WEB Development Learning Path Notes

JavaScript

# JavaScript fundamentals

## Comments

Programming is often highly collaborative. In addition, our own code can quickly become difficult to understand when we return to it— sometimes only an hour later! For these reasons, it’s often useful to leave notes in our code for other developers or ourselves.

As we write JavaScript, we can write comments in our code that the computer will ignore as our program runs. These comments exist just for human readers.

Comments can explain what the code is doing, leave instructions for developers using the code, or add any other useful annotations.

There are two types of code comments in JavaScript:

1. A single line comment will comment out a single line and is denoted with two forward slashes // preceding it.
2. // Prints 5 to the console

console.log(5);

You can also use a single line comment to comment after a line of code:

console.log(5); // Prints 5

1. A multi-line comment will comment out multiple lines and is denoted with /\* to begin the comment, and \*/ to end the comment.
2. /\*
3. This is all commented
4. console.log(10);
5. None of this is going to run!
6. console.log(99);

\*/

You can also use this syntax to comment something out in the middle of a line of code:

console.log(/\*IGNORED!\*/ 5); // Still just

## Data Types

Data types are the classifications we give to the different kinds of data that we use in programming. In JavaScript, there are seven fundamental data types:

1. Number: Any number, including numbers with decimals: 4, 8, 1516, 23.42.
2. String: Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: ' ... ' or double quotes " ... ". Though we prefer single quotes. Some people like to think of string as a fancy word for text.
3. Boolean: This data type only has two possible values— either true or false (without quotes). It’s helpful to think of booleans as on and off switches or as the answers to a “yes” or “no” question.
4. Null: This data type represents the intentional absence of a value, and is represented by the keyword null (without quotes).
5. Undefined: This data type is denoted by the keyword undefined (without quotes). It also represents the absence of a value though it has a different use than null.
6. Symbol: A newer feature to the language, symbols are unique identifiers, useful in more complex coding. No need to worry about these for now.
7. Object: Collections of related data.

The first 6 of those types are considered primitive data types. They are the most basic data types in the language. Objects are more complex, and you’ll learn much more about them as you progress through JavaScript. At first, seven types may not seem like that many, but soon you’ll observe the world opens with possibilities once you start leveraging each one. As you learn more about objects, you’ll be able to create complex collections of data.

But before we do that, let’s get comfortable with strings and numbers!

console.log('Location of Codecademy headquarters: 575 Broadway, New York City');

console.log(40);

In the example above, we first printed a string. Our string isn’t just a single word; it includes both capital and lowercase letters, spaces, and punctuation.

Next, we printed the number 40, notice we did not use quotes.

## Arithmetic Operators

Basic arithmetic often comes in handy when programming.

An *operator* is a character that performs a task in our code. JavaScript has several built-in in *arithmetic operators*, that allow us to perform mathematical calculations on numbers. These include the following operators and their corresponding symbols:

1. Add: +
2. Subtract: -
3. Multiply: \*
4. Divide: /
5. Remainder: %

The first four work how you might guess:

console.log(3 + 4); // Prints 7

console.log(5 - 1); // Prints 4

console.log(4 \* 2); // Prints 8

console.log(9 / 3); // Prints 3

Note that when we console.log() the computer will evaluate the expression inside the parentheses and print that result to the console. If we wanted to print the characters 3 + 4, we would wrap them in quotes and print them as a string.

The remainder operator, sometimes called *modulo*, returns the number that remains after the right-hand number divides into the left-hand number as many times as it evenly can: 11 % 3 equals 2 because 3 fits into 11 three times, leaving 2 as the remainder.

## String Concatination

Operators aren’t just for numbers! When a + operator is used on two strings, it appends the right string to the left string:

console.log('hi' + 'ya'); // Prints 'hiya'

console.log('wo' + 'ah'); // Prints 'woah'

console.log('I love to ' + 'code.')

// Prints 'I love to code.'

This process of appending one string to another is called *concatenation*. Notice in the third example we had to make sure to include a space at the end of the first string. The computer will join the strings exactly, so we needed to make sure to include the space we wanted between the two strings.

console.log('front ' + 'space');

// Prints 'front space'

console.log('back' + ' space');

// Prints 'back space'

console.log('no' + 'space');

// Prints 'nospace'

console.log('middle' + ' ' + 'space');

// Prints 'middle space'

Just like with regular math, we can combine, or chain, our operations to get a final result:

console.log('One' + ', ' + 'two' + ', ' + 'three!');

// Prints 'One, two, three!'

## Properties

When you introduce a new piece of data into a JavaScript program, the browser saves it as an instance of the data type. Every string instance has a property called length that stores the number of characters in that string. You can retrieve property information by appending the string with a period and the property name:

console.log('Hello'.length); // Prints 5

The . is another operator! We call it the *dot operator*.

In the example above, the value saved to the length property is retrieved from the instance of the string, 'Hello'. The program prints 5 to the console, because Hello has five characters in it.

## Methods

Remember that methods are actions we can perform. JavaScript provides a number of string methods.

We call, or use, these methods by appending an instance with a period (the dot operator), the name of the method, and opening and closing parentheses: e.g. 'example string'.methodName().

Does that syntax look a little familiar? When we use console.log() we’re calling the .log() method on the console object. Let’s see console.log() and some real string methods in action!

console.log('hello'.toUpperCase()); // Prints 'HELLO'

console.log('Hey'.startsWith('H')); // Prints true

Let’s look at each of the lines above:

* On the first line, the .toUpperCase() method is called on the string instance 'hello'. The result is logged to the console. This method returns a string in all capital letters: 'HELLO'.
* On the second line, the .startsWith() method is called on the string instance 'Hey'. This method also accepts the character 'H' as an input, or argument, between the parentheses. Since the string 'Hey' does start with the letter 'H', the method returns the boolean true.

You can find a list of built-in string methods in the [JavaScript documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/prototype). Developers use documentation as a reference tool. It describes JavaScript’s keywords, methods, and syntax.

## Built in Objects

In addition to console, there are other [objects built into JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects). Down the line, you’ll build your own objects, but for now these “built-in” objects are full of useful functionality.

For example, if you wanted to perform more complex mathematical operations than arithmetic, JavaScript has the built-in Math object.

The great thing about objects is that they have methods! Let’s call the .random() method from the built-in Math object:

console.log(Math.random()); // Prints a random number between 0 and 1

In the example above, we called the .random() method by appending the object name with the dot operator, the name of the method, and opening and closing parentheses. This method returns a random number between 0 and 1.

To generate a random number between 0 and 50, we could multiply this result by 50, like so:

Math.random() \* 50;

The example above will likely evaluate to a decimal. To ensure the answer is a whole number, we can take advantage of another useful Math method called Math.floor().

Math.floor() takes a decimal number, and rounds down to the nearest whole number. You can use Math.floor() to round down a random number like this:

Math.floor(Math.random() \* 50);

In this case:

1. Math.random generates a random number between 0 and 1.
2. We then multiply that number by 50, so now we have a number between 0 and 50.
3. Then, Math.floor() rounds the number down to the nearest whole number.

To see all of the properties and methods on the Math object, take a look at [the documentation here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math).

## Variables

There were a lot of changes introduced in the ES6 version of JavaScript in 2015. One of the biggest changes was two new keywords, let and const, to create, or declare, variables. Prior to the ES6, programmers could only use the var keyword to declare variables.

var myName = 'Arya';

console.log(myName);

// Output: Arya

Let’s consider the example above:

1. var, short for variable, is a JavaScript keyword that creates, or declares, a new variable.
2. myName is the variable’s name. Capitalizing in this way is a standard convention in JavaScript called camel casing. In camel casing you group words into one, the first word is lowercase, then every word that follows will have its first letter uppercased. (e.g. camelCaseEverything).
3. = is the assignment operator. It assigns the value ('Arya') to the variable (myName).
4. 'Arya' is the value assigned (=) to the variable myName. You can also say that the myName variable is initialized with a value of 'Arya'.
5. After the variable is declared, the string value 'Arya' is printed to the console by referencing the variable name: console.log(myName).

There are a few general rules for naming variables:

* Variable names cannot start with numbers.
* Variable names are case sensitive, so myName and myname would be different variables. It is bad practice to create two variables that have the same name using different cases.
* Variable names cannot be the same as keywords. For a comprehensive list of keywords check out [MDN’s keyword documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords).

Note: In the next exercises, we will learn why ES6’s let and const are the preferred variable keywords by many programmers. Because there is still a ton of code written prior to ES6, it’s helpful to be familiar with the pre-ES6 var keyword.

If you want to learn more about var and the quirks associated with it, check out the [MDN var documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var).

### Let

As mentioned in the previous exercise, the let keyword was introduced in ES6. The let keyword signals that the variable can be reassigned a different value. Take a look at the example:

let meal = 'Enchiladas';

console.log(meal); // Output: Enchiladas

meal = 'Burrito';

console.log(meal); // Output: Burrito

Another concept that we should be aware of when using let (and even var) is that we can declare a variable without assigning the variable a value. In such a case, the variable will be automatically initialized with a value of undefined:

let price;

console.log(price); // Output: undefined

price = 350;

console.log(price); // Output: 350

Notice in the example above:

* If we don’t assign a value to a variable declared using the let keyword, it automatically has a value of undefined.
* We can reassign the value of the variable.

### Const

The const keyword was also introduced in ES6, and is short for the word constant. Just like with var and let you can store any value in a const variable. The way you declare a const variable and assign a value to it follows the same structure as let and var. Take a look at the following example:

const myName = 'Gilberto';

console.log(myName); // Output: Gilberto

However, a const variable cannot be reassigned because it is *constant*. If you try to reassign a const variable, you’ll get a TypeError.

Constant variables *must* be assigned a value when declared. If you try to declare a const variable without a value, you’ll get a SyntaxError.

If you’re trying to decide between which keyword to use, let or const, think about whether you’ll need to reassign the variable later on. If you do need to reassign the variable use let, otherwise, use const.

## Conditional Statements

In life, we make decisions based on circumstances. Think of an everyday decision as mundane as falling asleep— if we are tired, we go to bed, otherwise, we wake up and start our day.

These if-else decisions can be modeled in code by creating *conditional statements*. A conditional statement checks specific condition(s) and performs a task based on the condition(s).

In this lesson we will explore how programs make decisions by evaluating conditions and introduce logic into our code! We’ll be covering the following concepts:

* if, else if, and else statements.
* comparison operators.
* logical operators.
* truthy vs falsy values.
* ternary operators.
* the switch statement.

### If , else and else if

We often perform a task based on a condition. For example, if the weather is nice today, then we will go outside. If the alarm clock rings, then we’ll shut it off. If we’re tired, then we’ll go to sleep.

In programming, we can also perform a task based on a condition using an if statement:

if (true) {

console.log('This message will print!');

}

// Prints "This message will print!"

Notice in the example above, we have an if statement. The if statement is composed of:

* The if keyword followed by a set of parentheses () which is followed by a *code block*, or *block statement*, indicated by a set of curly braces {}.
* Inside the parentheses (), a condition is provided that evaluates to true or false.
* If the condition evaluates to true, the code inside the curly braces {} runs, or *executes*.
* If the condition evaluates to false, the block won’t execute.

In many cases, we’ll have code we want to run if our condition evaluates to false. If we wanted to add some default behavior to the if statement, we can add an else statement to run a block of code when the condition evaluates to false. Take a look at the inclusion of an else statement:

if (false) {

console.log('The code in this block will not run.');

} else {

console.log('But the code in this block will!');

}

// Prints "But the code in this block will!"

An else statement must be paired with an if statement, and together they are referred to as an if...else statement. In the example above, the else statement:

* Uses the else keyword following the code block of an if statement.
* Has a code block that is wrapped by a set of curly braces {}.
* The code inside the else statement code block will execute when the if statement’s condition evaluates to false.

if...else statements allow us to automate solutions to yes-or-no questions, also known as *binary decisions*.

When writing conditional statements, sometimes we need to use different types of operators to compare values. These operators are called *comparison operators*.

Here is a list of some handy comparison operators and their syntax:

* Less than: <
* Greater than: >
* Less than or equal to: <=
* Greater than or equal to: >=
* Is equal to: ===
* Is NOT equal to: !==

Comparison operators compare the value on the left with the value on the right. For instance:

10 < 12 // Evaluates to true

It can be helpful to think of comparison statements as questions. When the answer is “yes”, the statement evaluates to true, and when the answer is “no”, the statement evaluates to false. The code above would be asking: is 10 less than 12? Yes! So 10 < 12 evaluates to true.

We can also use comparison operators on different data types like strings:

'apples' === 'oranges' // false

In the example above, we’re using the *identity operator* (===) to check if the string 'apples' is the same as the string 'oranges'. Since the two strings are not the same, the comparison statement evaluates to false.

All comparison statements evaluate to either true or false and are made up of:

* Two values that will be compared.
* An operator that separates the values and compares them accordingly (>, <, <=,>=,===,!==).

Working with conditionals means that we will be using booleans, true or false values. In JavaScript, there are operators that work with boolean values known as *logical operators*. We can use logical operators to add more sophisticated logic to our conditionals. There are three logical operators:

* the *and* operator (&&)
* the *or* operator (||)
* the *not* operator, otherwise known as the *bang* operator (!)

When we use the && operator, we are checking that two things are true:

if (stopLight === 'green' && pedestrians === 0) {

console.log('Go!');

} else {

console.log('Stop');

}

When using the && operator, both conditions *must* evaluate to true for the entire condition to evaluate to true and execute. Otherwise, if either condition is false, the && condition will evaluate to false and the else block will execute.

If we only care about either condition being true, we can use the || operator:

if (day === 'Saturday' || day === 'Sunday') {

console.log('Enjoy the weekend!');

} else {

console.log('Do some work.');

}

When using the || operator, only one of the conditions must evaluate to true for the overall statement to evaluate to true. In the code example above, if either day === 'Saturday' or day === 'Sunday' evaluates to true the if‘s condition will evaluate to true and its code block will execute. If the first condition in an || statement evaluates to true, the second condition won’t even be checked. Only if day === 'Saturday' evaluates to false will day === 'Sunday' be evaluated. The code in the else statement above will execute only if both comparisons evaluate to false.

The ! *not operator* reverses, or *negates*, the value of a boolean:

let excited = true;

console.log(!excited); // Prints false

let sleepy = false;

console.log(!sleepy); // Prints true

Essentially, the ! operator will either take a true value and pass back false, or it will take a false value and pass back true.

Logical operators are often used in conditional statements to add another layer of logic to our code.

Let’s consider how non-boolean data types, like strings or numbers, are evaluated when checked inside a condition.

Sometimes, you’ll want to check if a variable exists and you won’t necessarily want it to equal a specific value— you’ll only check to see if the variable has been assigned a value.

Here’s an example:

let myVariable = 'I Exist!';

if (myVariable) {

console.log(myVariable)

} else {

console.log('The variable does not exist.')

}

The code block in the if statement will run because myVariable has a *truthy* value; even though the value of myVariable is not explicitly the value true, when used in a boolean or conditional context, it evaluates to true because it has been assigned a non-falsy value.

So which values are *falsy*— or evaluate to false when checked as a condition? The list of falsy values includes:

* 0
* Empty strings like "" or ''
* null which represent when there is no value at all
* undefined which represent when a declared variable lacks a value
* NaN, or Not a Number

Here’s an example with numbers:

let numberOfApples = 0;

if (numberOfApples){

console.log('Let us eat apples!');

} else {

console.log('No apples left!');

}

// Prints 'No apples left!'

The condition evaluates to false because the value of the numberOfApples is 0. Since 0 is a falsy value, the code block in the else statement will run.

Truthy and falsy evaluations open a world of short-hand possibilities!

Say you have a website and want to take a user’s username to make a personalized greeting. Sometimes, the user does not have an account, making the username variable falsy. The code below checks if username is defined and assigns a default string if it is not:

let defaultName;

if (username) {

defaultName = username;

} else {

defaultName = 'Stranger';

}

If you combine your knowledge of logical operators you can use a short-hand for the code above. In a boolean condition, JavaScript assigns the truthy value to a variable if you use the || operator in your assignment:

let defaultName = username || 'Stranger';

Because || or statements check the left-hand condition first, the variable defaultName will be assigned the actual value of username if is truthy, and it will be assigned the value of 'Stranger' if username is falsy. This concept is also referred to as *short-circuit evaluation*.

### Ternary Operator

In the spirit of using short-hand syntax, we can use a *ternary operator* to simplify an if...else statement.

Take a look at the if...else statement example:

let isNightTime = true;

if (isNightTime) {

console.log('Turn on the lights!');

} else {

console.log('Turn off the lights!');

}

We can use a *ternary operator* to perform the same functionality:

isNightTime ? console.log('Turn on the lights!') : console.log('Turn off the lights!');

In the example above:

* The condition, isNightTime, is provided before the ?.
* Two expressions follow the ? and are separated by a colon :.
* If the condition evaluates to true, the first expression executes.
* If the condition evaluates to false, the second expression executes.

Like if...else statements, ternary operators can be used for conditions which evaluate to true or false.

### Else If

We can add more conditions to our if...else with an else if statement. The else if statement allows for more than two possible outcomes. You can add as many else if statements as you’d like, to make more complex conditionals!

The else if statement always comes after the if statement and before the else statement. The else if statement also takes a condition. Let’s take a look at the syntax:

let stopLight = 'yellow';

if (stopLight === 'red') {

console.log('Stop!');

} else if (stopLight === 'yellow') {

console.log('Slow down.');

} else if (stopLight === 'green') {

console.log('Go!');

} else {

console.log('Caution, unknown!');

}

The else if statements allow you to have multiple possible outcomes. if/else if/else statements are read from top to bottom, so the first condition that evaluates to true from the top to bottom is the block that gets executed.

In the example above, since stopLight === 'red' evaluates to false and stopLight === 'yellow' evaluates to true, the code inside the first else if statement is executed. The rest of the conditions are not evaluated. If none of the conditions evaluated to true, then the code in the else statement would have executed.

### Switch

else if statements are a great tool if we need to check multiple conditions. In programming, we often find ourselves needing to check multiple values and handling each of them differently. For example:

let groceryItem = 'papaya';

if (groceryItem === 'tomato') {

console.log('Tomatoes are $0.49');

} else if (groceryItem === 'papaya'){

console.log('Papayas are $1.29');

} else {

console.log('Invalid item');

}

In the code above, we have a series of conditions checking for a value that matches a groceryItem variable. Our code works fine, but imagine if we needed to check 100 different values! Having to write that many else if statements sounds like a pain!

A switch statement provides an alternative syntax that is easier to read and write. A switch statement looks like this:

let groceryItem = 'papaya';

switch (groceryItem) {

case 'tomato':

console.log('Tomatoes are $0.49');

break;

case 'lime':

console.log('Limes are $1.49');

break;

case 'papaya':

console.log('Papayas are $1.29');

break;

default:

console.log('Invalid item');

break;

}

// Prints 'Papayas are $1.29'

* The switch keyword initiates the statement and is followed by ( ... ), which contains the value that each case will compare. In the example, the value or expression of the switch statement is groceryItem.
* Inside the block, { ... }, there are multiple cases. The case keyword checks if the expression matches the specified value that comes after it. The value following the first case is 'tomato'. If the value of groceryItem equalled 'tomato', that case‘s console.log() would run.
* The value of groceryItem is 'papaya', so the third case runs— Papayas are $1.29 is logged to the console.
* The break keyword tells the computer to exit the block and not execute any more code or check any other cases inside the code block. Note: Without the break keyword at the end of each case, the program would execute the code for all matching cases and the default code as well. This behavior is different from if/else conditional statements which execute only one block of code.
* At the end of each switch statement, there is a default statement. If none of the cases are true, then the code in the default statement will run.

## Functions

### 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).

### Calling Functions

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.

### Parameters and arguments

So far, the functions we’ve created execute a task without an input. However, some functions can take inputs and use the inputs to perform a task. When declaring a function, we can specify its *parameters*. Parameters allow functions to accept input(s) and perform a task using the input(s). We use parameters as placeholders for information that will be passed to the function when it is called.

Let’s observe how to specify parameters in our function declaration:

In the diagram above, calculateArea(), computes the area of a rectangle, based on two inputs, width and height. The parameters are specified between the parenthesis as width and height, and inside the function body, they act just like regular variables. width and height act as placeholders for values that will be multiplied together.

When calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called *arguments*. Arguments can be passed to the function as values or variables.

In the function call above, the number 10 is passed as the width and 6 is passed as height. Notice that the order in which arguments are passed and assigned follows the order that the parameters are declared.

The variables rectWidth and rectHeight are initialized with the values for the height and width of a rectangle before being used in the function call.

By using parameters, calculateArea() can be reused to compute the area of any rectangle! Functions are a powerful tool in computer programming so let’s practice creating and calling functions with parameters.

#### 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.

### 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.

### 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.

### 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.

### 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.

#### Concise 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.

Zero parameters

const functionName = () =>?

One parameter

const functionName = parameter =>?

Two or more parameters

const functionName = (param1, param2) =>?

1. 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*.

One line

const sumNumbers = number => number + number

multi line

const sumNumbers = number =>{

const sum = number + number;

return sum;

}

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.