Lesson - 16

Functions
Function Expressions. Arrow functions

SkillUp by Dobrea Vladislav

Lesson Plan

- Functions
- Functions Expressions
- Arrow function

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!' );
}
```

Function Declaration

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 ...
}
```

Function Declaration

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.

Outer variables

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.

Outer variables

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 (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.

Parameters

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`.

Default values

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
```

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
}
```

Default Values

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.

Default parameters old-style

Old editions of JavaScript did not support default parameters. So there are alternative ways to support them, that you can find mostly in the old scripts.

For instance, an explicit check for being `undefined`:

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

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

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).

Returning a value

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' );
}
```

Returning a value

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

```
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`.

Returning a value

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`:

```
1 function doNothing() { /* empty */ }
2
3 alert( doNothing() === undefined ); // true
```

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

```
1 function doNothing() {
2   return;
3 }
4
5 alert( doNothing() === undefined ); // true
```

Returning a value

⚠️ **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.

Naming function

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 functions 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
  }
}
```

Functions == comments

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;
}
```

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.

Function expressions

In JavaScript, a function is not a “magical language structure”, but a special kind of value.

The syntax that we used before is called a *Function Declaration*:

```
function sayHi() {
  alert( "Hello" );
}
```

There is another syntax for creating a function that is called a *Function Expression*.

It looks like this:

```
let sayHi = function() {
  alert( "Hello" );
};
```

Here, the function is created and assigned to the variable explicitly, like any other value. No matter how the function is defined, it’s just a value stored in the variable `sayHi`.

The meaning of these code samples is the same: "create a function and put it into the variable `sayHi`".

Function expressions

We can even print out that value using `alert`:

```
1  function sayHi() {
2    alert( "Hello" );
3  }
4
5  alert( sayHi ); // shows the function code
```

Please note that the last line does not run the function, because there are no parentheses after `sayHi`. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.

In JavaScript, a function is a value, so we can deal with it as a value. The code above shows its string representation, which is the source code.

Surely, a function is a special value, in the sense that we can call it like `sayHi()`.

But it's still a value. So we can work with it like with other kinds of values.

Function expressions

We can copy a function to another variable:

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    //     this still works too (why wouldn't it)
```

Here's what happens above in detail:

1. The Function Declaration `(1)` creates the function and puts it into the variable named `sayHi`.
2. Line `(2)` copies it into the variable `func`. Please note again: there are no parentheses after `sayHi`. If there were, then `func = sayHi()` would write *the result of the call* `sayHi()` into `func`, not *the function* `sayHi` itself.
3. Now the function can be called as both `sayHi()` and `func()`.

Function expressions

Why is there a semicolon at the end?

You might wonder, why does Function Expression have a semicolon `;` at the end, but Function Declaration does not:

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

The answer is simple:

- There's no need for `;` at the end of code blocks and syntax structures that use them like `if { ... }`, `for { }`, `function f { }` etc.
- A Function Expression is used inside the statement: `let sayHi = ...;`, as a value. It's not a code block, but rather an assignment. The semicolon `;` is recommended at the end of statements, no matter what the value is. So the semicolon here is not related to the Function Expression itself, it just terminates the statement.

Callback functions

Let's look at more examples of passing functions as values and using function expressions.

We'll write a function `ask(question, yes, no)` with three parameters:

`question`

Text of the question

`yes`

Function to run if the answer is "Yes"

`no`

Function to run if the answer is "No"

The function should ask the `question` and, depending on the user's answer, call `yes()` or `no()`:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "You agreed." );
}

function showCancel() {
  alert( "You canceled the execution." );
}

// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

Callback functions

In practice, such functions are quite useful. The major difference between a real-life `ask` and the example above is that real-life functions use more complex ways to interact with the user than a simple `confirm`. In the browser, such function usually draws a nice-looking question window. But that's another story.

The arguments `showOk` and `showCancel` of `ask` are called *callback functions* or just *callbacks*.

The idea is that we pass a function and expect it to be "called back" later if necessary. In our case, `showOk` becomes the callback for "yes" answer, and `showCancel` for "no" answer.

We can use Function Expressions to write the same function much shorter:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

Here, functions are declared right inside the `ask(...)` call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of `ask` (because they are not assigned to variables), but that's just what we want here.

Such code appears in our scripts very naturally, it's in the spirit of JavaScript.

Function expression vs declaration

Let's formulate the key differences between Function Declarations and Expressions.

First, the syntax: how to differentiate between them in the code.

- *Function Declaration:* a function, declared as a separate statement, in the main code flow.

```
// Function Declaration
function sum(a, b) {
  return a + b;
}
```

- *Function Expression:* a function, created inside an expression or inside another syntax construct. Here, the function is created at the right side of the "assignment expression" `=`:

```
// Function Expression
let sum = function(a, b) {
  return a + b;
};
```

The more subtle difference is *when* a function is created by the JavaScript engine.

Function expression vs declaration

A Function Expression is created when the execution reaches it and is usable only from that moment.

Once the execution flow passes to the right side of the assignment `let sum = function…` – here we go, the function is created and can be used (assigned, called, etc.) from now on.

Function Declarations are different.

A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is.

That’s due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

And after all Function Declarations are processed, the code is executed. So it has access to these functions.

Function expression vs declaration

For example, this works:

```
1  sayHi("John"); // Hello, John
2
3  function sayHi(name) {
4    alert( `Hello, ${name}` );
5  }
```

The Function Declaration `sayHi` is created when JavaScript is preparing to start the script and is visible everywhere in it.

...If it were a Function Expression, then it wouldn't work:

```
1  sayHi("John"); // error!
2
3  let sayHi = function(name) {  // (*) no magic any more
4    alert( `Hello, ${name}` );
5  };
```

Function Expressions are created when the execution reaches them. That would happen only in the line `(*)`. Too late.

Function expression vs declaration

In strict mode, when a Function Declaration is within a code block, it's visible everywhere inside that block. But not outside of it.

For instance, let's imagine that we need to declare a function `welcome()` depending on the `age` variable that we get during runtime. And then we plan to use it some time later.

If we use Function Declaration, it won't work as intended:

```
let age = prompt("What is your age?", 18);

// conditionally declare a function
if (age < 18) {

  function welcome() {
    alert("Hello!");
  }

} else {

  function welcome() {
    alert("Greetings!");
  }

}

// ...use it later
welcome(); // Error: welcome is not defined
```

Function expression vs declaration

That’s because a Function Declaration is only visible inside the code block in which it resides.

Here’s another example:

```
let age = 16; // take 16 as an example

if (age < 18) {
  welcome();               // \   (runs)
                           //  |
  function welcome() {     //  |
    alert("Hello!");       //  |  Function Declaration is available
  }                        //  |  everywhere in the block where it's declared
                           //  |
  welcome();               // /   (runs)

} else {

  function welcome() {
    alert("Greetings!");
  }
}

// Here we're out of curly braces,
// so we can not see Function Declarations made inside of them.

welcome(); // Error: welcome is not defined
```

Function expression vs declaration

What can we do to make `welcome` visible outside of `if`?

The correct approach would be to use a Function Expression and assign `welcome` to the variable that is declared outside of `if` and has the proper visibility.

This code works as intended:

```
let age = prompt("What is your age?", 18);

let welcome;

if (age < 18) {

  welcome = function() {
    alert("Hello!");
  };

} else {

  welcome = function() {
    alert("Greetings!");
  };

}

welcome(); // ok now
```

Summary

When to choose Function Declaration versus Function Expression?

As a rule of thumb, when we need to declare a function, the first to consider is Function Declaration syntax. It gives more freedom in how to organize our code, because we can call such functions before they are declared.

That's also better for readability, as it's easier to look up `function f(…) {…}` in the code than `let f = function(…) {…};`. Function Declarations are more "eye-catching".

...But if a Function Declaration does not suit us for some reason, or we need a conditional declaration (we've just seen an example), then Function Expression should be used.

- Functions are values. They can be assigned, copied or declared in any place of the code.
- If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".
- If the function is created as a part of an expression, it's called a "Function Expression".
- Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
- Function Expressions are created when the execution flow reaches them.

In most cases when we need to declare a function, a Function Declaration is preferable, because it is visible prior to the declaration itself. That gives us more flexibility in code organization, and is usually more readable.

So we should use a Function Expression only when a Function Declaration is not fit for the task. We've seen a couple of examples of that in this chapter, and will see more in the future.

Arrow functions

There’s another very simple and concise syntax for creating functions, that’s often better than Function Expressions.

It’s called “arrow functions”, because it looks like this:

```
let func = (arg1, arg2, ...argN) => expression
```

...This creates a function `func` that accepts arguments `arg1..argN`, then evaluates the `expression` on the right side with their use and returns its result.

In other words, it’s the shorter version of:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Arrow functions

Let’s see a concrete example:

```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

As you can, see `(a, b) => a + b` means a function that accepts two arguments named `a` and `b`. Upon the execution, it evaluates the expression `a + b` and returns the result.

Arrow functions

- If we have only one argument, then parentheses around parameters can be omitted, making that even shorter.

  For example:

```
let double = n => n * 2;
// roughly the same as: let double = function(n) { return n * 2 }

alert( double(3) ); // 6
```

- If there are no arguments, parentheses will be empty (but they should be present):

```
let sayHi = () => alert("Hello!");

sayHi();
```

Arrow functions can be used in the same way as Function Expressions.

For instance, to dynamically create a function:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");

welcome();
```

Arrow functions

The examples above took arguments from the left of `=>` and evaluated the right-side expression with them.

Sometimes we need something a little bit more complex, like multiple expressions or statements. It is also possible, but we should enclose them in curly braces. Then use a normal `return` within them.

Like this:

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit "return"
};

alert( sum(1, 2) ); // 3
```

More to come

Here we praised arrow functions for brevity. But that's not all!

Arrow functions have other interesting features.

To study them in-depth, we first need to get to know some other aspects of JavaScript, so we'll return to arrow functions later in the chapter Arrow functions revisited.

For now, we can already use arrow functions for one-line actions and callbacks.

Learning Resources

1. **Functions Declaration:**
    1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions
    2. https://javascript.info/function-basics
    3. https://www.w3schools.com/js/js_functions.asp
    4. https://codeburst.io/javascript-functions-understanding-the-basics-207dbf42ed99
2. **Function Expressions:**
    1. https://javascript.info/function-expressions
3. **Arrow functions**
    1. https://javascript.info/arrow-functions-basics

Home Work

1. Create a new project in GitHub lesson-16-hw
2. Create a index.html file
3. Create a script.js file and attach it to index.html
4. Write a function that calculates if a year is leap year (https://en.wikipedia.org/wiki/Leap_year).
   - According to the rules of the Gregorian calendar. A year is a leap year if it is:
     1. divisible by 4 **but(AND)** not 100,
     2. **or** else is divisible by 400.
   - Bonus: try to find what can be extracted as a separate function.
5. Write two temperature convertor functions following this rules https://www.mathsisfun.com/temperature-conversion.html
   1. Write a function that converts **celsius to fahrenheit** and outputs “12°C is 53.6°F”.
   2. Write a function that converts **fahrenheit to celsius** and outputs “53.6°F is 12°C”.