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 (empty in the example above) and finally the code of the function, also named “the function body”, between curly braces.

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. So an occasional modification may happen if we forget `let`.

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).

Usually, a function declares all variables specific to its task. Global variables only store project-level data, and it's important that these variables are accessible from anywhere. Modern code has few or no globals. Most variables reside in their functions.

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
```

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 a 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"

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() is called every time showMessage() is called without the text parameter. This is in contrast to some other languages like Python, where any default parameters are evaluated only once during the initial interpretation.