**Introduction to Functions**

A function is a block of code designed to perform a task.

*Functions* are like recipes. They accept data, perform actions on that data, and return a result. The beauty of functions is that they allow us to write a block of code once, then we can reuse it over and over without rewriting the same code.

Take a look at the code in this exercise. This code turns the calculator on if it is currently off, and turns it off if the calculator is currently on.

See if you can figure out how this code works. In the next exercise, we'll walk through it line by line.

**Instructions**

**1.**

Execute the code. When you think you understand what the code does, proceed to the next exercise.

|  |
| --- |
| let calculatorIsOn = false;  const pressPowerButton = () => {  if (calculatorIsOn) {  console.log('Calculator turning off.');  calculatorIsOn = false;  } else {  console.log('Calculator turning on.');  calculatorIsOn = true;  }  };  pressPowerButton();  // Output: Calculator turning on.  pressPowerButton();  // Output: Calculator turning off. |

|  |
| --- |
| Calculator turning on.  Calculator turning off. |

**Functions**

How does this code work?

let calculatorIsOn = false; const pressPowerButton = () => { if (calculatorIsOn) { console.log('Calculator turning off.'); calculatorIsOn = false; } else { console.log('Calculator turning on.'); calculatorIsOn = true; } }; pressPowerButton(); // Output: Calculator turning on. pressPowerButton(); // Output: Calculator turning off.

Let's explore each part in detail.

1. We created a function named pressPowerButton.

* const pressPowerButton creates a variable with a given name written in *camelCase*.
* The variable is then set equal = to a set of parentheses followed by an arrow token () =>, indicating the variable stores a function. This syntax is known as *arrow function* syntax.
* Finally, between the curly braces {} is the *function body*, or the JavaScript statements that define the function. This is followed by a semi-colon ;. In JavaScript, any code between curly braces is also known as a *block*.

2. Inside the function body is an if/else statement.

3. On the last few lines, we call the function by writing its name followed by a semi-colon pressPowerButton();. This executes the function, running all code within the function body.

4. We executed the code in the function body twice without having to write the same set of instructions twice. Functions can make code reusable!

**1.**

Imagine you work at a pizza restaurant and you want to write a JavaScript program to take orders so you don't have to write orders by hand. You can write a function to perform this task!

Start by writing a function using the keyword const and the name takeOrder. Then set the variable = to a set of parentheses followed by an arrow () =>. Inside of its block {}, use console.log() to print 'Order: pizza'.

Hint

const takeOrder = () => { ... };

**2.**

Under the function, let's take an order.

Call the takeOrder() function on the last line.

Hint

To call a function, type its name followed by a set of parentheses and a semi-colon. You should see the result from the takeOrder() function in the console. The console should read Order: pizza

|  |
| --- |
| const takeOrder = () => {  console.log('Order: pizza');  }  takeOrder(); |

**FUNCTIONS**

**Parameters**

So far our function has not required any input. We can also write functions that accept data. We do this with *parameters*. Parameters are variables in a function definition that represent data we can input into the function.

const multiplyByThirteen = (inputNumber) => { console.log(inputNumber \* 13); }; multiplyByThirteen(9); // Output: 117

Let's explore how this function works:

1. We add inputNumber within the parentheses () => of the multiplyByThirteen function. inputNumber is a parameter.
2. Inside the multiplyByThirteen() function, we use console.log to print the inputNumber multiplied by 13.
3. When we call the multiplyByThirteen() function on the last line, we set the inputNumber parameter. Here, we set it to 9. Then, inside the function block, 9 is multiplied by 13, resulting in 117printing to the console.
4. Note on terminology: inputNumber is a parameter, but when we call multiplyByThirteen(9), the 9 is called an *argument*. In other words, arguments are provided when you call a function, and parameters receive arguments as their value. When we set the value 9 as the argument, we *pass* a value to the function.

Parameters let us write logic inside functions that are modified when we call the function. This makes functions more flexible.

**Instructions**

**1.**

Let's include a parameter in the takeOrder()function to make the orders we log to the console more descriptive.

Inside the parentheses of the takeOrder()function, add a parameter named topping.

Hint

const takeOrder = (topping) => { ... };

**2.**

Now, let's include the topping parameter in the body of the takeOrder function.

Modify the console.log to interpolate the topping parameter to print a string like this:

Order: pizza topped with mushrooms

Hint

You should interpolate the topping parameter, like this:

'Order: pizza topped with ' + topping;

**3.**

At the end of the program, modify the takeOrder() function call to include an argument for topping.

Hint

To define topping when calling the takeOrder()function, include a string in the parentheses of the function call, like this:

takeOrder('mushrooms');

|  |
| --- |
| const takeOrder = (topping) => {  console.log('Order: pizza ' + topping);  };  takeOrder('mushrooms'); |

**Parameters II**

If we can set one parameter, can we set two?

We can set as many parameters as we'd like by adding them when we declare the function, separated by commas, like this:

const getAverage = (numberOne, numberTwo) => { const average = (numberOne + numberTwo) / 2 ; console.log(average); }; getAverage(365, 27); // Output: 196

1. The getAverage() function has two parameters: numberOne and numberTwo, both entered in the parentheses ().
2. When we call the getAverage() function on the last line, we include two numbers as the arguments, also separated by commas.

In this case, we are telling the function to assign numberOne the value of 365 and numberTwo the value of 27. We are passing in 365 and 27 to the getAverage() function.

1. When getAverage() runs, the function knows what numberOne and numberTwo equal since we passed in two arguments when we called the function. Therefore it evaluates (365 + 27) / 2and stores the result in the average variable. When logged to the console, the value of the average is 196.

By adding multiple parameters, we can build functions that are more flexible. Now the function has two variables that we can define when we call the function.

**Return**

Using console.log() as the result of a function isn't the best use of a function. The purpose of a function is to take some input, perform some task on that input, then return a result.

To return a result, we can use the return keyword. Take a look at our function from the last exercise, now re-written slightly:

const getAverage = (numberOne, numberTwo) => { const average = (numberOne + numberTwo) / 2; return average; } console.log(getAverage(365, 27)); // Output: 196

1. Instead of using console.log() inside the getAverage() function, we used the return keyword. return will take the value of the variable and return it.   
  
2. On the last line, we called the getAverage() function inside of a console.log() statement, which outputted the result of 196.   
  
3. This code achieved the same output as before, however now our code is better. Why? If we wanted to use the getAverage() function in another place in our program, we could without printing the result to the console. Using return is generally a best practice when writing functions, as it makes your code more maintainable and flexible.

**Instructions**

**1.**

Now that we have the pizza orders, you want to add them up to find the cost of the pizzas for the check. Let's imagine that each pizza is $7.50, no matter the topping and crust type.

We will need to do three things to write this in JavaScript:

* Create a variable to hold the number of pizzas ordered.
* Whenever a pizza is ordered, add one to the number of pizzas ordered.
* Take the total number of pizzas and multiply them by 7.5, since each pizza is $7.50.

Begin by creating a variable named orderCountset equal to 0 at the top of your code.

Hint

Make sure to use let to declare a variable to define orderCount above the takeOrder() function. We will increment the orderCount variable in the function in the next step.

**2.**

Inside the takeOrder() function, add 1 to the value of orderCount each time takeOrder() is called.

Stuck? Get a hint

**3.**

Now it's time to calculate the subtotal of the pizzas. This is the perfect job for a function.

On a new line, beneath the closing brackets of the takeOrder function, declare a new function named getSubTotal that has one parameter named itemCount.

Hint

The syntax of a function with one parameter looks like this:

const multiplyByThirteen = (inputNumber) => { // ... }

**4.**

Inside the getSubTotal() function's block, use return to output the itemCount multiplied by 7.5.

Hint

Your code may look similar to this:

const multiplyByThirteen = (inputNumber) => { return inputNumber \* 13; }

**5.**

On the last line of your program, after the takeOrder() function calls, call the getSubTotal() function inside a console.logstatement.

getSubTotal has a parameter that represents the number of items ordered. Pass in the orderCount as an argument when making the function call.

Nice work! Now you can see the orders taken and how much it costs.

Hint

To define an argument when calling a function, include it inside the function call's parentheses, like this:

multiplyByThirteen(42);

|  |
| --- |
| let orderCount = 0;  const takeOrder = (topping, crustType) => {  orderCount = orderCount + 1  console.log('Order: ' + crustType + ' pizza topped with ' + topping);    };  const getSubTotal = (itemCount) => {  return itemCount \* 7.5;  }  takeOrder('mushroom', 'thin crust');  takeOrder('spinach', 'whole wheat');  takeOrder('pepperoni', 'brooklyn style');  console.log(getSubTotal(orderCount)); |
| Order: thin crust pizza topped with mushroom  Order: whole wheat pizza topped with spinach  Order: brooklyn style pizza topped with pepperoni  22.5 |

**Return II**

In the last exercise, we pointed out that using returnmakes programs more maintainable and flexible, but how exactly?

When functions return their value, we can use them together and inside one another. If our calculator needed to have a Celsius to Fahrenheit operation, we could write it with two functions like so:

const multiplyByNineFifths = (celsius) => { return celsius \* (9/5); }; const getFahrenheit = (celsius) => { return multiplyByNineFifths(celsius) + 32; }; console.log('The temperature is ' + getFahrenheit(15) + '°F'); // Output: The temperature is 59°F

Take a look at the getFahrenheit() function. Inside of its block, we called multiplyByNineFifths() and passed it the degrees in celsius. The multiplyByNineFifths() function multiplied the celsius by (9/5). Then it returned its value so the getFahrenheit() function could continue on to add 32to it.

Finally, on the last line, we interpolated the function call within a console.log() statement. This works because getFahrenheit() returns its value.

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

**Instructions**

**1.**

It's your job to calculate two more numbers for each order:

* A sales tax of 6% needs to be calculated for every full order. This should be based on the subtotal.
* The total, which is the subtotal plus tax, should also be computed.

Let's start with calculating the tax. Under the getSubTotal() function, create a function expression using the variable const named getTax. This function will take one parameter named itemCount.

Hint

A function declaration looks like this:

const pressPowerButton = () => { ... };

**2.**

Inside the getTax() function, call your getSubTotal() function and pass it the argument itemCount to get the subtotal, and then multiply the returned value by 6% (0.06). Make sure to return the result of this operation.

Hint

Since you have a getSubTotal() function written, you can call that function inside the getTax() function's block. Since getSubTotal() needs an itemCount, make sure to pass it an argument. It may look like this:

const getTax = (someParameter) => { return getSubTotal(someParameter) };

**3.**

Nice work! Now that you calculated the tax, declare another function named getTotal()beneath the getTax() function. The getTotal() function should take in one parameter named itemCount.

Inside the getTotal() function's block, add the subtotal to the tax, then return the result.

Hint

You can utilize the getSubTotal() and getTax()functions in the block of the getTotal() function, since both return their results.

It may look like:

const getTotal = (someParameter) => { return getSubTotal(someParameter) + getTax(someParameter); };

**4.**

On the last line of the program, call the getTotal() function with the argument orderCount inside of a console.log()statement to view the result.

Way to go! You wrote four functions from scratch and even passed them into each other. That's incredible!

Hint

Remember to pass in orderCount when you call getTotal. Put that expression into a console.log to see the cost!

|  |
| --- |
| let orderCount = 0;  const takeOrder = (topping, crustType) => {  orderCount++;  console.log('Order: ' + crustType + ' pizza topped with ' + topping);  };  takeOrder('mushroom', 'thin crust');  takeOrder('spinach', 'whole wheat');  takeOrder('pepperoni', 'brooklyn style');  const getSubTotal = (itemCount) => {  return itemCount \* 7.5;  };  const getTax = (itemCount) => {  return getSubTotal(itemCount)\*0.06  }  const getTotal = (itemCount) => {  return getTax(itemCount) + getSubTotal(itemCount)  }  console.log(getTotal(orderCount)); |
| rder: thin crust pizza topped with mushroom  Order: whole wheat pizza topped with spinach  Order: brooklyn style pizza topped with pepperoni  23.85 |

**Function Declarations**

Now that we have an understanding of functions in JavaScript, let's take a broader look at the type of functions we'll see. Functions in JavaScript are generally declared as either a *function declaration* or a *function expression*.

A *function declaration* is a function that is bound to an identifier or name.

function square (number) { return number \* number; } console.log(square(5)); // Output: 25.

Function declarations require the keyword function, a name, and a function body. You can identify this by the use of function square() and the {} below. Function declarations do not end in a semi-colon.

Let's create a basic function declaration.

**Instructions**

**1.**

In **main.js**, use a function declaration to create a new function called isGreaterThan. The function should take two parameters, numberOne and numberTwo.

Hint

function isGreaterThan(numberOne, numberTwo) { }

**2.**

Inside the function, using an if/elsestatement, create the following logic:

If numberOne is greater than numberTwo, return true.

Otherwise, return false.

Hint

if (some condition) { return true; } else { return false; }

**3.**

Call the isGreaterThan() function with two arguments of your choice.

Hint

isGreaterThan(5, 7);

|  |
| --- |
| function isGreaterThan (numberOne, numberTwo){  if (numberOne>numberTwo){  return true;  }  else{  return false;  }  }  isGreaterThan(5,7) |

# Function Expressions

A function expression is similar to function declaration, with the exception that identifier can be omitted, creating an anonymous function. Function expressions are often stored in a variable. You can identify a function expression by the absence of a function name immediately trailing the function keyword.

const square = function (number) { return number \* number; }; console.log(square(5)); // Output: 25.

Also note function expressions end with a semi-colon since they are stored in a variable.

In this lesson, we have primarily been using a type of function expression known as an arrow function. Arrow function syntax is a shorter syntax for a function expression. You can identify arrow functions through the use of parentheses and the arrow token () =>.

const square = (number) => { return number \* number; }; console.log(square(5)); // Output: 25.

It's important to be familiar with the multiple ways of writing functions, since you will come across these in JavaScript code.

**Instructions**

**1.**

Refactor the function declaration to be a function expression using arrow function syntax.

Be sure to call the function at the end.

Hint

const varName = (x, y) => { // body of function }; varName(1, 2);

|  |
| --- |
| const isGreaterThan = (numberOne, numberTwo) => {  if(numberOne > numberTwo){  return true;  } else {  return false;  }  };  isGreaterThan(4, 8); |

**Arrow Functions**

JavaScript also provides several ways to refactor arrow function syntax. We'll explore a few of these techniques here, using an example function from a previous exercise.

const multiplyByNineFifths = (celsius) => { return celsius \* (9/5); }; const getFahrenheit = (celsius) => { return multiplyByNineFifths(celsius) + 32; }; console.log('The temperature is ' + getFahrenheit(15) + '°F');

We can refactor this function in three ways. The most condensed form of the function is known as *concise body*.

1. Functions that take a single parameter should not use parentheses. The code will still work, but it's better practice to omit the parentheses around single parameters. However, if a function takes zero or multiple parameters, parentheses are required.
2. A function composed of a sole single-line block is automatically returned. The contents of the block should immediately follow the arrow => and the returnkeyword can be removed. This is referred to as *implicit return*.
3. A function composed of a sole single-line block does not need brackets.

In other words, the previous code can be refactored like this:

const multiplyByNineFifths = celsius => celsius \* (9/5); const getFahrenheit = celsius => multiplyByNineFifths(celsius) + 32; console.log('The temperature is ' + getFahrenheit(15) + '°F');

You'll notice:

1. The parentheses around celsius have been removed, since it is a single parameter.
2. The return keyword has been removed since the function consists of a single-line block.
3. The {} have been removed, again, since the function consists of a single-line block.

**Instructions**

**1.**

Refactor the volumeOfSphere function using the principles described above.

Hint

const volumeOfSphere = diameter => (1/6) \* Math.PI \* diameter \* diameter \* diameter; console.log('The volume of the sphere is ' + volumeOfSphere(10) + ' cubic centimeters');

|  |
| --- |
| const volumeOfSphere = diameter => (1/6) \* Math.PI \* diameter \* diameter \* diameter;  console.log('The volume of a sphere is ' + volumeOfSphere(10) + ' cubic centimeters'); |

**Review Functions**

This unit introduced you to functions.

* *Functions* are written to perform a task.
* Functions take data, perform a set of tasks on the data, and then return the result.
* We can define parameters to be used when calling the function.
* When calling a function, we can pass in *arguments*, which will set the function's parameters.
* We can use return to return the result of a function which allows us to call functions anywhere, even inside other functions.