**Scope**

An important idea in programming is *scope*. Scope defines where variables can be accessed or referenced. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context.

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.

Over the next few exercises, we'll explore how scope relates to variables and learn best practices for variable declaration.

**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.

# 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.

**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.

**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.

**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 }; logSkyColor(); // 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 colorif 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!

**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.