**What are Functions?**

When first learning how to calculate the area of a rectangle, there’s a sequence of steps to calculate the correct answer:

1. Measure the width of the rectangle.
2. Measure the height of the rectangle.
3. Multiply the width and height of the rectangle.

With practice, you can calculate the area of the rectangle without being instructed with these three steps every time.

We can calculate the area of one rectangle with the following code:

const width = 10;

const height = 6;

const area = width \* height;

console.log(area); // Output: 60

Imagine being asked to calculate the area of three different rectangles:

// Area of the first rectangle

const width1 = 10;

const height1 = 6;

const area1 = width1 \* height1;

// Area of the second rectangle

const width2 = 4;

const height2 = 9;

const area2 = width2 \* height2;

// Area of the third rectangle

const width3 = 10;

const height3 = 10;

const area3 = width3 \* height3;

In programming, we often use code to perform a specific task multiple times. Instead of rewriting the same code, we can group a block of code together and associate it with one task, then we can reuse that block of code whenever we need to perform the task again. We achieve this by creating a *function*. A function is a reusable block of code that groups together a sequence of statements to perform a specific task.

In this lesson, you will learn how to create and use functions, and how they can be used to create clearer and more concise code.

**Instructions**

Take a look at the provided GIF. It shows a function, named addOneSide, adding an additional side to different shape inputs. Notice how there is only one function, represented by the box, that is used to transform individual shapes (inputs) into new shapes (outputs).

**Function Declarations**

In JavaScript, there are many ways to create a function. One way to create a function is by using a *function declaration*. Just like how a variable declaration binds a value to a variable name, a function declaration binds a function to a name, or an *identifier*. Take a look at the anatomy of a function declaration below:

A function declaration consists of:

* The function keyword.
* The name of the function, or its identifier, followed by parentheses.
* A function body, or the block of statements required to perform a specific task, enclosed in the function’s curly brackets, { }.

A function declaration is a function that is bound to an identifier, or name. In the next exercise we’ll go over how to run the code inside the function body.

We should also be aware of the *hoisting* feature in JavaScript which allows access to function declarations before they’re defined.

Take a look at example of hoisting:

console.log(greetWorld()); // Output: Hello, World!

function greetWorld() {

console.log('Hello, World!');

}

Notice how hoisting allowed greetWorld() to be called before the greetWorld() function was defined! Since hoisting isn’t considered good practice, we simply want you to be aware of this feature.

If you want to read more about hoisting, check out [MDN documentation on hoisting](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting).

**Instructions**

**1.**

Let’s create a function that prints a reminder to the console. Using a function declaration, create a function called getReminder().

Hint

Syntax for a function declaration:

function functionName() {}

**2.**

In the function body of getReminder(), log the following reminder to the console: 'Water the plants.'

Hint

To write to the console:

console.log('Insert Message Here');

Include your code between the curly braces:

function functionName() {

// Code body between curly braces

}

**3.**

Let’s create another function that prints a useful Spanish travel phrase to the console.

Using a function declaration, create a function called greetInSpanish().

Hint

Syntax for a function declaration:

function functionName() {}

**4.**

Add code to the function body of greetInSpanish():

* In the function body console.log() the following Spanish phrase to the console: 'Buenas Tardes.'

Hint

To write to the console:

console.log('Insert Message Here');

Include your code between the curly braces:

function functionName() {

// Code body between curly braces

}

function getReminder(){

  console.log('Water the plants.')

}

function greetInSpanish(){

  console.log('Buenas Tardes.')

}

**Calling a Function**

As we saw in previous exercises, a function declaration binds a function to an identifier.

However, a function declaration does not ask the code inside the function body to run, it just declares the existence of the function. The code inside a function body runs, or *executes*, only when the function is *called*.

To call a function in your code, you type the function name followed by parentheses.

This *function call* executes the function body, or all of the statements between the curly braces in the function declaration.

We can call the same function as many times as needed.

Let’s practice calling functions in our code.

**Instructions**

**1.**

Imagine that you manage an online store. When a customer places an order, you send them a thank you note. Let’s create a function to complete this task:

* Define a function called sayThanks() as a function declaration.
* In the function body of sayThanks(), add code such that the function writes the following thank you message to the console when called: 'Thank you for your purchase! We appreciate your business.'

Hint

Syntax for calling a function:

function functionName(){

// Function body

}

To write messages to the console:

console.log('Enter your message here.');

**2.**

Call sayThanks() to view the thank you message in the console.

Hint

Syntax for calling a function:

functionName();

**3.**

Functions can be called as many times as you need them.

Imagine that three customers placed an order and you wanted to send each of them a thank you message. Update your code to call sayThanks() three times.

Hint

The syntax for calling a function once:

functionName();

So we could call that same function three times with the following code:

functionName();

functionName();

functionName();

function sayThanks(){

   console.log

   ('Thank you for your purchase! We appreciate your business.');

}

sayThanks();

sayThanks();

sayThanks();

**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:

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.

**Instructions**

**1.**

The function makeShoppingList() creates a shopping list based on the items that are passed to the function as arguments.

Imagine that you always purchase milk, bread, and eggs every time you go shopping for groceries. To make creating a grocery list easier, let’s assign default values to the parameters in makeShoppingList().

Change the parameters of makeShoppingList() into default parameters :

* Assign ‘milk’ as the default value of item1.
* Assign ‘bread’ as the default value of item2.
* Assign ‘eggs’ as the default value of item3.

function makeShoppingList(item1 = 'milk', item2 = 'bread', item3 = 'eggs'){

  console.log(`Remember to buy ${item1}`);

  console.log(`Remember to buy ${item2}`);

  console.log(`Remember to buy ${item3}`);

}

# 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 height parameter. 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.

**Instructions**

**1.**

Imagine if we needed to order monitors for everyone in an office and this office is conveniently arranged in a grid shape. We could use a function to help us calculate the number of monitors needed!

Declare a function monitorCount() that has two parameters. The first parameter is rows and the second parameter is columns.

Hint

To write a function, use the following syntax:

function functionName(firstParameter, secondParameter){

};

**2.**

Let’s compute the number of monitors by multiplying rows and columns and then returning the value.

In the function body of the function you just wrote, use the return keyword to return rows \* columns.

Hint

To return a value in a function expression, use the return keyword. Take a look at the following syntax:

function functionName(parameterName){

return parameterName;

};

**3.**

Now that the function is defined, we can compute the number of monitors needed. Let’s say that the office has 5 rows and 4 columns.

Declare a variable named numOfMonitors using the const keyword and assign numOfMonitors the value of invoking monitorCount() with the arguments 5 and 4.

Hint

Since you want to save a value, declare a variable named numOfMonitors using the const keyword. We assign its value using the = operator and calling monitorCount(5, 4). monitorCount(5, 4) will get evaluated to 20. So the value of numOfMonitors will be 20. Remember to separate the arguments using a comma.

const numOfMonitors = monitorCount(5, 4);

**4.**

To check that the function worked properly, log numOfMonitors to the console.

Hint

To log the value of a variable to the console, pass in the variable to a console.log() statement.

const hello = "Hello World!";

console.log(hello); // Output: "Hello World!"

function monitorCount(rows, columns) {

  return rows \* columns;

}

const numOfMonitors = monitorCount(5, 4);

console.log(numOfMonitors);

**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 15 as 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 in getFahrenheit().
* 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.

**Instructions**

**1.**

In the previous exercise, we created a function to find the number of monitors to order for an office. Now let’s write another function that uses the monitorCount function to figure out the price.

Below monitorCount Create a function declaration named costOfMonitors that has two parameters, the first parameter is rows and the second parameter is columns. Leave the function body empty for now.

Hint

To write a function declaration, use the following syntax:

function newFunction(firstParameter, secondParameter) {

};

**2.**

Time to add some code to the function body of costOfMonitors to calculate the total cost.

Add a return statement that returns the value of calling monitorCount(rows, columns) multiplied by 200.

Hint

To add a return statement that calls another function inside costOfMonitors, follow the sample syntax:

function newFunction(firstParameter, secondParameter) {

return helperFunction(firstParameter, secondParameter) \* 1000;

};

**3.**

We should save the cost to a variable.

Declare a variable named totalCost using the const keyword. Assign to totalCost the value of calling costOfMonitors() with the arguments 5 and 4 respectively.

Hint

To save the return value of a function to a variable, follow the sample syntax:

const saveValue = newFunction(10, 20);

**4.**

To check that the function worked properly, log totalCost to the console.

function monitorCount(rows, columns) {

  return rows \* columns;

}

function costOfMonitors(rows,columns){

  return monitorCount(rows,columns) \* 200;

}

const totalCost=costOfMonitors(5,4);

console.log(totalCost);

**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 function keyword. 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.

**Instructions**

**1.**

Let’s say we have a plant that we need to water once a week on Wednesdays. We could define a function expression to help us check the day of the week and the plant needs to be watered:

* Create a variable named plantNeedsWater using the const variable keyword.
* Assign an anonymous function that takes in a parameter of day to plantNeedsWater.

Hint

To write a function expression, use the following syntax:

const variableName = function(paramGoesHere){

};

**2.**

Now we need to add some code to the function body of plantNeedsWater():

* In the function body add an if conditional that checks day === 'Wednesday'.
* If the conditional is truthy, inside the if code block, use the return keyword to return true.

Hint

We’re adding an if conditional inside the function body of plantNeedsWater() to check if day === 'Wednesday':

const plantNeedsWater = function(day){

if(day === 'Wednesday'){

}

};

Then, inside the if conditional code block, return true.

const plantNeedsWater = function(day){

if(day === 'Wednesday'){

return true;

}

};

**3.**

On days that aren’t 'Wednesday', plantNeedsWater() should return false:

* Add an else statement after the if statement.
* Inside the else statement use the return keyword to return false.

Hint

First add an else statement after the if statement:

const plantNeedsWater = function(day){

if(day === 'Wednesday'){

return true;

} else {

}

};

Then inside the else statement, return `false.

const plantNeedsWater = function(day){

if(day === 'Wednesday'){

return true;

} else {

return false;

}

};

**4.**

Call the plantNeedsWater() and pass in 'Tuesday' as an argument.

Hint

To call a function, add a set of parentheses to the name of the function and pass in the necessary arguments:

const someFunction = function(foo) {

return foo;

};

// To call the function:

someFunction('an argument');

**5.**

Let’s check that plantNeedsWater() returned the expected value.

Log plantNeedsWater('Tuesday') to the console. If it worked correctly, you should see false logged to the console.

Hint

To log the value of plantNeedsWater('Tuesday') to the console we can use the following syntax:

console.log(plantNeedsWater('Tuesday')); // Output: false

You can also save the value of plantNeedsWater('Tuesday') to a variable and log the variable like so:

const waterCheck = plantNeedsWater('Tuesday');

console.log(waterCheck); // Output: false

const plantNeedsWater = function(day){

  if(day === 'Wednesday'){

    return true;

  }

    return false;

}

console.log(plantNeedsWater('Tuesday'));

# Arrow Functions

ES6 introduced arrow function syntax, a shorter way to write functions 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.

**Instructions**

**1.**

Change plantNeedsWater() to use arrow function syntax.

Hint

An arrow function uses a set of parentheses () followed by an arrow token => followed by the function body like so:

const newFunction = () => {

// Function body goes here

};

const plantNeedsWater = (day) => {

  if (day === 'Wednesday') {

    return true;

  } else {

    return false;

  }

};

const plantNeedsWater = day => day === 'Wednesday' ? true : false;