Lesson Tuesday

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

Text Cheat sheet

In this lesson, we'll explain another method of defining a function called **function expressions**. We'll be using function expressions in the next lesson when we learn about event handling with event handler properties.

First, let's start with a review.

Review

Formatting Code in the DevTools Console

It's optional to code along with this lesson. If you want to code along, review how to format code in the DevTools Console:

- 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.

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.

Function Declarations

Here's a function called `duplicate()` that we've defined using a function declaration:

```
function duplicate(phrase) {
  return phrase.concat(" ").concat(phrase);
}
```

When we write a function like this, it's called a **function declaration**. This means we've defined a function starting with the `function` keyword. Here's the breakdown:

- The `function` keyword indicates that we are declaring a function.
- `duplicate` is the name of the function, and we always include parentheses `()` after the function name. The parentheses are to hold optional parameters.
- `phrase` is a parameter: a placeholder for any values that we want to pass into the function when we call it. In this example, the data type of `phrase` should be a string.
- 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 it.

When we call this function, we need to pass an argument in. An **argument** is the value that is passed *into* a parameter. The function call will look like this:

```
> duplicate("echo");
"echo echo"
```

Function declarations are also called **function statements**, though that is less common.

Defining Functions with Function Expressions

We can also define a function with a **function expression**. This involves storing a function inside a variable like this:

```
// this is a function expression
const add = function(number1, number2) {
  return number1 + number2;
};
```

This style of declaring a function is also called a **function literal**, though this term is less commonly used.

You might be wondering why we added a semicolon to the end of the function expression. Doesn't the curly brace indicate the end of the function? Yes, it does, but we are treating this similarly to other statements where we assign a value to a variable. For that reason, we are adding a semicolon to the end.

We can then call this function by doing the following:

```
> add(3, 5)
8
```

As we can see, calling the function works in exactly the same way as it does when we write the function as a function declaration:

```
// this is a function declaration
function add(number1, number2) {
   return number1 + number2;
}
```

Differences Between Function Declarations and Function Expressions

The main difference between a function expression and a function declaration is the function name, which can be omitted in function expressions to create anonymous functions. The example we've seen so far has been an **anonymous function expression**. Even though we're storing the function into a variable `add`, and we can call the function by adding parens at the end of the variable name `add()`, the function expression is still considered anonymous.

To write a **named function expression**, it would look like this:

```
// this is a **named** function expression
const calc = function add(number1, number2) {
  return number1 + number2;
};
```

Notice that we've re-written the original `add()` function expression. Now, we've given the name `add` to our function expression (`function add() {...}`), and we're storing it in a variable called `calc`. If we wanted to call this named function expression, we would do so like this:

```
> calc(2,3);
5
```

The use cases for named function expressions are limited and you likely won't use them in the program. You are welcome to explore more about these use cases by referencing this section on MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function#named_function_expression)

Another important difference between these two methods of defining a function have to do with hoisting. JS interpreters (programs that interpret JS code, like Google Chrome's V8 JavaScript engine) perform **hoisting**, which is the process of moving the declaration of functions and variables to the top of their scope, prior to execution of the code. Let's understand this with an example. Copy and paste this code into your DevTools console

```
const result = add(3, 5);
function add(number1, number2) {
   return number1 + number2;
}
result;
```

Take note that we've called the `add()` function before we've declared it, and declared our `add()` function with a function declaration. Will this code execute properly?

When we hit 'enter' the code executes without error and we get `8` as the result. In this case we can call `add()` before it has been defined because of **hoisting**. This only happens with function declarations.

Now refresh your page and your DevTools console and copy/paste this code and hit enter:

```
const result = add(3, 5);
const add = function(number1, number2) {
   return number1 + number2;
}
result;
```

And what do we get? An error, specifically this one:

```
Uncaught ReferenceError: add is not defined
```

The difference with the second code snippet is that we've defined our `add()` function with a function expression, which does not get hoisted. Code gets read from top to bottom, and in this case when we call `add(3,5)`, our browser's JS interpreter can't find this function because it hasn't been hoisted and literally doesn't exist yet.

Which Method of Defining a Function Should I Use?

Generally speaking you should stick to using function declarations because that code gets hoisted. However, you'll see a lot of examples online that use function expressions, so it's very important for you to be able to recognize and use them.

In this course section, we'll learn how to use function expressions in the context of event handling. We'll get started on this in the next lesson!

Later on, we'll also learn how to use function declarations in event handling. When we get there, we'll cover the differences between the two and when we would choose to use one over the other.

On MDN

There are more ways to define a function than just function declarations and function expressions. If you want to learn more, check out MDN's reference page on functions:

- **Functions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions)**

There's quite a lot on that reference page, including more information and examples about function declarations and expressions. If you find the additional information on MDN confusing or overwhelming, that's entirely normal when you are first starting out. **Take note that everything that you'll need to be successful in terms of writing functions can be found in this section's content,** so there's no immediate need to reference MDN.

Previous (/introduction-to-programming/javascript-and-web-browsers/practice-accessing-html-element-attributes-and-properties-in-the-dom)

Next (/introduction-to-programming/javascript-and-web-browsers/event-handling-with-event-handler-properties)

Lesson 50 of 75

Last updated more than 3 months ago.

disable dark mode

Epicodus (http://www.epicodus.com)