## Let

**Block Scope**

A confusing point for developers coming from different languages is the way variable [scope](https://en.wikipedia.org/wiki/Scope_(computer_science)) behaves in JavaScript.

  *Scope* refers to the lifecycle of a variable, i.e. where in the code it’s visible and for how long

In Java and C++ there is the concept of block scope, a block is any code that is wrapped in { and }, like so:

{

// This is a block

}

// This is not part of the block

This means that in those languages if you declare a variable inside a block, it’s only visible inside that block and any nested blocks inside that block.

But in ES5 JavaScript we only have two scopes, the ***global scope*** and ***function scope***.

So if I wrote:

{

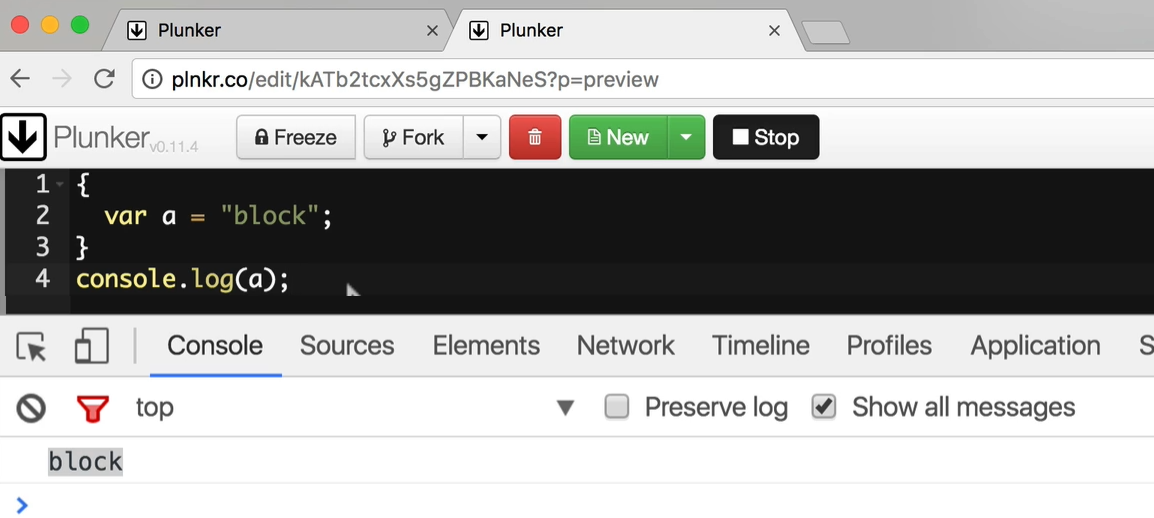
var a = "block";

}

console.log(a);

Those coming from Java or C++ backgrounds might think that this would throw an error because we are trying to use the variable a outside of the block it was created in. But **in JavaScript it doesn’t, this code is perfectly legal**.

The variable a, as we’ve declared it above, exists in *global scope* so this means it’s visible from *everywhere* in our application and exists for the lifetime of the application.



In ES5 apart from global scope, the only other scope is *function scope*, so if we wrote.

function hello() {

var a = "function";

}

hello();

console.log(a);

If we ran the above we would get an error, like so:

Uncaught ReferenceError: a is not defined(

…

)

This is because **the a variable is declared *inside* a function** and is therefore only visible inside *that* function, trying to access it outside the function results in an error.

But again this isn’t *block level scope* as we can see if we add a *for loop* inside the function, like so:

function hello() {

var a = "function";

for (var i = 0; i < 10; i++) {

var a = "block";

}

console.log(a);

}

hello();

What gets printed out here is **block** not function despite the fact we are outside the for loop, that’s because the body of the for loop is not its own scope.

* i.e. again the curly braces DON’T signify the block level scope

### IIFE

This issue of *no block level scope* has plagued JavaScript developers since its inception.

One common workaround in the past has been to use something called an Immediately Invoked Function Expression (IIFE) like so:

function hello() {

var a = "function";

for (var i=0; i<5; i++) {

(function() {

var a = "block";

})();

}

console.log(a);

}

hello();

This now prints out

**function**

.

If this syntax looks a bit strange to you:

(

function()

{

var a = "block";

})()

;

Know its just a shorter way of writing:

function something() {

var a = "block";

}

something();

* What we are doing above is just declaring a function and calling it straight away(immediately).

It’s a function that we call *immediately* after defining it.

Since functions have their own scope, using an IIFE has the same effect as if we had block level scope, the variable a inside the IIFE isn’t visible *outside* the IIFE.

* This is how JavaScript developers used to simulating the block level scope for years before ES6

**Let**

IIFEs work but it’s a pretty long winded way of solving this problem. So with ES6 we now have the new let keyword, we use it in place of the var keyword and it creates a variable **with** block level scope, like so:

function hello() {

var a = "function";

for (var i = 0; i < 5; i++) {

let a = "block";

}

console.log(a);

}

hello();

Now the a declared in the for loop body only exists between the { and } i.e. block level scope, and the code snippet above prints out **function** as expected.

**Using let in for loops**

So a classic interview question to test JavaScript developers knowledge of the lack of block level scope is this one:

var funcs = [];

for (var i = 0; i < 6; i += 1) {

var y = i;

funcs.push(function () {

console.log(y);

})

}

funcs.forEach(function (func) {

func()

})

;

What gets printed out?

You might expect

0

1

2

3

4

But in fact it’s

5

5

5

5

5

The reason for this is that the variable y is not *block level*, it doesn’t *only* exist inside its enclosing {} In fact it’s a global variable and by the time any of the functions are called it’s already been set to 5.

In the above example if we replace var y = i with let y = i then the variable y only exists inside it’s block, like so:

var funcs = [];

for (var i = 0; i < **5**; i += 1) {

let y = i;

funcs.push(function () {

console.log(y);

})

}

funcs.forEach(function (func) {

func()

})

;

And so executing this now results in:

0

1

2

3

4

**The for loop short-cut**

The above construct is so common we have a shortcut, we can declare the index variable i with let in the for loop expression, like so:

var funcs = [];

for (let i = 0; i < 5; i += 1) {

funcs.push(function () {

console.log(i);

})

}

funcs.forEach(function (func) {

func()

})

;

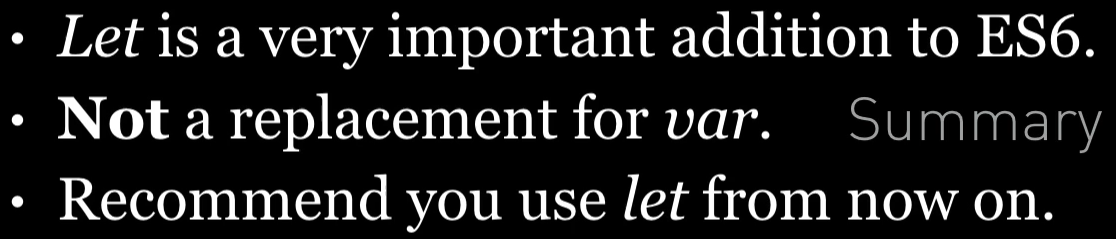
Even though let i = 0 is strictly declared *outside* of the for block { }, any variables declared in the for loop expression with let has block level scope in the for loop.

**Summary**

let is a very important addition the javascript language in ES6.

It’s not a replacement to var, var can still be used even in ES6 and has the same semantics as ES5.

However unless you have a specific reason to use var I would expect all variables you define from now on to use let.



**Listing**

<http://plnkr.co/edit/k7150PqX1DML7EFHKuRc?p=preview>

*script.js*

/\* let \*/

var funcs = [];

for (let i = 0; i < 5; i += 1) {

funcs.push(function () {

console.log(i);

})

}

funcs.forEach(function (func) {

func()

})

;

*index.html*

<!

DOCTYPE html

>

<

html

>

<

head

>

<script src="script.js"></script>

<

/head

>

<

/html

>