Back to Week 1

JS | Functions Intro

LESSON

Learning goals

After this lesson, you will be able to:

- properly name, declare and invoke functions,
- use parameters and arguments in functions to enhance code reuse,
- understand what the return statement is,
- use best practices when it comes to naming functions and refactoring them.

Introduction

Often we need to **perform a similar action in many places in our application**. For example, you want to show a message to the users when they successfully or unsuccessfully complete some action.
To avoid repeating the same code in multiple places in your code, you can use a function to wrap that code and reuse it by calling the function whenever we need to do so. This being said, we can proceed to define **functions as reusable pieces of code that perform specific actions**.

Functions are the main “building blocks” of any program. They allow the code to be reused many times without repetition. They keep our code DRY (*Don’t Repeat Yourself*).

Expressions we will use and explain as we go:

The **function declaration** is the process of **creating** a function, but not executing it.

```
function functionName(parameters) {
  // ....
}
```

Explain this code

The process of executing (calling) the function is known as **function invocation**.

```
functionName(arguments);
```

Explain this code

We can declare functions in 3 different ways:

- as function declarations (aka statements),

- as function expressions and
- as an instance of the global Function object constructor. This looks like a sequence of random words for now, and that is understandable. When we start learning about object-oriented programming, it will become much more clear. Up to then, don't stress if you don't understand it completely.

Function syntax

> When talking about function definition, very commonly used synonym is *function declaration*.

The syntax to declare a **function** is:

```
function <name> ([<parameters>]) {
     <instructions>

     [return <expression>;]
}

```

Explain this code

The previous syntax corresponds to the **`function statement`** way of declaring functions. We can often hear that this way has been called *named functions* as well.

We are going to break it down a bit into each part, but before we do that, let's understand the symbols in the definition:

Syntax symbols

- `function, return`: Reserved words and should be typed as they are.
- `<something>`: any given name. Angle brackets (`<`, `>`). ie: `myFunction`
- `[something]`: optional. Square brackets (`[`, `]`). In our case, - `[parameters]` is optional since some functions won't have any parameters as we saw in the previous `sayHelloWorld` example. More about parameters in the rest of the lesson.

To summarize, when declaring a function, we have to make sure these exist:

- `function` keyword,
- the name of the function,
- parameters (if any, if not then just `()`),
- body of the function - which is all the code (instructions) between the curly braces `{}`.

```
// keyword          function parameters (if any)
// ^          name             ___|
// |            |             |
function sayHelloWorld(someParameter(s)?) {
  // the code or so-called the body of the function
}
```

Explain this code

> There is another way to define functions called a `function expression`, which we will explain later. The way you just learned is called `function declaration`.

Function name

- The name we define for each function. However, that doesn't mean we can use any word to name our functions.
- The name should be very descriptive and should express what the function **does**.
- As a rule of thumb, we will try to use **verbs** that describe **actions** (ex. getUsers, showErrorMessage, showSuccessMessage, etc.).
- In JavaScript, we prefer to name functions the same as variables using the **camelCase**
  - `addTwoNumbers`
  - `sayHello`
- Function name always begin with a lowercase letter:
  - `lowerCase` 👍
  - ~~`LowerCase`~~

Function parameters (and function arguments)

A function can accept zero, one, or multiple parameters. If there are multiple parameters, you need to separate them by commas (`,`).

But what are parameters? Let's declare (define) our first function and learn, through example, what are and how to use parameters.

```
// function declaration

function sayHello() {
  console.log('Hello there!');
}
```

Explain this code

Now, let's invoke (call) this function, so it executes:

```
sayHello();

// output in the console:
// Hello there!
```

Explain this code

This is the example of a function with zero parameters. So when do we need them actually?

```
function sayHelloMary() {
  console.log('Hello, Mary!');
}
```

Explain this code

```
function sayHelloJohn() {
  console.log('Hello, John!');
}
```

Explain this code

And then, we will have to call every function:

```
sayHelloMary(); // output: 'Hello, Mary!'
sayHelloJohn(); // output: 'Hello, John!'
```

Explain this code

Okay, it is obvious that we can do this in a simpler and cleaner way. Like, imagine the following scenario: you have to create a function that calculates the sum of two numbers. It would be ridiculous to imagine that you would have to create a new function for every possible combination of numbers.

In these situations, when we want to tell the computer to do very similar things, but not exactly the same each time is when **parameters** come to rescue!

The same example but this time adapted to be reused when greeting whoever you want:

```
function sayHello(name) {
  console.log(`Hello ${name}!`);
}

sayHello('Mary');
// name = Mary
// output: 'Hello Mary!'

sayHello('John');
// name = John
// output: 'Hello John!'

sayHello('Lucy');
// name = Lucy
// output: 'Hello Lucy!'
```

Explain this code

This is the example of a function with a single parameter. However, functions can have as many parameters as we need.

```
// function definition/declaration
function sayHello(classmate1, classmate2, classmate3) {
  console.log(`Hello ${classmate1}, ${classmate2} and ${classmate3}!`);
}

// function call/invocation
sayHello('Mat', 'Jo', 'Maria');
// output: Hello Mat, Jo and Maria!
```

Explain this code

Function arguments

Now we will introduce one similar but not the same term, which is very often interchangeably used but shouldn't be: **arguments**.

> **Parameters** are variables that are part of the function declaration (the names which we use when we define some function; i.e., `classmate1`, `classmate2`). These are also known as *placeholders* since they don't have to represent real values passed to the function, as we can see in this example. `classmate1`, `classmate2` could be any words when we define a function.
> **Arguments** are (real) values passed to function in the moment of its invocation (i.e. *Mat*, *Jo*, *Maria*).

Returning value(s)

> *Potential interview question:*
> A JavaScript function **always returns something**.
> When a returning value is not specified, the function returns `undefined`.

We already covered **undefined** as primitive data type, but here you can refresh your knowledge.

Let's see an example:

```
function printName(name) {
  console.log(`Name is ${name}.`);
}

printName('Ana');

// output:
// Name is Ana.
// undefined
```

Explain this code

Since our previous function didn't have any *return* statement, after the output (from `console.log()`), it returned *undefined*.
So what exactly is `return`?

> The `return` statement delivers value as a final result of the bundled actions that took place in the function body.

We will take the same example:

```
function printName(name) {
  return `Name is ${name}.`;
}

printName('Peter'); // => Name is Peter.
```

Explain this code

> The `return` statement is the very last piece of code that will execute in the function. Any code you add after the `return` doesn't exist for the function since it finishes the execution with the *return* statement.

Here is an example:

```
function printName(name) {
  return `Name is ${name}.`;
  console.log('Hello, I am after the return statement.');
}

printName('Yo'); // => Name is Yo.
```

Explain this code

Multiple returns

One function can have more than one return statement.

```
function printName(name) {
  if (name.length === 0) {
    return 'Please provide a valid name!';
  }

  return `Name is ${name}.`;
}

printName(''); // => Please provide a valid name!
printName('George'); // => Name is George.
```

Explain this code

In the previous example, we took care of the *edge* case in case we accidentally invoked the function without passing an argument. In this case, the condition in the `if` statement would be true, and we would get the message as a return from the function. Since *return* is the last statement in every function, the rest of the code wouldn't execute.

Return multiple values

In all examples so far, we showed the return of a single value. The previous function returned statement `Please provide a valid name!` *OR* statement `Name is George.`. What if we want to return multiple values in a single return? Definitely doable, and let's see how.

Return values in an object

Imagine the following scenario: you are asking the user for their personal information, and then you want to return their first and last name to be used in some other function or some other piece of code. You could concatenate them into a string (`fullName`) and return that string (`return fullName`). But that has its limitations. If you need to use just their first name later, you would have to do some additional work to get it from that string. Instead of that, let's return the user's first and last name in the object so they could be used if needed.

```js
function getUserInfo(firstName, lastName) {
  const userInfo = {
    firstName: firstName,
    lastName: lastName
  };

  return userInfo;
}

getUserInfo('ana', 'martinez'); // => { firstName: 'ana', lastName: 'martinez' }
```

Explain this code

To utilize the *return*, we can do the following:

```js
const userData = getUserInfo('ana', 'martinez');
const firstName = userData.firstName;
console.log(firstName); // => ana
```

Explain this code

If this `const userData = getUserInfo('ana', 'martinez');` is not too clear, give it a bit. When we come to function expressions, this will become much more understandable.

Return values in an array

Similar to the previous example, *return* statement can return the array.

```js
function getFavorites(fav1, fav2, fav3) {
  const favorites = [fav1, fav2, fav3];

  return favorites;
}

getFavorites('javascript', 'html', 'css'); // => [ 'javascript', 'html', 'css' ]
```

Explain this code

To use this data, we can do the following:

```js
const favoritesArray = getFavorites('javascript', 'html', 'css');
const favorite1 = favoritesArray[0];
const favorite3 = favoritesArray[2];
console.log(favorite1, favorite3); // => javascript css
```

Explain this code

To wrap up about return statements, functions can’t, by default, return multiple values. To surpass this limitation, you can pack return values into an object or array and return it.

If this syntax is a bit robust (and it is), don’t worry. Soon you will know how to use some cool object and array destructuring features, and the previous code will look much nicer.

Check for understanding

1. Create a function that accepts 3 numbers as parameters, and returns their sum.
2. Create a function named `isNameOddOrEven()` that accepts a string as a parameter. The function should return whether a received string has an odd or even number of letters. The expected return should be in the following format - string: '<name> has an even/odd number of letters'.

Writing good functions

Functions are one of the pillars of programming. Functions help us to keep our code clean and well organized, and as we write more and more code. The following are some of the main characteristics of good functions:

- **Reuse code** refers to the possibility to call a function as many times as we need it in our code, but we only need to define once how it works.
- **Abstraction** is a technique that allows us to think at higher, more *abstract* levels. We will learn about abstraction later but to visualize what we mean - we really don't know how `.substring()` works, but we know when to use it and what results to expect.
- **Separation of Concerns** refers to the fact that functions allow us to split a big problem into multiple smaller ones.
- **Single Responsibility Principle**, as a name says, refers to the fact that a function should do just one thing. The name of the function has to be very clear so you can identify what is doing just reading the name.

Code reuse and division of responsibilities

From generalization, code reuse arises naturally: now, we can perform the same operation in different places without repeating a single line of code. We are reusing the function.

The division of responsibilities refers to the level of isolation. **One function should only do one thing**. It sounds simple, but mastering the division of responsibilities is not that easy. Here are some tips:

- Name your functions with verbs, but only **one verb** per function.
- If your function is more than 20 lines of code, you are most likely doing it wrong.
- If you are grouping a bunch of instructions, you are probably doing more than one thing.

> Use a straightforward rule to check if you really separated the concerns in a function: when you try to describe what a specific function does, if you use `AND` while doing that, most likely, that function could be split into two or more.
> Example: This function *calculates the total price* AND *displays it to the users*. This function should be split into two.

Refactoring

> **Code Refactoring** is a technique in software development by which we change the way the code is structured, keeping the same functionality.

It is a good practice to refactor our code often, as it will help us to make it better, more modular, and easier to maintain.

Examples of **refactoring techniques** may include techniques such as:

- Choosing better names for variables, functions, etc.
- Taking pieces of functionality and abstracting them in separate functions.

Let's look at our `avg()` function:

```
function avg(array) {
  // !array.length is the same as writing array.length === 0
  if (!array.length) return;

  for (let sum = 0, i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum / array.length;
}
```

Explain this code

If we think about it, it actually does two separate things:

1. it calculates the sum of all the items in the array and
2. it divides the total sum by the length of the array.

We can further improve this by isolating one of those calculations into a separate function. We need to break down the code so that it does the same thing, but it is easier to understand and maintain it.

Let's call the first step `sum()` and make it into a separate function. Then the `avg()` could be rewritten, now using our `sum` function:

```
function sum(array) {
  if (!array.length) return;

  for (let sum = 0, i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum;
}

function avg(array) {
  if (!array.length) return;

  return sum(array) / array.length;
}
```

Explain this code

As you can see, we are calling the function `sum()` as part of the expression for the `return` statement of the `avg()` function. Cool, right?

Summary

In this lesson, you have learned what function statements are and how to declare and invoke functions. You also learned that to enhance function reusability, when declaring them, we can pass parameters to functions. Function parameters are placeholders in a function definition, which become function arguments when we call the function to execute it. Functions can have one or multiple return statements, but just one of those statements will actually be the final return since, as soon as one of them triggers, the function stops executing. In case there is a need to have multiple values returned from a function, we can "pack" them into object or array and return it as a single value that holds all the others.
Knowing how to name your variables or functions actually is probably one of the most challenging things. If you name them wrongly, maintaining that code and building on top of it, becomes simply a nightmare. This being said, make sure you know some of the basic characteristics that make any function a good function.

Extra resources

- Functions - Mozilla Developer Network

Mark as completed

PREVIOUS
← LAB | JavaScript Basic Algorithms

NEXT
JS | Data Types in JavaScript - Arrays →