Lesson Monday

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

Text Cheat sheet

We were first introduced to JavaScript functions in the Functions (https://www.learnhowtoprogram.com/introduction-to-programming/javascript-and-web-browsers/functions) lesson. In that lesson we familiarized ourselves with function syntax and other important terminology and features. In this lesson, we'll review what we learned and start writing our own custom functions.

What do we know so far about functions?

- A function is a bundle of code that performs a set of operations.
- Functions allow us to *do* things in JavaScript.
- A method is a type of function that belongs to an object, and a function is just a set of operations that isn't necessarily a method.
- There are a few built-in functions in JavaScript, like `parseInt()`, but mostly we will be writing our own custom functions. This is in contrast to methods: we will only be using built-in JavaScript methods until we revisit objects in the Object-Oriented

JavaScript (https://www.learnhowtoprogram.com/intermediate-javascript/object-oriented-javascript/) course section.

Let's review function syntax, terminology and get started with writing our own custom functions.

Writing Functions

Formatting Code in the DevTools Console

Before we get into our first example function, let's review how to format code in the DevTools console to make 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`, within the DevTools window go to *Settings* > *Preferences* > scroll to the *Sources* section > set "Default indentation" to 2 spaces.

Function Declarations

We're going to write a very simple function declaration to start. We'll give the function the name `sayHi`.

```
// Here we are defining the sayHi() function; this is a "fu
nction declaration".
function sayHi() {
  return 'Hello from Epicodus!';
}
```

When we write a function like this, it's called a **function declaration**. That just means we've declared (defined) a function starting with the `function` keyword. There are other ways of writing

a function, but we won't be covering them in this lesson. Here's the breakdown:

- The `function` keyword indicates that we are declaring a function. "Declaring" a function means we're defining it.
- `sayHi` is the name of the function, and we always include parentheses `()` after the function name. The parentheses are to hold optional parameters, which we'll review later in this lesson.
- Everything enclosed in the curly brackets `{` and `}` is called the **function body**; this is where we write out all the code that we want the function to execute when we call on it.

Indentation and Spacing

Note, the code snippet above shows the proper indentation and spacing for functions. It's common convention to write JavaScript functions on multiple lines to make the code more readable. While we could write our `sayHi()` function declaration on one line like this:

```
// Don't write function in one line.
function sayHi() { return 'Hello from Epicodus!'; }
```

It's not considered best practice.

Semicolons in Function Declarations

Let's continue examining our function declaration:

```
// Here we are defining the sayHi function; this is a "func
tion declaration".
function sayHi() {
  return 'Hello from Epicodus!';
}
```

Notice how we are using semicolons here. The first line, which uses the function keyword and includes the name of the function, ends with an opening curly bracket `{`. It will *never* have a semicolon.

The next line has a `return` statement. Statements and expressions have semicolons at the end of the line regardless of whether or not they are inside function declarations.

The `return` keyword tells JavaScript to return anything to the right of the `return` keyword. In this case, we're returning the string `'Hello from Epicodus!'`. We could also return a variable instead:

```
function sayHi() {
  const greeting = 'Hello from Epicodus!';  // Each stateme
nt is on its own line.
  return greeting;                          // Each stateme
nt is on its own line.
}
```

Without a `return` statement, JavaScript will return `undefined` from the function, which is JavaScript's way of saying something does not have a value. Also notice that each new statement in the function body is written on a new line. This is also a best practice for code readability.

Finally, we close our function with a curly bracket `}`. We *don't* add a semicolon here. Why is this? Well, this is a part of the **function declaration**. It's *not* a statement. JavaScript knows that the curly bracket `}` represents the end of the function so it doesn't need a semicolon.

If semicolon placement feels fuzzy still, it's really not something to worry about. Understanding semicolon placement is important but there are much more important topics we need to focus on right now like learning about functions and how they work.

Calling a Function

For this next example, we're putting the code into the DevTools console. To call our `sayHi` function, this is what we do:

```
// First we input the function declaration.
> function sayHi() {
  return 'Hello from Epicodus!';
}
// Then we call the sayHi() function by including parens.
> sayHi();
'Hello from Epicodus!'
```

Every time we run the `sayHi()` function, it executes all JavaScript code between the opening and closing braces — `{` and `}`. In this case, our function returns the string `'Hello from Epicodus!'`. This isn't terribly useful so let's write a slightly more interesting function.

A Function with a Parameter — `saySomething(whatToSay)`

Let's write a function in the DevTools console that has one parameter. We'll name this function `saySomething`:

```
> function saySomething(whatToSay) {  // function declarati
on with 1 parameter
   return whatToSay;
}
> saySomething("hello!");             // function call with
1 argument
"hello!"
```

Let's review the parts of this new function:

- When we're declaring a new function, we start with the `function` **keyword**. A function declaration always starts with this keyword.
- Next, we have the **function name** `saySomething`. After the name of the function, we will always include parentheses to list any parameters (more on this below): `saySomething()`. Just like with naming variables, function names should:
    - Be lower camelCase, which means the first letter of the name should be lower-cased while the first letter of other words in the function (`Something` in this case) should be capitalized.
    - Be named descriptively so other developers can quickly understand the purpose of the function. In the example above, we are simply denoting that this function will say something.
- Next, we have the **function body**. This is enclosed in two curly brackets, `{` and `}`. The function body includes any code that should be executed when the function is called, in this case `return whatToSay;`. It is convention to have the opening curly bracket on the same line as the function declaration. Meanwhile, the closing curly bracket goes on the final line of the function.
- If a function has **parameters**, they go inside the parentheses next to the function name: `saySomething(whatToSay)`. A **parameter** is a placeholder variable in the function declaration that doesn't initially have a value. The value of parameters are set by the arguments we pass into functions when we call them. In other words, parameters are placeholder variables for any data (called "arguments") that we want to pass into a function when we call the function. Here's more of a breakdown:
    - `whatToSay` is the **parameter** in the `saySomething()` function.
    - When we call the function with an **argument**, like `saySomething("hello")`, we are setting the value of the

`whatToSay` parameter to the argument, which is the string `"hello"`.
    - Then, anywhere the parameter `whatToSay` is used in the function body, like in the return statement `return whatToSay;`, the argument's value `"hello"` is used. This makes the function return `"hello"`. If we called `saySomething("Howdy")`, then the function would return `"Howdy"`.
    - Note that we can create as many parameters as we want in a function.

If you're confused about the difference between arguments and parameters, just remember that the argument is the data you pass in, and the parameter is the variable that receives the argument as a value. In the `saySomething()` function, `"hello!"` is the argument, and `whatToSay` is the parameter. The parameter can then be used just like any other variable. Let's look at another example to better understand functions, and the difference between parameters and arguments.

A Function with Two Parameters — `add(number1, number2)`

Okay, on to another slightly more complex function:

```
> function add(number1, number2) {  // function declaration
with 2 parameters
   const sum = number1 + number2;
   return sum;
}
> add(3, 5);                        // function call with 2
arguments
8
```

Let's step through exactly what happens when we call `add(3, 5)`:

1. We call the `add(3, 5)` function with two arguments `3` and `5`: because the `add()` function has two parameters, we must pass in two arguments, one for each parameter.
2. The `add` function is run. The parameter `number1` is set equal to 3, and the parameter `number2` is set equal to 5.
3. The expression `3 + 5` is run and we set the value of the `sum` variable equal to the result of the expression, which is `8`.
4. The variable `sum` (with a value of `8`) is returned.
5. The `add` function ends.

Notice our variables names: `number1` and `number2`. We could have called them `n1` and `n2`, and it would have taken less typing. But the name `number1` very clearly expresses what the variable is for, whereas `n1` would require another programmer to figure it out from context (or your future self, when you come back to your code in a few months and don't remember exactly how it works). In this example, it is pretty obvious what `n1` is for. But if you practice choosing descriptive names now and resisting the temptation to abbreviate, you will be in the habit of doing it when you're writing more complex code where it matters more.

Documentation on MDN

To see the documentation on function declarations, including more examples, visit this page on MDN:

- **Function Declarations (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function)**

Next, let's practice what we've learned and write and call some functions!

Previous (/introduction-to-programming/javascript-and-web-browsers/practice-review-of-javascript-basics)
Next (/introduction-to-programming/javascript-and-web-browsers/practice-writing-functions)

disable dark mode

(http://www.epicodus.com)