Arrow Functions

- Due No Due Date
- Points 1
- Submitting a website url

 (https://github.com/learn-co-curriculum/phase-1-arrow-functions)  (https://github.com/learn-co-curriculum/phase-1-arrow-functions/issues/new)

Learning Goals

- Review declaring a function using a function expression
- Declare a function using arrow syntax
- Describe situations where arrow functions are used

Introduction

The original style for defining functions in JavaScript is the *function declaration*. But JavaScript has two other ways to write functions: the *function expression* and the *arrow function expression* (often simply called an *arrow function*). In this lesson, we will start by briefly reviewing function expressions, then we will learn how to write functions using *arrow syntax*.

Getting Started

If you haven't already, fork and clone this lab into your local environment. Remember to **fork** a copy into your GitHub account first, then **clone** from that copy. Navigate into its directory in the terminal, then run `code .` to open the files in Visual Studio Code.

Review: Declare a Function Using a Function Expression

The first method we learned for defining a function is the *function declaration*:

```
function foo() {
  return 'bar';
```

```
}
```

But, as we've learned, a function can also be written as follows:

```
const foo = function() {
  return 'bar';
}
```

The `function() {...}` to the right of the assignment operator (`=`) is called a *function expression*. The best way to understand function expressions is by analogy.

```
const sum = 1 + 1
```

Evaluate the expression `1 + 1` , returning `2` , and assign it to the variable `sum` .

```
const difference = 10 - 1;
```

Evaluate the expression `10 - 1` , returning `9` , and assign it to the variable `difference` .

```
const foo = function() {
  return 'bar';
}
```

Evaluate the expression `function() { return 'bar' }` , returning a thing that can be called, and assign it to the variable `foo` .

We've also learned that the function expression (again, the thing to the right of `=`) is known as an *anonymous function*. It doesn't have a name associated with it like you see in a *function declaration*.

However, when we assign an anonymous function to a variable, we have a name that points to a callable thing. We can call this anonymous function by invoking `foo()` . That anonymous function is now, for all practical purposes, named `foo` .

There are a few subtle differences between *function declarations* and *function expressions*, but they are very minute. Neither is really better than the other. Over time, conventions have evolved in the JavaScript programming community for when to use one vs. the other; you will develop a sense

for these as you continue to learn JavaScript. Ultimately, however, you are free to use whichever one you prefer.

Declare a Function Using An Arrow Function

The arrow syntax builds on the syntax of the function expression and provides a shorthand way to declare functions that doesn't require using the `function` keyword. In fact, in cases where the function body consists of one line of code, we can define it in a single line:

```
const add = (parameter1, parameter2) => parameter1 + parameter2;
add(2,3); //=> 5
```

Here, we're declaring a variable `add` and assigning an *anonymous function* as its value. Let's look to the right of the `=` :

```
(parameter1, parameter2) => parameter1 + parameter2;
// Parameter list ^^^^^   // Function Body ^^^^^^^^
```

This is a very short function body! It adds `parameter1` and `parameter2` .

There are a couple of things to be aware of in the code above: first, note that if the function body consists of a single expression, we no longer need to wrap it in curly braces. Second, **when there are no braces, arrow functions have an *implicit* return**, i.e., they *automatically* return the result of the last expression! **This is the only situation in which a JavaScript function doesn't require *explicit return* with the `return` keyword.**

To the left of the `=>` , you see the parameters that are defined for the function. This looks similar to what we would have done with a function declaration: list the parameters, separated by commas, inside of `()` .

If your arrow function has only one parameter, the `()` around the parameter becomes optional:

```
const twoAdder = x => x + 2;
// is the same as
const twoAdder = (x) => x + 2;
```

Almost all developers will drop the parentheses in this case.

If we need to do more work than return a single expression, we'll need `{}` to wrap the multiple lines of code, **and** we'll have to declare a `return`. That sweet no-`return` syntax is only available if your function body is one expression long.

```
const sum = (parameter1, parameter2) => {
  console.log(`Adding ${parameter1}`);
  console.log(`Adding ${parameter2}`);
  return parameter1 + parameter2;
}
sum(1,2); //=> 3
```

Describe Situations Where Arrow Functions Are Used

Arrow functions are often used in JavaScript's *iterator* methods. An iterator is a method that allows you to deal with a set of data one at a time. For example, if you had a group of students' essays, you could only grade them one at a time.

In addition to looping constructs such as `for`, JavaScript includes a number of *advanced iterators*; we'll learn about these later in this section. For now, to see an example of how arrow functions are used in these methods, we'll preview JavaScript's `.map()` method.

The `.map()` method is called on an `Array` and takes a function as an argument. It iterates through the array, passing each element in turn to the function. It then takes that function's return value and adds it to a new array, leaving the original array unchanged. That new array, containing the modified elements, is returned at the end after all iterations are complete.

```
const nums = [1,2,3];
const squares = nums.map(x => x ** 2);
squares; //=> [1,4,9]
nums; //=> [1,2,3]
```

Note that the argument being passed to `map` above is an arrow function! In each iteration through the `nums` array, `map` passes the value of the current element to the arrow function as an argument and it is assigned to the parameter `x`. That value is then squared and stored in a new array. After `map` has iterated through all of the elements, it returns the new array containing the squared values.

If all this math stuff seems a bit too textbook-y, be reassured that we can iterate through anything, not just numbers. In the following example, we can imagine that `overdueTodoItems` is a collection of DOM elements:

```
finishedItems = overdueTodoItems.map( item => item.className = "complete" );
header.innerText = `You finished ${finishedItems.length} items!`;
```

Or we might use `map` in billing software:

```
lapsedUserAccounts.map( u => sendBillTo(u.address) );
```

Don't worry if you don't completely follow everything that goes on here — we will cover advanced iterators in detail later in this section.

Instructions

You are going to write several methods. Write your code in the `index.js` file. Let the tests guide you through the process. Once the tests are passing, remember to commit and push your changes up to GitHub, then submit your work to Canvas using CodeGrade.

Conclusion

In this lesson you saw two different styles for declaring functions: function expressions and arrow functions. Neither is "better" than the standard function declaration we've been using. Arrow functions excel when a simple change or operation needs to be used repeatedly. But they're certainly used to write long, complex functions too! As you continue through the course, you'll see all three methods used to write functions, and develop a feel for when to use each.

Resources

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