26th December 2019

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, we mentioned three ways of variable declaration:

1. `let`
2. `const`
3. `var`

`let` and `const` behave exactly the same way in terms of Lexical Environments.

But `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, but then there's a chance that it bites you later.

From the first sight, `var` behaves similar to `let`. That is, declares a variable:

```
function sayHi() {
  var phrase = "Hello"; // local variable, "var" instead of "let"

  alert(phrase); // Hello
}

sayHi();

alert(phrase); // Error, phrase is not defined
```

…But here are the differences.

"var" has no block scope

Variables, declared with `var`, are either function-wide or global. 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); // Error: test is not defined
```

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

```
for (var i = 0; i < 10; i++) {
  // ...
}

alert(i); // 10, "i" 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); // Error: phrase is not defined (Check the Developer Console)
```

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" declarations are processed at the function start

`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 better to demonstrate with an example, like this:

```
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

As in the past 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() {

  let 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 meets `"function"` in the main code flow, 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:

```
// Try to declare and immediately call a function
function() { // <-- Error: Unexpected token (

  let message = "Hello";

  alert(message); // Hello
```

```
7
8 }();
```

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:

```
1 // syntax error because of parentheses below
2 function go() {
3
4 }(); // <-- 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:

```
1 // Ways to create IIFE
2
3 (function() {
4   alert("Parentheses around the function");
5 })();
6
7 (function() {
8   alert("Parentheses around the whole thing");
9 }());
10
11 !function() {
12   alert("Bitwise NOT operator starts the expression");
13 }();
14
15 +function() {
16   alert("Unary plus starts the expression");
17 }();
```

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

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

1. `var` variables have no block scope, they are visible minimum at the function level.
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.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)