**The old "var"**

**This article is for understanding old scripts**

The information in this article is useful for understanding old scripts.

That’s not how we write a new code.

In the very first chapter about [variables](https://javascript.info/variables), we mentioned three ways of variable declaration:

1. let
2. const
3. var

The var declaration is similar to let. Most of the time we can replace let by var or vice-versa and expect things to work:

var message = "Hi";

alert(message); // Hi

But internally var is a very different beast, that originates from very old times. It’s generally not used in modern scripts, but still lurks in the old ones.

If you don’t plan on meeting such scripts you may even skip this chapter or postpone it.

On the other hand, it’s important to understand differences when migrating old scripts from var to let, to avoid odd errors.

**[“var” has no block scope](https://javascript.info/var" \l "var-has-no-block-scope)**

Variables, declared with var, are either function-scoped or global-scoped. They are visible through blocks.

For instance:

if (true) {

var test = true; // use "var" instead of "let"

}

alert(test); // true, the variable lives after if

As var ignores code blocks, we’ve got a global variable test.

If we used let test instead of var test, then the variable would only be visible inside if:

if (true) {

let test = true; // use "let"

}

alert(test); // ReferenceError: test is not defined

The same thing for loops: var cannot be block- or loop-local:

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

var one = 1;

// ...

}

alert(i); // 10, "i" is visible after loop, it's a global variable

alert(one); // 1, "one" is visible after loop, it's a global variable

If a code block is inside a function, then var becomes a function-level variable:

function sayHi() {

if (true) {

var phrase = "Hello";

}

alert(phrase); // works

}

sayHi();

alert(phrase); // ReferenceError: phrase is not defined

As we can see, var pierces through if, for or other code blocks. That’s because a long time ago in JavaScript, blocks had no Lexical Environments, and var is a remnant of that.

**[“var” tolerates redeclarations](https://javascript.info/var" \l "var-tolerates-redeclarations)**

If we declare the same variable with let twice in the same scope, that’s an error:

let user;

let user; // SyntaxError: 'user' has already been declared

With var, we can redeclare a variable any number of times. If we use var with an already-declared variable, it’s just ignored:

var user = "Pete";

var user = "John"; // this "var" does nothing (already declared)

// ...it doesn't trigger an error

alert(user); // John

**[“var” variables can be declared below their use](https://javascript.info/var" \l "var-variables-can-be-declared-below-their-use)**

var declarations are processed when the function starts (or script starts for globals).

In other words, var variables are defined from the beginning of the function, no matter where the definition is (assuming that the definition is not in the nested function).

So this code:

function sayHi() {

phrase = "Hello";

alert(phrase);

var phrase;

}

sayHi();

…Is technically the same as this (moved var phrase above):

function sayHi() {

var phrase;

phrase = "Hello";

alert(phrase);

}

sayHi();

…Or even as this (remember, code blocks are ignored):

function sayHi() {

phrase = "Hello"; // (\*)

if (false) {

var phrase;

}

alert(phrase);

}

sayHi();

People also call such behavior “hoisting” (raising), because all var are “hoisted” (raised) to the top of the function.

So in the example above, if (false) branch never executes, but that doesn’t matter. The var inside it is processed in the beginning of the function, so at the moment of (\*) the variable exists.

**Declarations are hoisted, but assignments are not.**

That’s best demonstrated with an example:

function sayHi() {

alert(phrase);

var phrase = "Hello";

}

sayHi();

The line var phrase = "Hello" has two actions in it:

1. Variable declaration var
2. Variable assignment =.

The declaration is processed at the start of function execution (“hoisted”), but the assignment always works at the place where it appears. So the code works essentially like this:

function sayHi() {

var phrase; // declaration works at the start...

alert(phrase); // undefined

phrase = "Hello"; // ...assignment - when the execution reaches it.

}

sayHi();

Because all var declarations are processed at the function start, we can reference them at any place. But variables are undefined until the assignments.

In both examples above, alert runs without an error, because the variable phrase exists. But its value is not yet assigned, so it shows undefined.

**[IIFE](https://javascript.info/var" \l "iife)**

In the past, as there was only var, and it has no block-level visibility, programmers invented a way to emulate it. What they did was called “immediately-invoked function expressions” (abbreviated as IIFE).

That’s not something we should use nowadays, but you can find them in old scripts.

An IIFE looks like this:

(function() {

var message = "Hello";

alert(message); // Hello

})();

Here, a Function Expression is created and immediately called. So the code executes right away and has its own private variables.

The Function Expression is wrapped with parenthesis (function {...}), because when JavaScript engine encounters "function" in the main code, it understands it as the start of a Function Declaration. But a Function Declaration must have a name, so this kind of code will give an error:

// Tries to declare and immediately call a function

function() { // <-- SyntaxError: Function statements require a function name

var message = "Hello";

alert(message); // Hello

}();

Even if we say: “okay, let’s add a name”, that won’t work, as JavaScript does not allow Function Declarations to be called immediately:

// syntax error because of parentheses below

function go() {

}(); // <-- can't call Function Declaration immediately

So, the parentheses around the function is a trick to show JavaScript that the function is created in the context of another expression, and hence it’s a Function Expression: it needs no name and can be called immediately.

There exist other ways besides parentheses to tell JavaScript that we mean a Function Expression:

// Ways to create IIFE

(function() {

alert("Parentheses around the function");

})();

(function() {

alert("Parentheses around the whole thing");

}());

!function() {

alert("Bitwise NOT operator starts the expression");

}();

+function() {

alert("Unary plus starts the expression");

}();

In all the above cases we declare a Function Expression and run it immediately. Let’s note again: nowadays there’s no reason to write such code.

**[Summary](https://javascript.info/var" \l "summary)**

There are two main differences of var compared to let/const:

1. var variables have no block scope, their visibility is scoped to current function, or global, if declared outside function.
2. var declarations are processed at function start (script start for globals).

There’s one more very minor difference related to the global object, that we’ll cover in the next chapter.

These differences make var worse than let most of the time. Block-level variables is such a great thing. That’s why let was introduced in the standard long ago, and is now a major way (along with const) to declare a variable.