- 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);
```

# 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);
}
```

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;
```

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?

# 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);
```

# 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 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
```