Variable Scope

Scope in JavaScript refers to the set of functions and variables that are available for use. Learn where variables and functions are available and how the JavaScript engine gives us the values we need.

Overview

The concept of scope refers to the availability of variables in our code. Can we use a variable in a function if it was created in another function? If a function contains another function, can they access each other's variables? These questions are simple, but also a source of great confusion to those without an understanding of what's going on. Let's jump right in.

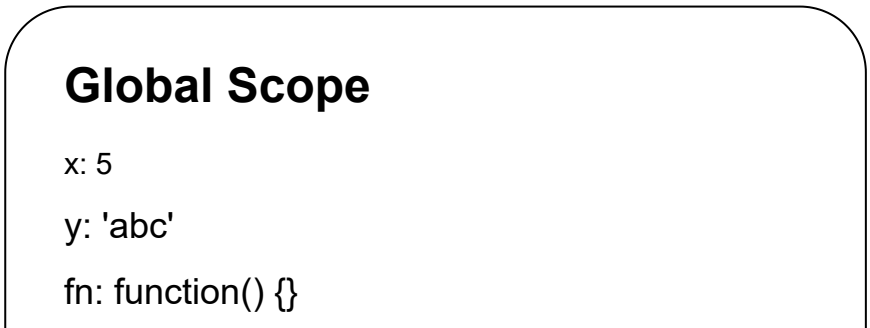
There are two types of scope in JavaScript: global scope and function scope.

Global Scope

When we start writing code in a file, we are automatically placed in the *global scope* of that file. When we declare variables in the file, we place them in the global scope.

```
var x = 5;
var y = 'abc';
function fn() {};
```

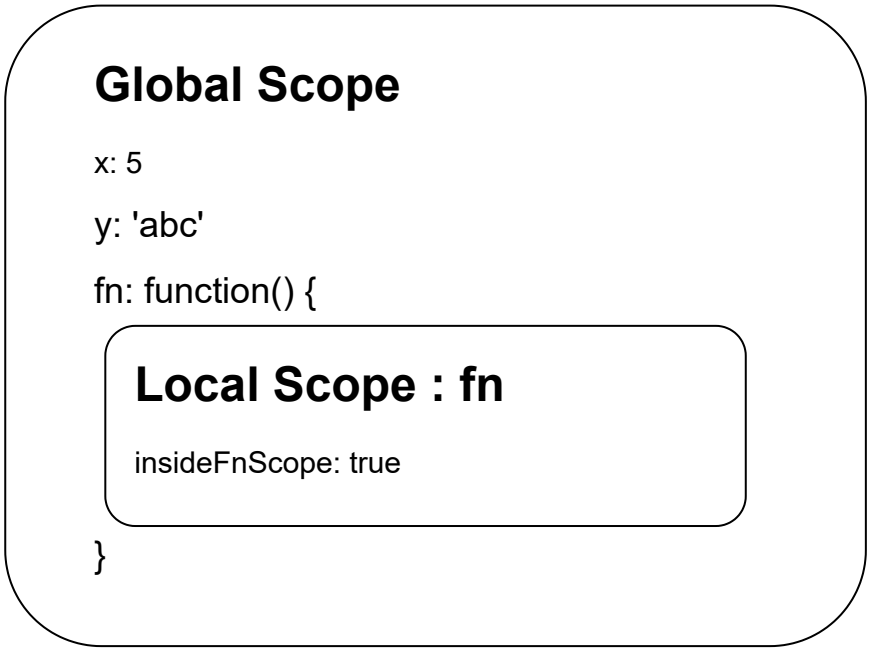
At this point, `x`, `y`, and `fn` are all available in the global scope. This is the highest scope available to us. Anything in the global scope can be accessed anywhere in the file.



Local Scope

Functions have their own *local scope* available to them. This local scope can be thought of as being nested *inside* the global scope.

```
var x = 5;
var y = 'abc';

function fn() {
    var insideFnScope = true;
};
```



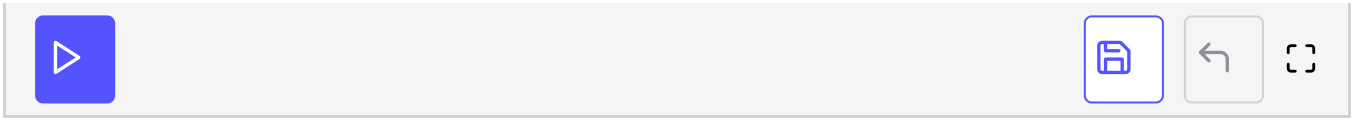
Inner scopes have access to their outer scopes. This means that while inside the function `fn`, we can access the variables `x` and `y`. If we're inside a "scope oval" as they're drawn here, we can access anything outside that oval. In a function, we can look outwards and access variables in the parent scope, in this case, the global scope.

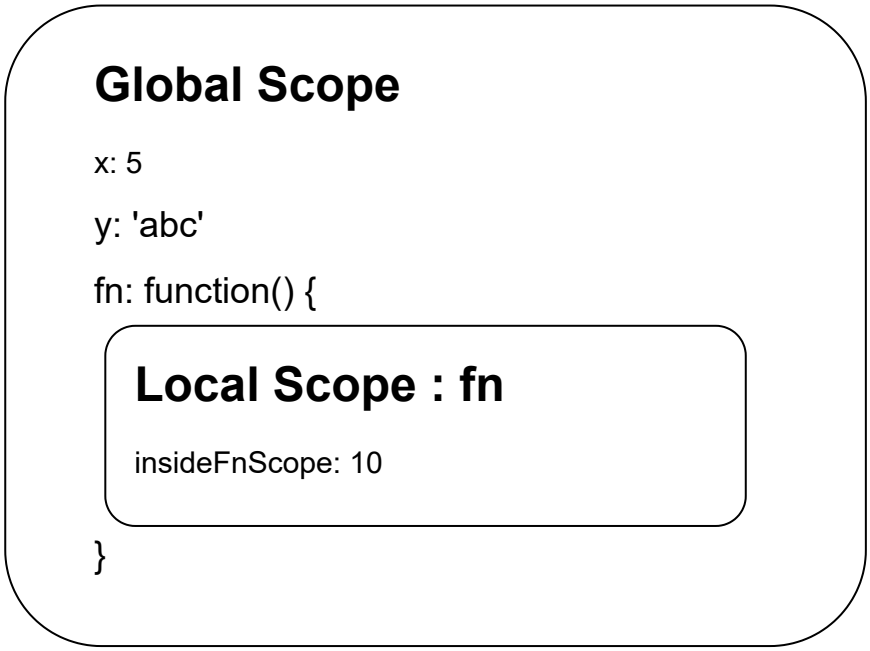
The opposite is not true. If we're writing code in the global scope, which means we're not writing in a function, we find ourselves in the larger oval. We can **not** look down into the scope of `fn` to access those variables.

We can look at the current scope level and outwards, but not inwards. Let's demonstrate this concept.

```
var x = 5;
```

```
function fn() {
    var insideFnScope = 10;
    console.log(insideFnScope);
    console.log(x);
};

fn();
// -> 10
// -> 5

console.log(insideFnScope);
// -> Uncaught ReferenceError: insideInnerScope is not defined
```





While we're inside `fn`, we can access all of `fn`'s variables and also the variables in the global scope because we have the ability to look outwards. We see `x` and `insideFnScope` logging correctly from inside the function.

When we're outside the function, we can't access variables declared inside the function. Attempting to do so results in a reference error.

Drilling Deeper

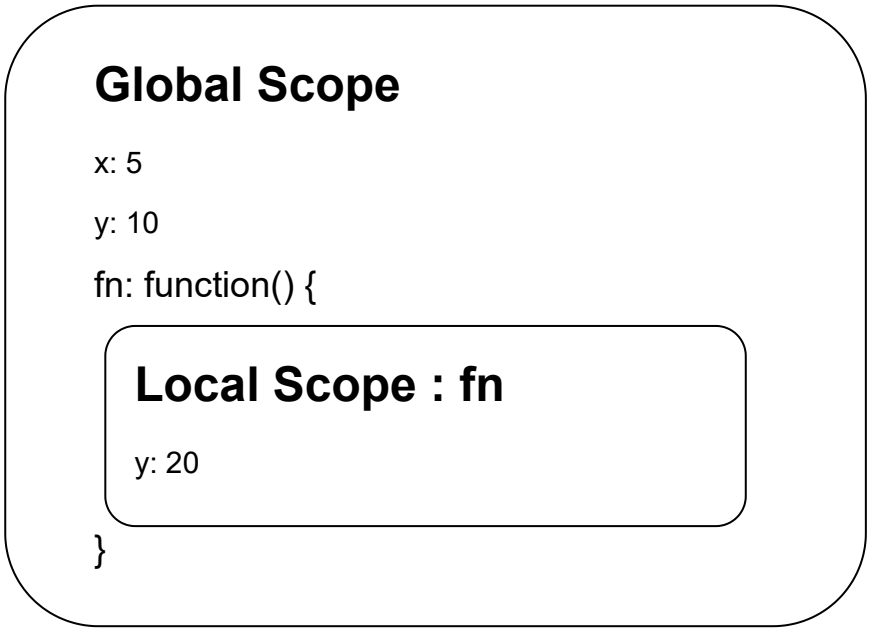
Let's go into these concepts a little further.

```
var x = 5;
var y = 10;

function fn() {
    var y = 20;
    console.log(x);
```

```
    console.log(y);
}

fn();
// -> 5
// -> 20

console.log(x); // -> 5
console.log(y); // -> 10
```

Let's describe what's happening here. We first create variables `x` and `y`. Since they're not in a function, they're in the global scope.

Inside the function, we create *another* variable `y` inside the local scope of the function. This is separate from the first `y`.



There's a mechanism behind variable lookup in JavaScript. When we try to access a variable, the JavasScript engine first looks at the current scope level. If the variable isn't found, it will jump upwards into the parent scope and look there, and keep going upwards and out until it either finds the variable or reaches the global scope. If it's not in the global scope, we get a reference error.

It only goes outwards. This means that a function can only access the global scope and the scope of functions it is inside of.

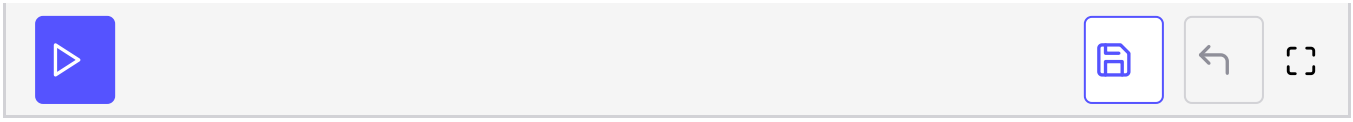
In this case, when we log `x` and `y` inside `fn`, the engine first checks the local function scope inside `fn` for the variables.

When looking for `x`, it doesn't find it in the local scope. It then jumps up one level and looks for it in the outer scope. It finds it there and uses that value. When looking for `y`, it finds it in the local scope and uses that value. It has no need to go up.

Changing the Outer Scope

We've seen how to access variables outside the local scope and how to declare new ones inside the local scope. Can we change variables in the outer scope when we're in the inner scope?

```
var x = 5;

function fn() {
    x = 10;
    y = 20;
};

fn();
console.log(x); // -> 10
console.log(y); // -> 20
```



This logged `10`... why? Shouldn't the first log statement print `5`, because the `x` inside `fn` is a new, local version of `x`? Shouldn't the second log statement cause an error?

Notice that inside `fn`, when we set `x` equal to `10` and `y` equal to `20`, we don't use the keyword `var`. This is the key.

Functions can access *and change* variables in their outer scope. When we omit `var`, the JavaScript engine will not create a new instance of a variable in the local scope. Instead, it will first attempt to find the variable we are changing. If it's found in the local scope, it'll change the variable in the local scope. If it's not, it'll go up continuously until it finds it.

In the case of `x`, it looks locally, doesn't find it, then looks upwards and finds it in the global scope. It will then change the value in the place it finds it.

In the case of `y`, it doesn't find it at all. When we're assigning a variable while omitting `var` and the engine can't find that variable, it'll automatically set the

variable in the global scope. That's why we see `20` printing out outside the function without any issues.

Best Practices

Setting new variables in the global scope is bad practice. It leads to confusion and bugs that are very difficult to find. It pollutes the global scope. If you ever find yourself needing to set a new variable in the global scope without using `var`, try refactoring your code. There's probably a better way to do what you're trying to do.

Quiz Time

Let's go through some examples. Try to figure out for yourself what'll happen here.

Scope Quiz

1 What will the following code print?

```
var a = 'abc';
var b = 'def';

function fn() {
    a = 123;
    var b = 456;
    console.log(a, b)
}

fn();
```

2

What will the following code print?

```
var a = 'abc';
var b = 'def';

function fn() {
    a = 123;
    var b = 456;
}

fn();

console.log(a, b);
```

3

What will the following code print?

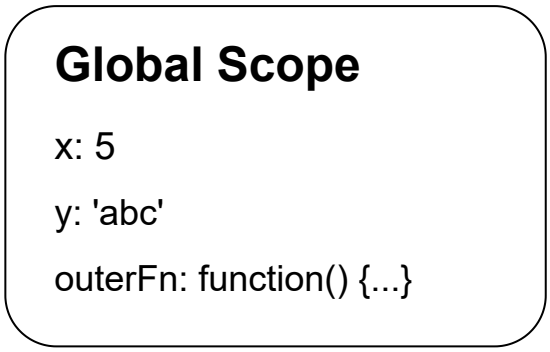
```
var x = 5;
var y = 'abc';

function outerFn() {
```

```
    var x = 10;
    y = 'xyz';

    function innerFn() {
        x = 20;
        var y = 'jkl';
    }

    innerFn();

    console.log(x);
    console.log(y);
}

outerFn();
// -> ?
// -> ?

console.log(x); // -> ?
console.log(y); // -> ?
```

Check Answers

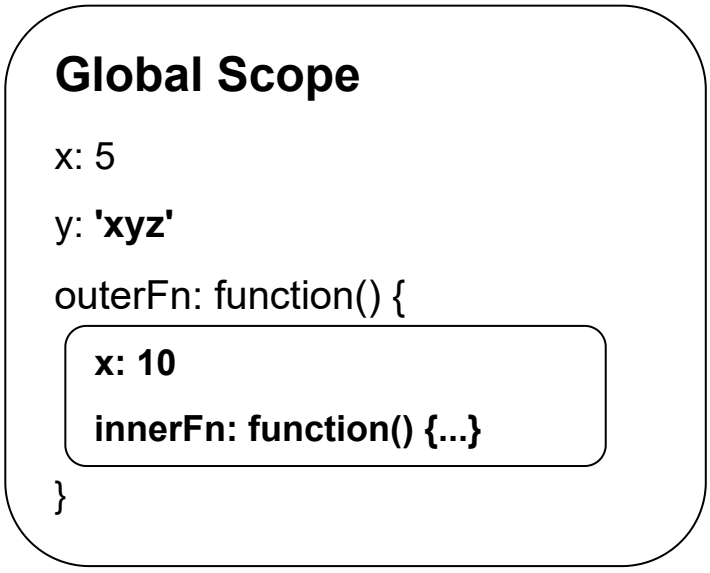
Let's review question 3 above.

Before any function runs, we have three items in the outer scope.



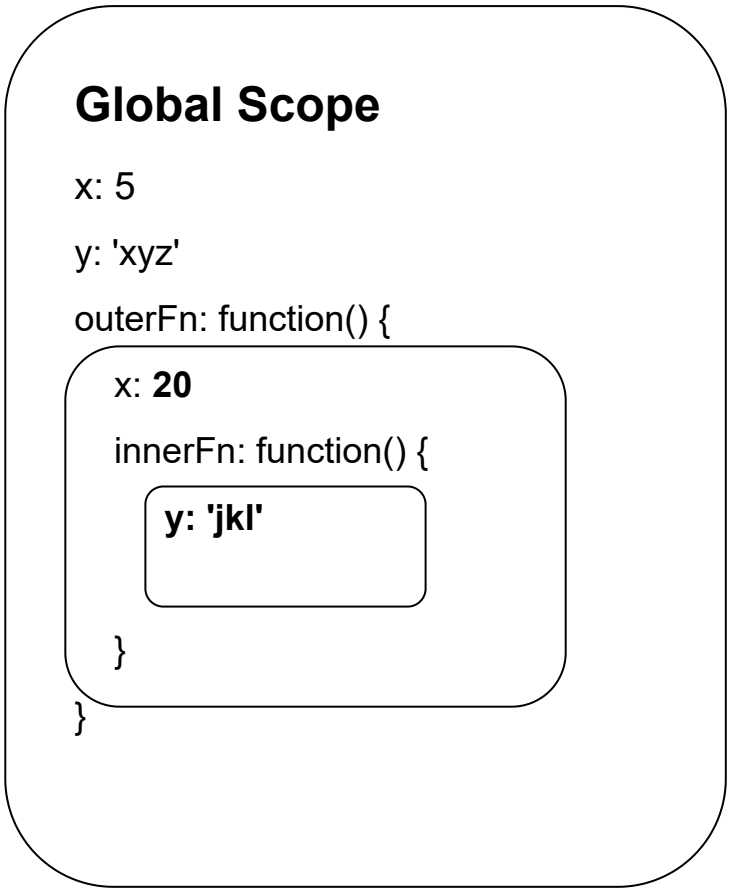
A scope diagram of variables before outerFn has run

When `outerFn` starts executing, it creates its own version of `x` and modifies the global `y`.



A scope diagram of variables after outerFn has started run but before innerFn has run

When `innerFn` runs, it modifies `outerFn`'s version of `x` and creates its own `y`. It leaves the global scope untouched. This is the final state of our scope before `innerFn` returns.



A scope diagram of variables after innerFn has started running

That’s it for Scope.