a new property to an existing object:

Example

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

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

Example

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

This way object properties can have default values.

Adding a Method to a Constructor

Your constructor function can also define methods:

Example

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

You cannot add a new method to an object constructor the same way you add a new method to an existing object.

Adding methods to an object must be done inside the constructor function:

Example

```
function Person(firstName, lastName, age, eyeColor) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
    this.eyeColor = eyeColor;  
    this.changeName = function (name) {  
        this.lastName = name;  
    };  
}
```

The `changeName()` function assigns the value of `name` to the person's `lastName` property.

Now You Can Try:

```
myMother.changeName("Doe");
```

JavaScript knows which person you are talking about by "substituting" `this` with `myMother`.

Built-in JavaScript Constructors

JavaScript has built-in constructors for native objects:

Example

```
var x1 = new Object();      // A new Object object
var x2 = new String();     // A new String object
var x3 = new Number();     // A new Number object
var x4 = new Boolean();    // A new Boolean object
var x5 = new Array();      // A new Array object
var x6 = new RegExp();     // A new RegExp object
var x7 = new Function();   // A new Function object
var x8 = new Date();       // A new Date object
```

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

Did You Know?

As you can see above, JavaScript has object versions of the primitive data types String, Number, and Boolean. But there is no reason to create complex objects. Primitive values are much faster.

ALSO:

Use object literals `{}` instead of `new Object()`.
Use string literals `""` instead of `new String()`.
Use number literals `12345` instead of `new Number()`.
Use boolean literals `true / false` instead of `new Boolean()`.
Use array literals `[]` instead of `new Array()`.
Use pattern literals `/()/.exec("text")` instead of `new RegExp()`.
Use function expressions `() {}` 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 = /()/.exec("text"); // new regexp object
var x7 = function(){}; // new function object
```

String Objects

Normally, strings are created as primitives: `var firstName = "John"`

But strings can also be created as objects using the `new` keyword: `var firstName = new String("John")`

Learn why strings should not be created as object in the chapter [JS Strings](#).

Number Objects

Normally, numbers are created as primitives: `var x = 123`

But numbers can also be created as objects using the `new` keyword: `var x = new Number(123)`

Learn why numbers should not be created as object in the chapter [JS Numbers](#).

Boolean Objects

Normally, booleans are created as primitives: `var x = false`

But booleans can also be created as objects using the `new` keyword: `var x = new Boolean(false)`

JavaScript Object Prototypes

All JavaScript objects inherit properties and methods from a prototype.

In the previous chapter we learned how to use an **object constructor**:

Example

```
function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
  
var myFather = new Person("John", "Doe", 50, "blue");  
var myMother = new Person("Sally", "Rally", 48, "green");
```

We also learned that you can **not** add a new property to an existing object constructor:

Example

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

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

Example

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

Prototype Inheritance

All JavaScript objects inherit properties and methods from a prototype.

Date objects inherit from Date.prototype. Array objects inherit from Array.prototype. Person objects inherit from Person.prototype.

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

Adding Properties and Methods to Objects

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

Using the **prototype** Property

The JavaScript prototype property allows you to add new properties to object constructors:

Example

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

The JavaScript prototype property also allows you to add new methods to objects constructors:

Example

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

Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.

JavaScript ES5 Object Methods

ECMAScript 5 added a lot of new Object Methods to JavaScript.

ES5 New Object Methods

```
// Adding or changing an object property
Object.defineProperty(object, property, descriptor)

// Adding or changing many object properties
Object.defineProperties(object, descriptors)

// Accessing Properties
Object.getOwnPropertyDescriptor(object, property)

// Returns all properties as an array
Object.getOwnPropertyNames(object)

// Returns enumerable properties as an array
Object.keys(object)

// Accessing the prototype
Object.getPrototypeOf(object)

// Prevents adding properties to an object
Object.preventExtensions(object)
// Returns true if properties can be added to an object
Object.isExtensible(object)

// Prevents changes of object properties (not values)
Object.seal(object)
// Returns true if object is sealed
Object.isSealed(object)

// Prevents any changes to an object
Object.freeze(object)
// Returns true if object is frozen
Object.isFrozen(object)
```

Changing a Property Value

Syntax

```
Object.defineProperty(object, property, {value : value})
```

This example changes a property value:

Example

```
var person = {
    firstName: "John",
    lastName : "Doe",
    language : "EN"
};
// Change a property
Object.defineProperty(person, "language", {value : "NO"});
```

Changing Meta Data

ES5 allows the following property meta data to be changed:

```
writable : true      // Property value can be changed
enumerable : true    // Property can be enumerated
configurable : true  // Property can be reconfigured
```

```
writable : false     // Property value can not be changed
enumerable : false   // Property can not be enumerated
configurable : false // Property can not be reconfigured
```

ES5 allows getters and setters to be changed:

```
// Defining a getter  
get: function() { return language }  
// Defining a setter  
set: function(value) { language = value }
```

This example makes language read-only:

```
Object.defineProperty(person, "language", {writable:false});
```

This example makes language not enumerable:

```
Object.defineProperty(person, "language", {enumerable:false});
```

Listing All Properties

This example list all properties of an object:

Example

```
var person = {  
    firstName: "John",  
    lastName : "Doe"  
    language : "EN"  
};  
Object.defineProperty(person, "language", {enumerable:false});  
Object.getOwnPropertyNames(person); // Returns an array of properties
```

Listing Enumerable Properties

This example lists only the enumerable properties of an object:

Example

```
var person = {
    firstName: "John",
    lastName : "Doe"
    language : "EN"
};
Object.defineProperty(person, "language", {enumerable:false});
Object.keys(person); // Returns an array of enumerable properties
```

Adding a Property

This example adds a new property to an object:

Example

```
// Create an object:
var person = {
    firstName: "John",
    lastName : "Doe",
    language : "EN"
};
// Add a property
Object.defineProperty(person, "year", {value:"2008"});
```

Adding Getters and Setters

The `Object.defineProperty()` method can also be used to add Getters and Setters:

Example

```
//Create an object
var person = {firstName:"John", lastName:"Doe"};

// Define a getter
Object.defineProperty(person, "fullName", {
    get : function () {return this.firstName + " " + this.lastName;}
});
```

A Counter Example

Example

```
// Define object
var obj = {counter:0};

// Define setters
Object.defineProperty(obj, "reset", {
    get : function () {this.counter = 0;}
});
Object.defineProperty(obj, "increment", {
    get : function () {this.counter++;}
});
Object.defineProperty(obj, "decrement", {
    get : function () {this.counter--;}
});
Object.defineProperty(obj, "add", {
    set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
    set : function (i) {this.counter += i;}
});

// Play with the counter:
obj.reset;
obj.add = 5;
obj.subtract = 1;
obj.increment;
obj.decrement;
```

JavaScript Function Definitions

JavaScript functions are **defined** with the **function** keyword.
You can use a function **declaration** or a function **expression**.

Function Declarations

Earlier in this tutorial, you learned that functions are **declared** with the following syntax:

```
function functionName(parameters) { code to be executed }
```

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

Example

```
function myFunction(a, b) {
    return a * b;
}
```

Semicolons are used to separate executable JavaScript statements. Since a function **declaration** is not an executable statement, it is not common to end it with a semicolon.

Function Expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

Example

```
var x = function (a, b) {return a * b};
```

After a function expression has been stored in a variable, the variable can be used as a function:

Example

```
var x = function (a, b) {return a * b};  
var z = x(4, 3);
```

The function above is actually an **anonymous function** (a function without a name). Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

The function above ends with a semicolon because it is a part of an executable statement.

The Function() Constructor

As you have seen in the previous examples, JavaScript functions are defined with the **function** keyword.

Functions can also be defined with a built-in JavaScript function constructor called `Function()`.

Example

```
var myFunction = new Function("a", "b", "return a * b");  
  
var x = myFunction(4, 3);
```

You actually don't have to use the function constructor. The example above is the same as writing:

Example

```
var myFunction = function (a, b) {return a * b};  
  
var x = myFunction(4, 3);
```

Most of the time, you can avoid using the **new** keyword in JavaScript.

Function Hoisting

Earlier in this tutorial, you learned about "hoisting".

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

Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

```
myFunction(5); function myFunction(y) { return y * y; }
```

Functions defined using an expression are not hoisted.

Self-Invoking Functions

Function expressions can be made "self-invoking".

A self-invoking expression is invoked (started) automatically, without being called.

Function expressions will execute automatically if the expression is followed by () .

You cannot self-invoke a function declaration.

You have to add parentheses around the function to indicate that it is a function expression:

Example

```
(function () {
    var x = "Hello!!";           // I will invoke myself
})();
```

The function above is actually an **anonymous self-invoking function** (function without name).

Functions Can Be Used as Values

JavaScript functions can be used as values:

Example

```
function myFunction(a, b) {
    return a * b;
}

var x = myFunction(4, 3);
```

JavaScript functions can be used in expressions:

Example

```
function myFunction(a, b) {  
    return a * b;  
}  
  
var x = myFunction(4, 3) * 2;
```

Functions are Objects

The **typeof** operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript functions have both **properties** and **methods**.

The arguments.length property returns the number of arguments received when the function was invoked:

Example

```
function myFunction(a, b) {  
    return arguments.length;  
}
```

The **toString()** method returns the function as a string:

Example

```
function myFunction(a, b) {  
    return a * b;  
}  
  
var txt = myFunction.toString();
```

A function defined as the property of an object, is called a method to the object. A function designed to create new objects, is called an object constructor.

JavaScript Function Parameters

A JavaScript function does not perform any checking on parameter values (arguments).

Function Parameters and Arguments

Earlier in this tutorial, you learned that functions can have **parameters**:

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

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

Function **arguments** are the real **values** passed to (and received by) the function.

Parameter Rules

JavaScript function definitions do not specify data types for parameters.

JavaScript functions do not perform type checking on the passed arguments.

JavaScript functions do not check the number of arguments received.

Parameter Defaults

If a function is called with **missing arguments** (less than declared), the missing values are set to: **undefined**

Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter:

Example

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

ECMAScript 2015 allows default parameters in the function call:

```
function (a=1, b=1) { // function code }
```

The Arguments Object

JavaScript functions have a built-in object called the arguments object.

The argument object contains an array of the arguments used when the function was called (invoked).

This way you can simply use a function to find (for instance) the highest value in a list of numbers:

Example

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
    var i;
    var max = -Infinity;
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}
```

Or create a function to sum all input values:

Example

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
    var i;
    var sum = 0;
    for (i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

If a function is called with **too many arguments** (more than declared), these arguments can be reached using **the arguments object**.

Arguments are Passed by Value

The parameters, in a function call, are the function's arguments.

JavaScript arguments are passed by **value**: The function only gets to know the values, not the argument's locations.

If a function changes an argument's value, it does not change the parameter's original value.

Changes to arguments are not visible (reflected) outside the function.

Objects are Passed by Reference

In JavaScript, object references are values.

Because of this, objects will behave like they are passed by **reference**:

If a function changes an object property, it changes the original value.

Changes to object properties are visible (reflected) outside the function.

JavaScript Function Invocation

The code inside a JavaScript function will execute when "something" invokes it.

Invoking a JavaScript Function

The code inside a function is not executed when the function is **defined**.

The code inside a function is executed when the function is **invoked**.

It is common to use the term "**call a function**" instead of "**invoke a function**".

It is also common to say "call upon a function", "start a function", or "execute a function".

In this tutorial, we will use **invoke**, because a JavaScript function can be invoked without being called.

Invoking a Function as a Function

Example

```
function myFunction(a, b) {  
    return a * b;  
}  
myFunction(10, 2);           // Will return 20
```

The function above does not belong to any object. But in JavaScript there is always a default global object.

In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.

In a browser the page object is the browser window. The function above automatically becomes a window function.

myFunction() and window.myFunction() is the same function:

Example

```
function myFunction(a, b) {  
    return a * b;  
}  
window.myFunction(10, 2);    // Will also return 20
```

This is a common way to invoke a JavaScript function, but not a very good practice.

Global variables, methods, or functions can easily create name conflicts and bugs in the global object.

The **this** Keyword

In JavaScript, the thing called **this**, is the object that "owns" the current code. The value of this, when used in a function, is the object that "owns" the function.

Note that **this** is not a variable. It is a keyword. You cannot change the value of **this**.

The Global Object

When a function is called without an owner object, the value of **this** becomes the global object.

In a web browser the global object is the browser window.

This example returns the window object as the value of **this**:

Example

```
var x = myFunction();           // x will be the window object

function myFunction() {
    return this;
}
```

Invoking a function as a global function, causes the value of **this** to be the global object.
Using the window object as a variable can easily crash your program.

Invoking a Function as a Method

In JavaScript you can define functions as object methods.

The following example creates an object (**myObject**), with two properties (**firstName** and **lastName**), and a method (**fullName**):

Example

```
var myObject = {
    firstName: "John",
    lastName: "Doe",
    fullName: function () {
        return this.firstName + " " + this.lastName;
    }
}
myObject.fullName();           // Will return "John Doe"
```

The **fullName** method is a function. The function belongs to the object. **myObject** is the owner of the function.

The thing called **this**, is the object that "owns" the JavaScript code. In this case the value of **this** is **myObject**.

Test it! Change the **fullName** method to return the value of **this**:

Example

```
var myObject = {
    firstName: "John",
    lastName: "Doe",
    fullName: function () {
        return this;
    }
}
myObject.fullName();           // Will return [object Object] (the owner
                             object)
```

Invoking a function as an object method, causes the value of **this** to be the object itself.

Invoking a Function with a Function Constructor

If a function invocation is preceded with the **new** keyword, it is a constructor invocation. It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

Example

```
// This is a function constructor:  
function myFunction(arg1, arg2) {  
    this.firstName = arg1;  
    this.lastName  = arg2;  
}  
  
// This creates a new object  
var x = new myFunction("John", "Doe");  
x.firstName;                                // Will return "John"
```

A constructor invocation creates a new object. The new object inherits the properties and methods from its constructor.

The **this** keyword in the constructor does not have a value. The value of **this** will be the new object created when the function is invoked.

JavaScript Function Call

Method Reuse

With the **call() method**, you can write a method that can be used on different objects.

All Functions are Methods

In JavaScript all functions are object methods.

If a function is not a method of a JavaScript object, it is a function of the global object (see previous chapter).

The example below creates an object with 3 properties, firstName, lastName, fullName.

Example

```
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    fullName: function () {  
        return this.firstName + " " + this.lastName;  
    }  
}  
person.fullName();           // Will return "John Doe"
```

The **this** Keyword

In a function definition, **this** refers to the "owner" of the function.

In the example above, **this** is the **person object** that "owns" the **fullName** function.

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

Read more about the **this** keyword at [JS this Keyword](#).

The JavaScript call() Method

The **call()** method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With **call()**, an object can use a method belonging to another object.

This example calls the **fullName** method of person, using it on **person1**:

Example

```
var person = {  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
  
var person1 = {  
    firstName: "John",  
    lastName: "Doe",  
}  
  
var person2 = {  
    firstName: "Mary",  
    lastName: "Doe",  
}  
  
person.fullName.call(person1); // Will return "John Doe"
```

This example calls the **fullName** method of person, using it on **person2**:

Example

```
var person = {  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
var person1 = {  
    firstName: "John",  
    lastName: "Doe",  
}  
var person2 = {  
    firstName: "Mary",  
    lastName: "Doe",  
}  
person.fullName.call(person2); // Will return "Mary Doe"
```

The call() Method with Arguments

The **call()** method can accept arguments:

Example

```
var person = {  
    fullName: function(city, country) {  
        return this.firstName + " " + this.lastName + "," + city + "," +  
country;  
    }  
}  
var person1 = {  
    firstName: "John",  
    lastName: "Doe",  
}  
person.fullName.call(person1, "Oslo", "Norway");
```

JavaScript Function Apply

Method Reuse

With the **apply()** method, you can write a method that can be used on different objects.

The JavaScript apply() Method

The **apply()** method is similar to the **call()** method (previous chapter).

In this example the **fullName** method of **person** is applied on **person1**:

Example

```
var person = {  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
  
var person1 = {  
    firstName: "Mary",  
    lastName: "Doe",  
}  
person.fullName.apply(person1); // Will return "Mary Doe"
```

The Difference Between call() and apply()

The difference is:

The **call()** method takes arguments **separately**.

The **apply()** method takes arguments as an **array**.

The **apply()** method is very handy if you want to use an array instead of an argument list.

The apply() Method with Arguments

The **apply()** method accepts arguments in an array:

Example

```
var person = {
    fullName: function(city, country) {
        return this.firstName + " " + this.lastName + "," + city + "," +
    country;
}
var person1 = {
    firstName: "John",
    lastName: "Doe",
}
person.fullName.apply(person1, ["Oslo", "Norway"]);
```

Compared with the **call()** method:

Example

```
var person = {
    fullName: function(city, country) {
        return this.firstName + " " + this.lastName + "," + city + "," +
    country;
}
var person1 = {
    firstName: "John",
    lastName: "Doe",
}
person.fullName.call(person1, "Oslo", "Norway");
```

Simulate a Max Method on Arrays

You can find the largest number (in a list of numbers) using the `Math.max()` method:

Example

```
Math.max(1,2,3); // Will return 3
```

Since JavaScript **arrays** do not have a `max()` method, you can apply the `Math.max()` method instead.

Example

```
Math.max.apply(null, [1,2,3]); // Will also return 3
```

The first argument (`null`) does not matter. It is not used in this example.

These examples will give the same result:

Example

```
Math.max.apply(Math, [1,2,3]); // Will also return 3
```

Example

```
Math.max.apply("", [1,2,3]); // Will also return 3
```

Example

```
Math.max.apply(0, [1,2,3]); // Will also return 3
```

JavaScript Strict Mode

In JavaScript strict mode, if the first argument of the apply() method is not an object, it becomes the owner (object) of the invoked function. In "non-strict" mode, it becomes the global object.

JavaScript Closures

JavaScript variables can belong to the **local** or **global** scope.
Global variables can be made local (private) with **closures**.

Global Variables

A function can access all variables defined **inside** the function, like this:

Example

```
function myFunction() {  
    var a = 4;  
    return a * a;  
}
```

But a function can also access variables defined **outside** the function, like this:

Example

```
var a = 4;  
function myFunction() {  
    return a * a;  
}
```

In the last example, **a** is a **global** variable.

In a web page, global variables belong to the window object.

Global variables can be used (and changed) by all scripts in the page (and in the window).

In the first example, **a** is a **local** variable.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Variables created **without** the keyword **var**, are always global, even if they are created inside a function.

Variable Lifetime

Global variables live as long as your application (your window / your web page) lives.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

A Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

Example

```
// Initiate counter
var counter = 0;

// Function to increment counter
function add() {
    counter += 1;
}

// Call add() 3 times
add();
add();
add();

// The counter should now be 3
```

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

The counter should be local to the add() function, to prevent other code from changing it:

Example

```
// Initiate counter
var counter = 0;

// Function to increment counter
function add() {
    var counter;
    counter += 1;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 0
```

It did not work because we display the global counter instead of the local counter. We can remove the global counter and access the local counter by letting the function return it:

Example

```
// Function to increment counter
function add() {
    var counter;
    counter += 1;
    return counter;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 1.
```

It did not work because we reset the local counter every time we call the function.

A JavaScript inner function can solve this.

JavaScript Nested Functions

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function **plus()** has access to the **counter** variable in the parent function:

Example

```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}
```

This could have solved the counter dilemma, if we could reach the **plus()** function from the outside.

We also need to find a way to execute **counter = 0** only once.

We need a closure.

JavaScript Closures

Remember self-invoking functions? What does this function do?

Example

```
var add = (function () {
    var counter = 0;
    return function () {counter += 1; return counter}
})();

add();
add();
add();

// the counter is now 3
```

Example Explained

The variable **add** is assigned the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure**. It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

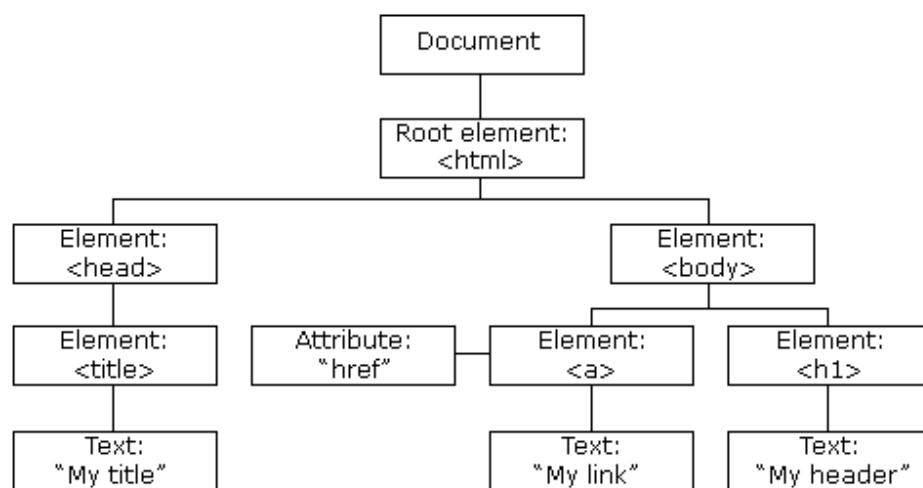
JavaScript HTML DOM

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

The HTML DOM document object is the owner of all other objects in your web page.

The HTML DOM Document Object

The document object represents your web page.

If you want to access any element 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.

Finding HTML Elements

Method	Description
<code>document.getElementById(<i>id</i>)</code>	Find an element by element id
<code>document.getElementsByTagName(<i>name</i>)</code>	Find elements by tag name
<code>document.getElementsByClassName(<i>name</i>)</code>	Find elements by class name

Changing HTML Elements

Method	Description
<code>element.innerHTML = new html content</code>	Change the inner HTML of an element
<code>element.attribute = new value</code>	Change the attribute value of an HTML element
<code>element.setAttribute(attribute, value)</code>	Change the attribute value of an HTML element
<code>element.style.property = new style</code>	Change the style of an HTML element

Adding and Deleting Elements

Method	Description
<code>document.createElement(element)</code>	Create an HTML element
<code>document.removeChild(element)</code>	Remove an HTML element
<code>document.appendChild(element)</code>	Add an HTML element
<code>document.replaceChild(element)</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
<code>document.anchors</code>	Returns all <code><a></code> elements that have a name attribute	1
<code>document.applets</code>	Returns all <code><applet></code> elements (Deprecated in HTML5)	1
<code>document.baseURI</code>	Returns the absolute base URI of the document	3
<code>document.body</code>	Returns the <code><body></code> element	1
<code>document.cookie</code>	Returns the document's cookie	1
<code>document.doctype</code>	Returns the document's doctype	3
<code>document.documentElement</code>	Returns the <code><html></code> element	3
<code>document.documentMode</code>	Returns the mode used by the browser	3

document.documentElement	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 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 CSS selectors
- Finding HTML elements by HTML object collections

Finding HTML Element by Id

The easiest way to find an HTML element in the DOM, is by using the element id.

This example finds the element with id="intro":

Example

```
var myElement = document.getElementById("intro");
```

If the element is found, the method will return the element as an object (in myElement).

If the element is not found, myElement will contain null.

Finding HTML Elements by Tag Name

This example finds all <p> elements:

Example

```
var x = document.getElementsByTagName("p");
```

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 getElementsByTagName().

This example returns a list of all elements with class="intro".

Example

```
var x = document.getElementsByClassName("intro");
```

Finding elements by class name does not work in Internet Explorer 8 and earlier versions.

Finding HTML Elements by CSS Selectors

If you want to find all HTML elements that matches a specified CSS selector (id, class names, types, attributes, values of attributes, etc), use the querySelectorAll() method.

This example returns a list of all <p> elements with class="intro".

Example

```
var x = document.querySelectorAll("p.intro");
```

The querySelectorAll() method does not work in Internet Explorer 8 and earlier versions.

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.forms["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: Tue Oct 02 2018 20:44:26 GMT+0900 (日本標準時)

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>
```

Example explained:

- The HTML document above contains a <p> element with id="p1"
- We use the HTML DOM to get the element with id="p1"
- A JavaScript changes the content (innerHTML) of that element to "New text!"

This example changes the content of an <h1> element:

Example

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

<h1 id="id01">Old Heading</h1>

<script>
var element = document.getElementById("id01");
element.innerHTML = "New Heading";
</script>

</body>
</html>
```

Example explained:

- The HTML document above contains an `<h1>` element with `id="id01"`
- We use the HTML DOM to get the element with `id="id01"`
- A JavaScript changes the content (`innerHTML`) of that element to "New Heading"

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 element:

Example

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



<script>
document.getElementById("myImage").src = "landscape.jpg";
</script>

</body>
</html>
```

Example explained:

- The HTML document above contains an 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>
```

More Examples

Visibility How to make an element invisible. Do you want to show the element or not?

HTML DOM Style Object Reference

For all HTML DOM style properties, look at our complete [HTML DOM Style Object Reference](#).

JavaScript HTML DOM Animation

Learn to create HTML animations using JavaScript.

A Basic Web Page

To demonstrate how to create HTML animations with JavaScript, we will use a simple web page:

Example

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

<h1>My First JavaScript Animation</h1>

<div id="animation">My animation will go here</div>

</body>
</html>
```

Create an Animation Container

All animations should be relative to a container element.

Example

```
<div id ="container">
  <div id ="animate">My animation will go here</div>
</div>
```

Style the Elements

The container element should be created with style = "position: relative".

The animation element should be created with style = "position: absolute".

Example

```
#container {  
    width: 400px;  
    height: 400px;  
    position: relative;  
    background: yellow;  
}  
  
#animate {  
    width: 50px;  
    height: 50px;  
    position: absolute;  
    background: red;  
}
```

Animation Code

JavaScript animations are done by programming gradual changes in an element's style. The changes are called by a timer. When the timer interval is small, the animation looks continuous.

The basic code is:

Example

```
var id = setInterval(frame, 5);  
  
function frame() {  
    if /* test for finished */ {  
        clearInterval(id);  
    } else {  
        /* code to change the element style */  
    }  
}
```

Create the Animation Using JavaScript

Example

```
function myMove() {  
    var elem = document.getElementById("animate");  
    var pos = 0;  
    var id = setInterval(frame, 5);  
    function frame() {  
        if (pos == 350) {  
            clearInterval(id);  
        } else {  
            pos++;  
            elem.style.top = pos + 'px';  
            elem.style.left = pos + 'px';  
        }  
    }  
}
```

JavaScript HTML DOM Events

HTML DOM allows JavaScript to react to HTML events:

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 = 'Ooops!'">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 = "Ooops!";
}
</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 = 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 is 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:

Mouse Over Me

[Try it Yourself »](#)

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.

Click Me

[Try it Yourself »](#)

More Examples

[onmousedown and onmouseup](#) Change an image when a user holds down the mouse button.

[onload](#) Display an alert box when the page has finished loading.

[onfocus](#) Change the background-color of an input field when it gets focus.

[Mouse Events](#) Change the color of an element when the cursor moves over it.

HTML DOM Event Object Reference

For a list of all HTML DOM events, look at our complete [HTML DOM Event Object Reference](#).

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 support 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					
addEventListener()	1.0	9.0	1.0	1.0	7.0
removeEventListener()	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 6.0 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);
}
```

HTML DOM Event Object Reference

For a list of all HTML DOM events, look at our complete [HTML DOM Event Object Reference](#).

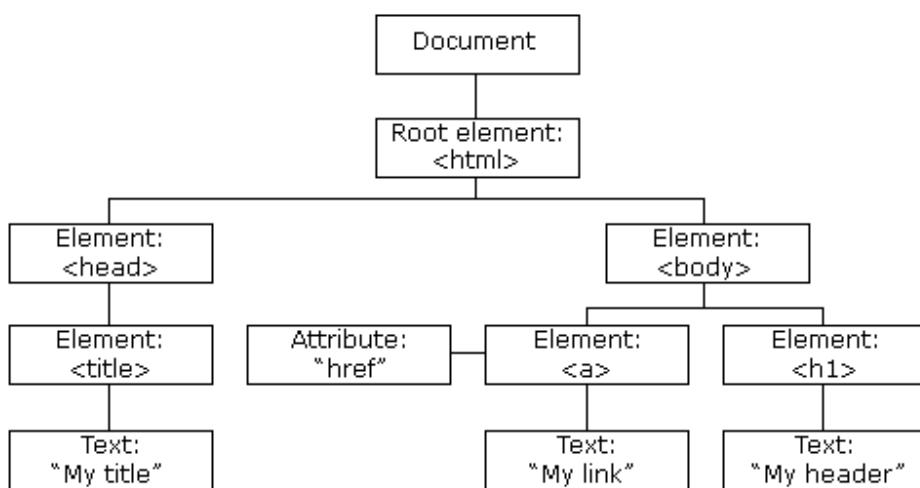
JavaScript HTML DOM Navigation

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 (deprecated)
- 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.

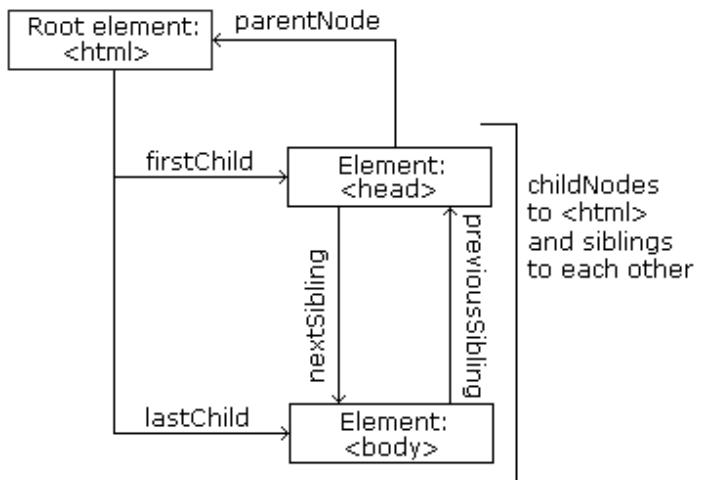
The terms **parent**, **child**, and **sibling** are used to describe the relationships.

- 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[nodenumber]
- firstChild
- lastChild
- nextSibling
- previousSibling

Child Nodes and Node Values

A common error in DOM processing is to expect an element node to contain text.

Example:

```
<title id="demo">DOM Tutorial</title>
```

The element node `<title>` (in the example above) 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:

Accessing the `innerHTML` property is the same as accessing the **nodeValue** of the first child:

```
var myTitle = document.getElementById("demo").firstChild.nodeValue;
```

Accessing the first child can also be done like this:

```
var myTitle = document.getElementById("demo").childNodes[0].nodeValue;
```

All the (3) following examples retrieves the text of an `<h1>` element and copies it into a `<p>` element:

Example

```
<html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").innerHTML;
</script>

</body>
</html>
```

Example

```
<html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").firstChild.nodeValue;
</script>

</body>
</html>
```

Example

```
<html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02">Hello!</p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").childNodes[0].nodeValue;
</script>

</body>
</html>
```

InnerHTML

In this tutorial we use the innerHTML property to retrieve the content of an HTML element. However, learning the other methods above is useful for understanding the tree structure and the navigation of the DOM.

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

Example

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").nodeName;
</script>
```

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 is read only. It returns the type of a node.

Example

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").nodeType;
</script>
```

The most important `nodeType` properties are:

Node	Type	Example
ELEMENT_NODE	1	<h1 class="heading">W3Schools</h1>
ATTRIBUTE_NODE	2	class = "heading" (deprecated)
TEXT_NODE	3	W3Schools
COMMENT_NODE	8	<!-- This is a comment -->
DOCUMENT_NODE	9	The HTML document itself (the parent of <html>)
DOCUMENT_TYPE_NODE	10	<!Doctype html>

Type 2 is deprecated in the HTML DOM (but works). It is not deprecated in the XML DOM.

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>
```

The `node.remove()` method does not work in any versions of the Internet Explorer browser.

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 Collections

The HTMLCollection Object

The `getElementsByName()` method returns an **HTMLCollection** object.

An **HTMLCollection** object is an array-like list (collection) of HTML elements.

The following code selects all `<p>` elements in a document:

Example

```
var x = document.getElementsByTagName("p");
```

The elements in the collection can be accessed by an index number.

To access the second `<p>` element you can write:

```
y = x[1];
```

Note: The index starts at 0.

HTML HTMLCollection Length

The `length` property defines the number of elements in an **HTMLCollection**:

Example

```
var myCollection = document.getElementsByTagName("p");
document.getElementById("demo").innerHTML = myCollection.length;
```

Example explained:

1. Create a collection of all `<p>` elements
2. Display the length of the collection

The `length` property is useful when you want to loop through the elements in a collection:

Example

Change the background color of all <p> elements:

```
var myCollection = document.getElementsByTagName("p");
var i;
for (i = 0; i < myCollection.length; i++) {
    myCollection[i].style.backgroundColor = "red";
}
```

An HTMLCollection is NOT an array!

An HTMLCollection may look like an array, but it is not.

You can loop through the list and refer to the elements with a number (just like an array).

However, you cannot use array methods like valueOf(), pop(), push(), or join() on an HTMLCollection.

JavaScript HTML DOM Node Lists

The HTML DOM NodeList Object

A **NodeList** object is a list (collection) of nodes extracted from a document.

A NodeList object is almost the same as an **HTMLCollection** object.

Some (older) browsers return a NodeList object instead of an **HTMLCollection** for methods like **getElementsByClassName()**.

All browsers return a NodeList object for the property **childNodes**.

Most browsers return a NodeList object for the method **querySelectorAll()**.

The following code selects all `<p>` nodes in a document:

Example

```
var myNodeList = document.querySelectorAll("p");
```

The elements in the NodeList can be accessed by an index number.

To access the second `<p>` node you can write:

```
y = myNodeList[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.querySelectorAll("p");
document.getElementById("demo").innerHTML = myNodelist.length;
```

Example explained:

1. Create a list of all `<p>` elements
2. Display the length of the 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.querySelectorAll("p");
var i;
for (i = 0; i < myNodelist.length; i++) {
  myNodelist[i].style.backgroundColor = "red";
}
```

The Difference Between an HTMLCollection and a NodeList

An HTMLCollection (previous chapter) is a collection of HTML elements.

A NodeList is a collection of document nodes.

A NodeList and an HTML collection is very much the same thing.

Both an HTMLCollection object and a NodeList object is an array-like list (collection) of objects.

Both have a length property defining the number of items in the list (collection).

Both provide an index (0, 1, 2, 3, 4, ...) to access each item like an array.

HTMLCollection items can be accessed by their name, id, or index number.

NodeList items can only be accessed by their index number.

Only the NodeList object can contain attribute nodes and text nodes.

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(), push(), pop(), or join() on a 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 represents 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

Two properties can be used to determine the size of the browser window.

Both properties return the sizes in pixels:

- `window.innerHeight` - the inner height of the browser window (in pixels)
- `window.innerWidth` - the inner width of the browser window (in pixels)

The browser window (the browser viewport) is NOT including toolbars and scrollbars.

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:

```
document.getElementById("demo").innerHTML =
"Screen Width: " + screen.width;
```

Result will be:

```
Screen Width: 1920
```

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:

```
document.getElementById("demo").innerHTML =  
"Screen Height: " + screen.height;
```

Result will be:

```
Screen Height: 1080
```

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:

```
document.getElementById("demo").innerHTML =  
"Available Screen Width: " + screen.availWidth;
```

Result will be:

```
Available Screen Width: 1920
```

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:

```
document.getElementById("demo").innerHTML =  
"Available Screen Height: " + screen.availHeight;
```

Result will be:

```
Available Screen Height: 1040
```

Window Screen Color Depth

The `screen.colorDepth` property returns the number of bits used to display one color. All modern computers use 24 bit or 32 bit hardware for color resolution:

- 24 bits = 16,777,216 different "True Colors"
- 32 bits = 4,294,967,296 different "Deep Colors"

Older computers used 16 bits: 65,536 different "High Colors" resolution.

Very old computers, and old cell phones used 8 bits: 256 different "VGA colors".

Example

Display the color depth of the screen in bits:

```
document.getElementById("demo").innerHTML =  
"Screen Color Depth: " + screen.colorDepth;
```

Result will be:

```
Screen Color Depth: 24
```

The `#rrggbb` (rgb) values used in HTML represents "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:

```
document.getElementById("demo").innerHTML =  
"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:

```
document.getElementById("demo").innerHTML =
"Page location is " + window.location.href;
```

Result is:

```
Page location is https://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:

```
document.getElementById("demo").innerHTML =  
"Page hostname is " + window.location.hostname;
```

Result is:

```
Page hostname is www.w3schools.com
```

Window Location Pathname

The **window.location.pathname** property returns the pathname of the current page.

Example

Display the path name of the current URL:

```
document.getElementById("demo").innerHTML =  
"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:

```
document.getElementById("demo").innerHTML =  
"Page protocol is " + window.location.protocol;
```

Result is:

```
Page protocol is https:
```

Window Location Port

The **window.location.port** property returns the number of the internet host port (of the current page).

Example

Display the name of the host:

```
document.getElementById("demo").innerHTML =  
"Port number is " + window.location.port;
```

Result is:

```
Port name is
```

Most browsers will not display default port numbers (80 for http and 443 for https)

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("https://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 browser's 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:

Back

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:

Forward

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`

Browser Cookies

The **cookieEnabled** property returns true if cookies are enabled, otherwise false:

Example

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

<script>
document.getElementById("demo").innerHTML =
"cookiesEnabled is " + navigator.cookieEnabled;
</script>
```

Browser Application Name

The **appName** property returns the application name of the browser:

Example

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

<script>
document.getElementById("demo").innerHTML =
"navigator.appName is " + navigator.appName;
</script>
```

Strange enough, "Netscape" is the application name for both IE11, Chrome, Firefox, and Safari.

Browser Application Code Name

The **appCodeName** property returns the application code name of the browser:

Example

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

<script>
document.getElementById("demo").innerHTML =
"navigator.appCodeName is " + navigator.appCodeName;
</script>
```

"Mozilla" is the application code name for both Chrome, Firefox, IE, Safari, and Opera.

The Browser Engine

The **product** property returns the product name of the browser engine:

Example

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

<script>
document.getElementById("demo").innerHTML =
"navigator.product is " + navigator.product;
</script>
```

Do not rely on this. Most browsers returns "Gecko" as product name !!

The Browser Version

The **appVersion** property returns version information about the browser:

Example

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

<script>
document.getElementById("demo").innerHTML = navigator.appVersion;
</script>
```

The Browser Agent

The **userAgent** property returns the user-agent header sent by the browser to the server:

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 **platform** property returns the browser platform (operating system):

Example

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

<script>
document.getElementById("demo").innerHTML = navigator.platform;
</script>
```

The Browser Language

The **language** property returns the browser's language:

Example

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

<script>
document.getElementById("demo").innerHTML = navigator.language;
</script>
```

Is The Browser Online?

The **onLine** property returns true if the browser is online:

Example

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

<script>
document.getElementById("demo").innerHTML = navigator.onLine;
</script>
```

Is Java Enabled?

The **javaEnabled()** method 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

```
if (confirm("Press a button!")) {
    txt = "You pressed OK!";
} else {
    txt = "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 || person == "") {
    txt = "User cancelled the prompt.";
} else {
    txt = "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

JavaScript can be executed in time-intervals.
This is called timing events.

Timing Events

The window object allows execution of code at specified time intervals.

These time intervals are called timing events.

The two key methods to use with JavaScript are:

- `setTimeout(function, milliseconds)`Executes a function, after waiting a specified number of milliseconds.
- `setInterval(function, milliseconds)`Same as `setTimeout()`, but repeats the execution of the function continuously.

The `setTimeout()` and `setInterval()` are both methods of the HTML DOM Window object.

The `setTimeout()` Method

```
window.setTimeout(function, milliseconds);
```

The **window.setTimeout()** method can be written without the window prefix.

The first parameter is a function to be executed.

The second parameter indicates the number of milliseconds before execution.

Example

Click a button. Wait 3 seconds, and the page will alert "Hello":

```
<button onclick="setTimeout(myFunction, 3000)">Try it</button>

<script>
function myFunction() {
    alert('Hello');
}
</script>
```

How to Stop the Execution?

The `clearTimeout()` method stops the execution of the function specified in `setTimeout()`.

```
window.clearTimeout(timeoutVariable)
```

The **window.clearTimeout()** method can be written without the window prefix.

The `clearTimeout()` method uses the variable returned from `setTimeout()`:

```
myVar = setTimeout(function, milliseconds); clearTimeout(myVar);
```

If the function has not already been executed, you can stop the execution by calling the `clearTimeout()` method:

Example

Same example as above, but with an added "Stop" button:

```
<button onclick="myVar = setTimeout(myFunction, 3000)">Try it</button>  
<button onclick="clearTimeout(myVar)">Stop it</button>
```

The setInterval() Method

The `setInterval()` method repeats a given function at every given time-interval.

```
window.setInterval(function, milliseconds);
```

The **window.setInterval()** method can be written without the window prefix.

The first parameter is the function to be executed.

The second parameter indicates the length of the time-interval between each execution.

This example executes a function called "myTimer" once every second (like a digital watch).

Example

Display the current time:

```
var myVar = setInterval(myTimer, 1000);

function myTimer() {
    var d = new Date();
    document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
```

There are 1000 milliseconds in one second.

How to Stop the Execution?

The clearInterval() method stops the executions of the function specified in the setInterval() method.

```
window.clearInterval(timerVariable)
```

The **window.clearInterval()** method can be written without the window prefix.

The clearInterval() method uses the variable returned from setInterval():

```
myVar = setInterval(function, milliseconds); clearInterval(myVar);
```

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(myTimer, 1000);
function myTimer() {
    var d = new Date();
    document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
</script>
```

More Examples

[Another simple timing](#)

[A clock created with a timing event](#)

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 requests 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.

None of the examples below will work if your browser has local cookies support turned off.

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.

You don't have to specify a cookie value when you delete a cookie.

Just set the expires parameter to a passed date:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
```

You should define the cookie path to ensure that you delete the right cookie.
Some browsers will not let you delete a cookie if you don't specify the path.

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;

[Display All Cookies](#)

[Create Cookie 1](#)

[Create Cookie 2](#)

[Delete Cookie 1](#)

[Delete Cookie 2](#)

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 + ";path=/";  
}
```

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 decodedCookie = decodeURIComponent(document.cookie);  
    var ca = decodedCookie.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 + "=").

Decode the cookie string, to handle cookies with special characters, e.g. '\$'

Split document.cookie on semicolons into an array called ca (ca = decodedCookie.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 + ";path=/";
}

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.

AJAX Introduction

AJAX is a developer's dream, because you can:

- Read data from a web server - after the page has loaded
- Update a web page without reloading the page
- Send data to a web server - in the background

AJAX Example

Let AJAX change this text

Change Content

AJAX Example Explained

HTML Page

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

<div id="demo">
  <h2>Let AJAX change this text</h2>
  <button type="button" onclick="loadDoc()">Change Content</button>
</div>

</body>
</html>
```

The HTML page contains a `<div>` section and a `<button>`.

The `<div>` section is used to display information from a server.

The `<button>` calls a function (if it is clicked).

The function requests data from a web server and displays it:

Function loadDoc()

```
function loadDoc() {  
    var xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
        if (this.readyState == 4 && this.status == 200) {  
            document.getElementById("demo").innerHTML = this.responseText;  
        }  
    };  
    xhttp.open("GET", "ajax_info.txt", true);  
    xhttp.send();  
}
```

What is AJAX?

AJAX = **A**synchronous **J**avaScript **A**nd **X**ML.

AJAX is not a programming language.

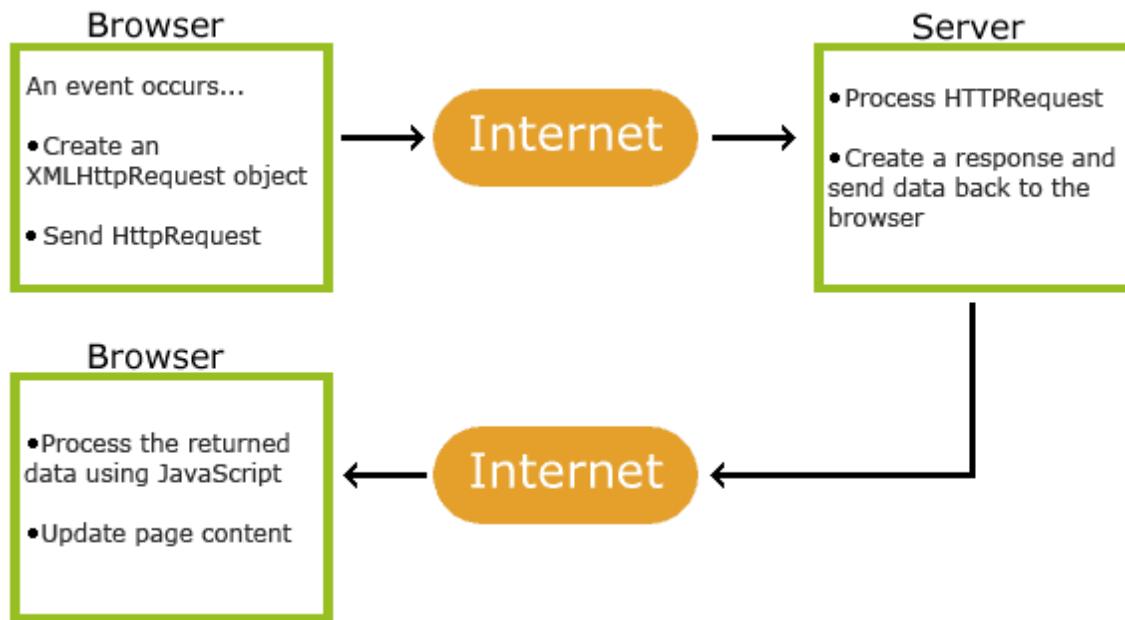
AJAX just uses a combination of:

- A browser built-in XMLHttpRequest object (to request data from a web server)
- JavaScript and HTML DOM (to display or use the data)

AJAX is a misleading name. AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text.

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

How AJAX Works



1. An event occurs in a web page (the page is loaded, a button is clicked)
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action (like page update) is performed by JavaScript

AJAX - The XMLHttpRequest Object

The keystone of AJAX is the XMLHttpRequest object.

The XMLHttpRequest Object

All modern browsers support the XMLHttpRequest object.

The XMLHttpRequest object can be used to exchange data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

Create an XMLHttpRequest Object

All modern browsers (Chrome, Firefox, IE7+, Edge, Safari, Opera) have a built-in XMLHttpRequest object.

Syntax for creating an XMLHttpRequest object:

```
variable = new XMLHttpRequest();
```

Example

```
var xhttp = new XMLHttpRequest();
```

Access Across Domains

For security reasons, modern browsers do not allow access across domains.

This means that both the web page and the XML file it tries to load, must be located on the same server.

The examples on W3Schools all open XML files located on the W3Schools domain.

If you want to use the example above on one of your own web pages, the XML files you load must be located on your own server.

Older Browsers (IE5 and IE6)

Old versions of Internet Explorer (5/6) use an ActiveX object instead of the XMLHttpRequest object:

```
variable = new ActiveXObject("Microsoft.XMLHTTP");
```

To handle IE5 and IE6, check if the browser supports the XMLHttpRequest object, or else create an ActiveX object:

Example

```
if (window.XMLHttpRequest) {  
    // code for modern browsers  
    xmlhttp = new XMLHttpRequest();  
} else {  
    // code for old IE browsers  
    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");  
}
```

XMLHttpRequest Object Methods

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(<i>method, url, async, user, psw</i>)	Specifies the request <i>method</i> : the request type GET or POST <i>url</i> : the file location <i>async</i> : true (asynchronous) or false (synchronous) <i>user</i> : optional user name <i>psw</i> : optional password
send()	Sends the request to the serverUsed for GET requests
send(<i>string</i>)	Sends the request to the server.Used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

XMLHttpRequest Object Properties

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number of a request 200: "OK" 403: "Forbidden" 404: "Not Found" For a complete list go to the Http Messages Reference
statusText	Returns the status-text (e.g. "OK" or "Not Found")

AJAX - Send a Request To a Server

The XMLHttpRequest object is used to exchange data with a server.

Send a Request To a Server

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object:

```
xhttp.open("GET", "ajax_info.txt", true); xhttp.send();
```

Method	Description
open(<i>method, url, async</i>)	Specifies the type of request <i>method</i> : the type of request: GET or POST <i>url</i> : the server (file) location <i>async</i> : true (asynchronous) or false (synchronous)
send()	Sends the request to the server (used for GET)
send(<i>string</i>)	Sends the request to the server (used for POST)

GET or POST?

GET is simpler and faster than POST, and can be used in most cases.

However, always use POST requests when:

- A cached file is not an option (update a file or database on the server).
- Sending a large amount of data to the server (POST has no size limitations).
- Sending user input (which can contain unknown characters), POST is more robust and secure than GET.

GET Requests

A simple GET request:

Example

```
xhttp.open("GET", "demo_get.asp", true);
xhttp.send();
```

In the example above, you may get a cached result. To avoid this, add a unique ID to the URL:

Example

```
xhttp.open("GET", "demo_get.asp?t=" + Math.random(), true);
xhttp.send();
```

If you want to send information with the GET method, add the information to the URL:

Example

```
xhttp.open("GET", "demo_get2.asp?fname=Henry&lname=Ford", true);
xhttp.send();
```

POST Requests

A simple POST request:

Example

```
xhttp.open("POST", "demo_post.asp", true);
xhttp.send();
```

To POST data like an HTML form, add an HTTP header with setRequestHeader(). Specify the data you want to send in the send() method:

Example

```
xhttp.open("POST", "ajax_test.asp", true);
xhttp.setRequestHeader("Content-type", "application/x-www-form-
urlencoded");
xhttp.send("fname=Henry&lname=Ford");
```

Method	Description
setRequestHeader(header, value)	Adds HTTP headers to the request <i>header</i> : specifies the header name <i>value</i> : specifies the header value

The url - A File On a Server

The url parameter of the open() method, is an address to a file on a server:

```
xhttp.open("GET", "ajax_test.asp", true);
```

The file can be any kind of file, like .txt and .xml, or server scripting files like .asp and .php (which can perform actions on the server before sending the response back).

Asynchronous - True or False?

Server requests should be sent asynchronously.

The async parameter of the open() method should be set to true:

```
xhttp.open("GET", "ajax_test.asp", true);
```

By sending asynchronously, the JavaScript does not have to wait for the server response, but can instead:

- execute other scripts while waiting for server response
- deal with the response after the response is ready

The onreadystatechange Property

With the XMLHttpRequest object you can define a function to be executed when the request receives an answer.

The function is defined in the **onreadystatechange** property of the XMLHttpRequest object:

Example

```
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    document.getElementById("demo").innerHTML = this.responseText;
  }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

You will learn more about onreadystatechange in a later chapter.

Synchronous Request

To execute a synchronous request, change the third parameter in the open() method to false:

```
xhttp.open("GET", "ajax_info.txt", false);
```

Sometimes async = false are used for quick testing. You will also find synchronous requests in older JavaScript code.

Since the code will wait for server completion, there is no need for an onreadystatechange function:

Example

```
xhttp.open("GET", "ajax_info.txt", false);
xhttp.send();
document.getElementById("demo").innerHTML = xhttp.responseText;
```

Synchronous XMLHttpRequest (async = false) is not recommended because the JavaScript will stop executing until the server response is ready. If the server is busy or slow, the application will hang or stop.

Synchronous XMLHttpRequest is in the process of being removed from the web standard, but this process can take many years.

Modern developer tools are encouraged to warn about using synchronous requests and may throw an `InvalidAccessError` exception when it occurs.

AJAX - Server Response

The onreadystatechange Property

The **readyState** property holds the status of the XMLHttpRequest.

The **onreadystatechange** property defines a function to be executed when the readyState changes.

The **status** property and the **statusText** property holds the status of the XMLHttpRequest object.

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
status	200: "OK" 403: "Forbidden" 404: "Page not found" For a complete list go to the Http Messages Reference
statusText	Returns the status-text (e.g. "OK" or "Not Found")

The onreadystatechange function is called every time the readyState changes.

When readyState is 4 and status is 200, the response is ready:

Example

```
function loadDoc() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("demo").innerHTML =
            this.responseText;
        }
    };
    xhttp.open("GET", "ajax_info.txt", true);
    xhttp.send();
}
```

The onreadystatechange event is triggered four times (1-4), one time for each change in the readyState.

Using a Callback Function

A callback function is a function passed as a parameter to another function.

If you have more than one AJAX task in a website, you should create one function for executing the XMLHttpRequest object, and one callback function for each AJAX task.

The function call should contain the URL and what function to call when the response is ready.

Example

```
loadDoc("url-1", myFunction1);

loadDoc("url-2", myFunction2);

function loadDoc(url, cFunction) {
    var xhttp;
    xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            cFunction(this);
        }
    };
    xhttp.open("GET", url, true);
    xhttp.send();
}

function myFunction1(xhttp) {
    // action goes here
}
function myFunction2(xhttp) {
    // action goes here
}
```

Server Response Properties

Property	Description
responseText	get the response data as a string
responseXML	get the response data as XML data

Server Response Methods

Method	Description
getResponseHeader()	Returns specific header information from the server resource
getAllResponseHeaders()	Returns all the header information from the server resource

The **responseText** Property

The **responseText** property returns the server response as a JavaScript string, and you can use it accordingly:

Example

```
document.getElementById("demo").innerHTML = xhttp.responseText;
```

The responseXML Property

The XML XMLHttpRequest object has an in-built XML parser.

The **responseXML** property returns the server response as an XML DOM object.

Using this property you can parse the response as an XML DOM object:

Example

Request the file [cd_catalog.xml](#) and parse the response:

```
xmlDoc = xhttp.responseXML;
txt = "";
x = xmlDoc.getElementsByTagName("ARTIST");
for (i = 0; i < x.length; i++) {
    txt += x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
xhttp.open("GET", "cd_catalog.xml", true);
xhttp.send();
```

You will learn a lot more about XML DOM in the DOM chapters of this tutorial.

The getAllResponseHeaders() Method

The **getAllResponseHeaders()** method returns all header information from the server response.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        document.getElementById("demo").innerHTML =
            this.getAllResponseHeaders();
    }
};
```

The getResponseHeader() Method

The **getResponseHeader()** method returns specific header information from the server response.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    document.getElementById("demo").innerHTML =
      this.getResponseHeader("Last-Modified");
  }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

AJAX XML Example

AJAX can be used for interactive communication with an XML file.

AJAX XML Example

The following example will demonstrate how a web page can fetch information from an XML file with AJAX:

Example

Get CD info

Example Explained

When a user clicks on the "Get CD info" button above, the loadDoc() function is executed. The loadDoc() function creates an XMLHttpRequest object, adds the function to be executed when the server response is ready, and sends the request off to the server.

When the server response is ready, an HTML table is built, nodes (elements) are extracted from the XML file, and it finally updates the element "demo" with the HTML table filled with XML data:

LoadXMLDoc()

```
function loadDoc() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            myFunction(this);
        }
    };
    xhttp.open("GET", "cd_catalog.xml", true);
    xhttp.send();
}
function myFunction(xml) {
    var i;
    var xmlDoc = xml.responseXML;
    var table=<tr><th>Artist</th><th>Title</th></tr>;
    var x = xmlDoc.getElementsByTagName("CD");
    for (i = 0; i <x.length; i++) {
        table += "<tr><td>" +
        x[i].getElementsByTagName("ARTIST")[0].childNodes[0].nodeValue +
        "</td><td>" +
        x[i].getElementsByTagName("TITLE")[0].childNodes[0].nodeValue +
        "</td></tr>";
    }
    document.getElementById("demo").innerHTML = table;
}
```

The XML File

The XML file used in the example above looks like this: "[cd_catalog.xml](#)".

JSON - Introduction

JSON: **J**ava**S**cript **O**bject **N**otation.

JSON is a syntax for storing and exchanging data.

JSON is text, written with JavaScript object notation.

Exchanging Data

When exchanging data between a browser and a server, the data can only be text.

JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server.

We can also convert any JSON received from the server into JavaScript objects.

This way we can work with the data as JavaScript objects, with no complicated parsing and translations.

Sending Data

If you have data stored in a JavaScript object, you can convert the object into JSON, and send it to a server:

Example

```
var myObj = {"name": "John", "age": 31, "city": "New York"};
var myJSON = JSON.stringify(myObj);
window.location = "demo_json.php?x=" + myJSON;
```

You will learn more about the `JSON.stringify()` function later in this tutorial.

Receiving Data

If you receive data in JSON format, you can convert it into a JavaScript object:

Example

```
var myJSON = '{"name":"John", "age":31, "city":"New York"}';
var myObj = JSON.parse(myJSON);
document.getElementById("demo").innerHTML = myObj.name;
```

You will learn more about the `JSON.parse()` function later in this tutorial.

Storing Data

When storing data, the data has to be a certain format, and regardless of where you choose to store it, *text* is always one of the legal formats.

JSON makes it possible to store JavaScript objects as text.

Example

Storing data in local storage

```
//Storing data:  
myObj = {name: "John", age: 31, city: "New York"};  
myJSON = JSON.stringify(myObj);  
localStorage.setItem("testJSON", myJSON);  
  
//Retrieving data:  
text = localStorage.getItem("testJSON");  
obj = JSON.parse(text);  
document.getElementById("demo").innerHTML = obj.name;
```

What is JSON?

- JSON stands for **JavaScript Object Notation**
- JSON is a lightweight data-interchange format
- JSON is "self-describing" and easy to understand
- JSON is language independent *

*JSON uses JavaScript syntax, but the JSON format is text only. Text can be read and used as a data format by any programming language.

The JSON format was originally specified by [Douglas Crockford](#).

Why use JSON?

Since the JSON format is text only, it can easily be sent to and from a server, and used as a data format by any programming language.

JavaScript has a built in function to convert a string, written in JSON format, into native JavaScript objects:

JSON.parse()

So, if you receive data from a server, in JSON format, you can use it like any other JavaScript object.

JSON Syntax

The JSON syntax is a subset of the JavaScript syntax.

JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

Example

```
"name": "John"
```

JSON names require double quotes. JavaScript names don't.

JSON - Evaluates to JavaScript Objects

The JSON format is almost identical to JavaScript objects.

In JSON, *keys* must be strings, written with double quotes:

JSON

```
{ "name": "John" }
```

In JavaScript, keys can be strings, numbers, or identifier names:

JavaScript

```
{ name: "John" }
```

In **JSON**, *values* must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- null

In **JavaScript** values can be all of the above, plus any other valid JavaScript expression, including:

- a function
- a date
- undefined

In JSON, *string values* must be written with double quotes:

JSON

```
{ "name": "John" }
```

In JavaScript, you can write string values with double *or* single quotes:

JavaScript

```
{ name: 'John' }
```

JSON Uses JavaScript Syntax

Because JSON syntax is derived from JavaScript object notation, very little extra software is needed to work with JSON within JavaScript.

With JavaScript you can create an object and assign data to it, like this:

Example

```
var person = { name: "John", age: 31, city: "New York" };
```

You can access a JavaScript object like this:

Example

```
// returns John  
person.name;
```

It can also be accessed like this:

Example

```
// returns John  
person["name"];
```

Data can be modified like this:

Example

```
person.name = "Gilbert";
```

It can also be modified like this:

Example

```
person["name"] = "Gilbert";
```

You will learn how to convert JavaScript objects into JSON later in this tutorial.

JavaScript Arrays as JSON

The same way JavaScript objects can be used as JSON, JavaScript arrays can also be used as JSON.

You will learn more about arrays as JSON later in this tutorial.

JSON Files

- The file type for JSON files is ".json"
- The MIME type for JSON text is "application/json"

JSON vs XML

Both JSON and XML can be used to receive data from a web server.

The following JSON and XML examples both define an employees object, with an array of 3 employees:

JSON Example

```
{"employees": [
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

XML Example

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSON is Like XML Because

- Both JSON and XML are "self describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages
- Both JSON and XML can be fetched with an XMLHttpRequest

JSON is Unlike XML Because

- JSON doesn't use end tag
- JSON is shorter
- JSON is quicker to read and write
- JSON can use arrays

The biggest difference is:

XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

Why JSON is Better Than XML

XML is much more difficult to parse than JSON. JSON is parsed into a ready-to-use JavaScript object.

For AJAX applications, JSON is faster and easier than XML:

Using XML

- Fetch an XML document
- Use the XML DOM to loop through the document
- Extract values and store in variables

Using JSON

- Fetch a JSON string
- JSON.Parse the JSON string

JSON Data Types

Valid Data Types

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- *null*

JSON values **cannot** be one of the following data types:

- a function
- a date
- *undefined*

JSON Strings

Strings in JSON must be written in double quotes.

Example

```
{ "name": "John" }
```

JSON Numbers

Numbers in JSON must be an integer or a floating point.

Example

```
{ "age": 30 }
```

JSON Objects

Values in JSON can be objects.

Example

```
{  
  "employee":{ "name":"John", "age":30, "city":"New York" }  
}
```

Objects as values in JSON must follow the same rules as JSON objects.

JSON Arrays

Values in JSON can be arrays.

Example

```
{  
  "employees":[ "John", "Anna", "Peter" ]  
}
```

JSON Booleans

Values in JSON can be true/false.

Example

```
{ "sale":true }
```

JSON null

Values in JSON can be null.

Example

```
{ "middlename":null }
```

JSON.parse()

A common use of JSON is to exchange data to/from a web server.

When receiving data from a web server, the data is always a string.

Parse the data with `JSON.parse()`, and the data becomes a JavaScript object.

Example - Parsing JSON

Imagine we received this text from a web server:

```
{ "name": "John", "age": 30, "city": "New York"}
```

Use the JavaScript function `JSON.parse()` to convert text into a JavaScript object:

```
var obj = JSON.parse('{ "name": "John", "age": 30, "city": "New York"}');
```

Make sure the text is written in JSON format, or else you will get a syntax error.

Use the JavaScript object in your page:

Example

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

<script>
document.getElementById("demo").innerHTML = obj.name + ", " + obj.age;
</script>
```

JSON From the Server

You can request JSON from the server by using an AJAX request

As long as the response from the server is written in JSON format, you can parse the string into a JavaScript object.

Example

Use the XMLHttpRequest to get data from the server:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        var myObj = JSON.parse(this.responseText);
        document.getElementById("demo").innerHTML = myObj.name;
    }
};
xmlhttp.open("GET", "json_demo.txt", true);
xmlhttp.send();
```

Take a look at [json_demo.txt](#)

Array as JSON

When using the JSON.parse() on a JSON derived from an array, the method will return a JavaScript array, instead of a JavaScript object.

Example

The JSON returned from the server is an array:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        var myArr = JSON.parse(this.responseText);
        document.getElementById("demo").innerHTML = myArr[0];
    }
};
xmlhttp.open("GET", "json_demo_array.txt", true);
xmlhttp.send();
```

Take a look at [json_demo_array.txt](#)

Exceptions

Parsing Dates

Date objects are not allowed in JSON.

If you need to include a date, write it as a string.

You can convert it back into a date object later:

Example

Convert a string into a date:

```
var text = '{ "name":"John", "birth":"1986-12-14", "city":"New York"}';
var obj = JSON.parse(text);
obj.birth = new Date(obj.birth);

document.getElementById("demo").innerHTML = obj.name + ", " + obj.birth;
```

Or, you can use the second parameter, of the `JSON.parse()` function, called *reviver*.

The *reviver* parameter is a function that checks each property, before returning the value.

Example

Convert a string into a date, using the *reviver* function:

```
var text = '{ "name":"John", "birth":"1986-12-14", "city":"New York"}';
var obj = JSON.parse(text, function (key, value) {
  if (key == "birth") {
    return new Date(value);
  } else {
    return value;
}});

document.getElementById("demo").innerHTML = obj.name + ", " + obj.birth;
```

Parsing Functions

Functions are not allowed in JSON.

If you need to include a function, write it as a string.

You can convert it back into a function later:

Example

Convert a string into a function:

```
var text = '{ "name":"John", "age":function () {return 30;}, "city":"New York"}';
var obj = JSON.parse(text);
obj.age = eval("(" + obj.age + ")");

document.getElementById("demo").innerHTML = obj.name + ", " + obj.age();
```

You should avoid using functions in JSON, the functions will lose their scope, and you would have to use eval() to convert them back into functions.

Browser Support

The JSON.parse() function is included in all major browsers and in the latest ECMAScript (JavaScript) standard:

Web Browsers Support

- Firefox 3.5
- Internet Explorer 8
- Chrome
- Opera 10
- Safari 4

For older browsers, a JavaScript library is available at

<https://github.com/douglascrockford/JSON-js>.

JSON.stringify()

A common use of JSON is to exchange data to/from a web server.
When sending data to a web server, the data has to be a string.
Convert a JavaScript object into a string with `JSON.stringify()`.

Stringify a JavaScript Object

Imagine we have this object in JavaScript:

```
var obj = { name: "John", age: 30, city: "New York" };
```

Use the JavaScript function `JSON.stringify()` to convert it into a string.

```
var myJSON = JSON.stringify(obj);
```

The result will be a string following the JSON notation.

`myJSON` is now a string, and ready to be sent to a server:

Example

```
var obj = { name: "John", age: 30, city: "New York" };
var myJSON = JSON.stringify(obj);
document.getElementById("demo").innerHTML = myJSON;
```

You will learn how to send JSON to the server in the next chapter.

Stringify a JavaScript Array

It is also possible to stringify JavaScript arrays:

Imagine we have this array in JavaScript:

```
var arr = [ "John", "Peter", "Sally", "Jane" ];
```

Use the JavaScript function `JSON.stringify()` to convert it into a string.

```
var myJSON = JSON.stringify(arr);
```

The result will be a string following the JSON notation.

myJSON is now a string, and ready to be sent to a server:

Example

```
var arr = [ "John", "Peter", "Sally", "Jane" ];
var myJSON = JSON.stringify(arr);
document.getElementById("demo").innerHTML = myJSON;
```

You will learn how to send JSON to the server in the next chapter.

Exceptions

Stringify Dates

In JSON, date objects are not allowed. The `JSON.stringify()` function will convert any dates into strings.

Example

```
var obj = { name: "John", today: new Date(), city : "New York" };
var myJSON = JSON.stringify(obj);

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

You can convert the string back into a date object at the receiver.

Stringify Functions

In JSON, functions are not allowed as object values.

The `JSON.stringify()` function will remove any functions from a JavaScript object, both the key and the value:

Example

```
var obj = { name: "John", age: function () {return 30;}, city: "New York"};
var myJSON = JSON.stringify(obj);

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

This can be omitted if you convert your functions into strings before running the `JSON.stringify()` function.

Example

```
var obj = { name: "John", age: function () {return 30;}, city: "New York"
};
obj.age = obj.age.toString();
var myJSON = JSON.stringify(obj);

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

You should avoid using functions in JSON, the functions will lose their scope, and you would have to use `eval()` to convert them back into functions.

Browser Support

The `JSON.stringify()` function is included in all major browsers and in the latest ECMAScript (JavaScript) standard:

Web Browsers Support

- Firefox 3.5
- Internet Explorer 8
- Chrome
- Opera 10
- Safari 4

JSON Objects

Object Syntax

Example

```
{ "name": "John", "age": 30, "car": null }
```

JSON objects are surrounded by curly braces {}.

JSON objects are written in key/value pairs.

Keys must be strings, and values must be a valid JSON data type (string, number, object, array, boolean or null).

Keys and values are separated by a colon.

Each key/value pair is separated by a comma.

Accessing Object Values

You can access the object values by using dot (.) notation:

Example

```
myObj = { "name": "John", "age": 30, "car": null };
x = myObj.name;
```

You can also access the object values by using bracket ([]) notation:

Example

```
myObj = { "name": "John", "age": 30, "car": null };
x = myObj["name"];
```

Looping an Object

You can loop through object properties by using the for-in loop:

Example

```
myObj = { "name":"John", "age":30, "car":null };
for (x in myObj) {
    document.getElementById("demo").innerHTML += x;
}
```

In a for-in loop, use the bracket notation to access the property *values*:

Example

```
myObj = { "name":"John", "age":30, "car":null };
for (x in myObj) {
    document.getElementById("demo").innerHTML += myObj[x];
}
```

Nested JSON Objects

Values in a JSON object can be another JSON object.

Example

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

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

Example

```
x = myObj.cars.car2;  
//or:  
x = myObj.cars["car2"];
```

Modify Values

You can use the dot notation to modify any value in a JSON object:

Example

```
myObj.cars.car2 = "Mercedes";
```

You can also use the bracket notation to modify a value in a JSON object:

Example

```
myObj.cars["car2"] = "Mercedes";
```

Delete Object Properties

Use the delete keyword to delete properties from a JSON object:

Example

```
delete myObj.cars.car2;
```

JSON Arrays

Arrays as JSON Objects

Example

```
[ "Ford", "BMW", "Fiat" ]
```

Arrays in JSON are almost the same as arrays in JavaScript.

In JSON, array values must be of type string, number, object, array, boolean or *null*.

In JavaScript, array values can be all of the above, plus any other valid JavaScript expression, including functions, dates, and *undefined*.

Arrays in JSON Objects

Arrays can be values of an object property:

Example

```
{
  "name": "John",
  "age": 30,
  "cars": [ "Ford", "BMW", "Fiat" ]
}
```

Accessing Array Values

You access the array values by using the index number:

Example

```
x = myObj.cars[0];
```

Looping Through an Array

You can access array values by using a for-in loop:

Example

```
for (i in myObj.cars) {  
    x += myObj.cars[i];  
}
```

Or you can use a for loop:

Example

```
for (i = 0; i < myObj.cars.length; i++) {  
    x += myObj.cars[i];  
}
```

Nested Arrays in JSON Objects

Values in an array can also be another array, or even another JSON object:

Example

```
myObj = {  
    "name": "John",  
    "age": 30,  
    "cars": [  
        { "name": "Ford", "models": [ "Fiesta", "Focus", "Mustang" ] },  
        { "name": "BMW", "models": [ "320", "X3", "X5" ] },  
        { "name": "Fiat", "models": [ "500", "Panda" ] }  
    ]  
}
```

To access arrays inside arrays, use a for-in loop for each array:

Example

```
for (i in myObj.cars) {  
    x += "<h1>" + myObj.cars[i].name + "</h1>";  
    for (j in myObj.cars[i].models) {  
        x += myObj.cars[i].models[j];  
    }  
}
```

Modify Array Values

Use the index number to modify an array:

Example

```
myObj.cars[1] = "Mercedes";
```

Delete Array Items

Use the delete keyword to delete items from an array:

Example

```
delete myObj.cars[1];
```

JSON PHP

A common use of JSON is to read data from a web server, and display the data in a web page.

This chapter will teach you how to exchange JSON data between the client and a PHP server.

The PHP File

PHP has some built-in functions to handle JSON.

Objects in PHP can be converted into JSON by using the PHP function `json_encode()` :

PHP file

```
<?php
$myObj->name = "John";
$myObj->age = 30;
$myObj->city = "New York";

$json = json_encode($myObj);

echo $json;
?>
```

[Show PHP file »](#)

The Client JavaScript

Here is a JavaScript on the client, using an AJAX call to request the PHP file from the example above:

Example

Use `JSON.parse()` to convert the result into a JavaScript object:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        var myObj = JSON.parse(this.responseText);
        document.getElementById("demo").innerHTML = myObj.name;
    }
};
xmlhttp.open("GET", "demo_file.php", true);
xmlhttp.send();
```

PHP Array

Arrays in PHP will also be converted into JSON when using the PHP function `json_encode()`:

PHP file

```
<?php
$myArr = array("John", "Mary", "Peter", "Sally");

$json = json_encode($myArr);

echo $json;
?>
```

[Show PHP file »](#)

The Client JavaScript

Here is a JavaScript on the client, using an AJAX call to request the PHP file from the array example above:

Example

Use JSON.parse() to convert the result into a JavaScript array:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        var myObj = JSON.parse(this.responseText);
        document.getElementById("demo").innerHTML = myObj[2];
    }
};
xmlhttp.open("GET", "demo_file_array.php", true);
xmlhttp.send();
```

PHP Database

PHP is a server side programming language, and should be used for operations that can only be performed by a server, like accessing a database.

Imagine you have a database on the server, containing customers, products, and suppliers.

You want to make a request to the server where you ask for the first 10 records in the "customers" table:

Example

Use JSON.stringify() to convert the JavaScript object into JSON:

```
obj = { "table":"customers", "limit":10 };
dbParam = JSON.stringify(obj);
xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        document.getElementById("demo").innerHTML = this.responseText;
    }
};
xmlhttp.open("GET", "json_demo_db.php?x=" + dbParam, true);
xmlhttp.send();
```

Example explained:

- Define an object containing a table property and a limit property.
- Convert the object into a JSON string.
- Send a request to the PHP file, with the JSON string as a parameter.
- Wait until the request returns with the result (as JSON)
- Display the result received from the PHP file.

Take a look at the PHP file:

PHP file

```
<?php
header("Content-Type: application/json; charset=UTF-8");
$obj = json_decode($_GET["x"], false);

$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");
$stmt = $conn->prepare("SELECT name FROM ? LIMIT ?");
$stmt->bind_param("ss", $obj->table, $obj->limit);
$stmt->execute();
$result = $stmt->get_result();
$outp = $result->fetch_all(MYSQLI_ASSOC);

echo json_encode($outp);
?>
```

PHP File explained:

- Convert the request into an object, using the PHP function `json_decode()`.
- Access the database, and fill an array with the requested data.
- Add the array to an object, and return the object as JSON using the `json_encode()` function.

Loop Through the Result

Convert the result received from the PHP file into a JavaScript object, or in this case, a JavaScript array:

Example

Use `JSON.parse()` to convert the JSON into a JavaScript object:

```
...
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myObj = JSON.parse(this.responseText);
        for (x in myObj) {
            txt += myObj[x].name + "<br>";
        }
        document.getElementById("demo").innerHTML = txt;
    }
};
...
```

PHP Method = POST

When sending data to the server, it is often best to use the HTTP POST method. To send AJAX requests using the POST method, specify the method, and the correct header. The data sent to the server must now be an argument to the .send() method:

Example

```
obj = { "table":"customers", "limit":10 };
dbParam = JSON.stringify(obj);
xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myObj = JSON.parse(this.responseText);
        for (x in myObj) {
            txt += myObj[x].name + "<br>";
        }
        document.getElementById("demo").innerHTML = txt;
    }
};
xmlhttp.open("POST", "json_demo_db_post.php", true);
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xmlhttp.send("x=" + dbParam);
```

The only difference in the PHP file is the method for getting the transferred data.

PHP file

Use `$_POST` instead of `$_GET`:

```
<?php
header("Content-Type: application/json; charset=UTF-8");
$obj = json_decode($_POST["x"], false);

$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");
$stmt = $conn->prepare("SELECT name FROM ? LIMIT ?");
$stmt->bind_param("ss", $obj->table, $obj->limit);
$stmt->execute();
$result = $stmt->get_result();
$outp = $result->fetch_all(MYSQLI_ASSOC);

echo json_encode($outp);
?>
```

JSON HTML

JSON can very easily be translated into JavaScript.
JavaScript can be used to make HTML in your web pages.

HTML Table

Make an HTML table with data received as JSON:

Example

```
obj = { table: "customers", limit: 20 };
dbParam = JSON.stringify(obj);
xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myObj = JSON.parse(this.responseText);
        txt += "<table border='1'>"
        for (x in myObj) {
            txt += "<tr><td>" + myObj[x].name + "</td></tr>";
        }
        txt += "</table>"
        document.getElementById("demo").innerHTML = txt;
    }
}
xmlhttp.open("POST", "json_demo_db_post.php", true);
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xmlhttp.send("x=" + dbParam);
```

Dynamic HTML Table

Make the HTML table based on the value of a drop down menu:

Example

```
<select id="myselect" onchange="change_myselect(this.value)">
    <option value="">Choose an option:</option>
    <option value="customers">Customers</option>
    <option value="products">Products</option>
    <option value="suppliers">Suppliers</option>
</select>

<script>
function change_myselect(sel) {
    var obj, dbParam, xmlhttp, myObj, x, txt = "";
    obj = { table: sel, limit: 20 };
    dbParam = JSON.stringify(obj);
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            myObj = JSON.parse(this.responseText);
            txt += "<table border='1'>";
            for (x in myObj) {
                txt += "<tr><td>" + myObj[x].name + "</td></tr>";
            }
            txt += "</table>";
            document.getElementById("demo").innerHTML = txt;
        }
    };
    xmlhttp.open("POST", "json_demo_db_post.php", true);
    xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    xmlhttp.send("x=" + dbParam);
}
</script>
```

HTML Drop Down List

Make an HTML drop down list with data received as JSON:

Example

```
obj = { table: "customers", limit: 20 };
dbParam = JSON.stringify(obj);
xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myObj = JSON.parse(this.responseText);
        txt += "<select>"
        for (x in myObj) {
            txt += "<option>" + myObj[x].name;
        }
        txt += "</select>"
        document.getElementById("demo").innerHTML = txt;
    }
}
xmlhttp.open("POST", "json_demo_db_post.php", true);
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xmlhttp.send("x=" + dbParam);
```

JSONP

JSONP is a method for sending JSON data without worrying about cross-domain issues.
JSONP does not use the XMLHttpRequest object.
JSONP uses the <script> tag instead.

JSONP Intro

JSONP stands for JSON with Padding.

Requesting a file from another domain can cause problems, due to cross-domain policy.
Requesting an external *script* from another domain does not have this problem.
JSONP uses this advantage, and request files using the script tag instead of the XMLHttpRequest object.

```
<script src="demo_jsonp.php">
```

The Server File

The file on the server wraps the result inside a function call:

Example

```
<?php
$myJSON = '{ "name":"John", "age":30, "city":"New York" }';

echo "myFunc(\".$myJSON.\")";
?>
```

[Show PHP file »](#)

The result returns a call to a function named "myFunc" with the JSON data as a parameter.
Make sure that the function exists on the client.

The JavaScript function

The function named "myFunc" is located on the client, and ready to handle JSON data:

Example

```
function myFunc(myObj) {  
    document.getElementById("demo").innerHTML = myObj.name;  
}
```

Creating a Dynamic Script Tag

The example above will execute the "myFunc" function when the page is loading, based on where you put the script tag, which is not very satisfying.

The script tag should only be created when needed:

Example

Create and insert the <script> tag when a button is clicked:

```
function clickButton() {  
    var s = document.createElement("script");  
    s.src = "demo_jsonp.php";  
    document.body.appendChild(s);  
}
```

Dynamic JSONP Result

The examples above are still very static.

Make the example dynamic by sending JSON to the php file, and let the php file return a JSON object based on the information it gets.

PHP file

```
<?php
header("Content-Type: application/json; charset=UTF-8");
$obj = json_decode($_GET["x"], false);

$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");
$result = $conn->query("SELECT name FROM ".$obj->$table." LIMIT ".$obj->$limit);
$outp = array();
$outp = $result->fetch_all(MYSQLI_ASSOC);

echo "myFunc(".json_encode($outp).")";
?>
```

PHP File explained:

- Convert the request into an object, using the PHP function `json_decode()`.
- Access the database, and fill an array with the requested data.
- Add the array to an object.
- Convert the array into JSON using the `json_encode()` function.
- Wrap "myFunc()" around the return object.

JavaScript Example

The "myFunc" function will be called from the php file:

```
function clickButton() {
    var obj, s
    obj = { table: "products", limit: 10 };
    s = document.createElement("script");
    s.src = "jsonp_demo_db.php?x=" + JSON.stringify(obj);
    document.body.appendChild(s);
}
function myFunc(myObj) {
    var x, txt = "";
    for (x in myObj) {
        txt += myObj[x].name + "<br>";
    }
    document.getElementById("demo").innerHTML = txt;
}
```

Callback Function

When you have no control over the server file, how do you get the server file to call the correct function?

Sometimes the server file offers a callback function as a parameter:

Example

The php file will call the function you pass as a callback parameter:

```
function clickButton() {
    var s = document.createElement("script");
    s.src = "jsonp_demo_db.php?callback=myDisplayFunction";
    document.body.appendChild(s);
}
```