JavaScript Scope

What is Scope?

Scope determines where variables, functions, and objects are accessible in your code. It defines the visibility and lifetime of these identifiers.

Global Scope

Variables declared outside any function or block have global scope. They're accessible from anywhere in your code.

```
var globalVar = "I'm global";
let globalLet = "Also global";

function example() {
  console.log(globalVar); // Accessible here
}

console.log(globalVar); // And here
```

In browsers, global variables become properties of the window object. However, overusing global scope can lead to naming conflicts and makes code harder to maintain.

Local (Function) Scope

Variables declared inside a function are locally scoped to that function. They exist only within that function and can't be accessed from outside.

```
function myFunction() {
  var localVar = "I'm local";
  let anotherLocal = "Me too";

  console.log(localVar); // Works fine
}

myFunction();
console.log(localVar); // ReferenceError: localVar is not defined
```

Each function creates its own scope, and nested functions can access variables from their parent function's scope (this is called lexical scoping or closure).

Block Scope

Introduced with ES6, let and const create block-scoped variables. A block is any code wrapped in curly braces, like if statements, loops, or standalone blocks.

```
if (true) {
  var notBlockScoped = "I escape!";
  let blockScoped = "I'm trapped";
  const alsoBlockScoped = "Me too";
}

console.log(notBlockScoped); // Works - var ignores block scope
console.log(blockScoped); // ReferenceError: blockScoped is not defined
```

```
This also applies to loops:
```

```
for (let i = 0; i < 3; i++) {
  // i is scoped to this loop
}
console.log(i); // ReferenceError

for (var j = 0; j < 3; j++) {
  // j escapes the loop
}
console.log(j); // 3
```

Key Differences Between var, let, and const

The main distinction is that var is function-scoped (ignores block boundaries), while let and const are block-scoped. This makes let and const more predictable and is why they're generally preferred in modern JavaScript. Function scope applies to all three declaration types, creating a local scope that prevents outside access.

Variables and Functions: Scope in Action

Variables Defined Within a Function

A variable defined inside a function does **not** exist outside that function. It's confined to the function's local scope:

```
function myFunction() {
  let insideVar = "I'm inside";
  console.log(insideVar); // Works fine
}

myFunction();
console.log(insideVar); // ReferenceError: insideVar is not defined
```

The variable insideVar is created when the function runs and destroyed when it finishes. The outside world has no access to it.

Variables Defined Before the Function

If you define a variable **before** a function, it exists in the outer scope and **is accessible** inside the function:

```
let outsideVar = "I'm outside";

function myFunction() {
  console.log(outsideVar); // "I'm outside" - works!
}

myFunction();
console.log(outsideVar); // "I'm outside" - also works!
```

The function can "see" variables from its parent scope thanks to **lexical scoping**.

Variables Defined After the Function

If you define a variable **after** a function declaration (but before calling it), it depends on whether it's in the same scope:

```
function myFunction() {
  console.log(afterVar); // Can access it
}

let afterVar = "Defined after";
myFunction(); // "Defined after"
```

However, **be careful with hoisting and the temporal dead zone** with let/const.

Understanding Scope Through Examples

Scope is fundamental to understanding how JavaScript manages variable accessibility. Here's the core principle:

- **Functions create boundaries**: Variables inside can't escape out
- **Functions can look outward**: Variables outside can be accessed inside
- **Block scope adds precision**: let and const respect block boundaries, while var does not

This concept of scope determines which variables are visible where in your code, making it one of the most important concepts to master in JavaScript.