JavaScript

JavaScript Introduction

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

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

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

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

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

</body>
</html>
```

JavaScript accepts both double and single quotes:

JavaScript Can Change HTML Attribute Values

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

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

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

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

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

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

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

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

</body>
</html>
```

JavaScript Can Change HTML Styles (CSS)

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

Example

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

or

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

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

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

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

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

</body>
</html>
```

JavaScript Can Hide HTML Elements

Hiding HTML elements can be done by changing the display style

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

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

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

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

</body>
</html>
```

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

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

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

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

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

</body>
</html>
```

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.

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

This example links to a script located in the same folder as the current page:
<script src="myScript1.js"></script>

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.

Example

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

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

```
</script>

</body>
</html>
```

JavaScript Statements

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

<h2>JavaScript Statements</h2>

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

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

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

document.getElementById("demo").innerHTML =
"The value of z is " + z + ".";
</script>

</body>
</html>
```

JavaScript Programs

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

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

JavaScript Statements

JavaScript statements are composed of:
Values, Operators, Expressions, Keywords, and Comments.

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":
Example
```
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 2 spaces of indentation for code blocks.

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
```

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.

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:

```
// Change heading:
document.getElementById("myH").innerHTML = "JavaScript Comments";
```

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

```
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:

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

```
in my web page:
*/
```

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:

```
//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:

```
/*
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;
```

Much Like Algebra
In this example, price1, price2, and total, are variables:

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

<h2>JavaScript Variables</h2>

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

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

</body>
</html>
```

In programming, just like in algebra, we use variables (like price1) to hold values.
In programming, just like in algebra, we use variables in expressions (total = price1 + price2).
From the example above, you can calculate the total to be 11.
JavaScript variables are containers for storing data values.

JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.
These unique names are called **identifiers**.
Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).
The general rules for constructing names for variables (unique identifiers) are:

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

JavaScript identifiers are case-sensitive.

The Assignment Operator

In JavaScript, the equal sign (=) is an "assignment" operator, not an "equal to" operator.
This is different from algebra. The following does not make sense in algebra:
x = x + 5

In JavaScript, however, it makes perfect sense: it assigns the value of x + 5 to x.
(It calculates the value of x + 5 and puts the result into x. The value of x is incremented by 5.)

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

JavaScript Data Types

JavaScript variables can hold numbers like 100 and text values like "John Doe".
In programming, text values are called text strings.
JavaScript can handle many types of data, but for now, just think of numbers and strings.
Strings are written inside double or single quotes. Numbers are written without quotes.
If you put a number in quotes, it will be treated as a text string.
Example

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

<h2>JavaScript Variables</h2>

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

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

<script>
var pi = 3.14;
var person = "John Doe";
var answer = 'Yes I am!';

document.getElementById("demo").innerHTML =
pi + "<br>" + person + "<br>" + answer;
</script>

</body>
</html>

Declaring (Creating) JavaScript Variables

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

After the declaration, the variable has no value (technically it has the value of undefined).
To assign a value to the variable, use the equal sign:
carName = "Volvo";

You can also assign a value to the variable when you declare it:
var carName = "Volvo";

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

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;    //result is :523
```
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";     //result is:55
```

JavaScript Operators

Assignment The assignment operator (=) assigns a value to a variable.
```
var x = 10;
```

Adding The addition operator (+) adds numbers:
```
var x = 5;
var y = 2;
var z = x + y;
```

Multiplying The multiplication operator (*) multiplies numbers.
```
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
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

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;

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

JavaScript Comparison Operators

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

JavaScript Logical Operators

Operator	**Description**
&&	logical and
||	logical or

!	logical not

JavaScript Type Operators

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

JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.
Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	**Description**	**Example**	**Same as**	**Result**	**Decimal**
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	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.
~00000000000000000000000000000101 will return 11111111111111111111111111111010

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";      //16Volvo
```

Example

```
var x = "Volvo" + 16;     //Volvo16
```

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

Example

```
var x = 16 + 4 + "Volvo";     //20Volvo
```

Example

```
var x = "Volvo" + 16 + 4;     // Volvo164
```

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
```

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 carName1 = "Volvo XC60";   // Using double quotes
var carName2 = 'Volvo XC60';   // Using single quotes
```

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

Example

```
var answer1 = "It's alright";             // Single quote inside double quotes
var answer2 = "He is called 'Johnny'";    // Single quotes inside double quotes
var answer3 = 'He is called "Johnny"';    // Double quotes inside single quotes
```

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

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

Example

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

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
```

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

<h2>JavaScript Booleans</h2>

<p>Booleans can have two values: true or false:</p>

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

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

</body>
</html>
```

Booleans are often used in conditional testing.

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):

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

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

JavaScript Objects

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

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

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
```

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

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

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

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 type 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.
in JavaScript, the data type of null is an object.

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`

objectThe 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

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

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)

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

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

<h2>JavaScript Functions</h2>

<p>Accessing a function without () will return the function definition instead of the function
result:</p>
<p id="demo"></p>

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

</body>
</html>
```

Functions Used as Variable Values

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

Example

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

```
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";
```

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

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.

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:

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.

JavaScript Events

HTML events are "things" that happen to HTML elements.
When JavaScript is used in HTML pages, JavaScript can "react" on these events.

HTML Events

An HTML event can be something the browser does, or something a user does.
Here are some examples of HTML events:

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

Often, when events happen, you may want to do something.
JavaScript lets you execute code when events are detected.
HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.
With single quotes:

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

With double quotes:

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

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

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

Common HTML Events

Here is a list of some common HTML events:

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

What can JavaScript Do?

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

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

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

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

JavaScript Strings

JavaScript strings are used for storing and manipulating text.

A JavaScript string is zero or more characters written inside quotes.
Example
var x = "John Doe";

You can use single or double quotes:

String Length

The length of a string is found with the built-in length property :
Example
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
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.";
```

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

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.

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);

The **lastIndexOf() methods searches backwards**, meaning: if the second parameter is 15, the search starts at position 15, counting from the end, and searches to the beginning of the string.
Example
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("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).

Extracting String Parts
There are 3 methods for extracting a part of a string:

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

The slice() Method
slice() extracts a part of a string and returns the extracted part in a new string.
The method takes 2 parameters: the start position, and the end position (end not included).

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

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.

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

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript String Methods</h2>
<p>Replace "Microsoft" with "W3Schools" in the paragraph below:</p>
<button onclick="myFunction()">Try it</button>
<p id="demo">Please visit Microsoft!</p>
<script>
function myFunction() {
  var str = document.getElementById("demo").innerHTML;
  var txt = str.replace("Microsoft","W3Schools");
  document.getElementById("demo").innerHTML = txt;
}
</script>
</body>
</html>
```

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");
```

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()

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

Example

```
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 makeProperty 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
```

s 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
```

Converting a String to an Array

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

Example

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

If the separator is omitted, the returned array will contain the whole string in index [0].
If the separator is "", the returned array will be an array of single characters:

Example

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

JavaScript Numbers

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

Example

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

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

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

JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.
JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

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

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

Precision

Integers (numbers without a period or exponent notation) are **accurate up to 15 digits.**
Example
```
var x = 999999999999999;   // x will be 999999999999999
var y = 9999999999999999;  // y will be 10000000000000000
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:
Example
```
var x = 0.2 + 0.1;         // x will be 0.30000000000000004
```

To solve the problem above, it helps to multiply and divide:
Example
```
var x = (0.2 * 10 + 0.1 * 10) / 10;       // x will be 0.3
```

Adding Numbers and Strings

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)
```

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;
}
```

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;                    // typeof x returns number
var y = new Number(123);        // typeof y returns object
```

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

When using the == 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

The toString() method returns a number as a string.

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

```
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
```

The parameter is optional. If you don't specify it, JavaScript will not round the number.

The toFixed() Method

toFixed() returns a string, with the number written with a specified number of decimals:
Example

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

toFixed(2) is perfect for working with money.

The toPrecision() Method

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

Example

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

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

Or Global JavaScript Methods

JavaScript global methods can be used on all JavaScript data types.
These are the most relevant methods, when working with numbers:

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

The Number() Method

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

Example

```
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.

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.

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];
```

Note: Array indexes start with 0.
[0] is the first element. [1] is the second element.

Changing an Array Element

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

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

Example

```
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
```

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 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:
```
<script>
var fruits, text, fLen, i;
fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[6] = "Lemon";

fLen = fruits.length;
text = "";
for (i = 0; i < fLen; i++) {
  text += fruits[i] + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
```

And result like this…
Banana
Orange
Apple
Mango
undefined
undefined
Lemon

Associative Arrays   // no use in js
Many programming languages support arrays with named indexes.
Arrays with named indexes are called associative arrays (or hashes).
JavaScript does not support arrays with named indexes.
In JavaScript, arrays always use numbered indexes.
Example
```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;     // person.length will return 3
var y = person[0];         // person[0] will return "John"
```

The Difference Between Arrays and Objects
In JavaScript, arrays use numbered indexes.

In JavaScript, objects use named indexes.
Arrays are a special kind of objects, with numbered indexes.

When to Use Arrays. When to use Objects.

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

Avoid new Array()

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

Use [] instead.
These two different statements both create a new empty array named points:

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

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

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

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

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
```

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();
```
Result:
```
Banana,Orange,Apple,Mango
```

*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(" * ");
```
Result:
```
Banana * Orange * Apple * Mango
```

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

```
<h2>JavaScript Array Methods</h2>

<h2>push()</h2>

<p>The push() method appends a new element to an array.</p>

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

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

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

function myFunction() {
  fruits.push("Kiwi");
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

The push() method returns the new array length:

Example

```
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.
The splice() method returns an array with the deleted items:
Example

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

Using splice() to Remove Elements

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

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

<h2>JavaScript Array Methods</h2>
```

```
<h2>splice()</h2>

<p>The splice() methods can be used to remove array elements.</p>

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

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

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
function myFunction() {
  fruits.splice(0, 1);
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

The first parameter (0) defines the position where new elements should be added (spliced in).
The second parameter (1) defines how many elements should be removed.
The rest of the parameters are omitted. No new elements will be added.

Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays:
Example (Merging Two Arrays)

```
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"):

```
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:

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

All JavaScript objects have a toString() method.

JavaScript Sorting Arrays

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.
If the result is negative a is sorted before b.
If the result is positive b is sorted before a.
If the result is 0 no changes is done with the sort order of the two values.

Example:

The compare function compares all the values in the array, two values at the time (a, b).

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

The function calculates 40 - 100 (a - b), and since the result is negative (-60), the sort function will sort 40 as a value lower than 100.

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

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

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

<script>
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(null, [1, 2, 3]) is equivalent to Math.max(1, 2, 3).

Using Math.min() on an Array

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

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

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

My Min / Max JavaScript Methods

The fastest solution is to use a "home made" method.
This function loops through an array comparing each value with the highest value found:
Example (Find Max)

```
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.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.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.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.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.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.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 someOver18 = 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.indexOf()

The indexOf() method searches an array for an element value and returns its position.
Note: The first item has position 0, the second item has position 1, and so on.
Example
Search an array for the item "Apple":

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

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");
```

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.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

JavaScript Date Objects

Example

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

<h2>JavaScript new Date()</h2>

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

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

</body>
</html>
```

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);

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

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);

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");

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.

Continues…

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 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");
```

JavaScript Short Dates.

Short dates are written with an "MM/DD/YYYY" syntax like this:
Example

```
var d = new Date("03/25/2015");
```

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.

Continues…

JavaScript Set Date Methods

Set Date methods let you set date values (years, months, days, hours, minutes, seconds, milliseconds) for a Date Object.

Set Date Methods

Set Date methods are used for setting a part of a date:

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

Compare Dates

Dates can easily be compared.

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

Example

```
var today, someday, text;
today = new Date();
someday = new Date();
someday.setFullYear(2100, 0, 14);

if (someday > today) {
  text = "Today is before January 14, 2100.";
} else {
  text = "Today is after January 14, 2100.";
}
document.getElementById("demo").innerHTML = text;
```

continues...

JavaScript Math Object

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

Example

```
Math.PI;            // returns 3.141592653589793
```

Math.round()

Math.round(x) returns the value of x rounded to its nearest integer:

Example

```
Math.round(4.7);    // returns 5
Math.round(4.4);    // returns 4
```

Math.pow()

Math.pow(x, y) returns the value of x to the power of y:

Example

```
Math.pow(8, 2);     // returns 64
```

Math.sqrt()

Math.sqrt(x) returns the square root of x:

Example

```
Math.sqrt(64);      // returns 8
```

Math.abs()

Math.abs(x) returns the absolute (positive) value of x:

Example

```
Math.abs(-4.7);     // returns 4.7
```

Math.ceil()

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

Example

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

Math.floor()

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

Example

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

Math.sin()

Math.sin(x) returns the sine (a value between -1 and 1) of the angle x (given in radians).

If you want to use degrees instead of radians, you have to convert degrees to radians:
Angle in radians = Angle in degrees x PI / 180.

Example

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

Math.cos()

Math.cos(x) returns the cosine (a value between -1 and 1) of the angle x (given in radians).

If you want to use degrees instead of radians, you have to convert degrees to radians:
Angle in radians = Angle in degrees x PI / 180.

Example

```
Math.cos(0 * Math.PI / 180);     // returns 1 (the cos of 0 degrees)
```

Math.min() and Math.max()

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

Example

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

Example

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

Math.random()

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

Example

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

Math Properties (Constants)

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

Example

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

Math Constructor

Unlike other global objects, the Math object has no constructor. Methods and properties are static.
All methods and properties (constants) can be used without creating a Math object first.

Math Object Methods

Method	**Description**
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
asin(x)	Returns the arcsine of x, in radians
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
atan2(y, x)	Returns the arctangent of the quotient of its arguments
ceil(x)	Returns the value of x rounded up to its nearest integer
cos(x)	Returns the cosine of x (x is in radians)
exp(x)	Returns the value of E_x
floor(x)	Returns the value of x rounded down to its nearest integer
log(x)	Returns the natural logarithm (base E) of x
max(x, y, z, ..., n)	Returns the number with the highest value
min(x, y, z, ..., n)	Returns the number with the lowest value
pow(x, y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Returns the value of x rounded to its nearest integer
sin(x)	Returns the sine of x (x is in radians)
sqrt(x)	Returns the square root of x

tan(x)	Returns the tangent of an angle

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() * 10) + 1; // returns a random integer **from 1 to 10**

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

The Boolean value of an expression is the basis for all JavaScript comparisons and conditions.

Everything With a "Value" is True except 0.

Everything Without a "Value" is False

example
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
```

JavaScript Comparison and Logical Operators

Comparison and Logical operators are used to test for true or false.

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Given that `x = 5`, the table below explains the comparison operators:

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	true

		x !== 8	true
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

How Can it be Used

Comparison operators can be used in conditional statements to compare values and take action depending on the result:

if (age < 18) text = "Too young";

Logical Operators

Logical operators are used to determine the logic between variables or values.
Given that x = 6 and y = 3, the table below explains the logical operators:

Operator	**Description**	**Example**
&&	and	(x < 10 && y > 1) is true
||	or	(x == 5 || y == 5) is false
!	not	!(x == y) is true

Conditional (Ternary) Operator

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

Syntax

variablename = (condition) ? value1:value2

Example

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**

2 < 12	true
2 < "12"	true
2 < "John"	false
2 > "John"	false
2 == "John"	false
"2" < "12"	false
"2" > "12"	true
"2" == "12"	false

When comparing two strings, "2" will be greater than "12", because (alphabetically) 1 is less than 2.

To secure a proper result, **variables should be converted to the proper type** before comparison:

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

<h2>JavaScript Comparisons</h2>

<p>Input your age and click the button:</p>

<input id="age" value="18" />

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

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

<script>
function myFunction() {
  var age, voteable;
  age = Number(document.getElementById("age").value);
  if (isNaN(age)) {
    voteable = "Input is not a number";
  } else {
    voteable = (age < 18) ? "Too young" : "Old enough";
  }
  document.getElementById("demo").innerHTML = voteable;
}
</script>

</body>
</html>
```

JavaScript if else and else if

Continues...

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 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
}
```

JavaScript Switch Statement

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

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:
Saturday

The break Keyword

When JavaScript reaches a break keyword, it breaks out of the switch block.
This will stop the execution of inside the block.
It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

Note: If you omit the break statement, the next case will be executed even if the evaluation does not match the case.

The default Keyword

The default keyword specifies the code to run if there is no case match:
Example
The getDay() method returns the weekday as a number between 0 and 6.
If today is neither Saturday (6) nor Sunday (0), write a default message:

```
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:
Today is Saturday

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.

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:

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

<h2>JavaScript Loops</h2>

<p>The for/in statement loops through the properties of an object.</p>
```

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

<script>
var txt = "";
var person = {fname:"John", lname:"Doe", age:25};
var x;
for (x in person) {
  txt += person[x] + " ";
}
document.getElementById("demo").innerHTML = txt;
</script>

</body>
</html>
```

Result :John Doe 25

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!

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

<h2>JavaScript break</h2>

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

<script>
var cars = ["BMW", "Volvo", "Saab", "Ford"];
var text = "";

list: {
  text += cars[0] + "<br>";
  text += cars[1] + "<br>";
  break list;
  text += cars[2] + "<br>";
  text += cars[3] + "<br>";
}

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

</body>
</html>
```

Result : BMW
Volvo

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 6 types of objects:

- `Object`
- `Date`
- `Array`
- `String`
- `Number`
- `Boolean`

And 2 data types that cannot contain values:

- `null`
- `undefined`
-

You cannot use typeof to determine if a JavaScript object is an array (or a date).

The constructor Property

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

Example

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

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

Example

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

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()
```

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
```

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"
```

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
```

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
```

JavaScript Bitwise Operators

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Examples

Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
5 | 1	5	0101 | 0001	0101
~ 5	10	~0101	1010
5 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

JavaScript Uses 32 bits Bitwise Operands

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.

Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers.

After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.
The examples above uses 4 bits unsigned binary numbers. Because of this ~ 5 returns 10.

Since JavaScript uses 32 bits signed integers, it will not return 10. It will return -6.
00000000000000000000000000000101 (5)
11111111111111111111111111111010 (~5 = -6)

A signed integer uses the leftmost bit as the minus sign.

JavaScript Regular Expressions

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
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

Regular Expression Patterns

Brackets are used to find a range of characters:

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

Metacharacters are characters with a special meaning:

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

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one *n*
n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*

Using the RegExp Object

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

Using test()
The test() method is a RegExp expression method.
It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":
Example
```
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 as an object.
If no match is found, it returns an empty (null) object.

The following example searches a string for the character "e":
Example 1
/e/.exec("The best things in life are free!");

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">
```

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

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

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

Eval Error
An EvalError indicates an error in the eval() function.
Newer versions of JavaScript do not throw EvalError. Use SyntaxError instead.

Range Error
A RangeError is thrown if you use a number that is outside the range of legal values.

For example: You cannot set the number of significant digits of a number to 500.

Example
```
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

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".
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!

JavaScript only hoists declarations, not initializations.

Declare Your Variables At the Top !
Hoisting is (to many developers) an unknown or overlooked behavior of JavaScript.
If a developer doesn't understand hoisting, programs may contain bugs (errors).
To avoid bugs, always declare all variables at the beginning of every scope.
Since this is how JavaScript interprets the code, it is always a good rule.

JavaScript in strict mode does not allow variables to be used if they are not declared.

JavaScript Use Strict

continues...

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.

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 her
```

Re-declaring Variables

Re-declaring 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
```

Re-declaring 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
```

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 variables defined with the **let keyword do not belong to the window object:**
Example
```
let carName = "Volvo";
// code here can not use window.carName
```

Re-declaring 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
}
```

Variables defined with let are not hoisted to the top.

Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Example

```
// you can NOT use carName here
let carName;
```

JavaScript Const

ES2015 intoduced 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
```

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
```

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
}
```

Variables defined with const are not hoisted to the top.
A const variable cannot be used before it is declared:
Example
```
carName = "Volvo";    // You can NOT use carName here
const carName = "Volvo";
```

JavaScript Debugging

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>
```

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;
```

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.

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 2 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.

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 lowercase 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) { // function code }
```

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

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 thought 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}
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 getElementsByTagName) 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

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

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

Words marked with* are new in ECMAScript 5 and 6.

Removed Reserved Words

The following reserved words has been removed from the ECMAScript 5/6 standard:

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

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

Array	Date	eval	function

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

Java Reserved Words

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

getClass	java	JavaArray	javaClass
JavaObject	JavaPackage		

Other Reserved Words

JavaScript can be used as the programming language in many applications. You should also avoid using the name of HTML and Window objects and properties:

alert	all	anchor	anchors
area	assign	blur	button
checkbox	clearInterval	clearTimeout	clientInformation
close	closed	confirm	constructor
crypto	decodeURI	decodeURIComponent	defaultStatus
document	element	elements	embed
embeds	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.
Examples:

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).

Continues…

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 Syntactic 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.

String.trim()

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

Example

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

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"}');
```

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;
```

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;
```

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;
```

ES5 New Object Methods

ECMAScript 5 added a lot of new Object Methods to JavaScript:

```
// 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)
```

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.

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

ECMAScript 5 allows string literals over multiple lines if escaped with a backslash:

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"}
```

ECMAScript 6 - ECMAScript 2015

What is ECMAScript 6?

ECMAScript 6 is also known as ES6 and ECMAScript 2015.
Some people call it JavaScript 6.
This chapter will introduce some of the new features in ES6.

- JavaScript `let`
- JavaScript `const`
- Exponentiation (`**`)
- 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
```

Exponentiation Operator

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

Example

```
var x = 5;
var z = x ** 2;          // result is 25
```

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

Example

```
var x = 5;
var z = Math.pow(x,2);   // result is 25
```

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

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 are not hoisted. They must be defined before they are used.
Using const is safer than using var, because a function expression is always constant value.
You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always 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 **J**ava**S**cript **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):

```
{
"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>
```

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 is often used to validate numeric input:

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

<h2>JavaScript Can Validate Input</h2>

<p>Please input a number between 1 and 10:</p>

<input id="numb">

<button type="button" onclick="myFunction()">Submit</button>

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

<script>
function myFunction() {
  var x, text;

  // Get the value of the input field with id="numb"
  x = document.getElementById("numb").value;

  // If x is Not a Number or less than one or greater than 10
  if (isNaN(x) || x < 1 || x > 10) {
    text = "Input not valid";
  } else {
    text = "Input OK";
  }
  document.getElementById("demo").innerHTML = text;
}
</script>
```

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

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

These are few to name…

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

These are few to name…

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:

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

<p>Enter a number and click OK:</p>
```

```
<input id="id1" type="number" min="100" max="300" required>
<button onclick="myFunction()">OK</button>

<p>If the number is less than 100 or greater than 300, an error message will be
displayed.</p>

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

<script>
function myFunction() {
  var inpObj = document.getElementById("id1");
  if (!inpObj.checkValidity()) {
    document.getElementById("demo").innerHTML = inpObj.validationMessage;
  } else {
    document.getElementById("demo").innerHTML = "Input OK";
  }
}
</script>

</body>
</html>
```

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>
```

Example
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.

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"
};
```

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

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

<h2>JavaScript Object Properties</h2>

<p>You can add new properties to existing objects.</p>

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

<script>
var person = {
  firstname:"John",
  lastname:"Doe",
  age:50,
  eyecolor:"blue"
};

person.nationality = "English";
document.getElementById("demo").innerHTML =
person.firstname + " is " + person.nationality + ".";
</script>
```

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

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**.

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

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

<h2>JavaScript Getters and Setters</h2>

<p>Perfect for creating a counter object:</p>

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

<script>
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;
```

```
// Display the counter:
document.getElementById("demo").innerHTML = obj.counter;
</script>

</body>
</html>
```

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 (value) {this.counter -= value;}
});

// Play with the counter:
obj.reset;
obj.add = 5;
obj.subtract = 1;
obj.increment;
obj.decrement;
```

JavaScript Object Constructors

Example

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}
```

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:

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

<h2>JavaScript Object Constructors</h2>

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

<script>

// Constructor function for Person objects
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

// Create two Person objects
var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48, "green");

// Display age
document.getElementById("demo").innerHTML =
"My father is " + myFather.age + ". My mother is " + myMother.age + ".";

</script>

</body>
</html>
```

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

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

Example

```
myFather.name = function () {
  return this.firstName + " " + this.lastName;
};
```

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:

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");

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 /()/ 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 = /()/           // new regexp object
var x7 = function(){};  // new function object
```

JavaScript Object Prototypes

All JavaScript objects inherit properties and methods from a prototype.

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:

```
var person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};
// Change a property
Object.defineProperty(person, "language", {value : "NO"});
```

Changing Metadata

ES5 allows the following property metadata 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 be not enumerated
configurable : false // Property can be not 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:

```
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 list only the enumerable properties of an object:

```
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:

```
// 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:

```
//Create an object
var person = {firstName:"John", lastName:"Doe"};

// Define a getter
Object.defineProperty(person, "fullName", {
  get : function () {return this.firstName + " " + this.lastName;}
});
```

JS Functions

JavaScript Function Definitions

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
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;
```

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.

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 are not hoisted. They must be defined before they are used.
Using const is safer than using var, because a function expression is always constant value.
You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:
Example

```
const x = (x, y) => { return x * y };
```

Arrow functions are not supported in IE11 or earlier.

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:

```
function 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".
we will use invoke, because a JavaScript function can be invoked without being called.

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.

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.

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

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:

```
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"
```

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 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"]);
```

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 = 0;
  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 = 0;
  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:

```
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

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

<h2>JavaScript Closures</h2>

<p>Counting with a local variable.</p>

<button type="button" onclick="myFunction();">Count!</button>

<p id="demo">0</p>

<script>
var add = (function () {
  var counter = 0;
  return function () {counter += 1; return counter;}
})();

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

</body>
</html>
```

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.

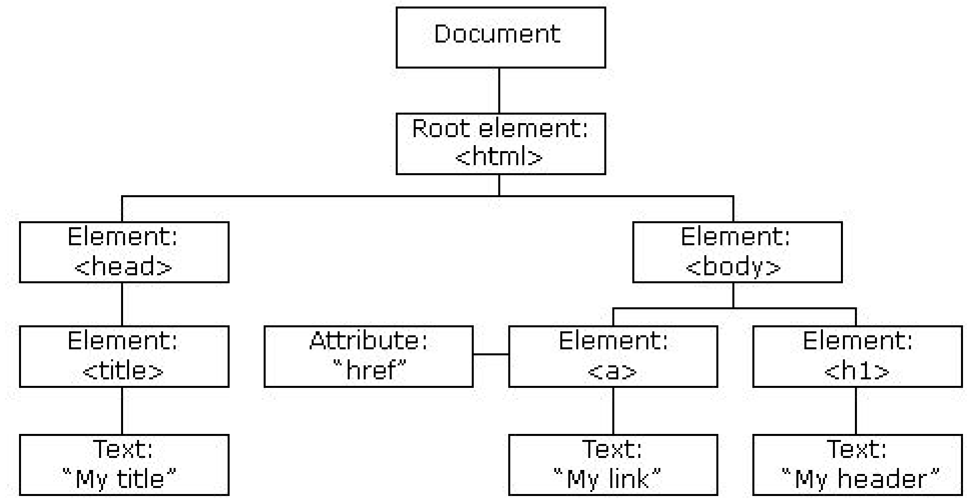
JS HTML DOM

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 **D**ocument **O**bject **M**odel 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).

The following example changes the content (the innerHTML) of the <p> element with id="demo":

```
<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**
document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name

document.getElementsByClassName(*name*)	Find elements by class name

Changing HTML Elements

Property	**Description**
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element

Method	**Description**
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element

Adding and Deleting Elements

Method	**Description**
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*new, old*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

Adding Events Handlers

Method	**Description**
document.getElementById(*id*).onclick = function(){*code*}	Adding event handler code to an onclick event

Finding HTML Objects

The first HTML DOM Level 1 (1998), defined 11 HTML objects, object collections, and properties. These are still valid in HTML5.
Later, in HTML DOM Level 3, more objects, collections, and properties were added.

Property	**Description**	**DOM**
document.anchors	Returns all <a> elements that have a name attribute	1
document.applets	Returns all <applet> elements (Deprecated in HTML5)	1
document.baseURI	Returns the absolute base URI of the document	3
document.body	Returns the <body> element	1
document.cookie	Returns the document's cookie	1
document.doctype	Returns the document's doctype	3
document.documentElement	Returns the <html> element	3
document.documentMode	Returns the mode used by the browser	3
document.documentURI	Returns the URI of the document	3
document.domain	Returns the domain name of the document server	1
document.domConfig	Obsolete. Returns the DOM configuration	3
document.embeds	Returns all <embed> elements	3
document.forms	Returns all <form> elements	1
document.head	Returns the <head> element	3
document.images	Returns all <img> elements	1
document.implementation	Returns the DOM implementation	3
document.inputEncoding	Returns the document's encoding (character set)	3
document.lastModified	Returns the date and time the document was updated	3

document.links	Returns all <area> and <a> elements that have a href attribute	1
document.readyState	Returns the (loading) status of the document	3
document.referrer	Returns the URI of the referrer (the linking document)	1
document.scripts	Returns all <script> elements	3
document.strictErrorChecking	Returns if error checking is enforced	3
document.title	Returns the <title> element	1
document.URL	Returns the complete URL of the document	1

JavaScript HTML DOM Elements

This page teaches you how to find and access HTML elements in an HTML page.

Finding HTML Elements

Often, with JavaScript, you want to manipulate HTML elements.
To do so, you have to find the elements first. There are several 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":
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");

Finding HTML Elements by Class Name

If you want to find all HTML elements with the same class name, use getElementsByClassName().

This example returns a list of all elements with class="intro".
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 match 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".
var x = document.querySelectorAll("p.intro");

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