JavaScript Scope

The Scope Chain

Learning Objective: By the end of this lesson, students will be able to describe how JavaScript's scope chain locates variables and functions, and identify common errors in variable access across different scopes.

What is the scope chain?

The scope chain is a mechanism that allows JavaScript to find variables and functions when they are referenced in code. It's a chain of scopes, where each scope is a collection of variables and functions accessible within that scope.

When a variable or function is referenced in code, the JavaScript engine searches the scope chain for that variable or function, starting from the innermost scope and working its way out. If the variable or function is not found in any of the scopes, the app will crash due to a `ReferenceError`.

Let's see the scope chain in action:

```
// Global scope
let friendName = "Burt";

const greet = () => {
  // Function scope
  let message = "Hello, " + friendName + "!";
  console.log(message);
}

greet();
```

When this code is run, JavaScript will first look for the value of `friendName` in the `greet()` function, the scope where it is used. When it doesn't find it there, JavaScript will look one level up in the **global scope** where it will find `friendName` declared and set to the value "Burt".

In the example above, the `friendName` variable is defined in global scope, making it accessible from anywhere in the code. The `message` variable is declared in the function scope of the `greet()` function, so it's only accessible inside of that function.

You can go up the scope chain but not down it!

A key takeaway is that functions have access to the set of variables and functions defined within their own scope AND in the **outer** scopes. When a line of code accesses a variable (or function), JS will traverse up the scope chain until it finds what it's looking for.

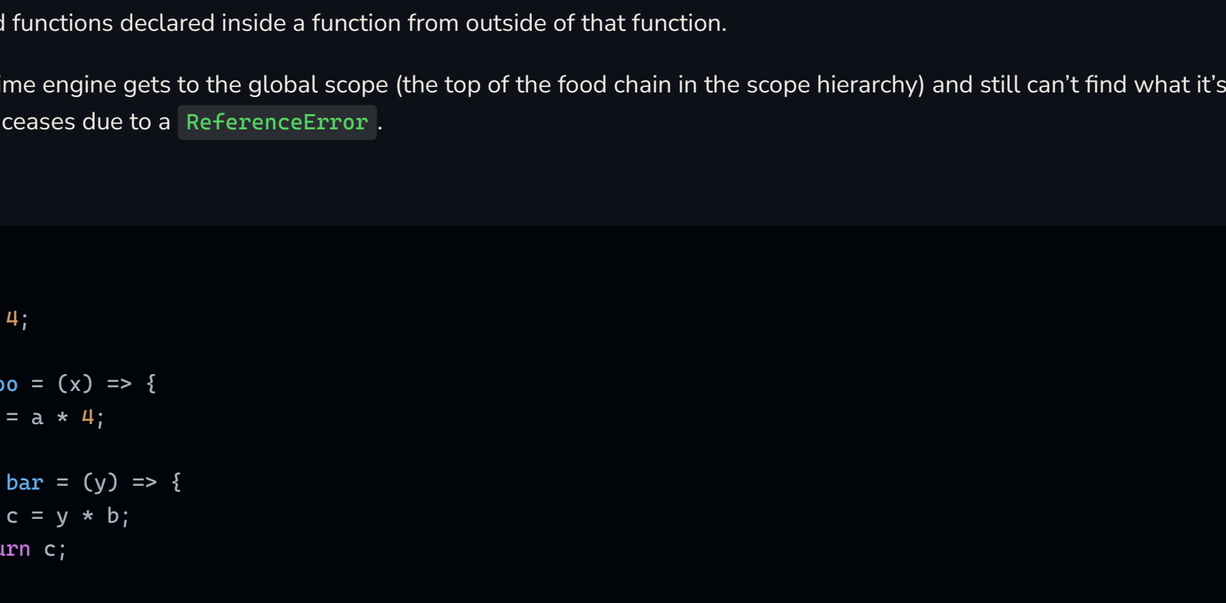
This means we can access variables and functions declared in the global scope from within any function. However, we cannot access variables and functions declared inside a function from outside of that function.

If the JS runtime engine gets to the global scope (the top of the food chain in the scope hierarchy) and still can't find what it's looking for, our program ceases due to a `ReferenceError`.

Try it out!

```
let a = 4;

const foo = (x) => {
  let b = a * 4;

  const bar = (y) => {
    let c = y * b;
    return c;
  }

  return bar(b);
}

console.log(foo(a));
```

Here's a diagram to help visualize how the different scopes in the code above are interacting with one another:



This diagram identifies three different scopes and the identifiers (variables and functions) that live within each scope.

> ❓ Does the above function `foo` have access to the variable `c`?