**Default Parameters**

One of the features added in ES6 is the ability to use *default parameters*. Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is undefined when called.

Take a look at the code snippet below that uses a default parameter below:

function greeting (name = 'stranger') { console.log(`Hello, ${name}!`) } greeting('Nick') // Output: Hello, Nick! greeting() // Output: Hello, stranger!

* In the example above, we used the =operator to assign the parameter name a default value of 'stranger'. This is useful to have in case we ever want to include a non-personalized default greeting!
* When the code calls greeting('Nick') the value of the argument is passed in and, 'Nick', will override the default parameter of 'stranger' to log 'Hello, Nick!' to the console.
* When there isn't an argument passed into greeting(), the default value of 'stranger'is used, and 'Hello, stranger!' is logged to the console.

By using a default parameter, we account for situations when an argument isn't passed into a function that is expecting an argument.

Let’s practice creating functions that use default parameters.

# Return

When a function is called, the computer will run through the function's code and evaluate the result of calling the function. By default that resulting value is undefined.

function rectangleArea(width, height) { let area = width \* height } console.log(rectangleArea(5, 7)) // Prints undefined

In the code example, we defined our function to calculate the area of a width and heightparameter. Then rectangleArea() is invoked with the arguments 5 and 7. But when we went to print the results we got undefined. Did we write our function wrong? No! In fact, the function worked fine, and the computer did calculate the area as 35, but we didn't capture it. So how can we do that? With the keyword return!

To pass back information from the function call, we use a return statement. To create a return statement, we use the return keyword followed by the value that we wish to return. Like we saw above, if the value is omitted, undefined is returned instead.

When a return statement is used in a function body, the execution of the function is stopped and the code that follows it will not be executed. Look at the example below:

function rectangleArea(width, height) { if (width < 0 || height < 0) { return 'You need positive integers to calculate area!'; } return width \* height; }

If an argument for width or height is less than 0, then rectangleArea() will return 'You need positive integers to calculate area!'. The second return statement width \* height will not run.

The return keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use.

**Helper Functions**

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as *helper functions*. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

If we wanted to define a function that converts the temperature from Celsius to Fahrenheit, we could write two functions like:

function multiplyByNineFifths(number) { return number \* (9/5); }; function getFahrenheit(celsius) { return multiplyByNineFifths(celsius) + 32; }; getFahrenheit(15); // Returns 59

In the example above:

* getFahrenheit() is called and 15 is passed as an argument.
* The code block inside of getFahrenheit()calls multiplyByNineFifths() and passes 15as an argument.
* multiplyByNineFifths() takes the argument of 15 for the number parameter.
* The code block inside of multiplyByNineFifths() function multiplies 15 by (9/5), which evaluates to 27.
* 27 is returned back to the function call ingetFahrenheit().
* getFahrenheit() continues to execute. It adds 32 to 27, which evaluates to 59.
* Finally, 59 is returned back to the function call getFahrenheit(15).

We can use functions to section off small bits of logic or tasks, then use them when we need to. Writing helper functions can help take large and difficult tasks and break them into smaller and more manageable tasks.

# Function Expressions

Another way to define a function is to use a function expression. To define a function inside an expression, we can use the functionkeyword. In a function expression, the function name is usually omitted. A function with no name is called an anonymous function. A function expression is often stored in a variable in order to refer to it.

Consider the following function expression:

To declare a function expression:

1. Declare a variable to make the variable’s name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use const as the keyword to declare the variable.
2. Assign as that variable's value an anonymous function created by using the function keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments being passed into the function.

variableName(argument1, argument2)

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

Let’s define a new function using a function expression.

# Arrow Functions

ES6 introduced arrow function syntax, a shorter way to write function by using the special "fat arrow" () => notation.

Arrow functions remove the need to type out the keyword function every time you need to create a function. Instead, you first include the parameters inside the ( ) and then add an arrow => that points to the function body surrounded in { } like this:

const rectangleArea = (width, height) => { let area = width \* height; return area }

It's important to be familiar with the multiple ways of writing functions because you will come across each of these when reading other JavaScript code.

**Concise Body Arrow Functions**

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as *concise body*. We'll explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.
2. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow => and the return keyword can be removed. This is referred to as *implicit return*.

So if we have a function:

const squareNum = (num) => { return num \* num; };

We can refactor the function to:

const squareNum = num => num \* num;

Notice the following changes:

* The parentheses around num have been removed, since it has a single parameter.
* The curly braces { } have been removed since the function consists of a single-line block.
* The return keyword has been removed since the function consists of a single-line block.

# Review Functions

Give yourself a pat on the back, you just navigated through functions!

In this lesson, we covered some important concepts about functions:

* A function is a reusable block of code that groups together a sequence of statements to perform a specific task.
* A function declaration :
* A parameter is a named variable inside a function's block which will be assigned the value of the argument passed in when the function is invoked:
* To call a function in your code:
* ES6 introduces new ways of handling arbitrary parameters through default parameters which allow us to assign a default value to a parameter in case no argument is passed into the function.
* To return a value from a function, we use a return statement.
* To define a function using function expressions:
* To define a function using arrow function notation:
* Function definition can be made concise using concise arrow notation:

It's good to be aware of the differences between function expressions, arrow functions, and function declarations. As you program more in JavaScript, you'll see a wide variety of how these function types are used.