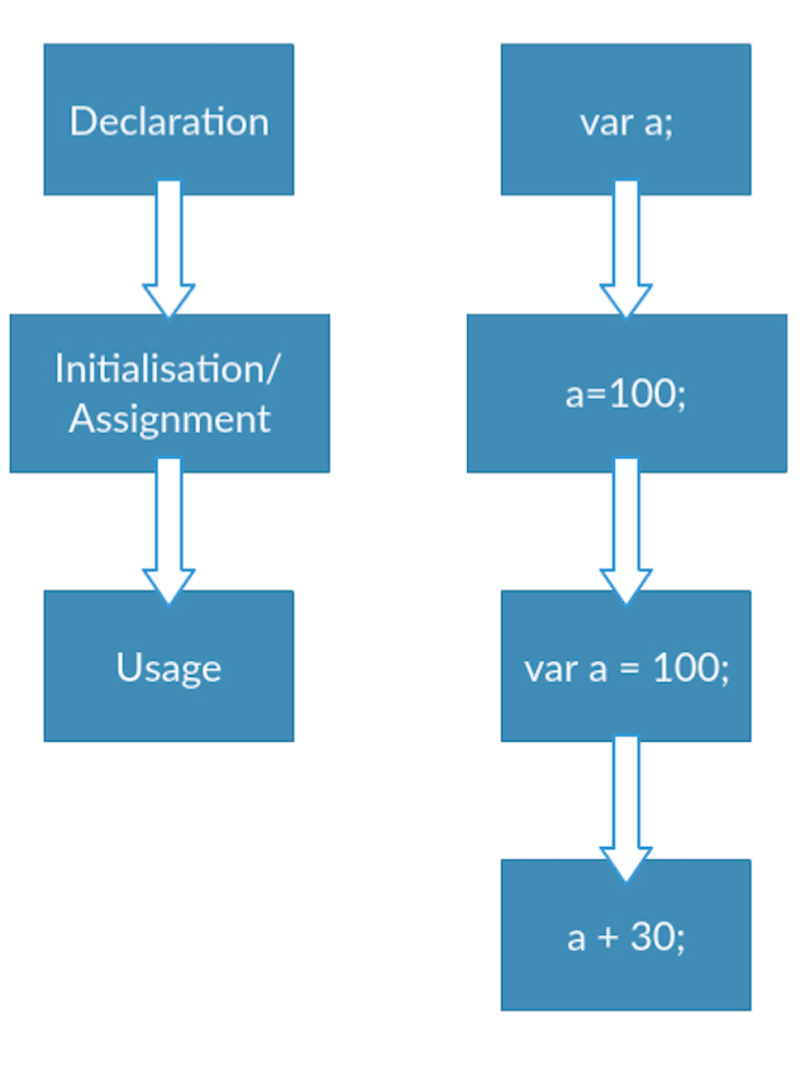
Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.



function hoist() {

a = 20;

var b = 100;

}

hoist();

console.log(a);

/\*

Accessible as a global variable outside hoist() function

Output: 20

\*/

console.log(b);

/\*

Since it was declared, it is confined to the hoist() function scope.

We can't print it out outside the confines of the hoist() function.

Output: ReferenceError: b is not defined

\*/

**var**

The scope of a variable declared with the keyword var is its *current execution context*. This is either the **enclosing function** or for variables declared outside any function, **global**. Let's look at a few examples to identify what this means:

**global variables**

console.log(hoist); // Output: undefined

var hoist = 'The variable has been hoisted.';

**Strict Mode**

Thanks to a utility of the es5 version of JavaScript known as strict-mode, we can be more careful about how we declare our variables. By enabling [*strict mode*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode), we opt into a restricted variant of JavaScript that will not tolerate the usage of variables before they are declared.

Running our code in strict mode:

1. Eliminates some silent JavaScript errors by changing them to explicit throw errors which will be spit out by the interpreter.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimisations.
3. Prohibits some syntax likely to be defined in future versions of JavaScript.

We enable strict mode by prefacing our file or function with

'use strict';

// OR

"use strict";

Es6

[ECMAScript 6](https://github.com/lukehoban/es6features), ECMAScript 2015 also known as ES6 is the latest version of the ECMAScript standard, as the writing of this article, Jan 2017 and introduces a few changes to es5.

Of interest to us is how changes in the standard affect the declaration and initialisation of JavaScript variables.

### let

Before we start, to be noted is the fact that variables declared with the keyword let are block scoped and not function scoped. That's significant, but it shouldn't trouble us here. Briefly, however, it just means that the variable's scope is bound to the block in which it is declared and not the function in which it is declared.

Let's start by looking at the let keyword's behaviour.

console.log(hoist); // Output: ReferenceError: hoist is not defined ...

let hoist = 'The variable has been hoisted.';

Like before, for the var keyword, we expect the output of the log to be undefined. However, since the es6 let doesn't take kindly on us using undeclared variables, the interpreter explicitly spits out a Reference error.

This ensures that we **always** declare our variables first.

However, we still have to be careful here. An implementation like the following will result in an ouput of undefined instead of a Reference error.

let hoist;

console.log(hoist); // Output: undefined

hoist = 'Hoisted'

Hence, to err on the side of caution, we should declare then assign our variables to a value before using them.

### const

The const keyword was introduced in es6 to allow immutable variables. That is, variables whose value cannot be modified once assigned.

With const, just as with let, the variable is hoisted to the top of the block.

Let's see what happens if we try to reassign the value attached to a const variable.

const PI = 3.142;

PI = 22/7; // Let's reassign the value of PI

console.log(PI); // Output: TypeError: Assignment to constant variable.

How does const alter variable declaration? Let's take a look.

console.log(hoist); // Output: ReferenceError: hoist is not defined

const hoist = 'The variable has been hoisted.';

Much like the let keyword, instead of silently exiting with an undefined, the interpreter saves us by explicitly throwing a Reference error.

The same occurs when using const within functions.

function getCircumference(radius) {

console.log(circumference)

circumference = PI\*radius\*2;

const PI = 22/7;

}

getCircumference(2) // ReferenceError: circumference is not defined

With const , es6 goes further. The interpreter throws an error if we use a constant before declaring and initialising it.

Our linter is also quick to inform us of this felony:

PI was used before it was declared, which is illegal for const variables.

Globally,

const PI;

console.log(PI); // Ouput: SyntaxError: Missing initializer in const declaration

PI=3.142;

Therefore, a constant variable must be both declared and initialised before use.

As a prologue to this section, it's important to note that indeed, JavaScript hoists variables declared with es6 let and const. The difference in this case is how it initialises them. Variables declared with let and const remain **uninitialised** at the beginning of execution whilst variables declared with var are initialised with a value of **undefined**.

## [Hoisting functions](https://scotch.io/tutorials/understanding-hoisting-in-javascript#toc-hoisting-functions)

JavaScript functions can be loosely classified as the following:

1. Function declarations
2. Function expressions

We'll investigate how hoisting is affected by both function types.

### Function declarations

These are of the following form and are hoisted completely to the top. Now, we can understand why JavaScript enable us to invoke a function seemingly before declaring it.

hoisted(); // Output: "This function has been hoisted."

function hoisted() {

console.log('This function has been hoisted.');

};

### Function expressions

Function expressions, however are not hoisted.

expression(); //Output: "TypeError: expression is not a function

var expression = function() {

console.log('Will this work?');

};

Let's try the combination of a function declaration and expression.

expression(); // Ouput: TypeError: expression is not a function

var expression = function hoisting() {

console.log('Will this work?');

};

As we can see above, the variable declaration var expression is hoisted but it's assignment to a function is not. Therefore, the intepreter throws a TypeError since it sees expression as a variable and not a function.

## [Order of precedence](https://scotch.io/tutorials/understanding-hoisting-in-javascript#toc-order-of-precedence)