# All About Scope

The **scope** of a program in JavaScript is the set of variables that are available for use within the program. If a variable or other expression is not in the current scope, then it is unavailable for use. If we declare a variable, this variable will only be valid in the scope where we declared it. We can have nested scopes, but we'll see that in a little bit.

When we declare a variable in a certain scope, it will evaluate to a specific value **in that scope**. We have been using the concept of scope in our code all along! Now we are just giving this concept a name.

By the end of this reading you should be able to predict the evaluation of code that utilizes local scope, block scope, lexical scope, and scope chaining

## Advantages of utilizing scope

Before we start talking about different types of scope we'll be talking about the two main advantages that scope gives us:

1. **Security**- Scope adds security to our code by ensuring that variables can only be accessed by pre-defined parts of our programs.
2. **Reduced Variable Name Collisions**- Scope reduces variable name collisions, also known as namespace collisions, by ensuring you can use the same variable name multiple times in different scopes without accidentally overwriting those variable's values.

## Different kinds of scope

There are three types of scope in JavaScript: `global scope`, `local scope`, and`block scope`.

### Global scope

Let's start by talking about the widest scope there is: *global scope*. The*global scope* is represented by the `window`object in the browser and the`global`object in Node.js. Adding attributes to these objects makes them available throughout the entire program. We can show this with a quick example:

```
let myName = "Apples";

console.log(myName);
// this myName references the myName variable from this scope,
// so myName will evaluate to "Apples"
```

The variable `myName`above is not inside a function, it is just lying out in the open in our code. The `myName`variable is part of *global scope*. The Global scope is the largest scope that exists, it is the outermost scope that exists.

While useful on occasion, global variables are best avoided. Every time a variable is declared on the global scope, the chance of a name collision increases. If we are unaware of the global variables in our code, we may accidentally overwrite variables.

### Local scope

The **scope** of a function is the set of variables that are available for use within that function. We call the scope within a function: *local scope*. The *local scope* of a function includes:

1. the function's arguments
2. any local variables declared inside the function
3. **any variables that were already declared when the function was defined**

In JavaScript when we enter a new function we enter a **new scope**:

```
// global scope
let myName = "global";

function function1() {
  // function1's scope
  let myName = "func1";
  console.log("function1 myName: " + myName);
}

function function2() {
  // function2's scope
  let myName = "func2";
  console.log("function2 myName: " + myName);
}

function1(); // function1 myName: func1
function2(); // function2 myName: func2
console.log("global myName: " + myName); // global myName: global
```

In the code above we are dealing with three different scopes: the global scope, `function1`, and `function2`. Since each of the `myName` variables were declared in separate scopes, we *are* allowed to reuse variable names without any issues. This is because each of the `myName` variables is bound to their respective functions.

## Block scope

A block in JavaScript is denoted by a pair of curly braces (`{}`). Examples of block statements in JavaScript are `if` conditionals or `for` and `while` loops.

When using the keywords `let` or `const` the variables defined within the curly braces will be *block scoped*. Let's look at an example:

```
// global scope
let dog = "woof";

// block scope
if (true) {
  let dog = "bowwow";
  console.log(dog); // will print "bowwow"
}

console.log(dog); // will print "woof"
```

## Scope chaining: variables and scope

A key scoping rule in JavaScript is the fact that **an *inner* scope does have access to variables in the *outer* scope**.

Let's look at a simple example:

```
let name = "Fiona";

// we aren't passing in or defining and variables
function hungryHippo() {
  console.log(name + " is hungry!");
}

hungryHippo(); // => "Fiona is hungry"
```

So when the `hungryHippo` function is declared a new local scope will be created for that function. Continuing on that line of thought what happens when we refer to `name` inside of `hungryHippo`? If the `name` variable is not found in the immediate scope, JavaScript will search all of the accessible outer scopes until it finds a variable name that matches the one we are referencing. Once it finds

the first matching variable, it will stop searching. In JavaScript this is called *scope chaining*.

Now let's look at an example of scope chaining with nested scope. Just like functions in JavaScript, a scope can be nested within another scope. Take a look at the example below:

```
// global scope
let person = "Rae";

// sayHello function's local scope
function sayHello() {
  let person = "Jeff";

  // greet function's local scope
  function greet() {
    console.log("Hi, " + person + "!");
  }
  greet();
}

sayHello(); // logs 'Hi, Jeff!'
```

In the example above, the variable `person` is referenced by `greet`, even though it was never declared within `greet`! When this code is executed JavaScript will attempt to run the `greet` function - notice there is no `person` variable within the scope of the `greet` function and move on to seeing if that variable is defined in an outer scope.

Notice that the `greet` function prints out `Hi, Jeff!` instead of `Hi, Rae!`. This is because JavaScript will start at the inner most scope looking for a variable named `person`. Then JavaScript will work it's way outward looking for a variable with a matching name of `person`. Since the `person` variable within `sayHello` is in the next level of scope above `greet` JavaScript then stops it's scope chaining search and assigns the value of the `person` variable.

Functions such as `greet` that use (ie. **capture**) variables like the person variable are called **closures**. We'll be talking a lot more about closures very soon!

**Important** An inner scope can reference outer variables, but an outer scope cannot reference inner variables:

```
function potatoMaker() {
  let name = "potato";
  console.log(name);
}

potatoMaker(); // => "potato"

console.log(name); // => ReferenceError: name is not defined
```

## Lexical scope

There is one last important concept to talk about when we refer to scope - and that is *lexical scope*. Whenever you run a piece of JavaScript that code is first parsed before it is actually run. This is known as the *lexing time*. In the *lexing time* your parser resolves variable names to their values when functions are nested.

The main take away is that *lexical scope* is determined at *lexing time* so we can determine the values of variables without having to run any code. JavaScript is a language **without dynamic** scoping. This means that by looking at a piece of code we can determine the values of variables just by looking at the different scopes involved.

Let's look at a quick example:

```
function outer() {
  let x = 5;

  function inner() {
    // here we know the value of x because scope chaining will
    // go into the scope above this one looking for variable named x.
    // We do not need to run this code in order to determine the value of x!
    console.log(x);
  }
  inner();
}
```

In the `inner`function above we don't need to run the `outer`function to know what the value of `x` will be because of *lexical scoping*.

## What you learned

The **scope** of a program in JavaScript is the set of variables that are available for use within the program. Due to *lexical scoping*we can determine the value of a variable by looking at various scopes without having to run our code. *Scope Chaining*allows code within an *inner*scope to access variables declared in an *outer*scope.

There are three different scopes:

- *global scope*- the global space is JavaScript
- *local scope*- created when a function is defined
- *block scope*- created by entering a pair of curly braces