**JavaScript Basics #2: Functions**

Table of Contents

* [Define a function](https://www.ericsdevblog.com/index.php/2021/08/javascript-basics-2-functions/#Define_a_function)
  + [Bindings and Scopes](https://www.ericsdevblog.com/index.php/2021/08/javascript-basics-2-functions/#Bindings_and_Scopes)
* [Optional Arguments](https://www.ericsdevblog.com/index.php/2021/08/javascript-basics-2-functions/#Optional_Arguments)
  + [Rest Parameters](https://www.ericsdevblog.com/index.php/2021/08/javascript-basics-2-functions/#Rest_Parameters)
* [Recursion](https://www.ericsdevblog.com/index.php/2021/08/javascript-basics-2-functions/#Recursion)

**DEFINE A FUNCTION**

In this article, we are going to focus on defining our own custom functions in JavaScript.

A function can be seen as a piece of code wrapped in a value, which allows us to reuse that piece of code over and over again. In this article, we are going to talk about three different ways we can define a function in JavaScript.

The first method is to define functions as values, and bind that value to a name (like how we defined variables in the previous article).

let square = function(x) {

return x\*x;

};

The function is created with the keyword function, and it will take a set of parameters as input, in this case, only x.

A function should also have a body where you return an output using the keyword return, or have some kind of side effect.

Finally, the function as a value will be assigned to the name square, which we need to use to execute/call this function.

Also, remember that the semicolon (;) at the end is necessary, because it is still a full statement where you declare a binding, except the value here is a function.

console.log(square(10));

// -> 100

A function can have more than one parameter or no parameters at all (an empty set of parameters).

const sleep = function() {

console.log("zzzzzzzzzzzzzzzzzzzzzz");

};

var multiply3 = function(x, y, z) {

return x \* y \* z;

};

As you can see, it is possible for a function to have only a side effect and not return anything.

The second method is slightly shorter, by declaring a function using the function keyword, and it doesn’t require a semicolon at the end:

function square(x) {

return x \* x;

}

The method also allows us to do something like this:

sleep();

multiply3(2,3,4);

function sleep() {

console.log("zzzzzzzzzzzzzzzzzzzzzz");

}

function multiply3(x, y, z) {

return x \* y \* z;

}

Here we put the function declarations after the statement that calls them, and the code still works. Now, we can put all the functions in one place, which is a good thing for future maintenance.

Finally, let’s talk about arrow functions. Instead of the keyword function, we can use an arrow (=>) to declare a function.

const square = (x) => {

return x \* x;

}

This is the exact same square() function we saw before, and it works exactly the same. Then why does JavaScript have both arrow functions and the function keyword? While, in some cases, it allows us to write shorter functions.

If the function only has one parameter, then you can omit the parentheses around the parameter list. And if there is only one statement in the function body, the curly braces and the return keyword can also be omitted.

Then, our square() function becomes:

const square = x => x \* x;

**Bindings and Scopes**

Before we go deeper into the topic of functions, let’s go back to the first method. You may have noticed that we defined the functions in the examples using different keywords, let, const and var. What exactly are their differences?

First, we need to understand the concept of scope. It is the part of the program in which the binding is accessible. If a binding is defined outside of any functions or blocks (blocks can be if statements, for or while loops, etc.), then you can refer to that binding wherever you want. This is called a global binding.

If the binding is declared inside a function or block using let or const, that binding will only be accessible from inside the function/block, and that is called a local binding. However, if the binding is defined using the keyword var, then that binding will also be accessible from outside of the function/block.

let x = 10;

if (true) {

let y = 20;

var z = 30;

console.log(x + y + z); // -> all three variables are accessible here

// -> 60

}

console.log(x + z); // -> you cannot "see" y from here, but z is still accessible

Now, what are the differences between let and const? As the name suggests, const stands for constant, meaning once a binding is declared using const, you cannot change its value (unlike let).

Graphical user interface, text, application, email

Description automatically generated

**OPTIONAL ARGUMENTS**

JavaScript is very broad-minded when it comes to the number of parameters you pass to the function. For example, we have the square() function we defined before, which is supposed to take one argument.

function square(x) { return x \* x; }

console.log(square(4, true, "qwerty"));

In this example, we gave the square() function more than one argument, and it simply ignores the extra arguments and computes the square of the first one.

And if we passed too few arguments, those missing parameters will be assigned the value undefined instead of giving you an error.

The downside of this is, of course, when you accidentally make a mistake, no one will tell you about it. So, even though it technically works, you should never rely on this, it could give you some unexpected results. Instead, you should always be careful how many parameters you need, and how many arguments you are passing to the function.

**Rest Parameters**

However, what if you don’t know how many parameters you need? For example, you are designing a function that finds the maximum number in a series of numbers, but you don’t know how many numbers are in the series, so you need to design a function that takes any number of arguments.

To write a function like this, you need to put three dots before the function’s last parameter:

function max(...numbers) {

let result = -Infinity;

for (let number of numbers) {

if (number > result) {

result = number;

}

}

return result;

}

max(1, 2, 3, 4, 5, 6, 7);

Now, the parameter numbers (it is called the rest parameter) will be bound to an array, and the function will return the maximum number in that array.

An array is a list of items, in this case, we have [ 1, 2, 3, 4, 5, 6, 7 ], and for (let number of numbers) is how we can iterate over all items in this array. We’ll discuss arrays in the next article.

**RECURSION**

Finally, let’s talk about the concept of recursion. Recursion is when a function calls itself. The most typical example is how we calculate the power of a number.

function power(base, exponent) {

if (exponent == 0) {

return 1;

} else {

return base \* power(base, exponent - 1);

}

}

Notice that in line 5, the function power called itself with parameters base and exponent - 1.

I know this is a bit confusing, but don’t worry, to understand this code, let’s plug in some numbers. Let’s try to calculate 10^5 (10 to the power of 5).

A picture containing chart

Description automatically generated

In the first step, we simply plug in the numbers, and the function returns 10 \* power(10, 4). Then we need to calculate power(10, 4). Plug in the numbers, and we get 10 \* power(10, 3), which means power(10, 5) equals 10 \* 10 \* power(10, 3).

And we keep repeating the same steps until we get 10 \* 10 \* 10 \* 10 \* 10 \* power(10, 0). Because power(10, 0) returns 1, eventually we get power(10, 5) equals 10 \* 10 \* 10 \* 10 \* 10.

This is a very elegant way of defining exponentiation, but unfortunately, this method is about three times slower than using loops in JavaScript. This is a dilemma that programmers face all the time, we have to choose between simplicity and speed because almost any program can be made faster by making it bigger. It’s up to the programmer to decide on an appropriate balance.