

Part 2: Var vs Const vs Let

This article is Part 2 for the Series “Modern ES6+ Javascript for those who know only a little about that old Javascript.”

[Harry Manchanda](#) Sep 17

CONST vs LET vs VAR

ES6 Conventions:

1. Use ``const`` by default.
2. Use ``let`` if you have to rebind a variable.
3. Use ``var`` to signal untouched legacy code

Source: <https://twitter.com/raganwald/status/564792624934961152>

JS

Source: <https://twitter.com/raganwald/status/564792624934961152>

Credits: *This Part of the Article series has been taken up from the [book](#) “Understanding ECMAScript 6 by Nicholas C. Zakas”.*

Looking for Part 1? [Click Here](#).

Traditionally, the way variable declarations work has been that weird part of programming in JavaScript. Variables creation depend on how you

declare them, and **ES6** offers options to make controlling scope easier. This article will look to clear on why those classic `var` declarations can be confusing and will also introduce block-level bindings aka `const` and `let`.

Variable declarations using `var` get treated as if they are at the top of the function (or global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called **hoisting**. See the example below to see what hoisting does:

```
function getValue(condition) {
  if (condition) {
    var value = "blue";
    // other code
    return value;
  } else {
    // value exists here with a value of undefined
    return null;
  }
  // value exists here with a value of undefined
}
```

If you are new to JavaScript, you might think that the variable value only to be created if the condition evaluates to true. In fact, this is not how JavaScript engines work behind the scenes; the variable value gets created regardless as the engine changes the `getValue` function to look something like this:

```
function getValue(condition) {
  var value;

  if (condition) {
    value = "blue";
    // other code
    return value;
  } else {
    return null;
  }
}
```

The declaration of value is hoisted to the top, while the initialization

remains in the same spot. That means the variable value is still accessible from within the else clause. It happens to be like this as the variable would just have a value of undefined the other way around as it hasn't been initialized.

It is obvious and understandable that will not be easy for new JavaScript developers to learn declaration hoisting, but please note that misunderstanding this unique behavior can end up causing bugs. To resolve this ES6 introduced block level scoping options to make the controlling a variable's lifecycle a little more powerful.

Block-Level Declarations

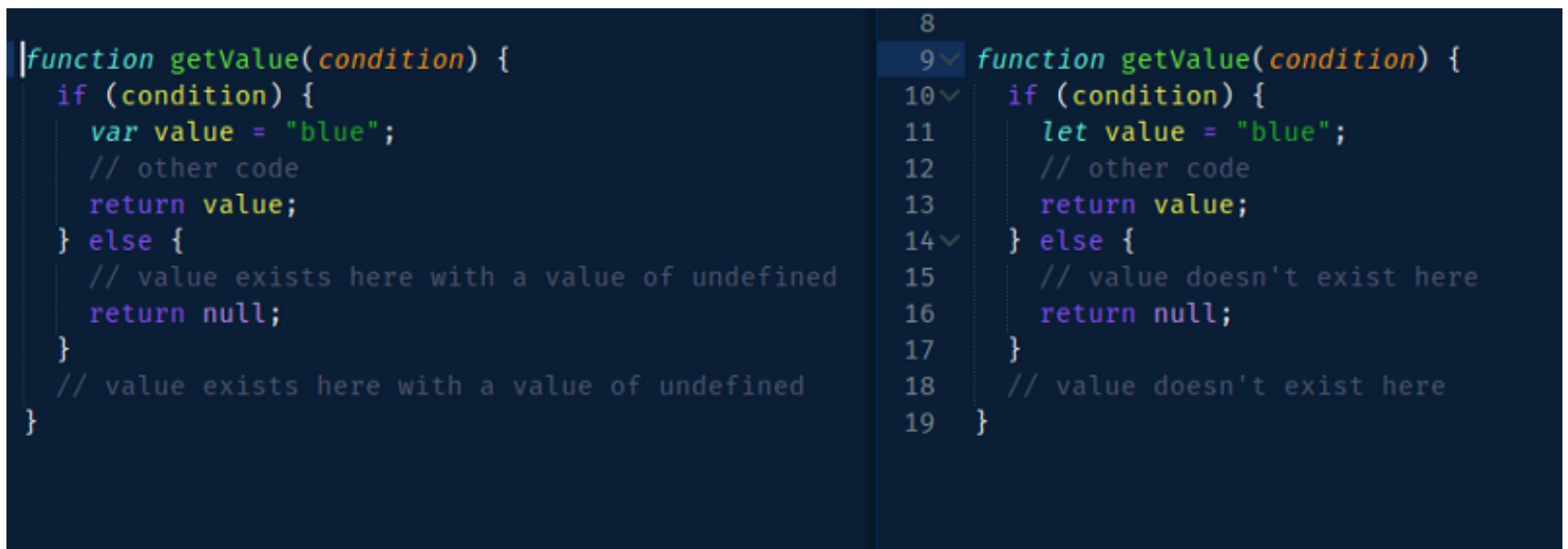
Block-level declarations are the ones that declare variables that are far outside of a given block scope. Block scopes, also known as lexical scopes, are created either inside of a function or inside of a block (indicated by the `{` and `}` characters). Block scoping is how many C-based languages work, and the introduction of block-level declarations in ECMAScript 6 is intended to bring that same flexibility (and uniformity) to JavaScript.

Let Declaration

The `let` declaration syntax is the same as `var`. You can replace `var` with `let` to declare a variable, this will limit the variable's scope to only that current code block. Since `let` declarations are not hoisted to the top of the enclosing block, you may want always to place `let` declarations first in the block, so that they are available to the entire block. Here's a quick example:

```
function getValue(condition) {
  if (condition) {
    let value = "blue";
    // other code
    return value;
  } else {
    // value doesn't exist here
    return null;
  }
  // value doesn't exist here
}
```

Here below is the screenshot that shows the difference between `var` and `let` (Check the comments within the code block).

A screenshot of a code editor with a dark background, showing two versions of a JavaScript function named `getValue`. The left version uses `var` for variable declaration, and the right version uses `let`. Both functions have an `if` block that sets `value` to "blue" if the condition is true, and returns `null` otherwise. Comments indicate that with `var`, the variable `value` exists throughout the function scope, while with `let`, it only exists within the `if` block's scope.

```
function getValue(condition) {  
  if (condition) {  
    var value = "blue";  
    // other code  
    return value;  
  } else {  
    // value exists here with a value of undefined  
    return null;  
  }  
  // value exists here with a value of undefined  
}  
  
function getValue(condition) {  
  if (condition) {  
    let value = "blue";  
    // other code  
    return value;  
  } else {  
    // value doesn't exist here  
    return null;  
  }  
  // value doesn't exist here  
}
```

As you can see, the `getValue` function with `let` behaves similar to other programming languages. As, variable `value` is declared using `let` instead of `var`, the declaration isn't hoisted to the top of the function definition, and the variable `value` is no longer accessible once execution flows out of the `if` block. If `condition` evaluates to false, then `value` is never declared or initialized.

No Redeclaration

If a identifier has already been defined within the scope, then using identifier in a `let` declaration inside that scope throws an error. Check below:

```
var count = 30;  
  
// Error: `count` has already been declared.  
let count = 40;
```

In this example, the `count` is declared twice: once with `var` and once with `let`. As `let` will not redefine an identifier that already exists in the same scope, the `let` declaration will throw an error.

On the other hand, no error shows up if the `let` declaration creates a new

variable with the same name as a variable in its containing scope, check the code below:

```
var count = 30;

if (condition) {
  // Does not throw an error
  let count = 40;

  // more code
}
```

This `let` declaration does not throw any error as it creates a new variable called `count` within the `if` statement, instead of creating the `count` in the surrounding block. Inside the `if` block, this new variable shadows the global `count`, which prevents access to it until execution leaves the block.

Constant Declarations

Also, you can define variables in ES6 with the `const` declaration syntax. Variables that are declared using the `const` keyword are considered constants, which means that their values can't be changed once set. Thus, each `const` variable must be initialized on the declaration, as shown below:

```
// Valid constant
const maxItems = 30;

// Syntax error: missing initialization
const name;
```

The `maxItems` variable is initialized, so its `const` declaration should work without any problems. The `name` variable, however, would cause a syntax error if you tried to run the program containing this code, because `name` is not initialized.

Thanks a lot...

If you liked my article and also my passion for teaching, please

don't forget to follow me and also CodeBurst by clicking the links below.

If you would like to hire me for your next cool project, or just want to say hello... my twitter handle is [@harmanmanchanda](#) for getting in touch with me! My DM's are open to the public so just hit me up.