**JavaScript Training**

JavaScript is an open source & most popular client side scripting language supported by all browsers. JavaScript used mainly for enhancing the interaction of a user with the webpage.

**An Introduction**

[**What is JavaScript?**](https://javascript.info/intro#what-is-javascript)

*JavaScript* was initially created to *“make web pages alive”*.

The programs in this language are called *scripts*. They can be written right in a web page’s HTML and run automatically as the page loads.

Scripts are provided and executed as plain text. They do not need special preparation or compilation to run.

**Code editors**

A code editor is the place where programmers spend most of their time.

There are two main types of code editors: IDEs and lightweight editors. Many people use one tool of each type.

[**IDE**](https://javascript.info/code-editors#ide)

The term [IDE](https://en.wikipedia.org/wiki/Integrated_development_environment) (Integrated Development Environment) refers to a powerful editor with many features that usually operates on a “whole project.” As the name suggests, it’s not just an editor, but a full-scale “development environment.”

An IDE loads the project (which can be many files), allows navigation between files, provides auto completion based on the whole project (not just the open file), and integrates with a version management system (like [git](https://git-scm.com/)), a testing environment, and other “project-level” stuff.

If you have not selected an IDE yet, consider the following options:

[Visual Studio Code](https://code.visualstudio.com/) (cross-platform, free).

[Web Storm](http://www.jetbrains.com/webstorm/) (cross-platform, paid).

For Windows, there is also “Visual Studio”, not to be confused with “Visual Studio Code”. “Visual Studio” is a paid and mighty Windows-only editor, well suited for the .NET platform. It is also good at JavaScript. There is also a free version [Visual Studio Community](https://www.visualstudio.com/vs/community/).

Many IDEs are paid, but have a trial period. Their cost is usually negligible compared to a qualified developer’s salary, so just choose the best one for you.

## 

## [Lightweight editors](https://javascript.info/code-editors#lightweight-editors)

“Lightweight editors” are not as powerful as IDEs, but they’re fast, elegant and simple.

They are mainly used to open and edit a file instantly.

The following options deserve your attention:

[Atom](https://atom.io/) (cross-platform, free).

[Visual Studio Code](https://code.visualstudio.com/) (cross-platform, free).

[Sublime Text](http://www.sublimetext.com/) (cross-platform, shareware).

[Notepad++](https://notepad-plus-plus.org/) (Windows, free).

[Vim](http://www.vim.org/) and [Emacs](https://www.gnu.org/software/emacs/) are also cool if you know how to use them.

# **Developer console**

Code is prone to errors. You will quite likely make errors… Oh, what am I talking about? You are absolutely going to make errors, at least if you are a human, not a robot.

However, in the browser, users do not see errors by default. Therefore, if something goes wrong in the script, we will not see what is broken and cannot fix it.

To see errors and get a lot of other useful information about scripts, “developer tools” have been embedded in browsers.

Most developers lean towards Chrome or Firefox for development because those browsers have the best developer tools. Other browsers also provide developer tools, sometimes with special features, but are usually playing “catch-up” to Chrome or Firefox. So most developers have a “favorite” browser and switch to others if a problem is browser-specific.

## [Google Chrome](https://javascript.info/devtools" \l "google-chrome)

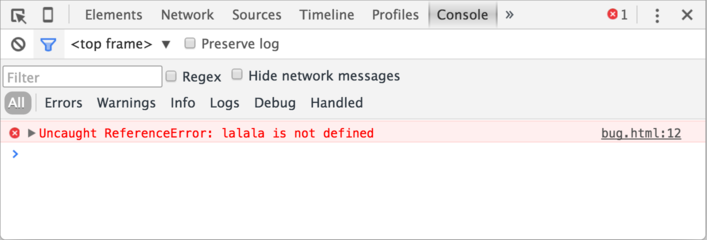
Open the page [bug.html](https://javascript.info/article/devtools/bug.html).

There is an error in the JavaScript code on it. It is hidden from a regular visitor’s eyes, so let’s open developer tools to see it.

Press F12 or, if you are on Mac, then Cmd+Opt+J.

The developer tools will open on the Console tab by default.

It looks somewhat like this:



## [Firefox, Edge, and others](https://javascript.info/devtools" \l "firefox-edge-and-others)

Most other browsers use F12 to open developer tools.

The look & feel of them is quite similar. Once you know how to use one of these tools (you can start with Chrome), you can easily switch to another.

**JavaScript Fundamentals**

**Hello World!**

The “script” tag

JavaScript programs can be inserted into any part of an HTML document with the help of the <script> tag.

[**External scripts**](https://javascript.info/hello-world#external-scripts)

If we have a lot of JavaScript code, we can put it into a separate file.

Script files are attached to HTML with the src attribute:

<Script src="/path/to/script.js"></script>

**Please note:**

As a rule, only the simplest scripts are put into HTML. More complex ones reside in separate files.

The benefit of a separate file is that the browser will download it and store it in its cache.

Other pages that reference the same script will take it from the cache instead of downloading it, so the file is actually downloaded only once.

That reduces traffic and makes pages faster.

**Code structure**

**Statements**

Statements are syntax constructs and commands that perform actions.

We’ve already seen a statement, alert('Hello, world!'), which shows the message “Hello, world!”.

We can have as many statements in our code as we want. Statements can be separated with a semicolon.

For example, here we split “Hello World” into two alerts:

alert('Hello'); alert('World');

Usually, statements are written on separate lines to make the code more readable:

alert('Hello');

alert('World');

**Semicolons :** A semicolon may be omitted in most cases when a line break exists.

This would also work:

alert('Hello')

alert('World')

another example :

alert(3 +

1

+ 2);

**Comments**

Single Line Comments

// This comment occupies a line of its own

alert('Hello');

alert('World'); // This comment follows the statement

Multiline Comments

/\* An example with two messages.

This is a multiline comment.

\*/

alert('Hello');

alert('World');

**Variables**

Variables are used to store this information.

Most of the time, a JavaScript application needs to work with information. Here are two examples:

An online shop – the information might include goods being sold and a shopping cart.

A chat application – the information might include users, messages, and much more.

A variable is a “named storage” for data. We can use variables to store goodies, visitors, and other data.

To create a variable in JavaScript, use the let keyword.

The statement below creates (in other words: declares) a variable with the name “message”:

let message;

Now, we can put some data into it by using the assignment operator =:

let message;

message = 'Hello'; // store the string

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

let message;

message = 'Hello!';

alert(message); // shows the variable content

**Variable Naming Convention**

There are two limitations on variable names in JavaScript:

* The name must contain only letters, digits, or the symbols $ and \_.
* The first character must not be a digit.

Examples of valid names:

let userName;

let test123;

Examples of incorrect variable names:

let 1a; // cannot start with a digit

let my-name; // hyphens '-' aren't allowed in the name

**Constants**

To declare a constant (unchanging) variable, use const instead of let:

const myBirthday = '18.04.1982';

Variables declared using const are called “constants”. They cannot be reassigned. An attempt to do so would cause an error:

const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, can't reassign the constant!

When a programmer is sure that a variable will never change, they can declare it with const to guarantee and clearly communicate that fact to everyone.

**Data Types**

A variable in JavaScript can contain any data. A variable can at one moment be a string and at another be a number:

// no error

let message = "hello";

message = 123456;

Programming languages that allow such things are called “dynamically typed”, meaning that there are data types, but variables are not bound to any of them.

**Number**

let n = 123;

n = 12.345;

The *number* type represents both integer and floating point numbers.

There are many operations for numbers, e.g. multiplication \*, division /, addition +, subtraction -, and so on.

**String**

A string in JavaScript must be surrounded by quotes.

let str = "Hello";

let str2 = 'Single quotes are ok too';

let phrase = `can embed another ${str}`;

In JavaScript, there are 3 types of quotes.

* Double quotes: "Hello".
* Single quotes: 'Hello'.
* Backticks: `Hello`.

**Boolean**

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.

For instance:

let nameFieldChecked = true; // yes, name field is checked

let ageFieldChecked = false; // no, age field is not checked

Boolean values also come as a result of comparisons:

let isGreater = 4 > 1;

alert( isGreater ); // true (the comparison result is "yes")

**The Null Value**

The special null value does not belong to any of the types described above.

It forms a separate type of its own which contains only the null value:

let age = null;

In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

It’s just a special value which represents “nothing”, “empty” or “value unknown”.

The code above states that age is unknown or empty for some reason.

**The Undefined Value**

The special value undefined also stands apart. It makes a type of its own, just like null.

The meaning of undefined is “value is not assigned”.

If a variable is declared, but not assigned, then its value is undefined:

let x;

alert(x); // shows "undefined"

Technically, it is possible to assign undefined to any variable:

let x = 123;

x = undefined;

alert(x); // "undefined"

**The typeof Operator**

The typeof operator returns the type of the argument. It’s useful when we want to process values of different types differently or just want to do a quick check.

It supports two forms of syntax:

1. As an operator: typeof x.
2. As a function: typeof(x).

In other words, it works with parentheses or without them. The result is the same.

The call to typeof x returns a string with the type name:

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

There are 8 basic data types in JavaScript.

* number for numbers of any kind: integer or floating-point, integers are limited by ±253.
* bigint is for integer numbers of arbitrary length.
* string for strings. A string may have one or more characters, there’s no separate single-character type.
* boolean for true/false.
* null for unknown values – a standalone type that has a single value null.
* undefined for unassigned values – a standalone type that has a single value undefined.
* object for more complex data structures.
* symbol for unique identifiers.
* The typeof operator allows us to see which type is stored in a variable.
* Two forms: typeof x or typeof(x).
* Returns a string with the name of the type, like "string".
* For null returns "object" – this is an error in the language, it’s not actually an object.

**Type Conversions**

Most of the time, operators and functions automatically convert the values given to them to the right type.

For example, alert automatically converts any value to a string to show it. Mathematical operations convert values to numbers.

There are also cases when we need to explicitly convert a value to the expected type.

**String Conversion**

String conversion happens when we need the string form of a value.

For example, alert(value) does it to show the value.

We can also call the String(value) function to convert a value to a string:

let value = true;

alert(typeof value); // boolean

value = String(value); // now value is a string "true"

alert(typeof value); // string

**Numeric Conversion**

Numeric conversion happens in mathematical functions and expressions automatically.

For example, when division / is applied to non-numbers:

alert( "6" / "2" ); // 3, strings are converted to numbers

We can use the Number(value) function to explicitly convert a value to a number:

let str = "123";

alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number

**Boolean Conversion**

Boolean conversion is the simplest one.

It happens in logical operations (later we’ll meet condition tests and other similar things) but can also be performed explicitly with a call to Boolean(value).

The conversion rule:

* Values that are intuitively “empty”, like 0, an empty string, null, undefined, and NaN, become false.
* Other values become true.

For instance:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true

alert( Boolean("") ); // false

**Operators**

We know many operators from school. They are things like addition +, multiplication \*, subtraction -, and so on.

In this chapter, we’ll concentrate on aspects of operators that are not covered by school arithmetic.

**Terms : “unary”, “binary”, “operand”**

Before we move on, let’s grasp some common terminology.

* *An operand* – is what operators are applied to. For instance, in the multiplication of 5 \* 2 there are two operands: the left operand is 5 and the right operand is 2. Sometimes, people call these “arguments” instead of “operands”.
* An operator is *unary* if it has a single operand. For example, the unary negation - reverses the sign of a number:

let x = 1;

x = -x;

alert( x ); // -1, unary negation was applied

* An operator is *binary* if it has two operands. The same minus exists in binary form as well:

let x = 1, y = 3;

alert( y - x ); // 2, binary minus subtracts values

Formally, in the examples above we have two different operators that share the same symbol: the negation operator, a unary operator that reverses the sign, and the subtraction operator, a binary operator that subtracts one number from another.

**String concatenation, binary +**

Now, let’s see special features of JavaScript operators that are beyond school arithmetics.

Usually, the plus operator + sums numbers.

But, if the binary + is applied to strings, it merges (concatenates) them:

let s = "my" + "string";

alert(s); // mystring

Note that if one of the operands is a string, the other one is converted to a string too.

For example:

alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"

**Assignment**

Let’s note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 3.

That’s why, when we assign a variable, like x = 2 \* 2 + 1, the calculations are done first and then the = is evaluated, storing the result in x.

let x = 2 \* 2 + 1;

alert( x ); // 5

**Remainder**

The remainder operator %, despite its appearance, is not related to percents.

The result of a % b is the remainder of the integer division of a by b.

For instance:

alert( 5 % 2 ); // 1 is a remainder of 5 divided by 2

alert( 8 % 3 ); // 2 is a remainder of 8 divided by 3

alert( 6 % 3 ); // 0 is a remainder of 6 divided by 3

**Increment/Decrement**

Increasing or decreasing a number by one is among the most common numerical operations.

So, there are special operators for it:

**Increment** ++ increases a variable by 1:

let counter = 2;

counter++; // works the same as counter = counter + 1, but is shorter

alert( counter ); // 3

**Decrement** -- decreases a variable by 1:

let counter = 2;

counter--; // works the same as counter = counter - 1, but is shorter

alert( counter ); // 1

**Comparisons**

We know many comparison operators from maths:

* Greater/less than: a > b, a < b.
* Greater/less than or equals: a >= b, a <= b.
* Equals: a == b (please note the double equals sign =. A single symbol a = b would mean an assignment).
* Not equals. In maths the notation is ≠, but in JavaScript it’s written as an assignment with an exclamation sign before it: a != b.

**Boolean is the result**

Like all other operators, a comparison returns a value. In this case, the value is a boolean.

true – means “yes”, “correct” or “the truth”.

false – means “no”, “wrong” or “not the truth”.

For example:

alert( 2 > 1 ); // true (correct)

alert( 2 == 1 ); // false (wrong)

alert( 2 != 1 ); // true (correct)

A comparison result can be assigned to a variable, just like any value:

let result = 5 > 4; // assign the result of the comparison

alert( result ); // true

**String Comparison**

To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order.

In other words, strings are compared letter-by-letter.

For example:

alert( 'Z' > 'A' ); // true

alert( 'Glow' > 'Glee' ); // true

alert( 'Bee' > 'Be' ); // true

**Interaction : alert, prompt, confirm**

In this part of the tutorial we cover JavaScript language “as is”, without environment-specific tweaks.

But we’ll still be using the browser as our demo environment, so we should know at least a few of its user-interface functions. In this chapter, we’ll get familiar with the browser functions alert, prompt and confirm.

**Alert**

Syntax:

alert(message);

This shows a message and pauses script execution until the user presses “OK”.

For example:

alert("Hello");

**Prompt**

The function prompt accepts two arguments:

result = prompt(title, [default]);

It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.

**title**

The text to show the visitor.

**default**

An optional second parameter, the initial value for the input field.

The visitor may type something in the prompt input field and press OK. Or they can cancel the input by pressing Cancel or hitting the Esc key.

The call to prompt returns the text from the input field or null if the input was canceled.

For instance:

let age = prompt('How old are you?', 100);

alert(`You are ${age} years old!`); // You are 100 years old!

**Confirm**

The syntax:

result = confirm(question);

The function confirm shows a modal window with a question and two buttons: OK and Cancel.

The result is true if OK is pressed and false otherwise.

For example:

let isBoss = confirm("Are you the boss?");

alert( isBoss ); // true if OK is pressed

**Conditional operator: if, ?**

Sometimes, we need to perform different actions based on different conditions.

To do that, we can use the if statement and the conditional operator ?, that’s also called a “question mark” operator.

**The “If” Statement**

The if(...) statement evaluates a condition in parentheses and, if the result is true, executes a block of code.

For example:

let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year == 2015) alert( 'You are right!' );

In the example above, the condition is a simple equality check (year == 2015), but it can be much more complex.

**Boolean conversion**

The if (…) statement evaluates the expression in its parentheses and converts the result to a boolean.

Let’s recall the conversion rules from the chapter [Type Conversions](https://javascript.info/type-conversions):

* A number 0, an empty string "", null, undefined, and NaN all become false. Because of that they are called “falsy” values.
* Other values become true, so they are called “truthy”.

We can also pass a pre-evaluated boolean value to if, like this:

let cond = (year == 2015); // equality evaluates to true or false

if (cond) {

...

}

**The “else” clause**

The if statement may contain an optional “else” block. It executes when the condition is false.

For example:

let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year == 2015) {

alert( 'You guessed it right!' );

} else {

alert( 'How can you be so wrong?' ); // any value except 2015

}

**Several conditions: “else if”**

Sometimes, we’d like to test several variants of a condition. The else if clause lets us do that.

For example:

let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year < 2015) {

alert( 'Too early...' );

} else if (year > 2015) {

alert( 'Too late' );

} else {

alert( 'Exactly!' );

}

In the code above, JavaScript first checks year < 2015. If that is falsy, it goes to the next condition year > 2015. If that is also falsy, it shows the last alert.

There can be more else if blocks. The final else is optional.

**Conditional operator ‘?’**

Sometimes, we need to assign a variable depending on a condition.

For instance:

let accessAllowed;

let age = prompt('How old are you?', '');

if (age > 18) {

accessAllowed = true;

} else {

accessAllowed = false;

}

alert(accessAllowed);

The so-called “conditional” or “question mark” operator lets us do that in a shorter and simpler way.

The operator is represented by a question mark ?. Sometimes it’s called “ternary”, because the operator has three operands. It is actually the one and only operator in JavaScript which has that many.

The syntax is:

let result = condition ? value1 : value2;

The condition is evaluated: if it’s truthy then value1 is returned, otherwise – value2.

For example:

let accessAllowed = (age > 18) ? true : false;

**Logical Operators**

There are three logical operators in JavaScript: || (OR), && (AND), ! (NOT).

**|| ( OR )**

The “OR” operator is represented with two vertical line symbols:

result = a || b;

There are four possible logical combinations:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

For example:

let hour = 9;

if (hour < 10 || hour > 18) {

alert( 'The office is closed.' );

}

We can pass more conditions:

let hour = 12;

let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {

alert( 'The office is closed.' ); // it is the weekend

}

**&& ( AND )**

The AND operator is represented with two ampersands &&:

result = a && b;

In classical programming, AND returns true if both operands are truthy and false otherwise:

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

An example with if:

let hour = 12;

let minute = 30;

if (hour == 12 && minute == 30) {

alert( 'The time is 12:30' ); }

**! ( NOT )**

The boolean NOT operator is represented with an exclamation sign !.

The syntax is pretty simple:

result = !value;

The operator accepts a single argument and does the following:

* Converts the operand to boolean type: true/false.
* Returns the inverse value.

For instance:

alert( !true ); // false

alert( !0 ); // true

**Loops: while and for**

We often need to repeat actions.

For example, outputting goods from a list one after another or just running the same code for each number from 1 to 10.

Loops are a way to repeat the same code multiple times.

**The “while” loop**

The while loop has the following syntax:

while (condition) {

// code

// so-called "loop body"

}

While the condition is truthy, the code from the loop body is executed.

For instance, the loop below outputs i while i < 3:

let i = 0;

while (i < 3) { // shows 0, then 1, then 2

alert( i );

i++;

}

**The “do…while” loop**

The condition check can be moved below the loop body using the do..while syntax:

do {

// loop body

} while (condition);

The loop will first execute the body, then check the condition, and, while it’s truthy, execute it again and again.

For example:

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

**The “for” loop**

The for loop is more complex, but it’s also the most commonly used loop.

It looks like this:

for (begin; condition; step) {

// ... loop body ...

}

Let’s learn the meaning of these parts by example. The loop below runs alert(i) for i from 0 up to (but not including) 3:

for (let i = 0; i < 3; i++) { // shows 0, then 1, then 2

alert(i);

}

Let’s examine the for statement part-by-part:

| **part** |  |  |
| --- | --- | --- |
| begin | i = 0 | Executes once upon entering the loop. |
| condition | i < 3 | Checked before every loop iteration. If false, the loop stops. |
| body | alert(i) | Runs again and again while the condition is truthy. |
| step | i++ | Executes after the body on each iteration. |

**The “switch” statement**

A switch statement can replace multiple if checks.

It gives a more descriptive way to compare a value with multiple variants.

## [The syntax](https://javascript.info/switch" \l "the-syntax)

The switch has one or more case blocks and an optional default.

It looks like this:

switch(x) {

case 'value1': // if (x === 'value1')

...

[break]

case 'value2': // if (x === 'value2')

...

[break]

default:

...

[break]

}

* The value of x is checked for a strict equality to the value from the first case (that is, value1) then to the second (value2) and so on.
* If the equality is found, switch starts to execute the code starting from the corresponding case, until the nearest break (or until the end of switch).
* If no case is matched then the default code is executed (if it exists).

An example of switch (the executed code is highlighted):

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Too small' );

break;

case 4:

alert( 'Exactly!' );

break;

case 5:

alert( 'Too large' );

break;

default:

alert( "I don't know such values" ); }

**Functions**

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

We’ve already seen examples of built-in functions, like alert(message), prompt(message, default) and confirm(question). But we can create functions of our own as well.

**Function Declaration**

To create a function we can use a *function declaration*.

It looks like this:

function showMessage() {

alert( 'Hello everyone!' );

}

The function keyword goes first, then goes the *name of the function*, then a list of *parameters* between the parentheses (comma-separated, empty in the example above) and finally the code of the function, also named “the function body”, between curly braces.

function name(parameters) {

...body...

}

Our new function can be called by its name: showMessage().

For instance:

function showMessage() {

alert( 'Hello everyone!' );

}

showMessage();

showMessage();

The call showMessage() executes the code of the function. Here we will see the message two times.

This example clearly demonstrates one of the main purposes of functions: to avoid code duplication.

If we ever need to change the message or the way it is shown, it’s enough to modify the code in one place: the function which outputs it.

**Local variables**

A variable declared inside a function is only visible inside that function.

For example:

function showMessage() {

let message = "Hello, I'm JavaScript!"; // local variable

alert( message );

}

showMessage(); // Hello, I'm JavaScript!

alert( message ); // <-- Error! The variable is local to the function

**Outer variables**

A function can access an outer variable as well, for example:

let userName = 'John';

function showMessage() {

let message = 'Hello, ' + userName;

alert(message);

}

showMessage(); // Hello, John

The function has full access to the outer variable. It can modify it as well.

For instance:

let userName = 'John';

function showMessage() {

userName = "Bob"; // (1) changed the outer variable

let message = 'Hello, ' + userName;

alert(message);

}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function

The outer variable is only used if there’s no local one.

If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local userName. The outer one is ignored:

let userName = 'John';

function showMessage() {

let userName = "Bob"; // declare a local variable

let message = 'Hello, ' + userName; // Bob

alert(message);

}

// the function will create and use its own userName

showMessage();

alert( userName ); // John, unchanged, the function did not access the outer variable

**Parameters**

We can pass arbitrary data to functions using parameters (also called *function arguments*) .

In the example below, the function has two parameters: from and text.

function showMessage(from, text) { // arguments: from, text

alert(from + ': ' + text);

}

showMessage('Ann', 'Hello!'); // Ann: Hello! (\*)

showMessage('Ann', "What's up?"); // Ann: What's up? (\*\*)

When the function is called in lines (\*) and (\*\*), the given values are copied to local variables from and text. Then the function uses them.

Here’s one more example: we have a variable from and pass it to the function. Please note: the function changes from, but the change is not seen outside, because a function always gets a copy of the value:

function showMessage(from, text) {

from = '\*' + from + '\*'; // make "from" look nicer

alert( from + ': ' + text );

}

let from = "Ann";

showMessage(from, "Hello"); // \*Ann\*: Hello

// the value of "from" is the same, the function modified a local copy

alert( from ); // Ann

**Returning a value**

A function can return a value back into the calling code as the result.

The simplest example would be a function that sums two values:

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

alert( result ); // 3

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

There may be many occurrences of return in a single function. For instance:

function checkAge(age) {

if (age >= 18) {

return true;

} else {

return confirm('Do you have permission from your parents?');

}

}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {

alert( 'Access granted' );

} else {

alert( 'Access denied' );

}

**Naming a function**

Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.

It is a widespread practice to start a function with a verbal prefix which vaguely describes the action. There must be an agreement within the team on the meaning of the prefixes.

For instance, functions that start with "show" usually show something.

Function starting with…

* "get…" – return a value,
* "calc…" – calculate something,
* "create…" – create something,
* "check…" – check something and return a boolean, etc.

Examples of such names:

showMessage(..) // shows a message

getAge(..) // returns the age (gets it somehow)

calcSum(..) // calculates a sum and returns the result

createForm(..) // creates a form (and usually returns it)

checkPermission(..) // checks a permission, returns true/false