Functions

Quite often we need to perform a similar action in many places of the script.

For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.

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, we’ll see examples later) and finally the code of the function, also named “the function body”, between curly braces.

```
function name(parameter1, parameter2, ... parameterN) {
 // 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
```
Global variables

Variables declared outside of any function, such as the outer `userName` in the code above, are called *global*.

Global variables are visible from any function (unless shadowed by locals).

It's a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Parameters

We can pass arbitrary data to functions using parameters.

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

```
function showMessage(from, text) { // parameters: 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
```
When a value is passed as a function parameter, it's also called an *argument*.

In other words, to put these terms straight:

- A parameter is the variable listed inside the parentheses in the function declaration (it's a declaration time term).
- An argument is the value that is passed to the function when it is called (it's a call time term).

We declare functions listing their parameters, then call them passing arguments.

In the example above, one might say: "the function `showMessage` is declared with two parameters, then called with two arguments: `from` and `"Hello"`".

Default values

If a function is called, but an argument is not provided, then the corresponding 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 a call would output `"*Ann*: undefined"`. As the value for `text` isn’t passed, it becomes `undefined`.

We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using `=`:

```
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"`.

The default value also jumps in if the parameter exists, but strictly equals `undefined`, like this:

```
showMessage("Ann", undefined); // Ann: no text given
```

Here `"no text given"` is a string, but it can be a more complex expression, which is only evaluated and assigned if the parameter is missing. So, this is also possible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() only executed if no text given
  // its result becomes the value of text
}
```

Evaluation of default parameters

In JavaScript, a default parameter is evaluated every time the function is called without the respective parameter.

In the example above, `anotherFunction()` isn’t called at all, if the `text` parameter is provided.

On the other hand, it’s independently called every time when `text` is missing.

Default parameters in old JavaScript code

Several years ago, JavaScript didn’t support the syntax for default parameters. So people used other ways to specify them.

Nowadays, we can come across them in old scripts.

For example, an explicit check for `undefined`:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'no text given';
  }

  alert( from + ": " + text );
}
```

...Or using the || operator:

```
function showMessage(from, text) {
  // If the value of text is falsy, assign the default value
  // this assumes that text == "" is the same as no text at all
  text = text || 'no text given';
  ...
}
```

Alternative default parameters

Sometimes it makes sense to assign default values for parameters at a later stage after the function declaration.

We can check if the parameter is passed during the function execution, by comparing it with `undefined`:

```
function showMessage(text) {
  // ...

  if (text === undefined) { // if the parameter is missing
    text = 'empty message';
  }

  alert(text);
}

showMessage(); // empty message
```

...Or we could use the || operator:

```
function showMessage(text) {
  // if text is undefined or otherwise falsy, set it to 'empty'
  text = text || 'empty';
  ...
}
```

Modern JavaScript engines support the nullish coalescing operator `??`, it’s better when most falsy values, such as `0`, should be considered “normal”:

```
function showCount(count) {
  // if count is undefined or null, show "unknown"
  alert(count ?? "unknown");
}

showCount(0); // 0
showCount(null); // unknown
showCount(); // unknown
```

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' );
}
```

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`.

A function with an empty `return` or without it returns `undefined`

If a function does not return a value, it is the same as if it returns `undefined`:

```
function doNothing() { /* empty */ }

alert( doNothing() === undefined ); // true
```

An empty `return` is also the same as `return undefined`:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

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.

If we want the returned expression to wrap across multiple lines, we should start it at the same line as `return`. Or at least put the opening parentheses there as follows:

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

And it will work just as we expect it to.

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
```

With prefixes in 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 independent 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).

A 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 it displays the `access granted/denied` message (should only perform the check and return the result).

These examples assume common meanings of prefixes. You and your team are free to agree on other meanings, but usually they're not much different. In any case, you should have a firm understanding of what a prefix means, what a prefixed function can and cannot do. All same-prefixed functions should obey the rules. And the team should share the knowledge.

Ultrashort function names
Functions that are used *very often* sometimes have ultrashort names.

For example, the jQuery framework defines a function with `$`. The Lodash library has its core function named _.

These are exceptions. Generally function names should be concise and descriptive.

Functions == Comments

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

A function declaration looks like this:

```
function name(parameters, delimited, by, comma) {
  /* code */
}
```

- Values passed to a 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've 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 into their advanced features.