**Scope**

*Scope* refers to where a variable can be accessed in a program. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context. Scope depends entirely on where a variable is declared.

You can think of scope like the view of the night sky from your window. Everyone who lives on the planet Earth is in the global scope of the stars. The stars are accessible *globally*. Meanwhile, if you live in a city, you may see the city skyline or the river. The skyline and river are only accessible *locally* in your city, but you can still see the stars that are available *globally*.

We'll learn more about scope in this lesson through the use of variables.

**Global Scope**

We'll start with *global scope*. Variables defined in the global scope are declared outside of a set of curly braces {}, referred to as a *block*, and are thus available throughout a program. We'll cover more on blocks in subsequent exercises.

Let's take a look at an example of global scope.

const color = 'blue' const colorOfSky = () => { return color; // blue }; console.log(colorOfSky()); // blue

1. Here the variable color is declared *outside* of the function block, giving it global scope.
2. In turn, color can be accessed within the colorOfSky function.

Global variables make data accessible from any place within a program.

**Instructions**

**1.**

At the top of **sky.js**, write two global variables using const, one named satellite set equal to 'The Moon', the other named galaxy set equal to 'The Milky Way'.

**2.**

Using let, write another global variable named stars and set it equal to 'North Star'.

**3.**

Below these variables, using const, write a function named myNightSky. Inside the function, include a return statement like this:

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

**4.**

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

You'll notice that the myNightSky()function is able to access the global variables without any problem since the variables are available *globally*.

|  |
| --- |
| const satellite = 'The Moon'  const galaxy = 'The Milky Way'  let stars = 'North Star'  const myNightSky = () => {  return 'Night Sky: ' + satellite + ', ' + stars + ', ' + galaxy;  }  console.log(myNightSky()) |

**Global Scope II**

While it's important to know what global scope is, it's better to avoid defining variables in the global scope. Globally scoped variables can collide with variables that are more locally scoped, causing unexpected behavior in our code.

Let's explore our program a little further.

**Instructions**

**1.**

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

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

stars = 'Sirius';

**2.**

Outside the function, under the previous console.log() statement, console.log() the value of stars.

You'll notice that the global variable stars was reassigned to 'Sirius'. In other words, we unexpectedly changed the value of the global variable, and this could impact our program in ways we do not intend.

|  |
| --- |
| const satellite = 'The Moon';  const galaxy = 'The Milky Way';  let stars = 'North Star';  const myNightSky = () => {  stars = 'Sirius';  return 'Night Sky: ' + satellite + ', ' + stars + ', ' + galaxy;  };  console.log(myNightSky());  console.log(stars); |
| Night Sky: The Moon, Sirius, The Milky Way  Sirius |

# Block Scope

Because of the challenges with global scope, it is preferable to define variables in block scope.

A block refers to the {} braces of a function, a loop, or an if statement, and serves as an important structural marker for our code. Block scope means that a variable defined in the block is only accessible within the curly braces.

Block scope works like this:

const colorOfSky = () => { let color = 'blue'; console.log(color); // blue }; colorOfSky(); // blue console.log(color); // ReferenceError

You'll notice:

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

**Block Scope II**

Let's take a look at another example of block scope, as defined within an if block:

const colorOfSky = () => { const dusk = true; let color = 'blue'; if (dusk) { let color = 'pink'; console.log(color); // pink } console.log(color); // blue }; colorOfSky(); // blue console.log(color); // ReferenceError

Here, you'll notice:

1. We create a variable dusk inside the colorOfSky() function.
2. After the if statement, we define a new code block with the {} braces. Here we assign a new value to the variable colorif the if statement is true.
3. Within the if block, the color variable holds the value pink, though outside the if block, in the function body, the colorvariable holds the value blue.

Block scope is a powerful tool in JavaScript, since it allows us to define variables with precision, and not pollute the global namespace.

**Block Scope III**

Let's take a look at one other common example of block scope, as defined within a for loop.

const cloudCount = () => { let i = 2; console.log(i); // 2 for (let i = 0; i < 10; i++) { console.log(i); // Numbers from 0 to 9 } }; cloudCount(); console.log(i); // ReferenceError

1. Here the variable i is defined in the cloudCount() function.
2. Within the for loop block, we again define i, as a value that will be incremented.
3. The local value of i, whether defined in the function block or the for loop, has no impact on the global scope of our program.

**Instructions**

**1.**

Using const, declare a new function called starCount().

**2.**

Within the starCount() function, declare a let variable named i and set it equal to 5.

**3.**

Right beneath the variable declaration, log the value of the i value to the console.

**4.**

Beneath the previous console.log()statement, create a for loop.

The for loop should begin counting when i = 0, as long as i < 12, and increment the value i by 1 each time the loop runs.

Within the block of the for loop, log the value of i to the console, as demonstrated in the example.

Hint

const starCount = () => { // ... for (let i = 0; i < 12; i++) { console.log(i); // All numbers from 0 to 11 } };

**5.**

Call starCount() function, from outside of the function.

**6.**

Finally, beneath the function call, log the value of i to the console from outside of the function.

You will notice that it throws a Reference Error! The value of i is contained in the block scope.

|  |
| --- |
| const starCount = () => {  let i =5;  console.log(i);  for (i = 0;i<12;i++){  console.log(i);  }  }  starCount();  //console.log(i); |
| 5  0  1  2  3  4  5  6  7  8  9  10  11 |

**Review: Scope**

This unit introduced you to scope.

* *Scope* is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
* *Global Scope* refers to variables that are accessible to every part of the program.
* *Block Scope* refers to variables that are accessible only within the block they are defined.