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

In JavaScript, you can declare variables with the var, let and 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](https://youtu.be/Gd_JG3e1g4A) 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.

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

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.