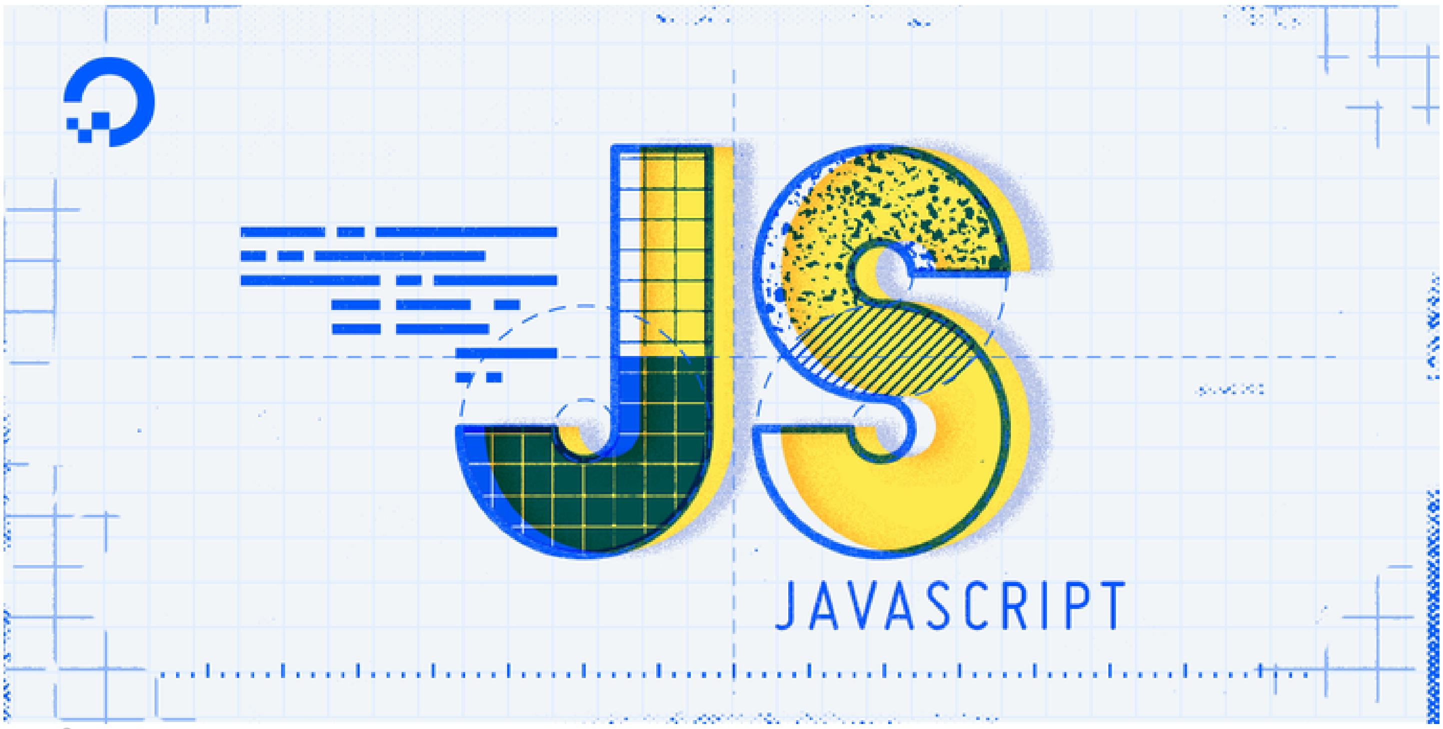
Subscribe



Understanding Variables, Scope, and Hoisting in JavaScript

12

Posted February 20, 2018 63.3k JAVASCRIPT DEVELOPMENT

By: Tania Rascia

Introduction

Variables are a fundamental part of many programming languages, and are among the first and most important concepts for novice coders to learn. There are a number of different properties of variables in JavaScript, as well as several rules which must be followed when naming them. In JavaScript, there are three keywords used to declare a variable — `var`, `let`, and `const` — and each one affects how the code will interpret the variable differently.

This tutorial will cover what variables are, how to declare and name them, and also take a closer look at the difference between `var`, `let`, and `const`. We will also review the effects of hoisting and the significance of global and local scope to a variable's behavior.

Understanding Variables

A variable is a named container used for storing values. A piece of information that we might reference multiple times can be stored in a variable for later use or modification. In JavaScript, the value contained inside a variable can be any JavaScript data type, including a number, string, or object.

Prior to the ECMAScript 2015 (ES6) language specification that today's JavaScript is based on, there was only one way to declare a variable — using the `var` keyword. As a result, most older code and learning resources will only use `var` for variables. We'll go over the differences between `var`, `let`, and `const` keywords in its own section below.

We can use `var` to demonstrate the concept of a variable itself. In the example below, we will *declare* a variable, and *assign* a value to it.

```
// Assign the string value Sammy to the username identifier
var username = "sammy_shark";
```

This statement consists of a few parts:

- The declaration of a variable using the `var` keyword
- The variable name (or identifier), `username`
- The assignment operation, represented by the `=` syntax
- The value being assigned, `"sammy_shark"`

Now we can use `username` in code. JavaScript will remember that `username` represents the string value `sammy_shark`.

```
// Check if variable is equal to value
if (username === "sammy_shark") {
  console.log(true);
```

```
Output
true
```

As mentioned previously, variables can be used to represent any JavaScript data type. In this example, we'll declare variables with string, number, object, Boolean, and null values.

```
// Assignment of various variables
var name = "Sammy";
var spartans = 300;
var kingdoms = [ "mammals", "birds", "fish" ];
var poem = { roses: "red", violets: "blue" };
var success = true;
var nothing = null;
```

Using `console.log`, we can see the value contained in a specific variable.

```
// Send spartans variable to the console
console.log(spartans);
```

```
Output
300
```

Variables store data in memory which can later be accessed and modified. Variables can also be reassigned and given a new value. The simplified example below demonstrates how a password might be stored to a variable and then updated.

```
// Assign value to password variable
var password = "hunter2";

// Reassign variable value with a new value
password = "hunter3";

console.log(password);
```

```
Output
'hunter3'
```

In an actual program, a password would most likely be securely stored in a database. This example, however, illustrates a situation in which we might need to update the value of a variable. The value of `password` was `hunter2`, but we reassigned it to `hunter3` which is the value JavaScript recognizes from that point forward.

Naming Variables

Variable names are known as *identifiers* in JavaScript. We discussed several of the rules of naming identifiers in Understanding Syntax and Code Structure in JavaScript, summarized here:

- Variable names can consist only of letters (`a-z`), numbers (`0-9`), dollar sign symbols (`$`), and underscores (`_`)
- Variable names cannot contain any whitespace characters (tabs or spaces)
- Numbers cannot begin the name of any variable
- There are several reserved keywords which cannot be used as the name of a variable
- Variable names are case sensitive

JavaScript also has the convention of using camel case (sometimes stylized as camelCase) in the names of functions and variables declared with `var` or `let`. This is the practice of writing the first word lowercase, and then capitalizing the first letter of every subsequent word with no spaces between them. Most variables that are not constants will follow this convention, with some exceptions. The names of variables that are constant, declared with the `const` keyword, are typically written in all uppercase.

This may seem like a lot of rules to learn, but it will very quickly become second nature to write valid and conventional variable names.

Difference Between `var`, `let`, and `const`

JavaScript has three different keywords to declare a variable, which adds an extra layer of intricacy to the language. The differences between the three are based on scope, hoisting, and reassignment.

Keyword	Scope	Hoisting	Can Be Reassigned	Can Be Redeclared
`var`	Function scope	Yes	Yes	Yes
`let`	Block scope	No	Yes	No

You may be wondering which of the three you should use in your own programs. A commonly accepted practice is to use `const` as much as possible, and `let` in the case of loops and reassignment. Generally, `var` can be avoided outside of working on legacy code.

Variable Scope

Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are *local* and *global*:

- **Global variables** are those declared outside of a block
- **Local variables** are those declared inside of a block

In the example below, we will create a global variable.

```
// Initialize a global variable
var creature = "wolf";
```

We learned that variables can be reassigned. Using local scope, we can actually create new variables with the same name as a variable in an outer scope without changing or reassigning the original value.

In the example below, we will create a global `species` variable. Within the function is a local variable with the same name. By sending them to the console, we can see how the variable's value is different depending on the scope, and the original value is not changed.

```
// Initialize a global variable
var species = "human";

function transform() {
  // Initialize a local, function-scoped variable
  var species = "werewolf";
  console.log(species);
}

// Log the global and local variable
console.log(species);
transform();
console.log(species);
```

```
Output
human
werewolf
human
```

In this example, the local variable is *function-scoped*. Variables declared with the `var` keyword are always function-scoped, meaning they recognize functions as having a separate scope. This locally-scoped variable is therefore not accessible from the global scope.

The new keywords `let` and `const`, however, are *block-scoped*. This means that a new, local scope is created from any kind of block, including function blocks, `if` statements, and `for` and `while` loops.

To illustrate the difference between function- and block-scoped variables, we will assign a new variable in an `if` block using `let`.

```
var fullMoon = true;

// Initialize a global variable
let species = "human";

if (fullMoon) {
  // Initialize a block-scoped variable
  let species = "werewolf";
  console.log(`It is a full moon. Lupin is currently a ${species}.`);
}

console.log(`It is not a full moon. Lupin is currently a ${species}.`);
```

```
Output
It is a full moon. Lupin is currently a werewolf.
It is not a full moon. Lupin is currently a human.
```

In this example, the `species` variable has one value globally (`human`), and another value locally (`werewolf`). If we were to use `var`, however, there would be a different result.

```
// Use var to initialize a variable
```

```
if (fullMoon) {
  // Attempt to create a new variable in a block
  var species = "werewolf";
  console.log(`It is a full moon. Lupin is currently a ${species}.`);
}

console.log(`It is not a full moon. Lupin is currently a ${species}.`);
```

```
Output
It is a full moon. Lupin is currently a werewolf.
It is not a full moon. Lupin is currently a werewolf.
```

In the result of this example, both the global variable and the block-scoped variable end up with the same value, `werewolf`. This is because instead of creating a new local variable with `var`, you are reassigning the same variable in the same scope. `var` does not recognize `if` to be part of a different, new scope. It is generally recommended that you declare variables that are block-scoped, as they produce code that is less likely to unintentionally override variable values.

Hoisting

In most of the examples so far, we've used `var` to *declare* a variable, and we have *initialized* it with a value. After declaring and initializing, we can access or reassign the variable.

If we attempt to use a variable before it has been declared and initialized, it will return `undefined`.

```
// Attempt to use a variable before declaring it
console.log(x);

// Variable assignment
var x = 100;
```

```
Output
undefined
```

However, if we omit the `var` keyword, we are no longer declaring the variable, only initializing it. It will return a ReferenceError and halt the execution of the script.

```
// Attempt to use a variable before declaring it
console.log(x);

// Variable assignment without var
x = 100;
```

```
Output
ReferenceError: x is not defined
```

The reason for this is due to *hoisting*, a behavior of JavaScript in which variable and function declarations are moved to the top of their scope. Since only the actual declaration is hoisted, not the initialization, the value in the first example returns `undefined`.

To demonstrate this concept more clearly, below is the code we wrote and how JavaScript actually interpreted it.

```
// The code we wrote
console.log(x);
var x = 100;

// How JavaScript interpreted it
var x;
console.log(x);
x = 100;
```

JavaScript saved `x` to memory as a variable before the execution of the script. Since it was still called before it was defined, the result is `undefined` and not `100`. However, it does not cause a `ReferenceError` and halt the script. Although the `var` keyword did not actually change location of the `var`, this is a helpful representation of how hoisting works. This behavior can cause issues, though, because the programmer who wrote this code likely expects the output of `x` to be `true`, when it is instead `undefined`.

We can also see how hoisting can lead to unpredictable results in the next example:

```
// Initialize x in the global scope
var x = 100;

function hoist() {
    var x = 200;
  }
  console.log(x);
}

hoist();
```

```
Output
undefined
```

In this example, we declared `x` to be `100` globally. Depending on an `if` statement, `x` could change to `200`, but since the condition was `false` it should not have affected the value of `x`. Instead, `x` was hoisted to the top of the `hoist()` function, and the value became `undefined`.

This type of unpredictable behavior can potentially cause bugs in a program. Since `let` and `const` are block-scoped, they will not hoist in this manner, as seen below.

```
// Initialize x in the global scope
let x = true;

function hoist() {
  // Initialize x in the function scope
  if (3 === 4) {
    let x = false;
  }
  console.log(x);
}

hoist();
```

```
Output
true
```

Duplicate declaration of variables, which is possible with `var`, will throw an error with `let` and `const`.

```
// Attempt to overwrite a variable declared with var
var x = 1;
```

```
console.log(x);
```

```
Output
2
```

```
// Attempt to overwrite a variable declared with let
let y = 1;
let y = 2;

console.log(y);
```

```
Output
Uncaught SyntaxError: Identifier 'y' has already been declared
```

To summarize, variables introduced with `var` have the potential of being affected by hoisting, a mechanism in JavaScript in which variable declarations are saved to memory. This may result in undefined variables in one's code. The introduction of `let` and `const` resolves this issue by throwing an error when attempting to use a variable before declaring it or attempting to declare a variable more than once.

Constants

Many programming languages feature *constants*, which are values that cannot be modified or changed. In JavaScript, the `const` identifier is modelled after constants, and the values assigned to a `const` cannot be reassigned.

It is common convention to write all `const` identifiers in uppercase. This marks them as readily distinguishable from other variable values.

In the example below, we initialize the variable `SPECIES` as a constant with the `const` keyword. Trying to reassign the variable will result in an error.

```
// Assign value to const
const SPECIES = "human";

// Attempt to reassign value
SPECIES = "werewolf";
```

```
Output
Uncaught TypeError: Assignment to constant variable.
```

Since `const` values cannot be reassigned, they need to be declared and initialized at the same time, or will also throw an error.

```
// Declare but do not initialize a const
const TODO;

console.log(TODO);
```

```
Output
Uncaught SyntaxError: Missing initializer in const declaration
```

Values that cannot change in programming are known as *immutable*, while values that can be changed are *mutable*. Although `const` values cannot be reassigned, they are mutable as it is possible to modify the properties of objects declared with `const`.

```
// Create a CAR object with two properties
const CAR = {
    color: "blue",
    price: 15000
}

// Modify a property of CAR
CAR.price = 20000;

console.log(CAR);
```

```
Output
{ color: 'blue', price: 20000 }
```

Constants are useful for making it clear to your future self and other programmers working on a project with you that the intended variable should not be reassigned. If you expect that a variable may be modified in the future, you will likely want to use `let` to declare the variable instead.

In this tutorial, we went over what a variable is, the rules of naming a variable, and how to reassign variable values. We also learned about scope and hoisting, some of the limitations of the original `var` keyword, as well as how `let` and `const` rectify those issues.

To compare how variables are used in other languages, you can read our tutorial on "How To Use Variables in Python 3."

By: Tania Rascia

Upvote (12) Subscribe

Editor:
Mark Drake

Tutorial Series

How To Code in JavaScript

JavaScript is a high-level, object-based, dynamic scripting language popular as a tool for making webpages interactive.

Show Tutorials

13 Comments

Leave a comment...

Log In to Comment

shawwesley *February 21, 2018*

0 will you be so kind, to add any additional articles and tutorials, please)

FredVasco *February 26, 2018*

0 It is not easy at all to use it

---

chasnz *April 9, 2018*

0 Hi Tania, Mark. Thanks for the post. One small typo in the hoisting section; You say "This behavior can cause issues, though, because the programmer who wrote this code likely expects the output of x to be true, when it is instead undefined." but I think you mean "This behavior can cause issues, though, because the programmer who wrote this code likely expects the output of x to be 100, when it is instead undefined."

---

emiralrashid *June 3, 2018*

0 WOAH, amazing tutorial!

paulharbuz *June 10, 2018*

0 Nice article, but i think you should add in this article the "temporal dead zone" for const and let declarations. Cheers

---

engineermarry123 *August 16, 2018*

0 Wow! Thanks, I got something new to learn.
I found good post on this topic which is http://www.visionfortech.com/2018/08/let-vs-var-in-javascript.html
So i thought let me share that post with all of you it is well written and well explained by its author.

---

ariananami642 *December 25, 2018*

0 اجاره خودرو

---

europcarkish *February 6, 2019*

0 Thanks for sharing

اجاره خودرو در کیش

Afshar *February 6, 2019*

0 Amazing article.
I will translate it into my personal website
https://www.nabatech.ir/

---

gapatour1988 *February 17, 2019*

0 Hello. thank you. You wrote a lot of fluent training. Please write a complete and comprehensive article about the JavaScript closures and the <this> keyword in the programming.

tours to iran are one of the best services of the gapatour.great memories with tour to iran

---

jainayushjain95 *February 20, 2019*

hghazad *February 20, 2019*

0 This was a great resource

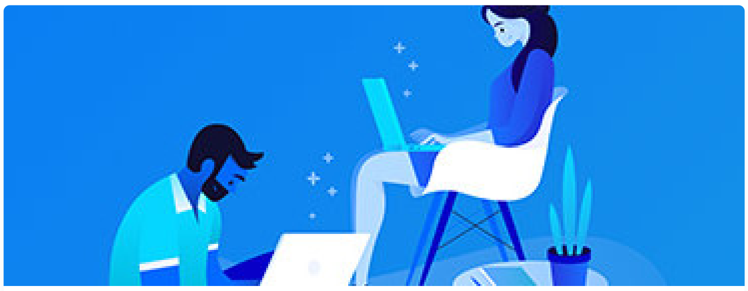
Thanks for the manager of this site

good lunch

richLeach *April 3, 2019*

0 Thanks so much, well done!

Running the code is what drives home the learning for me....

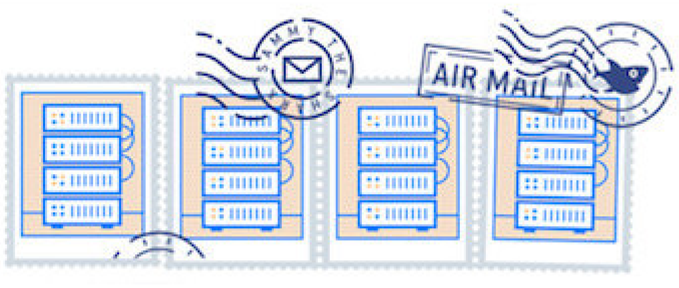


BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

CONNECT WITH OTHER DEVELOPERS

Find a DigitalOcean Meetup near you.



GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.

Featured on Community Intro to Kubernetes Learn Python 3 Machine Learning in Python Getting started with Go Migrate Node.js to Kubernetes

DigitalOcean Products Droplets Managed Databases Managed Kubernetes Spaces Object Storage Marketplace

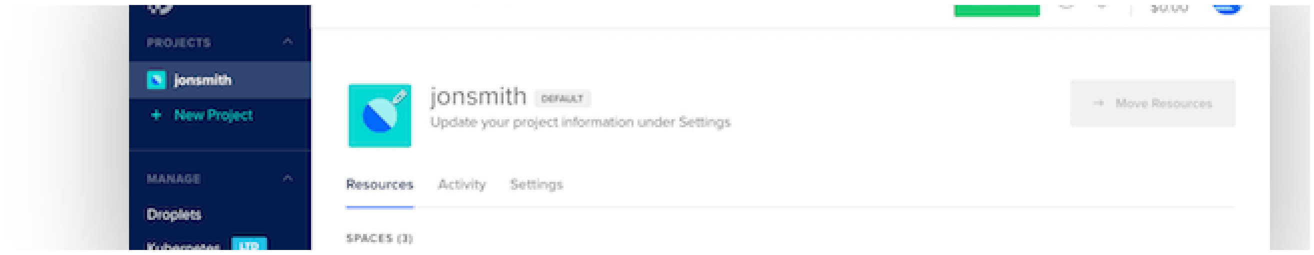
Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn More

Sign up for our newsletter. Get the latest tutorials on SysAdmin and open source topics.

Enter your email address Sign Up


PROJECTS
jonsmith
New Project
MANAGE
Droplets
jonsmith
DEFAULT
Update your project information under Settings
Move Resources
Resources
Activity
Settings
SPACES (3)