6

Functions

You have seen quite a lot of JavaScript already, and now you are ready for functions. Soon you will see that you have been using functions already, but now it is time to learn how to start writing your own. Functions are a great building block that will reduce the amount of code you will need in your app. You can call a function whenever you need it, and you can write it as a kind of template with variables. So, depending on how you've written it, you can reuse it in many situations.

They do require you to think differently about the structure of your code and this can be hard, especially in the beginning. Once you have got the hang of this way of thinking, functions will really help you to write nicely structured, reusable, and low-maintenance code. Let's dive into this new abstraction layer!

Along the way, we will cover the following topics:

- Basic functions
- Function arguments
- Return
- Variable scope in functions
- Recursive functions
- Nested functions
- Anonymous functions
- Function callbacks

Note: exercise, project and self-check quiz answers can be found in the *Appendix*.

Basic functions

We have been calling functions for a while already. Remember `prompt()`, `console.log()`, `push()`, and `sort()` for arrays? These are all functions. Functions are a group of statements, variable declarations, loops, and so on that are bundled together. Calling a function means an entire group of statements will get executed.

First, we are going to have a look at how we can invoke functions, and then we will see how to write functions of our own.

Invoking functions

We can recognize functions by the parentheses at the end. We can invoke functions like this:

```
nameOfTheFunction();
functionThatTakesInput("the input", 5, true);
```

This is invoking a function called `nameOfTheFunction` with no arguments, and a function called `functionThatTakesInput` with three required arguments. Let's have a look at what functions can look like when we start writing them.

Writing functions

Writing a function can be done using the `function` keyword. Here is the template syntax to do so:

```
function nameOfTheFunction() {
    //content of the function
}
```

The above function can be called like this:

```
nameOfTheFunction();
```

Let's write a function that asks for your name and then greets you:

```
function sayHello() {
  let you = prompt("What's your name? ");
  console.log("Hello", you + "!");
}
```

We add a space after the question mark to ensure the user starts typing their answer one space away from the question mark, rather than directly afterward. We call this function like this:

```
sayHello();
```

It will prompt:

```
What's your name? >
```

Let's go ahead and enter our name. The output will be:

```
Hello Maaike!
```

Take a moment to consider the relationship between functions and variables. As you have seen, functions can contain variables, which shape how they operate. The opposite is also true: variables can contain functions. Still with me? Here you can see an example of a variable containing a function (`varContainingFunction`) and a variable inside a function (`varInFunction`):

```
let varContainingFunction = function() {
    let varInFunction = "I'm in a function.";
    console.log("hi there!", varInFunction);
};

varContainingFunction();
```

Variables contain a certain value and *are* something; they do not *do* anything. Functions are actions. They are a bundle of statements that can be executed when they get called. JavaScript will not run the statements when the functions do not get invoked. We will return to the idea of storing functions in variables, and consider some of the benefits, in the *Anonymous functions* section, but for now let's move on to look at the best way to name your functions.

Naming functions

Giving your function a name might seem like a trivial task, but there are some best practices to keep in mind here. To keep it short:

- Use camelCase for your functions: this means that the first word starts with a lowercase letter and new words start with a capital. That makes it a lot easier to read and keeps your code consistent.
- Make sure that the name describes what the function is doing: it's better to call a number addition function `addNumbers` than `myFunc`.

- Use a verb to describe what the function is doing: make it an action. So instead of `hiThere`, call it `sayHi`.

Practice exercise 6.1

See if you can write a function for yourself. We want to write a function that adds two numbers.

1. Create a function that takes two parameters, adds the parameters together, and returns the result.
2. Set up two different variables with two different values.
3. Use your function on the two variables, and output the result using `console.log`.
4. Create a second call to the function using two more numbers as arguments sent to the function.

Practice exercise 6.2

We are going to create a program that will randomly describe an inputted name.

1. Create an array of descriptive words.
2. Create a function that contains a prompt asking the user for a name.
3. Select a random value from the array using `Math.random`.
4. Output into the console the prompt value and the randomly selected array value.
5. Invoke the function.

Parameters and arguments

You may have noticed that we are talking about parameters and arguments. Both terms are commonly used to mean the information that is passed into a function:

```
function tester(para1, para2){
    return para1 + " " + para2;
}
const arg1 = "argument 1";
const arg2 = "argument 2";
tester(arg1, arg2);
```

A parameter is defined as the variable listed inside the parentheses of the function definition, which defines the scope of the function. They are declared like so:

```
function myFunc(param1, param2) {
  // code of the function;
}
```

A practical example could be the following, which takes `x` and `y` as parameters:

```
function addTwoNumbers(x, y) {
  console.log(x + y);
}
```

When called, this function will simply add the parameters and log the result. However, to do this, we can call the function with arguments:

```
myFunc("arg1", "arg2");
```

We have seen various examples of arguments; for example:

```
console.log("this is an argument");
prompt("argument here too");

let arr = [];
arr.push("argument");
```

Depending on the arguments you are calling with the function, the outcome of the function can change, which makes the function a very powerful and flexible building block. A practical example using our `addTwoNumbers()` function looks like this:

```
addTwoNumbers(3, 4);
addTwoNumbers(12,-90);
```

This will output:

```
7
-78
```

As you can see, the function has a different outcome for both calls. This is because we call it with different arguments, which take the place of `x` and `y`, that are sent to the function to be used within the function scope.

Practice exercise 6.3

Create a basic calculator that takes two numbers and one string value indicating an operation. If the operation equals add, the two numbers should be added. If the operation equals subtract, the two numbers should be subtracted from one another. If there is no option specified, the value of the option should be `add`.

The result of this function needs to be logged. Test your function by invoking it with different operators and no operator specified.

1. Set up two variables containing number values.
2. Set up a variable to hold an operator, either + or -.
3. Create a function that retrieves the two values and the operator string value within its parameters. Use those values with a condition to check if the operator is + or -, and add or subtract the values accordingly (remember if not presented with a valid operator, the function should default to addition).
4. Within `console.log()`, call the function using your variables and output the response to the console.
5. Update the operator value to be the other operator type—either plus or minus—and call to the function again with the new updated arguments.

Default or unsuitable parameters

What happens if we call our `addTwoNumbers()` function without any arguments? Take a moment and decide what you think this should do:

```
addTwoNumbers();
```

Some languages might crash and cry, but not JavaScript. JavaScript just gives the variables a default type, which is undefined. And `undefined + undefined` equals:

```
NaN
```

Instead, we could tell JavaScript to take different default parameters. And that can be done like this:

```
function addTwoNumbers(x = 2, y = 3) {
  console.log(x + y);
}
```

If you call the function with no arguments now, it will automatically assign `2` to `x` and `3` to `y`, unless you override them by calling the function with arguments. The values that are used for invoking are prioritized over hardcoded arguments. So, given the above function, what will the output of these function calls be?

```
addTwoNumbers();
addTwoNumbers(6, 6);
addTwoNumbers(10);
```

The output will be:

```
5
12
13
```

The first one has the default values, so `x` is `2` and `y` is `3`. The second one assigns `6` to both `x` and `y`. The last one is a bit less obvious. We are only giving one argument, so which one will be given this value? Well, JavaScript does not like to overcomplicate things. It simply assigns the value to the first parameter, `x`. Therefore, `x` becomes `10` and `y` gets its default value `3`, and together that makes `13`.

If you call a function with more arguments than parameters, nothing will happen. JavaScript will just execute the function using the first arguments that can be mapped to parameters. Like this:

```
addTwoNumbers(1,2,3,4);
```

This will output:

```
3
```

It is just adding 1 and 2 and ignoring the last two arguments (`3` and `4`).

Special functions and operators

There are a few special ways of writing functions, as well as some special operators that will come in handy. We are talking about arrow functions and the spread and rest operators here. Arrow functions are great for sending functions around as parameters and using shorter notations. The spread and rest operators make our lives easier and are more flexible when sending arguments and working with arrays.

Arrow functions

Arrow functions are a special way of writing functions that can be confusing at first. Their use looks like this:

```
(param1, param2) => body of the function;
```

Or for no parameters:

```
() => body of the function;
```

Or for one parameter (no parentheses are needed here):

```
param => body of the function;
```

Or for a multiline function with two parameters:

```
(param1, param2) => {
  // line 1;
  // any number of lines;
};
```

Arrow functions are useful whenever you want to write an implementation on the spot, such as inside another function as an argument. This is because they are a shorthand notation for writing functions. They are most often used for functions that consist of only one statement. Let's start with a simple function that we will rewrite to an arrow function:

```
function doingStuff(x) {
  console.log(x);
}
```

To rewrite this as an arrow function, you will have to store it in a variable or send it in as an argument if you want to be able to use it. We use the name of the variable to execute the arrow function. In this case we only have one parameter, so it's optional to surround it with parentheses. We can write it like this:

```
let doingArrowStuff = x => console.log(x);
```

And invoke it like this:

```
doingArrowStuff("Great!");
```

This will log `Great!` to the console. If there is more than one argument, we will have to use parentheses, like this:

```
let addTwoNumbers = (x, y) => console.log(x + y);
```

We can call it like this:

```
addTwoNumbers(5, 3);
```

And then it will log `8` to the console. If there are no arguments, you must use the parentheses, like this:

```
let sayHi = () => console.log("hi");
```

If we call `sayHi()`, it will log `hi` to the console.

As a final example, we can combine the arrow function with certain built-in methods. For example, we can use the `foreach()` method on an array. This method executes a certain function for every element in the array. Have a look at this example:

```
const arr = ["squirrel", "alpaca", "buddy"];
arr.forEach(e => console.log(e));
```

It outputs:

```
squirrel
alpaca
buddy
```

For every element in the array, it takes the element as input and executing the arrow function for it. In this case, the function is to log the element. So the output is every single element in the array.

Using arrow functions combined with built-in functions is very powerful. We can do something for every element in the array, without counting or writing a complicated loop. We'll see more examples of great use cases for arrow functions later on.

Spread operator

The spread operator is a special operator. It consists of three dots used before a referenced expression or string, and it spreads out the arguments or elements of an array.

This might sound very complicated, so let's look at a simple example:

```
let spread = ["so", "much", "fun"];
let message = ["JavaScript", "is", ...spread, "and", "very",
               "powerful"];
```

The value of this array becomes:

```
['JavaScript', 'is', 'so', 'much', 'fun', 'and', 'very', 'powerful']
```

As you can see, the elements of the spread operator become individual elements in the array. The spread operator spreads the array to individual elements in the new array. It can also be used to send multiple arguments to a function, like this:

```
function addTwoNumbers(x, y) {
  console.log(x + y);
}
let arr = [5, 9];
addTwoNumbers(...arr);
```

This will log `14` to the console, since it is the same as calling the function with:

```
addTwoNumbers(5, 9);
```

This operator avoids having to copy a long array or string into a function, which saves time and reduces code complexity. You can call a function with multiple spread operators. It will use all the elements of the arrays as input. Here's an example:

```
function addFourNumbers(x, y, z, a) {
  console.log(x + y + z + a);
}
let arr = [5, 9];
let arr2 = [6, 7];
addFourNumbers(...arr, ...arr2);
```

This will output `27` to the console, calling the function like this:

```
addFourNumbers(5, 9, 6, 7);
```

Rest parameter

Similar to the spread operator, we have the rest parameter. It has the same symbol as the spread operator, but it is used inside the function parameter list. Remember what would happen if we were to send an argument too many times, as here:

```
function someFunction(param1, param2) {
  console.log(param1, param2);
}
someFunction("hi", "there!", "How are you?");
```

That's right. Nothing really: it would just pretend we only sent in two arguments and log `hi there!`. If we use the rest parameter, it allows us to send in any number of arguments and translate them into a parameter array. Here is an example:

```
function someFunction(param1, ...param2) {
  console.log(param1, param2);
}
someFunction("hi", "there!", "How are you?");
```

This will log:

```
hi [ 'there!', 'How are you?' ]
```

As you can see, the second parameter has changed into an array, containing our second and third arguments. This can be useful whenever you are not sure what number of arguments you will get. Using the rest parameter allows you to process this variable number of arguments, for example, using a loop.

Returning function values

We are still missing a very important piece to make functions as useful as they are: the return value. Functions can give back a result when we specify a return value. The return value can be stored in a variable. We have done this already – remember `prompt()`?

```
let favoriteSubject = prompt("What is your favorite subject?");
```

We are storing the result of our `prompt()` function in the variable `favoriteSubject`, which in this case would be whatever the user specifies. Let's see what happens if we store the result of our `addTwoNumbers()` function and log that variable:

```
let result = addTwoNumbers(4, 5);
console.log(result);
```

You may or may not have guessed it—this logs the following:

```
9
undefined
```

The value `9` is written to the console because `addTwoNumbers()` contains a `console.log()` statement. The `console.log(result)` line outputs `undefined`, because nothing is inserted into the function to store the result, meaning our function `addTwoNumbers()` does not send anything back. Since JavaScript does not like to cause trouble and crash, it will assign `undefined`. To counter this, we can rewrite our `addTwoNumbers()` function to actually return the value instead of logging it. This is much more powerful because we can store the result and continue working with the result of this function in the rest of our code:

```
function addTwoNumbers(x, y) {
  return x + y;
}
```

`return` ends the function and sends back whatever value comes after `return`. If it is an expression, like the one above, it will evaluate the expression to one result and then return that to where it was called (the `result` variable, in this instance):

```
let result = addTwoNumbers(4, 5);
console.log(result);
```

With these adjustments made, the code snippet logs `9` to the terminal.

What do you think this code does?

```
let resultsArr = [];

for(let i = 0; i < 10; i ++){
  let result = addTwoNumbers(i, 2*i);
  resultsArr.push(result);
}

console.log(resultsArr);
```

It logs an array of all the results to the screen. The function is being called in a loop. The first iteration, `i`, equals `0`. Therefore, the result is `0`. The last iteration, `i`, equals `9`, and therefore the last value of the array equals `27`. Here are the results:

```
[
   0,  3,  6,  9, 12,
  15, 18, 21, 24, 27
]
```

Practice exercise 6.4

Modify the calculator that you made in *Practice exercise 6.2* to return added values instead of printing them. Then, call the function 10 or more times in a loop, and store the results in an array. Once the loop finishes, output the final array into the console.

1. Set up an empty array to store the values that will be calculated within the loop.
2. Create a loop that runs 10 times, incrementing by 1 each time, creating two values each iteration. For the first value, multiply the value of the loop count by 5. For the second value, multiply the value of the loop counter by itself.
3. Create a function that returns the value of the two parameters passed into the function when it is called. Add the values together, returning the result.
4. Within the loop, call the calculation function, passing in the two values as arguments into the function and storing the returned result in a response variable.
5. Still within the loop, push the result values into the array as it iterates through the loop.
6. After the loop is complete, output the value of the array into the console.
7. You should see the values `[0, 6, 14, 24, 36, 50, 66, 84, 104, 126]` for the array in the console.

Returning with arrow functions

If we have a one-line arrow function, we can return without using the keyword `return`. So if we want to rewrite the function, we can write it like this to make an arrow function out of it:

```
let addTwoNumbers = (x, y) => x + y;
```

And we can call it and store the result like this:

```
let result = addTwoNumbers(12, 15);
console.log(result);
```

This will then log `27` to the console. If it's a multiline function, you will have to use the keyword `return` as demonstrated in the previous section. So, for example:

```
let addTwoNumbers = (x, y) => {
  console.log("Adding...");
  return x + y;
}
```

Variable scope in functions

In this section, we will discuss a topic that is often considered challenging. We will talk about scope. Scope defines where you can access a certain variable. When a variable is *in scope*, you can access it. When a variable is *out of scope*, you cannot access the variable. We will discuss this for both local and global variables.

Local variables in functions

Local variables are only in scope within the function they are defined. This is true for `let` variables and `var` variables. There is a difference between them, which we will touch upon here as well. The function parameters (they do not use `let` or `var`) are also local variables. This might sound very vague, but the next code snippet will demonstrate what this means:

```
function testAvailability(x) {
  console.log("Available here:", x);
}

testAvailability("Hi!");
console.log("Not available here:", x);
```

This will output:

```
Available here: Hi!
ReferenceError: x is not defined
```

When called inside the function, `x` will be logged. The statement outside of the function fails, because `x` is a local variable to the function `testAvailability()`. This is showing that the function parameters are not accessible outside of the function.

They are out of scope outside the function and in scope inside the function. Let's have a look at a variable defined inside a function:

```
function testAvailability() {
  let y = "Local variable!";
  console.log("Available here:", y);
}

testAvailability();
console.log("Not available here:", y);
```

This shows the following on the console:

```
Available here: Local variable!
ReferenceError: y is not defined
```

Variables defined inside the function are not available outside the function either.

For beginners, it can be confusing to combine local variables and `return`. Right now, we're telling you the local variables declared inside a function are not available outside of the function, but with `return` you can make their values available outside the function. So if you need their values outside a function, you can return the values. The key word here is *values*! You cannot return the variable itself. Instead, a value can be caught and stored in a different variable, like this:

```
function testAvailability() {
  let y = "I'll return";
  console.log("Available here:", y);
  return y;
}

let z = testAvailability();
console.log("Outside the function:", z);
console.log("Not available here:", y);
```

So, the returned value `I'll return` that was assigned to local variable `y` gets returned and stored in variable `z`.

This variable `z` could actually also have been called `y`, but that would have been confusing since it still would have been a different variable.

The output of this code snippet is as follows:

```
Available here: I'll return
Outside the function: I'll return
ReferenceError: y is not defined
```

let versus var variables

The difference between `let` and `var` is that `var` is function-scoped, which is the concept we described above. `let` is actually not function-scoped but block-scoped. A block is defined by two curly braces `{ }`. The code within those braces is where `let` is still available.

Let's see this distinction in action:

```
function doingStuff() {
  if (true) {
    var x = "local";
  }
  console.log(x);
}

doingStuff();
```

The output of this snippet will be:

```
local
```

If we use `var`, the variable becomes function-scoped and is available anywhere in the function block (even before defining with the value undefined). Thus, after the `if` block has ended, `x` can still be accessed.

Here is what happens with `let`:

```
function doingStuff() {
  if (true) {
    let x = "local";
  }
  console.log(x);
}

doingStuff();
```

This will produce the following output:

```
ReferenceError: x is not defined
```

Here we get the error that `x is not defined`. Since `let` is only block-scoped, `x` goes out of scope when the `if` block ends and can no longer be accessed after that.

A final difference between `let` and `var` relates to the order of declaration in a script. Try using the value of `x` before having defined it with `let`:

```
function doingStuff() {
  if (true) {
    console.log(x);
    let x = "local";
  }
}

doingStuff();
```

This will give a `ReferenceError` that `x` is not initialized. This is because variables declared with `let` cannot be accessed before being defined, even within the same block. What do you think will happen for a `var` declaration like this?

```
function doingStuff() {
  if (true) {
    console.log(x);
    var x = "local";
  }
}

doingStuff();
```

This time, we won't get an error. When we use a `var` variable before the `define` statement, we simply get `undefined`. This is due to a phenomenon called hoisting, which means using a `var` variable before it's been declared results in the variable being `undefined` rather than giving a `ReferenceError`.

Hoisting, and how to negate its effects if needed, are more complex topics that we will cover in *Chapter 12, Intermediate JavaScript*.

const scope

Constants are block-scoped, just like `let`. This is why the scope rules here are similar to those for `let`. Here is an example:

```
function doingStuff() {
  if (true) {
```

```
    const X = "local";
  }
  console.log(X);
}

doingStuff();
```

This will produce the following output:

```
ReferenceError: X is not defined
```

Using a `const` variable before having defined it will also give a `ReferenceError`, just as it does for a `let` variable.

Global variables

As you might have guessed, global variables are variables declared outside a function and not in some other code block. Variables are accessible in the scope (either function or block) where they're defined, plus any "lower" scopes. So, a variable defined outside of a function is available within the function as well as inside any functions or other code blocks inside that function. A variable defined at the top level of your program is therefore available everywhere in your program. This concept is called a global variable. You can see an example here:

```
let globalVar = "Accessible everywhere!";
console.log("Outside function:", globalVar);

function creatingNewScope(x) {
  console.log("Access to global vars inside function." , globalVar);
}

creatingNewScope("some parameter");

console.log("Still available:", globalVar);
```

This will output:

```
Outside function: Accessible everywhere!
Access to global vars inside function. Accessible everywhere!
Still available: Accessible everywhere!
```

As you can see, global variables are accessible from everywhere because they are not declared in a block. They are *always* in scope after they have been defined—it doesn't matter where you use them. However, you can hide their accessibility inside a function by specifying a new variable with the same name inside that scope; this can be done for `let`, `var`, and `const`. (This is not changing the value of the `const` variable; you are creating a new `const` variable that is going to override the first one in the inner scope.) In the same scope, you cannot specify two `let` or two `const` variables with the same name. You can do so for `var`, but you shouldn't do so, in order to avoid confusion.

If you create a variable with the same name inside a function, that variable's value will be used whenever you refer to that variable name within the scope of that particular function. Here you can see an example:

```
let x = "global";

function doingStuff() {
  let x = "local";
  console.log(x);
}

doingStuff();
console.log(x);
```

This will output:

```
local
global
```

As you can see, the value of `x` inside the `doingStuff()` function is `local`. However, outside the function the value is still `global`. This means that you'll have to be extra careful about mixing up names in local and global scopes. It is usually better to avoid this.

The same is also true for parameter names. If you have the same parameter name as a global variable, the value of the parameter will be used:

```
let x = "global";

function doingStuff(x) {
  console.log(x);
}

doingStuff("param");
```

This will log `param`.

There is a danger in relying on global variables too much. This is something you will come across soon when your applications grow. As we just saw, local variables override the value of global variables. It is best to work with local variables in functions; this way, you have more control over what you are working with. This might be a bit vague for now, but it will become clear when coding in the wild as things get bigger and more lines and files of code get involved.

There is only one more very important point to be made about scopes for now. Let's start with an example and see if you can figure out what this should log:

```
function confuseReader() {
  x = "Guess my scope...";
  console.log("Inside the function:", x);
}

confuseReader();
console.log("Outside of function:", x);
```

Answer ready? Here is the output:

```
Inside the function: Guess my scope...
Outside of function: Guess my scope...
```

Do not close the book—we'll explain what is going on. If you look carefully, the `x` in the function gets defined without the keyword `let` or `var`. There is no declaration of `x` above the code; this is all the code of the program. JavaScript does not see `let` or `var` and then decides, "this must be a global variable." Even though it gets defined inside the function, the declaration of `x` within the function gets global scope and can still be accessed outside of the function.

We really want to emphasize that this is a terrible practice. If you need a global variable, declare it at the top of your file.

Immediately invoked function expression

The **immediately invoked function expression** (**IIFE**) is a way of expressing a function so that it gets invoked immediately. It is anonymous, it doesn't have a name, and it is self-executing.

This can be useful when you want to initialize something using this function. It is also used in many design patterns, for example, to create private and public variables and functions.

This has to do with where functions and variables are accessible from. If you have an IIFE in the top-level scope, whatever is in there is not accessible from outside even though it is top level.

Here is how to define it:

```
(function () {
  console.log("IIFE!");
})();
```

The function itself is surrounded by parentheses, which makes it create a function instance. Without these parentheses around it, it would throw an error because our function does not have a name (this is worked around by assigning the function to a variable, though, where the output can be returned to the variable).

`();` executes the unnamed function—this must be done immediately following a function declaration. If your function were to require a parameter, you would pass it in within these final brackets.

You could also combine IIFE with other function patterns. For example, you could use an arrow function here to make the function even more concise:

```
(()=>{
    console.log("run right away");
})();
```

Again, we use `();` to invoke the function that you created.

Practice exercise 6.5

Use IIFE to create a few immediately invoked functions and observe how the scope is affected.

1. Create a variable value with `let` and assign a string value of 1000 to it.
2. Create an IIFE function and within this function scope assign a new value to a variable of the same name. Within the function, print the local value to the console.
3. Create an IIFE expression, assigning it to a new `result` variable, and assign a new value to a variable of the same name within this scope. Return this local value to the `result` variable and invoke the function. Print the `result` variable, along with the variable name you've been using: what value does it contain now?

4. Lastly, create an anonymous function that has a parameter. Add logic that will assign a passed-in value to the same variable name as the other steps, and print it as part of a string sentence. Invoke the function and pass in your desired value within the rounded brackets.

Recursive functions

In some cases, you want to call the same function from inside the function. It can be a beautiful solution to rather complex problems. There are some things to keep in mind though. What do you think this will do?

```
function getRecursive(nr) {
  console.log(nr);
  getRecursive(--nr);
}

getRecursive(3);
```

It prints `3` and then counts down and never stops. Why is it not stopping? Well, we are not saying when it should stop. Look at our improved version:

```
function getRecursive(nr) {
  console.log(nr);
  if (nr > 0) {
    getRecursive(--nr);
  }
}

getRecursive(3);
```

This function is going to call itself until the value of the parameter is no longer bigger than `0`. And then it stops.

What happens when we call a function recursively is that it goes one function deeper every time. The first function call is done last. For this function it goes like this:

- `getRecursive(3)`
    - `getRecursive(2)`
        - `getRecursive(1)`
            - `getRecursive(0)`
            - done with `getRecursive(0)` execution

        - done with `getRecursive(1)` execution
    - done with `getRecursive(2)` execution
- done with `getRecursive(3)` execution

The next recursive function will demonstrate that:

```
function logRecursive(nr) {
  console.log("Started function:", nr);
  if (nr > 0) {
    logRecursive(nr - 1);
  } else {
      console.log("done with recursion");
  }
  console.log("Ended function:", nr);
}

logRecursive(3);
```

It will output:

```
Started function: 3
Started function: 2
Started function: 1
Started function: 0
done with recursion
Ended function: 0
Ended function: 1
Ended function: 2
Ended function: 3
```

Recursive functions can be great in some contexts. When you feel the need to call the same function over and over again in a loop, you should probably consider recursion. An example could also be searching for something. Instead of looping over everything inside the same function, you can split up inside the function and call the function repeatedly from the inside.

However, it must be kept in mind that in general, the performance of recursion is slightly worse than the performance of regular iteration using a loop. So if this causes a bottleneck situation that would really slow down your application, then you might want to consider another approach.

Have a look at calculating the factorial using recursive functions in the following exercise.

Practice exercise 6.6

A common problem that we can solve with recursion is calculating the factorial.

Quick mathematics refresher about factorials:

The factorial of a number is the product of all positive integers bigger than 0, up to the number itself. So for example, the factorial of seven is 7 * 6 * 5 * 4 * 3 * 2 * 1. You can write this as 7!.

How are recursive functions going to help us calculate the factorial? We are going to call the function with a lower number until we reach 0. In this exercise, we will use recursion to calculate the factorial result of a numeric value set as the argument of a function.

1. Create a function that contains a condition within it checking if the argument value is `0`.
2. If the parameter is equal to `0`, it should return the value of `1`. Otherwise, it should return the value of the argument multiplied by the value returned from the function itself, subtracting one from the value of the argument that is provided. This will result in running the block of code until the value reaches `0`.
3. Invoke the function, providing an argument of whatever number you want to find the factorial of. The code should run whatever number is passed initially into the function, decreasing all the way to `0` and outputting the results of the calculation to the console. It could also contain a `console.log()` call to print the current value of the argument in the function as it gets invoked.
4. Change and update the number to see how it affects the results.

Nested functions

Just as with loops, `if` statements, and actually all other building blocks, we can have functions inside functions. This phenomenon is called nested functions:

```
function doOuterFunctionStuff(nr) {
  console.log("Outer function");
  doInnerFunctionStuff(nr);
  function doInnerFunctionStuff(x) {
    console.log(x + 7);
```

```
    console.log("I can access outer variables:", nr);
  }
}
doOuterFunctionStuff(2);
```

This will output:

```
Outer function
9
I can access outer variables: 2
```

As you can see, the outer function is calling its nested function. This nested function has access to the variables of the parent. The other way around, this is not the case. Variables defined inside the inner function have function scope. This means they are accessible inside the function where they are defined, which is in this case the inner function. Thus, this will throw a `ReferenceError`:

```
function doOuterFunctionStuff(nr) {
  doInnerFunctionStuff(nr);
  function doInnerFunctionStuff(x) {
    let z = 10;
  }
  console.log("Not accessible:", z);
}

doOuterFunctionStuff(2);
```

What do you think this will do?

```
function doOuterFunctionStuff(nr) {
  doInnerFunctionStuff(nr);
  function doInnerFunctionStuff(x) {
    let z = 10;
  }
}

doInnerFunctionStuff(3);
```

This will also throw a `ReferenceError`. Now, `doInnerFunctionStuff()` is defined inside the outer function, which means that it is only in scope inside `doOuterFunctionStuff()`. Outside this function, it is out of scope.

Practice exercise 6.7

Create a countdown loop starting at a dynamic value of `10`.

1. Set the `start` variable at a value of `10`, which will be used as the starting value for the loop.
2. Create a function that takes one argument, which is the countdown value.
3. Within the function, output the current value of the countdown into the console.
4. Add a condition to check if the value is less than 1; if it is, then return the function.
5. Add a condition to check if the value of the countdown is not less than 1, then continue to loop by calling the function within itself.
6. Make sure you add a decrement operator on the countdown so the preceding condition eventually will be true to end the loop. Every time it loops, the value will decrease until it reaches 0.
7. Update and create a second countdown using a condition if the value is greater than 0. If it is, decrease the value of the countdown by 1.
8. Use `return` to return the function, which then invokes it again and again until the condition is no longer true.
9. Make sure, when you send the new countdown value as an argument into the function, that there is a way out of the loop by using the `return` keyword and a condition that continues the loop if met.

Anonymous functions

So far, we have been naming our functions. We can also create functions without names if we store them inside variables. We call these functions anonymous. Here is a non-anonymous function:

```
function doingStuffAnonymously() {
  console.log("Not so secret though.");
}
```

Here is how to turn the previous function into an anonymous function:

```
function () {
  console.log("Not so secret though.");
};
```

As you can see, our function has no name. It is anonymous. So you may wonder how you can invoke this function. Well actually, you can't like this!

We will have to store it in a variable in order to call the anonymous function; we can store it like this:

```
let functionVariable = function () {
  console.log("Not so secret though.");
};
```

An anonymous function can be called using the variable name, like this:

```
functionVariable();
```

It will simply output `Not so secret though.`.

This might seem a bit useless, but it is a very powerful JavaScript construct. Storing functions inside variables enables us to do very cool things, like passing in functions as parameters. This concept adds another abstract layer to coding. This concept is called callbacks, and we will discuss it in the next section.

Practice exercise 6.8

1. Set a variable name and assign a function to it. Create a function expression with one parameter that outputs a provided argument to the console.
2. Pass an argument into the function.
3. Create the same function as a normal function declaration.

Function callbacks

Here is an example of passing a function as an argument to another function:

```
function doFlexibleStuff(executeStuff) {
  executeStuff();
  console.log("Inside doFlexibleStuffFunction.");
}
```

If we call this new function with our previously made anonymous function, `functionVariable`, like this:

```
doFlexibleStuff(functionVariable);
```

It will output:

```
Not so secret though.
Inside doFlexibleStuffFunction.
```

But we can also call it with another function, and then our `doFlexibleStuff` function will execute this other function. How cool is that?

```
let anotherFunctionVariable = function() {
  console.log("Another anonymous function implementation.");
}

doFlexibleStuff(anotherFunctionVariable);
```

This will produce the following output:

```
Another anonymous function implementation.
Inside doFlexibleStuffFunction.
```

So what happened? We created a function and stored it in the `anotherFunctionVariable` variable. We then sent that in as a function parameter to our `doFlexibleStuff()` function. And this function is simply executing whatever function gets sent in.

At this point you may wonder why the writers are so excited about this callback concept. It probably looks rather lame in the examples you have seen so far. Once we get to asynchronous functions later on, this concept is going to be of great help. To still satisfy your need for a more concrete example, we will give you one.

In JavaScript, there are many built-in functions, as you may know by now. One of them is the `setTimeout()` function. It is a very special function that is executing a certain function after a specified amount of time that it will wait first. It is also seemingly responsible for quite a few terribly performing web pages, but that is definitely not the fault of this poor misunderstood and misused function.

This code is really something you should try to understand:

```
let youGotThis = function () {
  console.log("You're doing really well, keep coding!");
};

setTimeout(youGotThis, 1000);
```

It is going to wait for `1000`ms (one second) and then print:

```
You're doing really well, keep coding!
```

If you need more encouragement, you can use the `setInterval()` function instead. It works very similarly, but instead of executing the specified function once, it will keep on executing it with the specified interval:

```
setInterval(youGotThis, 1000);
```

In this case, it will print our encouraging message every second until you kill the program.

This concept of the function executing the function after having been called itself is very useful for managing asynchronous program execution.

Chapter projects

Create a recursive function

Create a recursive function that counts up to 10. Invoke the function with different start numbers as the arguments that are passed into the function. The function should run until the value is greater than 10.

Set timeout order

Use the arrow format to create functions that output the values `one` and `two` to the console. Create a third function that outputs the value `three` to the console, and then invokes the first two functions.

Create a fourth function that outputs the word `four` to the console and also use `setTimeout()` to invoke the first function immediately and then the third function.

What does your output look like in the console? Try to get the console to output:

```
Four
Three
One
Two
One
```

Self-check quiz

1. What value is output into the console?

```
let val = 10;
function tester(val){
    val += 10;
    if(val < 100){
        return tester(val);
    }
    return val;
}
tester(val);
console.log(val);
```

2. What will be output into the console by the below code?

```
let testFunction = function(){
    console.log("Hello");
}();
```

3. What will be output to the console?

```
(function () {
    console.log("Welcome");
})();
(function () {
    let firstName = "Laurence";
})();
let result = (function () {
    let firstName = "Laurence";
    return firstName;
})();
console.log(result);
(function (firstName) {
    console.log("My Name is " + firstName);
})("Laurence");
```

4. What will be output to the console?

```
let test2 = (num) => num + 5;
console.log(test2(14));
```

5. What will be output to the console?

```
var addFive1 = function addFive1(num) {
return num + 2;
};
let addFive2 = (num) => num + 2;
console.log(addFive1(14));
```

Summary

In this chapter, we have covered functions. Functions are a great JavaScript building block that we can use to reuse lines of code. We can give our functions parameters, so that we can change the code depending on the arguments a function gets invoked with. Functions can return a result; we do so using the `return` keyword. And we can use `return` at the place where we call a function. We can store the result in a variable or use it in another function, for example.

We then met with variable scopes. The scope entails the places from where variables are accessible. Default `let` and `const` variables can be accessed inside the block where they're defined (and the inner blocks of that block) and `var` is just accessible from the line where it was defined.

We can also use recursive functions to elegantly solve problems that can be solved recursively by nature, such as calculating the factorial. Nested functions were the next topic we studied. They are not a big deal, just functions inside functions. Basic functions inside functions are not considered very pretty, but anonymous functions and arrow functions are not uncommon to see. Anonymous functions are functions without a name and arrow functions are a special case of anonymous functions, where we use an arrow to separate the parameters and the body.

In the next chapter, we'll consider classes, another powerful programming construct!