JavaScript Scope

Concepts

Learning Objective: By the end of this lesson, students will be able to clearly define lexical scope in JavaScript, illustrating its significance in structuring and accessing variables within code.

What is scope?

Think of scope in JavaScript as a set of rules for accessing tools (variables and functions) in a workshop. Only certain tools are available for use in certain places, and are separated by physical barriers. In JavaScript, the location of your code (like where you put a tool) determines what you can access. A tool placed in a special room can't be used in the main workshop area.

JavaScript scope is *lexical*, meaning the code's physical structure determines scope. When a line of code doesn't have access to a variable or function, we can describe that variable or function as being "out of scope".

> 📚 In programming languages with *lexical* scope, like JavaScript, the location of your variables and functions within your code determines where they can be used.

Just as the placement of tools in a workshop dictates *where* they can be used, the placement of variables within the code defines their accessibility. In JavaScript, curly braces `{}` act like walls or dividers in the workshop. They create distinct areas or zones. Variables declared inside these braces belong to that specific area and can't be accessed outside of it, just like tools stored in a locked cabinet can't be used elsewhere in the workshop.

Types of scope in JavaScript

JavaScript has three types of scope:

- **Global Scope**: This is like the main area of the workshop, where tools are available to everyone, no matter where they are in the workshop.
- **Function Scope**: Consider this a special room in the workshop. The tools found here are only available to those working in this room.
- **Block Scope**: Imagine a toolbox inside the special room. The tools in this box are only available to those using this particular toolbox.

Why does scope matter?

Scope is an essential concept because it determines exactly what variables you can access and where. At first, this may feel limiting, but it has many benefits for us as developers:

- Variable isolation: A variable's use is confined to its scope, making it easier to reduce unintended side effects.
- Code readability: Because a variable is confined to its scope, we can safely reuse the same variable name in different scopes. This can greatly assist readability. For example, if we have multiple loops, we can safely use the `i` variable name in each one.
- Memory efficiency: When a variable is no longer in scope, it can be automatically removed from a computer's memory. We, as developers, don't have to think about or manage this process. This is known as garbage collection and is a benefit in higher-level languages like JavaScript.

Why the different types of scope?

There's a concept in programming known as the principle of least privilege. This principle is based on the idea that limiting the accessibility of variables (and functions) helps reduce bugs in the code - think of it as a form of *code safety*.

> 📚 *Code safety* refers to practices that help to prevent or minimize errors, vulnerabilities, and unexpected behavior in a program, ensuring its stability, security, and reliability.