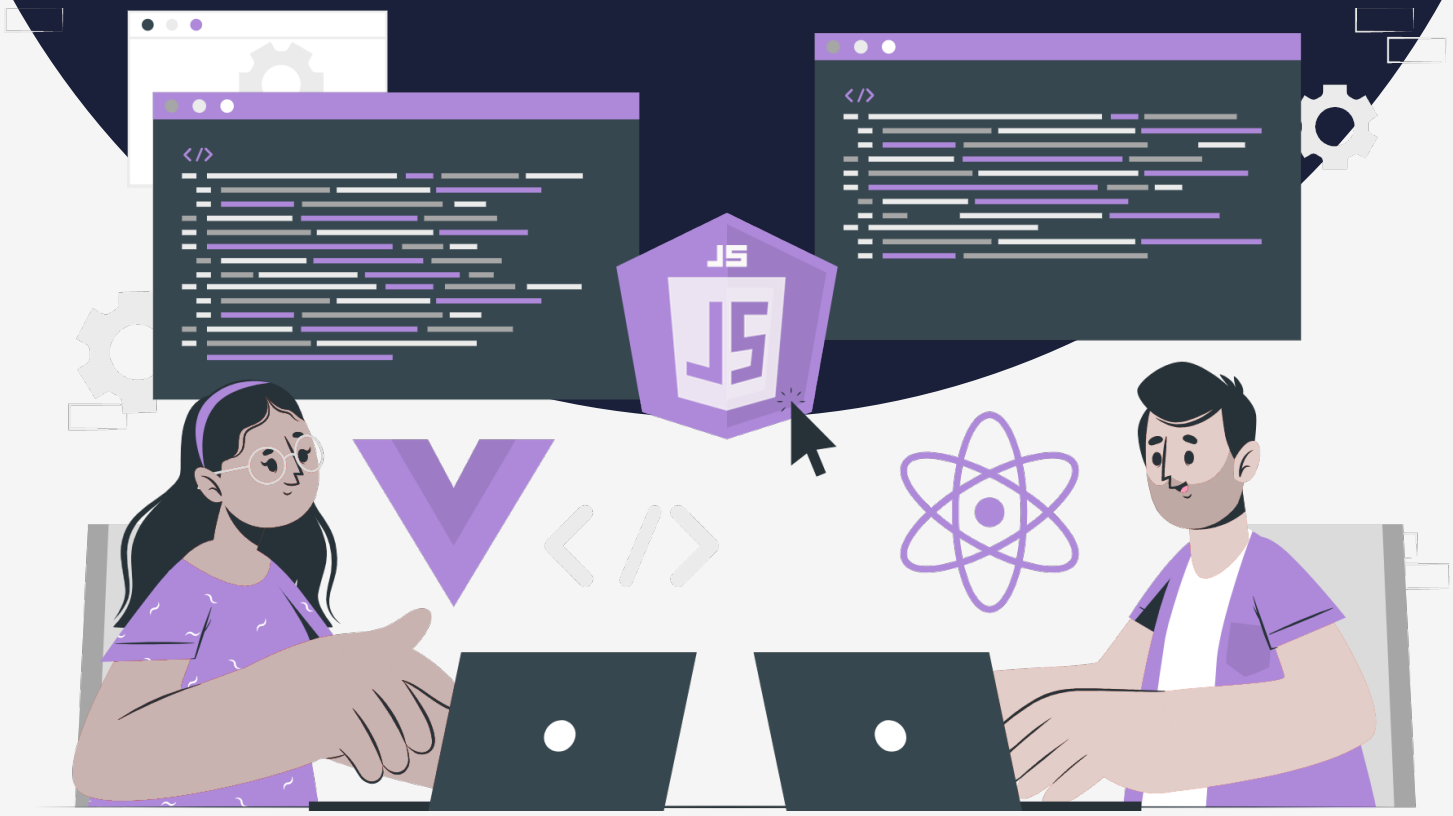
Lesson:

Scope / Lexical Scope / Block Scope



PW SKILLS

Topics Covered:

1. Introduction to Scope.
2. Global Scope.
3. Local Scope.
4. Lexical Scope.
5. Block Scope.

Scope in JavaScript refers to the area of a program where a particular variable or function ca n be accessed. In the programs we write, we declare several variables and functions. The scope determines where in your code a variable or function can be used, and which parts of the program can access it.

In JavaScript, there are two main types of scope: global scope and local scope. A variable with global scope is available throughout the entire program, while a variable with a local scope is only available within the block of code in which it was defined.

Let's look at them one by one in this lecture.

Global Scope
Global scope is like a public space that anyone can access. A variable declared in global scope is available to all parts of the program, including functions and other blocks of code.

Variables with a global scope are usually declared outside of any functions or blocks of code, making them available to the entire program. This can be convenient when you need to use a variable in multiple parts of your program, but it can also make your code harder to manage and prone to conflicts.

One important thing to keep in mind when using global variables is that they can be accessed and modified by any part of the program, which can make it difficult to track down errors. For this reason, it's generally a good practice to minimize the use of global variables and to use local variables whenever possible.

```
var name = "PW Skills";

function printName() {
  console.log(name);
}

printName();
console.log(name);

/**
OUTPUT:

PW Skills
PW Skills

 */
```

In the above code, the variable name has global scope. This means that it can be accessed from anywhere in the program, including the printName function.

When the printName function is called, it logs the value of the name variable to the console, which is "PW Skills". Because the name has global scope, it's available to the printName function even though it's defined outside of that function.

After that, the value of the name is also logged to the console, which again outputs "PW Skills", as the variable is accessible from the global scope.

Local Scope

Local scope, on the other hand, is like a private space that is only available within a specific block of code, such as a function. Variables declared within a function have local scope and can only be accessed within that function. This helps to prevent naming conflicts and makes your code easier to manage, but it can also make it more difficult to reuse code across different parts of your program.

Variables with the local scope are usually declared inside a function or block of code. This makes them available only to that specific function or block of code, and not to the rest of the program.

Another benefit of local scope is that it can help to prevent bugs and errors in your code. By keeping variables and functions within a specific block of code, you can help to ensure that they're used only in the way they were intended.

```
function printName() {
  var name = "PW Skills";
  console.log(name);
}

printName();
console.log(name);

/**
OUTPUT:

PW Skills
ReferenceError: name is not defined

 */
```

In the above code, the name variable is declared inside the printName function using the var keyword. This means that it has local scope and is accessible only within the printName function.

When the printName function is called, it logs the value of the name variable to the console, which is "PW Skills". However, when the console.log(name) statement is executed outside of the printName function, it generates a ReferenceError, because the name variable is not defined in the global scope.

This error occurs because the name has local scope and can be accessed only within the printName function. Once the function has completed execution, the name variable is no longer available.

Lexical Scope

Global and local scopes are useful, but they can become difficult to manage in larger programs with many nested functions. To solve this, the lexical scope was introduced to help manage the complexity of larger JavaScript programs.

With the lexical scope, a function can access variables defined in its own scope, as well as variables defined in its parent scope. This means that you can create nested functions the child function or the nested functions gets access to the scope of their parent functions, making it easier to manage variables and functions in a large program.

Imagine you have a program with several nested functions. Each function has its own local scope, which can make it difficult to keep track of which variables are available where. With the lexical scope, you can define variables in the parent function's scope and make them available to all the child functions.

```
function parentFunction() {
  var parentVariable = "I am a variable declared in parent function.";

  function childFunction() {
    var childVariable = "I am a variable declared in child function.";

    console.log(parentVariable);
    console.log(childVariable);
  }

  childFunction();
}

parentFunction();

/*

OUTPUT:

I am a variable declared in parent function.
I am a variable declared in child function.

*/
```

In the above code, parentFunction defines a variable parentVariable and a nested function childFunction. childFunction defines a variable childVariable.

When childFunction is called from within parentFunction, it has access to parentVariable because parentVariable is defined in the parent scope of childFunction. This is an example of lexical scope, where a nested function can access variables in its parent scope.

When childFunction is executed, it logs the value of parentVariable and childVariable to the console. parentVariable is accessible within childFunction because it is defined in the parent scope. This is how lexical scope works.

Block Scope

A block is a set of statements or codes enclosed within a pair of curly braces {}. A block can contain zero or more statements and can be used to group statements together and execute them as a single unit.

Block scope in JavaScript refers to the visibility or accessibility of variables or functions that are declared within a pair of curly braces, typically used to enclose statements or code blocks.

A block can be used in many places where a single statement is expected, such as in an if statement, a for loop, or a function declaration. A variable or function declared within a block can only be accessed within that block or within a nested block. This is valid only for variables declared with let and const. We will be looking at the difference between var, let, and const in upcoming lectures.

```
function printValues() {
  let a = 1;
  if (a == 1) {
    let b = 2;
    let c = 3;
    console.log(a, b, c); // OUTPUT: 1  2 3
  }
  console.log(a); // OUTPUT: 1
  console.log(b); // OUTPUT: ReferenceError: b is not defined
  console.log(c); // OUTPUT: ReferenceError: b is not defined
}

printValues();
```

Inside the printValues function, we first declare a variable a with the value of 1 using the let keyword.

Next, we enter an if block that checks if a is equal to 1. Inside the if block, we declare two more variables b and c using the let keyword, and assign them the values of 2 and 3, respectively. These variables are only accessible within the if block, and cannot be accessed outside of it.

Then we use console.log to print out the values of a, b, and c. Since all three variables are declared within the same block, we can access all of them from within the block, and the output is 1 2 3.

After that, we exit the if block and try to print out the values of a, b, and c again. However, since b and c were declared within the if block, they are not accessible outside of it, and trying to access them results in a ReferenceError. Only a, which was declared outside of the if block, is accessible outside of it, and its value is still 1, as expected.