**Blocks and Scope**

Before we talk more about scope, we first need to talk about *blocks*.

We’ve seen blocks used before in functions and if statements. A block is the code found inside a set of curly braces {}. Blocks help us group one or more statements together and serve as an important structural marker for our code.

A block of code could be a function, like this:

const logSkyColor = () => {

let color = 'blue';

console.log(color); // blue

};

Notice that the function body is actually a block of code.

Observe the block in an if statement:

if (dusk) {

let color = 'pink';

console.log(color); // pink

};

In the next few exercises, we’ll see how blocks define the scope of variables.

**Instructions**

**1.**

At the top of **main.js**, declare a const variable, named city set equal to 'New York City'. This variable will exist *outside* of the block.

**2.**

Below the city variable, write a function named logCitySkyline.

Hint

You can define the logCitySkyline() function using a function expression, arrow function syntax, or a function declaration.

**3.**

Inside of the function body of logCitySkyline(), write another variable using let named skyscraper and set it equal to 'Empire State Building'.

Hint

Make sure you’re declaring the let variable named skyscraper inside the function body of logCitySkyline().

**4.**

Inside the function, include a return statement like this:

return 'The stars over the ' + skyscraper + ' in ' + city;

Hint

You can also use string interpolation to return the string in step 4. Make sure you have the correct amount of whitespace.

**5.**

Beneath the logCitySkyline() function, use console.log() to log the value of logCitySkyline() to the console.

You’ll notice that the logCitySkyline() function is able to access both variables without any problems. In the next exercise we’ll consider why would it be preferable to have one variable outside of a block and the other inside of a block.

const city = 'New York City';

function logCitySkyline(){

  let skyscraper = 'Empire State Building';

  return 'The stars over the ' + skyscraper + ' in ' + city;

}

console.log(logCitySkyline());

# Global Scope

Scope is the context in which our variables are declared. We think about scope in relation to blocks because variables can exist either outside of or within these blocks.

In global scope, variables are declared outside of blocks. These variables are called global variables. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks.

Let’s take a look at an example of global scope:

const color = 'blue'

const returnSkyColor = () => {

return color; // blue

};

console.log(returnSkyColor()); // blue

* Even though the color variable is defined outside of the block, it can be accessed in the function block, giving it global scope.
* In turn, color can be accessed within the returnSkyColor function block.

Let’s work with global variables to see how data can be accessible from any place within a program.

**Instructions**

**1.**

At the top of **main.js**, write three global variables:

* Name the first variable satellite and set it equal to 'The Moon'.
* Name the second variable galaxy and set it equal to 'The Milky Way'.
* Name the third variable stars and set it equal to 'North Star'.

Hint

You can use any keyword to define these variables, though we recommend you use const since these variables will not be reassigned.

**2.**

Below the variables created in the previous step, write a function named callMyNightSky. Inside the function, include a return statement like this:

return 'Night Sky: ' + satellite + ', ' + stars + ', and ' + galaxy;

Hint

You can define the function using a function declaration, function expression, or arrow function notation. Inside the function you may use string concatenation or string interpolation, however, do not change the values of satellite, stars, or galaxy.

Make sure you return a string with the requested capitalization and punctuation.

**3.**

Beneath the callMyNightSky() function, use console.log() to log the value of callMyNightSky() to the console.

You’ll notice that the function block for callMyNightSky() is able to access the global variables freely since the variables are available to all lines of code in the file.

const satellite = 'The Moon';

const galaxy = 'The Milky Way';

const stars = 'North Star';

function callMyNightSky(){

  return 'Night Sky: ' + satellite + ', ' + stars + ', and ' + galaxy;

}

console.log(callMyNightSky());

**Block Scope**

The next context we’ll cover is *block scope*. When a variable is defined inside a block, it is only accessible to the code within the curly braces {}. We say that variable has *block scope* because it is *only* accessible to the lines of code within that block.

Variables that are declared with block scope are known as *local variables* because they are only available to the code that is part of the same block.

Block scope works like this:

const logSkyColor = () => {

let color = 'blue';

console.log(color); // blue

};

logSkyColor(); // blue

console.log(color); // ReferenceError

You’ll notice:

* We define a function logSkyColor().
* Within the function, the color variable is only available within the curly braces of the function.
* If we try to log the same variable outside the function, throws a ReferenceError.

**Instructions**

**1.**

In **main.js**, define a function logVisibleLightWaves().

Stuck? Get a hint

**2.**

Within the logVisibleLightWaves() function, using const, create a variable lightWaves and set it equal to 'Moonlight'.

**3.**

Within the logVisibleLightWaves() function, beneath the lightWaves variable, add a console.log() statement that will log the value of the lightWaves variable when the function runs.

Stuck? Get a hint

**4.**

Call the logVisibleLightWaves() function from outside the function.

Hint

Calling logVisibleLightWaves() will log the value of lightWaves within the scope of the function body of logVisibleLightWaves().

Your code should resemble:

const exampleFunct = () => {

// Function body of exampleFunct

};

exampleFunct();

**5.**

Beneath the function call, log the value of lightWaves to the console from outside the function.

You’ll notice that it logs a ReferenceError since the variable is tied to the block scope of the function!

function logVisibleLightWaves(){

  const lightWaves = 'Moonlight';

  console.log(lightWaves);

}

logVisibleLightWaves();

console.log(lightWaves);

**Scope Pollution**

It may seem like a great idea to always make your variables accessible, but having too many global variables can cause problems in a program.

When you declare global variables, they go to the *global namespace*. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.

*Scope pollution* is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different variables and sets us up for potential accidents. For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code.

Let’s look at an example of scope pollution in practice so we know how to avoid it:

let num = 50;

const logNum = () => {

num = 100; // Take note of this line of code

console.log(num);

};

logNum(); // Prints 100

console.log(num); // Prints 100

You’ll notice:

* We have a variable num.
* Inside the function body of logNum(), we want to declare a new variable but forgot to use the let keyword.
* When we call logNum(), num gets reassigned to 100.
* The reassignment inside logNum() affects the global variable num.
* Even though the reassignment is allowed and we won’t get an error, if we decided to use num later, we’ll unknowingly use the new value of num.

While it’s important to know what global scope is, it’s best practice to not define variables in the global scope.

**Instructions**

**1.**

Let’s see what happens if we create a variable that overwrites a global variable.

Inside the callMyNightSky() function, on the very first line of the function body, assign the variable stars to 'Sirius' as such:

stars = 'Sirius';

**2.**

Outside the function, under the current console.log() statement, add another console.log() statement to log stars to the console.

You’ll notice that the global variable stars was reassigned to 'Sirius'. In other words, we changed the value of the global stars variable but it’s not easy to read what exactly happened. This is bad practice in code maintainability and could impact our program in ways we do not intend.

const satellite = 'The Moon';

const galaxy = 'The Milky Way';

let stars = 'North Star';

const callMyNightSky = () => {

  stars = 'Sirius';

  return 'Night Sky: ' + satellite + ', ' + stars + ', ' + galaxy;

};

console.log(callMyNightSky());

console.log(stars);

**Practice Good Scoping**

Given the challenges with global variables and scope pollution, we should follow best practices for scoping our variables as tightly as possible using block scope.

Tightly scoping your variables will greatly improve your code in several ways:

* It will make your code more legible since the blocks will organize your code into discrete sections.
* It makes your code more understandable since it clarifies which variables are associated with different parts of the program rather than having to keep track of them line after line!
* It’s easier to maintain your code, since your code will be modular.
* It will save memory in your code because it will cease to exist after the block finishes running.

Here’s another example of how to use block scope, as defined within an if block:

const logSkyColor = () => {

const dusk = true;

let color = 'blue';

if (dusk) {

let color = 'pink';

console.log(color); // pink

}

console.log(color); // blue

};

console.log(color); // ReferenceError

Here, you’ll notice:

* We create a variable dusk inside the logSkyColor() function.
* After the if statement, we define a new code block with the {} braces. Here we assign a new value to the variable color if the if statement is truthy.
* Within the if block, the color variable holds the value 'pink', though outside the if block, in the function body, the color variable holds the value 'blue'.
* While we use block scope, we still pollute our namespace by reusing the same variable name twice. A better practice would be to rename the variable inside the block.

Block scope is a powerful tool in JavaScript, since it allows us to define variables with precision, and not pollute the global namespace. If a variable does not need to exist outside a block— it shouldn’t!

**Instructions**

**1.**

Inside the function body of logVisibleLightWaves(), beneath the region variable and before the provided console.log() statement, create an if statement that checks if the region is the 'The Arctic'.

Stuck? Get a hint

**2.**

Inside the if block, define a new let variable lightWaves and set it equal to 'Northern Lights'.

**3.**

Beneath the variable in the if block, use console.log() to log the value of the block variable inside the if block.

Run your code and notice the output. Inside the if block console.log(lightWaves) logs the value Northern Lights to the console. Outside the if block, but still within the function, the same statement logs Moonlight to the console.

const logVisibleLightWaves = () => {

  let lightWaves = 'Moonlight';

  let region = 'The Arctic';

  // Add if statement here:

  if(region === 'The Arctic'){

    let lightWaves = 'Northern Lights';

    console.log(lightWaves);

  }

  console.log(lightWaves);

};

logVisibleLightWaves();

# Review: Scope

In this lesson, you learned about scope and how it impacts the accessibility of different variables.

Let’s review the following terms:

* **Scope** is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
* **Blocks** are statements that exist within curly braces {}.
* **Global scope** refers to the context within which variables are accessible to every part of the program.
* **Global variables** are variables that exist within global scope.
* **Block scope** refers to the context within which variables that are accessible only within the block they are defined.
* **Local variables** are variables that exist within block scope.
* **Global namespace** is the space in our code that contains globally scoped information.
* **Scope pollution** is when too many variables exist in a namespace or variable names are reused.

As you continue your coding journey, remember to use best practices when declaring your variables! Scoping your variables tightly will ensure that your code has clean, organized, and modular logic.