**[Parameters](https://javascript.info/function-basics" \l "parameters)**

We can pass arbitrary data to function 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 next. 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

**[Default values](https://javascript.info/function-basics" \l "default-values)**

If a parameter is not provided, then its value becomes undefined.

For instance, the aforementioned function showMessage(from, text) can be called with a single argument:

showMessage("Ann");

That’s not an error. Such call would output "Ann: undefined". There’s no text, so it’s assumed that text === undefined.

If we want to use a “default” text in this case, then we can specify it after =:

function showMessage(from, text = "no text given") {

alert( from + ": " + text );

}

showMessage("Ann"); // Ann: no text given

Now if the text parameter is not passed, it will get the value "no text given"

[**Returning a value**](https://javascript.info/function-basics#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 occurences of return in a single function. For instance:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Got a permission from the parents?');

}

}

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

if ( checkAge(age) ) {

alert( 'Access granted' );

} else {

alert( 'Access denied' );

}

It is possible to use return without a value. That causes the function to exit immediately.

For example:

function showMovie(age) {

if ( !checkAge(age) ) {

return;

}

alert( "Showing you the movie" ); // (\*)

// ...

}

In the code above, if checkAge(age) returns false, then showMovie won’t proceed to the alert.

**Never add a newline between return and the value**

For a long expression in return, it might be tempting to put it on a separate line, like this:

return

(some + long + expression + or + whatever \* f(a) + f(b))

That doesn’t work, because JavaScript assumes a semicolon after return. That’ll work the same as:

return;

(some + long + expression + or + whatever \* f(a) + f(b))

So, it effectively becomes an empty return. We should put the value on the same line instead.

**[Naming a function](https://javascript.info/function-basics" \l "function-naming)**

Functions are actions. So their name is usually a verb. It should briefly, but as accurately as possible describe what the function does. So that a person who reads the code gets the right clue.

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

With prefixes at place, a glance at a function name gives an understanding what kind of work it does and what kind of value it returns.

**One function – one action**

A function should do exactly what is suggested by its name, no more.

Two independant actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).

Few examples of breaking this rule:

* getAge – would be bad if it shows an alert with the age (should only get).
* createForm – would be bad if it modifies the document, adding a form to it (should only create it and return).
* checkPermission – would be bad if displays the access granted/denied message (should only perform the check and return the result).

These examples assume common meanings of prefixes. What they mean for you is determined by you and your team. Maybe it’s pretty normal for your code to behave differently. But you should have a firm understanding of what a prefix means, what a prefixed function can and what it cannot do. All same-prefixed functions should obey the rules. And the team should share the knowledge.

**[Functions == Comments](https://javascript.info/function-basics" \l "functions-comments)**

Functions should be short and do exactly one thing. If that thing is big, maybe it’s worth to split the function into few smaller functions. Sometimes following this rule may not be that easy, but it’s a definitely good thing.

A separate function is not only easier to test and debug – its very existence is a great comment!

For instance, compare the two functions showPrimes(n) below. Each one outputs [prime numbers](https://en.wikipedia.org/wiki/Prime_number) up to n.

The first variant uses a label:

function showPrimes(n) {

nextPrime: for (let i = 2; i < n; i++) {

for (let j = 2; j < i; j++) {

if (i % j == 0) continue nextPrime;

}

alert( i ); // a prime

}

}

The second variant uses an additional function isPrime(n) to test for primality:

function showPrimes(n) {

for (let i = 2; i < n; i++) {

if (!isPrime(i)) continue;

alert(i); // a prime

}

}

function isPrime(n) {

for (let i = 2; i < n; i++) {

if ( n % i == 0) return false;

}

return true;

}

The second variant is easier to understand isn’t it? Instead of the code piece we see a name of the action (isPrime). Sometimes people refer to such code as *self-describing*.

So, functions can be created even if we don’t intend to reuse them. They structure the code and make it readable.

**[Summary](https://javascript.info/function-basics" \l "summary)**

A function declaration looks like this:

function name(parameters, delimited, by, comma) {

/\* code \*/

}

* Values passed to function as parameters are copied to its local variables.
* A function may access outer variables. But it works only from inside out. The code outside of the function doesn’t see its local variables.
* A function can return a value. If it doesn’t then its result is undefined.

To make the code clean and easy to understand, it’s recommended to use mainly local variables and parameters in the function, not outer variables.

It is always easier to understand a function which gets parameters, works with them and returns a result than a function which gets no parameters, but modifies outer variables as a side-effect.

Function naming:

* A name should clearly describe what the function does. When we see a function call in the code, a good name instantly gives us an understanding what it does and returns.
* A function is an action, so function names are usually verbal.
* There exist many well-known function prefixes like create…, show…, get…, check… and so on. Use them to hint what a function does.

Functions are the main building blocks of scripts. Now we covered the basics, so we actually can start creating and using them. But that’s only the beginning of the path. We are going to return to them many times, going more deeply in their advanced features.

**Questions and Exercises**

1. The following function returns true if the parameter age is greater than 18.

Otherwise it asks for a confirmation and returns its result:

function checkAge(age) {

if (age > 18) {

return true;

} else {

// ...

return confirm('Did parents allow you?');

}

}

Will the function work differently if else is removed?

function checkAge(age) {

if (age > 18) {

return true;

}

// ...

return confirm('Did parents allow you?');

}

Is there any difference in the behavior of these two variants?

Yes, the return statement will always be true no mater if it meets the condition of age>18

1. The following function returns true if the parameter age is greater than 18.

Otherwise it asks for a confirmation and returns its result.

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Do you have your parents permission to access this page?');

}

}

Rewrite it, to perform the same, but without if, in a single line.

Make two variants of checkAge:

1. Using a question mark operator '?'

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Do you have your parents permission to access this page?');

}

}

1. Using OR ||
2. Write a function min(a,b) which returns the least of two numbers a and b.

For instance:

min(2, 5) == 2

min(3, -1) == -1

min(1, 1) == 1

1. Write a function pow(x,n) that returns x in power n. Or, in other words, multiplies x by itself n times and returns the result.

pow(3, 2) = 3 \* 3 = 9

pow(3, 3) = 3 \* 3 \* 3 = 27

pow(1, 100) = 1 \* 1 \* ...\*1 = 1

Create a web-page that prompts for x and n, and then shows the result of pow(x,n).

1. Create the following list of functions (make a program to test each of them out):

**Adder(5,4)🡪**this procedure takes two *integers*, adds them together and displays the result in an alert box

**AreaOfCircle(10)🡪**takes one *single* number that represents the radius of a circle and displays the circle’s total area in a label (add a label to your form for use with this procedure)

**AreaOfRectangle(8,9)🡪**takes two *single* numbers that represent the length and width of a rectangle and displays its area in a textbox (add a textbox to your form for use with this procedure)

**AreaOfSquare(10)🡪**takes a single *integer* number that represents the length of one side of a square and calculates its area and displays the result in the title bar of the form

**VolumeOfBlock(4,5,7)🡪**takes three *single* numbers that represents the length, width and height of a 3D block, calculates its volume and displays the answer in a messagebox

**FinalCost(4.59)🡪**takes the cost of an item as a *single* number, adds the GST and PST to it and then indicates via a messagebox if the final cost is over or under a $100.00

**Average(55,99,76,98)🡪**takes 4 *integers* that represent the marks of four different classes, averages them and displays in a label the corresponding letter mark

**AreaOfTriangle(4.5,8,”cm”)🡪**takes two *single* numbers representing the base and height of a triangle, calculates it area and displays the final answer along with the correct unit in a messagebox (the last parameter, a string, is used to identify the unit of measurement)