Lesson Weekend

Introduction to Programming (/introduction-to-programming) / JavaScript and Web Browsers (/introduction-to-programming/javascript-and-web-browsers) / Functions

Text Cheat sheet

To help us better understand methods, let's get to know functions better. In this lesson, we'll review what a function is and what it looks like. We'll also learn some new terminology, and we'll take time to compare methods and functions so the distinction becomes clear.

If you are confused about any of the concepts covered in this lesson, that is normal. Know that we will get a lot of practice with functions in this section. Also, keep in mind that you don't need to know everything about how JavaScript works in order to use basic methods and functions. Take notes on anything you don't understand so that you can discuss it with your dev team, pair, or in Scrum. The time it takes to understand methods and functions will be different for everyone in your cohort, and discussing points of confusion is instrumental in getting to a solid understanding of these JavaScript concepts.

Functions in JavaScript

Remember, all methods are functions, but not all functions are methods. Let's take a look at a function that *isn't* a method. JavaScript has very few built-in functions, and a lot of built-in methods. So, to demonstrate functions, we'll write a custom function. We'll do this by **declaring a function**. This means to define a function by giving it a name and procedures. This is also called **function declaration**. Here's an example of a function with the name `addEmphasis`:

```
// This is a function declaration.
function addEmphasis(stringParam) {
  const result = stringParam.toUpperCase().concat("!!!");
  return result;
}
```

Take note that the code snippet above shows the proper indentation and spacing for functions. If you are going to code along with this lesson, remember that you can format your code in the Dev Tools console to include new lines and indentation:

- To create a new line, use `shift + enter`.
- To tab over multiple spaces for indentation, use `tab`. To configure the console to use 2 spaces for indentation with `tab`, in the DevTools window, go to *Settings* > *Preferences* > scroll to the *Sources* section > set "Default indentation" to 2 spaces.

Now, here's an example of **calling** the `addEmphasis` function with the **argument** `"I love my pet rabbit"`:

```
// This is a function call.
> addEmphasis("I love my pet rabbit");
"I LOVE MY PET RABBIT!!!"
```

Remember, **calling a function** means that we're asking the function to perform its actions by running through its procedures. Even being unfamiliar with function syntax, we should be able to tell from the function declaration and the function call that `addEmphasis(stringParam)` takes a string as an argument, and then capitalizes that string and adds three exclamation points to the end of it.

Note that the `addEmphasis(stringParam)` function takes a string as an argument but has no receiver that it is called on. How does this all happen? Let's get into it.

Semicolons

Before we continue, we've reached our first example where we *don't* add semicolons at the end of each line. Whenever we write a function starting with the `function` keyword, we do not need to end the function with a semicolon.

Also, often we'll have code that spans multiple lines. We should still have semicolons at the end of each **statement**. For instance, the line beginning with `return` is a statement, and so is the line in which we create the `result` variable. This will become clearer with practice and more examples.

Parameters

In the function definition, `stringParam` is a parameter. A **parameter** is a placeholder variable. It has no value yet.

When we call the function, we pass an **argument** into the **parameter**, and the parameter takes on the value of the argument. If, for some reason, we forget to pass an argument into a parameter, that parameter's value will be `undefined`.

Let's look at an example. Remember this?

```
> let stringVariable = "hi";
> stringVariable.toUpperCase();
"HI"
```

Just like we can call methods on variables (as shown above), we can pass variables in as arguments:

```
> function addEmphasis(stringParam) {  // function declarat
ion
  const result = string.toUpperCase().concat("!!!");
  return result;
}
> let stringArg = "hi";
> addEmphasis(stringArg); // function call
"HI!!!"
```

In this example, `stringArg` is the **argument** that we're passing into the `addEmphasis` function call, and `stringParam` is the **parameter** that we're using in the function definition as a placeholder variable for the argument. When we pass in `stringArg` as the argument, that means that the placeholder `stringParam` takes on its value of `"hi!"` in the function.

Return Statements

We can see in the function definition that `stringParam` is uppercased and then three exclamation points are added to the end of it. We then store the result in a variable called `result`, and we `return` the `result` variable.

The **`return`** keyword makes it so that the data saved in the `result` variable is available outside of the function. In other words, if we didn't have the `return` statement, the console would not show us the result, `"I LOVE MY PET RABBIT!!!"`. To try this out, enter this revised function definition into your browser DevTools console:

```
> function addEmphasisVersion2(stringParam) {
  const result = stringParam.toUpperCase().concat("!!!");
}
```

And call the function right after:

```
> addEmphasisVersion2("I love my pet rabbit");
```

Did you get the uppercased and concatenated result? No. Instead `undefined` was returned, meaning that no value was returned out of the function.

Another Look at Function Syntax

Let's break down our `addEmphasis(stringParam)` example further. It is based on the following syntax:

```
function nameOfFunction(/* parameters go here */) {
  // We define what the function does here.
  // We can call on any of the parameters here and apply me
thods or arithmetic to them.
  // We can use the return keyword to make
  // the data that's internal to the function sent outside
of it.
}
```

- We use the `function` keyword to indicate that we are defining a function.
- Next, comes the `nameOfFunction` part. In the function we wrote earlier, we called our function `addEmphasis` . We use lower camel case to name functions, just like with variable names.
- Next, comes this syntax `() { }`
    - the parens `()` are where we put any parameters, and

- the curly brackets `{ }` are where we define what the function does. This is called the **function body**. We can include as many lines of code as we need in the function body.

Soon we will be writing more custom functions so there is no need to worry about perfectly understanding or remembering the function syntax we have covered so far. It's okay if this is all still a bit fuzzy. After all, we're covering a lot of new concepts in this lesson!

Distinguishing Functions from Methods

Let's review how we call the `addEmphasis` function. If you haven't already, put the function definition (below) into the DevTools console. To make new lines, hold down the `shift` key while pressing `enter`.

```
> function addEmphasis(stringParam) {  // function definiti
on
  const result = string.toUpperCase().concat("!!!");
  return result;
}
```

Then call the function like this:

```
> addEmphasis("hi");
"HI!!!"
```

As you can see, **there's no receiver**. Instead, there's just an argument. What would happen if we tried to call this function on a receiver?

```
> "hi".addEmphasis();
```

We'll get the following error:

```
"hi".addEmphasis is not a function
```

How can this be the case?

Well, `addEmphasis("hi")` and `"hi".addEmphasis()` are two different functions. The former is a function that isn't called *on* anything. The latter is a type of function known as a method that is called *on* something — in this case, a string.

We'll be working with both methods and functions a lot in JavaScript, so being able to distinguish between the two is important. Fortunately, we'll get lots of practice with it.

Differentiating Parameters from Arguments

What can make understanding the difference between parameters and arguments especially confusing for beginners is when a variable has the *same* name as a parameter. In JavaScript, the following code is perfectly acceptable:

```
> function addEmphasis(string) {
  const result = string.toUpperCase().concat("!!!");
  return result;
}
> let string = "hi";
> addEmphasis(string);
HI!!!
```

Note that the parameter in `function addEmphasis(string)` has the same name as the variable `let string = "hi";`. This is so confusing!

When we actually call `addEmphasis(string)`, we are passing in the `string` variable (set to `"hi"`) as an argument. So `string` in that context is an argument and *not* the parameter we initially defined when we declared the function.

This is why we suggest making variable names different from parameter names. Doing so makes the distinction between parameters and arguments much more clear. You can add "Param" at the end of parameters, like `stringParam`:

```
> function addEmphasis(stringParam) {
  return stringParam.toUpperCase().concat("!!!");
}
```

And for arguments, you can give them a more descriptive name or use 'arg', short for argument' in the variable name:

```
> let greeting = "hi";
> addEmphasis(greeting);
HI!!!
```

```
> let userInput = "hi";
> addEmphasis(userInput);
HI!!!
```

```
> let arg1 = "hi"; // 'arg' is short for 'argument'
> addEmphasis(arg1);
HI!!!
```

It is completely expected that the difference between parameters and arguments will be at least a little confusing at first. Just keep reminding yourself of the distinction:

- A **parameter** is a placeholder for an argument. We use parameters when we define functions and methods. We'll learn more about declaring functions soon, when we actually start creating our own custom functions.
- An **argument** is the value that is passed *into* a parameter when we call the function (or method). Often we'll use variables for our arguments, which can be easily confused with parameters.

Once again, keep your variable names distinct from your parameter names to avoid confusion! Otherwise, it's easy to start thinking variables used as arguments are parameters, when they are not.

Summary

In this lesson, we've learned more about functions and we've better distinguished between methods and functions. A method is a kind of function that is called *on* something and belongs to a specific data type. A function is just a set of operations that isn't necessarily a method.

We've also learned about the difference between parameters and arguments. Once again, parameters are placeholders for arguments. Because an argument is passed into a parameter, it can be tempting to think arguments and parameters are interchangeable when they are not.

We will be working with these concepts every day at Epicodus — so even if they are not fully clear yet, they will be soon.

Previous (/introduction-to-programming/javascript-and-web-browsers/methods)
Next (/introduction-to-programming/javascript-and-web-browsers/string-and-number-methods)

Lesson 13 of 75
Last updated March 24, 2023

disable dark mode

(http://www.epicodus.com)