var, let, and const in JavaScript – the Differences Between These Keywords Explained

freecodecamp.org/news/differences-between-var-let-const-javascript

Dillion Megida January 11, 2023

Dillion Megida

In JavaScript, you can declare variables with the `var`, `let`, and `const` keywords. But what are the differences between them? That's what I'll explain in this tutorial.

I have a video version of this topic you can check out as well. 😇

If you're just starting out using JavaScript, a few things you may hear about these keywords are:

- `var` and `let` create variables that can be reassigned another value.
- `const` creates "constant" variables that cannot be reassigned another value.
- developers shouldn't use `var` anymore. They should use `let` or `const` instead.
- if you're not going to change the value of a variable, it is good practice to use `const`.

If you're just starting out using JavaScript, a few things you may hear about these keywords are:

- `var` and `let` create variables that can be reassigned another value.
- `const` creates "constant" variables that cannot be reassigned another value.
- developers shouldn't use `var` anymore. They should use `let` or `const` instead.
- if you're not going to change the value of a variable, it is good practice to use `const`.

The first two points are likely pretty self-explanatory. But what about why we shouldn't use var, or when to use let vs const? As we go through this tutorial, hopefully this will all make sense to you.

`var` vs `let` vs `const` – What's the Difference?

To analyze the differences between these keywords, I'll be using three factors:

- Scope of variables
- Redeclaration and reassignment
- Hoisting

I'm also writing separate tutorials about variable scope, variable hoisting, and variable redeclaration and reassignment – so you can learn more about them as well. They'll be out soon. :)

Let's start by looking at how these factors apply to variables declared with `var`.

How to Declare Variables with `var` in JavaScript

The scope of variables declared with `var`

Variables declared with `var` can have a **global** or **local** scope. Global scope is for variables declared outside functions, while local scope is for variables declared inside functions.

Let's see some examples, starting from global scope:

```
var number = 50

function print() {
  var square = number * number
  console.log(square)
}

console.log(number) // 50

print() // 2500
```

The `number` variable has a global scope – it's declared outside functions in the global space – so you can access it everywhere (inside and outside functions).

Let's see an example of local scope:

```
function print() {
  var number = 50
  var square = number * number
  console.log(square)
}

print() // 2500

console.log(number)
// ReferenceError: number is not defined
```

Here, we declared the `number` variable in the function `print`, so it has a local scope. This means that the variable can only be accessed inside that function. Any attempt to access the variable outside the function where it was declared will result in a **variable is not defined** reference error.

How to redeclare and reassign variables declared with `var`

Variables declared with `var` can be redeclared and reassigned. I'll explain what I mean with examples.

Here's how to declare a variable with `var`:

```
var number = 50
```

You have the `var` keyword, the name of the variable `number`, and an initial value **50**. If an initial value is not provided, the default value will be **undefined**:

```
var number

console.log(number)
// undefined
```

The `var` keyword allows for redeclaration. Here's an example:

```
var number = 50
console.log(number) // 50

var number = 100
console.log(number) // 100
```

As you can see, we have redeclared the variable `number` using the `var` keyword and an initial value of **100**.

The `var` keyword also allows for reassignment. In the code `var number = 50`, we assigned the **50** value to `number`. We can reassign another value anywhere in the code since it was declared with `var`. Here's what I mean:

```
var number = 50
console.log(number) // 50

number = 100
console.log(number) // 100

number = 200
console.log(number) // 200
```

Here, we're not redeclaring – rather, we're reassigning. After declaring the first time with an initial value of **50**, we reassign a new value of **100** and later on with a new value of **200**.

How to hoist variables declared with `var`

Variables declared with `var` are hoisted to the top of their global or local scope, which makes them accessible before the line they are declared. Here's an example:

```
console.log(number) // undefined

var number = 50

console.log(number) // 50
```

The `number` variable here has a global scope. Since it is declared with `var`, the variable is hoisted. This means that we can access the variable before the line where it was declared without errors.

But the variable is hoisted with a default value of **undefined**. So that's the value returned from the variable (until the line where the variable is declared with an initial value gets executed).

Let's see a local scope example:

```
function print() {
  var square1 = number * number
  console.log(square1)

  var number = 50

  var square2 = number * number
  console.log(square2)
}

print()
// NaN
// 2500
```

In the `print` function, `number` has a local scope. Due to hoisting, we can access the `number` variable before the line of declaration.

As we see in `square1`, we assign **number * number**. Since `number` is hoisted with a default value of **undefined**, `square1` will be **undefined * undefined** which results in **NaN**.

After the line of declaration with an initial value is executed, `number` will have a value of **50**. So in `square2`, **number * number** will be **50 * 50** which results in **2500**.

There are some problems with `var`, which we'll discuss at the end. Just know that it's generally not advisable to use it in your modern JavaScript projects.

How to Declare Variables with `let` in JavaScript

The scope of variables declared with `let`

Variables declared with `let` can have a **global**, **local**, or **block scope**. Block scope is for variables declared in a block. A block in JavaScript involves opening and closing curly braces:

```
{
  // a block
}
```

You can find blocks in `if`, `loop`, `switch`, and a couple of other statements. Any variables declared in such blocks with the `let` keyword will have a block scope. Also, you can't access these variables outside the block. Here's an example showing a global, local, and block scope:

```
let number = 50

function print() {
  let square = number * number

  if (number < 60) {
    var largerNumber = 80
    let anotherLargerNumber = 100

    console.log(square)
  }

  console.log(largerNumber)
  console.log(anotherLargerNumber)
}

print()
// 2500
// 80
// ReferenceError: anotherLargerNumber is not defined
```

In this example, we have a global scope variable `number` and a local scope variable `square`. There's also block scope variable `anotherLargerNumber` because it is declared with `let` in a block.

`largerNumber`, on the other hand – though declared in a block – does not have a block scope because it is declared with `var`. So `largerNumber` has a local scope as it is declared in the function `print`.

We can access `number` everywhere. We can only access `square` and `largerNumber` in the function because they have local scope. But accessing `anotherLargerNumber` outside the block throws an **anotherLargerNumber is not defined** error.

How to redeclare and reassign variables declared with `let`

Just like `var`, variables declared with `let` can be reassigned to other values, but they cannot be redeclared. Let's see a reassignment example:

```
let number = 50
console.log(number) // 50

number = 100
console.log(number) // 100
```

Here, we reassigned another value **100** after the initial declaration of **50**.

But redeclaring a variable with `let` will throw an error:

```
let number = 50

let number = 100
// SyntaxError: Identifier 'number' has already been declared
```

You see we get a syntax error: **Identifier 'number' has already been declared**.

How to hoist variables declared with `let`

Variables declared with `let` are hoisted to the top of their global, local, or block scope, but their hoisting is a little different from the one with `var`.

`var` variables are hoisted with a default value of **undefined**, which makes them accessible before their line of declaration (as we've seen above).

But, `let` variables are hoisted without a default initialization. So when you try to access such variables, instead of getting **undefined**, or **variable is not defined** error, you get **cannot access variable before initialization**. Let's see an example:

```
console.log(number)
// ReferenceError: Cannot access 'number' before initialization

let number = 50
```

Here, we have a global variable, `number` declared with `let`. By trying to access this variable before the line of declaration, we get **ReferenceError: Cannot access 'number' before initialization**.

Here's another example with a local scope variable:

```
function print() {
  let square = number * number

  let number = 50
}

print()
// ReferenceError: Cannot access 'number' before initialization
```

Here we have a local scope variable, `number`, declared with `let`. By accessing it before the line of declaration again, we get the **cannot access 'number' before initialization** reference error

How to Declare Variables with `const` in JavaScript

The scope of variables declared with `const`

Variables declared with `const` are similar to `let` in regards to **scope**. Such variables can have a **global**, **local**, or **block** scope.

Here is an example:

```
const number = 50

function print() {
  const square = number * number

  if (number < 60) {
    var largerNumber = 80
    const anotherLargerNumber = 100

    console.log(square)
  }

  console.log(largerNumber)
  console.log(anotherLargerNumber)
}

print()
// 2500
// 80
// ReferenceError: anotherLargerNumber is not defined
```

This is from our previous example, but I've replaced `let` with `const`. As you can see here, the `number` variable has a global scope, `square` has a local scope (declared in the `print` function), and `anotherLargeNumber` has a block scope (declared with `const`).

There's also `largeNumber`, declared in a block. But because it is with `var`, the variable only has a local scope. Therefore, it can be accessed outside the block.

Because `anotherLargeNumber` has a block scope, accessing it outside the block throws an **anotherLargerNumber is not defined**.

How to redeclare and reassign variables declared with `const`

In this regard, `const` is different from `var` and `let`. `const` is used for declaring **constant** variables – which are variables with values that cannot be changed. So such variables cannot be redeclared, and neither can they be reassigned to other values. Attempting such would throw an error.

Let's see an example with redeclaration:

```
const number = 50

const number = 100

// SyntaxError: Identifier 'number' has already been declared
```

Here, you can see the **Identifier has already been declared** syntax error.

Now, let's see an example with reassignment:

```
const number = 50

number = 100

// TypeError: Assignment to constant variable
```

Here, you can see the **Assignment to constant variable** type error.

How to hoist variables declared with `const`

Variables declared with `const`, just like `let`, are hoisted to the top of their global, local, or block scope – but without a default initialization.

`var` variables, as you've seen earlier, are hoisted with a default value of **undefined** so they can be accessed before declaration without errors. Accessing a variable declared with `const` before the line of declaration will throw a **cannot access variable before initialization** error.

Let's see an example:

```
console.log(number)
// ReferenceError: Cannot access 'number' before initialization

const number = 50
```

Here, `number` is a globally scoped variable declared with `const`. By trying to access this variable before the line of declaration, we get **ReferenceError: Cannot access 'number' before initialization**. The same will occur if it was a locally scoped variable.

Here's an article to learn more about Hoisting in JavaScript with let and const – and How it Differs from var.

Wrap up

Here's a table summary showing the differences between these keywords:

Keyword	Scope	Redeclaration & Reassignment	Hoisting
`var`	Global, Local	yes & yes	yes, with default value
`let`	Global, Local, Block	no & yes	yes, without default value
`const`	Global, Local, Block	no & no	yes, without default value

These factors I've explained, play a role in determining how you declare variables in JavaScript.

If you never want a variable to change, `const` is the keyword to use.

If you want to reassign values:

- and you want the hoisting behavior, `var` is the keyword to use
- if you don't want it, `let` is the keyword for you

The hoisting behavior can cause unexpected bugs in your application. That's why developers are generally advised to avoid `var` and stick to `let` and `cost`.

I hope you learned something from this article.

Dillion Megida
Developer Advocate and Content Creator passionate about sharing my knowledge on Tech. I simplify JavaScript / ReactJS / NodeJS / Frameworks / TypeScript / et al My YT channel: youtube.com/c/deeecode

---