**What is Scope?**

Scope in JavaScript refers to the accessibility or visibility of variables. That is, which parts of a program have access to the variable or where the variable is visible.

**Why is Scope Important?**

1. The main benefit of scope is security. That is, the variables can be accessed from only a certain area of the program. Using scope, we can avoid unintended modifications to the variables from other parts of the program.
2. The scope also reduces the namespace collisions. That is, we can use the same variable names in different scopes.

**Types of Scope**

There are three types of scope in JavaScript — 1) **Global Scope,**2)**Function Scope,**and,3)**Block Scope.**

**1. Global Scope**

Any variable that’s not inside any function or block (a pair of curly braces), is inside the global scope. The variables in global scope can be accessed from anywhere in the program. For example:

var greeting = 'Hello World!';function greet() {  
 console.log(greeting);  
}// Prints 'Hello World!'  
greet();

**2. Local Scope or Function Scope**

Variables declared inside a function is inside the local scope. They can only be accessed from within that function, that means they can’t be accessed from the outside code. For example:

function greet() {  
 var greeting = 'Hello World!';  
 console.log(greeting);  
}// Prints 'Hello World!'  
greet();// Uncaught ReferenceError: greeting is not defined  
console.log(greeting);

**3. Block Scope**

ES6 introduced **let** and **const** variables, unlike **var** variables, they can be scoped to the nearest pair of curly braces. That means, they can’t be accessed from outside that pair of curly braces. For example:

{  
 let greeting = 'Hello World!';  
 var lang = 'English';  
 console.log(greeting); // Prints 'Hello World!'  
}// Prints 'English'  
console.log(lang);// Uncaught ReferenceError: greeting is not defined  
console.log(greeting);

We can see that var variables can be used outside the block, that is, var variables are not block scoped.

**Nested Scope**

Just like functions in JavaScript, a scope can be nested inside another scope. For example:

var name = 'Peter';

function greet() {  
 var greeting = 'Hello';  
 {  
 let lang = 'English';  
 console.log(`${lang}: ${greeting} ${name}`);  
 }  
}greet();

Here we have 3 scopes nested within each other. First, the block scope (created due to the let variable) is nested inside the local or function scope which is in turn nested inside the global scope.

**Lexical Scope**

Lexical Scope (also known as Static Scope) literally means that scope is determined at the [lexing time](https://en.wikipedia.org/wiki/Lexical_analysis" \t "_blank) (generally referred to as compiling) rather than at runtime. For example:

let number = 42;

function printNumber() {  
 console.log(number);  
}function log() {  
 let number = 54;  
 printNumber();  
}// Prints 42  
log();

Here the console.log(number) will always print 42 no matter from where the function printNumber() is called. This is different from languages with the **dynamic scope** where the console.log(number) will print different value depending on from where the function printNumber() is called.

If the above code was written in a language that supports dynamic scoping, the console.log(number) would have printed 54 instead.

Using lexical scope we can determine the scope of the variable just by looking at the source code. Whereas in the case of dynamic scoping the scope can’t be determined until the code is executed.

Most of the programming languages support lexical or static scope such as C, C++, Java, JavaScript. Perl supports both static and dynamic scoping.

**Scope Chain**

When a variable is used in JavaScript, the JavaScript engine will try to find the variable’s value in the current scope. If it could not find the variable, it will look into the outer scope and will continue to do so until it finds the variable or reaches global scope.

If it’s still could not find the variable, it will either implicitly declare the variable in the global scope (if not in strict mode) or return an error.

For example:

let foo = 'foo';

function bar() {  
 let baz = 'baz'; // Prints 'baz'  
 console.log(baz); // Prints 'foo'  
 console.log(foo); number = 42;  
 console.log(number); // Prints 42  
}bar();

When the function bar() is executed, the JavaScript engine looks for the baz variable and finds it in the current scope. Next, it looks for foo variable in the current scope and it can’t find it there, so it looks for the variable in outer scope and finds it there (i.e global scope).

After that, we assign 42 to the number variable, so the JavaScript engine looks for the number variable in the current scope and after that in the outer scope.

If the script is not in strict mode, the engine will create a new variable named number and assign 42 to it or return an error (if not in strict mode).

So when a variable is used the engine will traverse the scope chain until it finds the variable.

**How Does Scope and Scope Chain Work?**

Up until now, we have discussed what scope is and types of scope. Now let’s understand how JavaScript engine determines the scope of variables and perform variable lookups under the hood.

In order to understand how JavaScript engine performs variable lookups, we have to understand the concept of lexical environment in JavaScript.

**What is a Lexical Environment?**

A *lexical environment* is a structure that holds **identifier-variable mapping**. (here **identifier** refers to the name of variables/functions, and the **variable**is the reference to actual object [including function object and array object] or primitive value).

Simply put, a lexical environment is a place where variables and references to the objects are stored.

**Note —** Don’t confuse lexical scope with the lexical environment, lexical scope is a scope that is determined at compile time and a lexical environment is a place where variables are stored during the program execution.

Conceptually a lexical environment looks like this:

lexicalEnvironment = {  
 a: 25,  
 obj: <ref. to the object>  
}

A new lexical environment is created for each lexical scope but only when the code in that scope is executed. The lexical environment also has a reference to its outer lexical environment ( i.e outer scope). For example:

lexicalEnvironment = {  
 a: 25,  
 obj: <ref. to the object>

outer: <outer lexical environemt>  
}

**How does JavaScript Engine Perform Variable Lookups?**

Now that we know what scope, scope chain and lexical environment are, let’s now understand how JavaScript engine uses the lexical environment to determine scope and scope chain.

Let’s take a look at below code snippet to understand the above concepts.

let greeting = 'Hello';

function greet() {  
 let name = 'Peter';  
 console.log(`${greeting} ${name}`);  
}greet();{  
 let greeting = 'Hello World!'  
 console.log(greeting);  
}

When the above script loads, a global lexical environment is created, which contains variables and functions defined inside the global scope. For example:

globalLexicalEnvironment = {  
 greeting: 'Hello'  
 greet: <ref. to greet function>

outer: <null>  
}

Here outer lexical environment is set to null because there is no outer scope after global scope.

After that, a call to the greet() function is encountered. So a new lexical environment is created for the greet() function. For example:

functionLexicalEnvironment = {  
 name: 'Peter'

outer: <globalLexicalEnvironment>  
}

Here outer lexical environment is set to globalLexicalEnvironment because its outer scope is the global scope.

After that, the JavaScript engine executes the console.log(`${greeting} ${name}`) statement.

The JavaScript engine tries to find the greeting and name variables inside the function’s lexical environment. It finds the name variable inside the current lexical environment but it won’t be able to find the greeting variable inside the current lexical environment.

So it looks inside the outer lexical environment (defined by the outer property i.e global lexical environment) and finds the greeting variable.

Next, the JavaScript engine executes at the code inside the block. So it creates a new lexical environment for that block. For example:

blockLexicalEnvironment = {  
 greeting: 'Hello World',

outer: <globalLexicalEnvironment>  
}

Next, the console.log(greeting) statement is executed, the JavaScript engine finds the variable in the current lexical environment and uses that variable. So it does not look inside the outer lexical environment (global lexical environment) for the variable.

**Note —**A new lexical environment is created only for let and const declarations, not var declarations. var declarations are added to the current lexical environment (global or function lexical environment).

So when a variable is used in a program, the JavaScript engine will try to find the variable in the current lexical environment and if it could not find the variable there, it looks inside the outer lexical environment to find the variable. So that’s how the JavaScript engine performs variable lookups.

**Conclusion**

So in a nutshell, a scope is an area where a variable is visible and accessible. Just like functions, scopes in JavaScript can be nested and the JavaScript engine traverses the scope chain to find the variables used in the program.

JavaScript uses lexical scope which means that scope of variables is determined at compile time. The JavaScript engine uses the lexical environment to store the variables during the program execution.

Scope and scope chain are fundamental concepts of JavaScript that every JavaScript developer should understand. Having a good knowledge of these concepts will help you to become a much more effective and better JavaScript developer.