scraperapi

What is the difference between LET and CONST in JavaScript?

The JavaScript let and const keywords provide block-level scope, but there is a slight difference in how they behave. With const, you can *not* re-assign a value to the variable. With let, you can.

Over time, JavaScript applications have grown in complexity. As a result of the increased code complexity programmers have been faced with a challenging dilemma: build applications that satisfy ever-evolving business requirements, yet continue to work with the same tools. It only makes sense that JavaScript would be in need of improvements, since for much of its history, functions were the only tools available to achieve scope. But, for several years, block-level scope was a feature that was sorely lacking. Then along came the ECMAScript-2015 specification that finally met that need with the **let** and **const** keywords.

The JavaScript **let** and **const** keywords are quite similar, in that they create block-level scope. However, they do differ a bit in the way that they behave. For example, the JavaScript **let** keyword is similar to the var keyword in that assignments can be changed. On the other hand, the JavaScript **const** keyword differs in that assignments can not be changed. So, once you declare a variable using the const keyword, you cannot re-assign a value to that variable. This does not mean that the variable is immutable. If you assign an object to a variable using the const keyword, you can mutate that object. You just can't re-assign that variable with a new value. Let's take a look at some examples.

Example # 1 A

```
var i = 100;

{
    let i = 50;

    console.log('(A) i = ' + i); //50
}

console.log('(B) i = ' + i); //100
```

let-example-1.js hosted with ❤ by GitHub view raw

Example # 1 B

```
//let-example-1: output
(A) i = 50
(B) i = 100
```

let-example-1-output.js hosted with ❤ by GitHub view raw

In Example # 1 A, we have two different versions of the **"i"** variable. I say *"two different versions"* because the same variable name exists in two difference scopes, the global scope and a block scope. The block scope exists between the two curly braces: **"{ }"**. Then inside of the two curly braces, I used the JavaScript let keyword to declare a second **"i"** variable. Because we used the let keyword, that particular **"i"** variable is scoped to the block in which it was declared. And because of this, the **console.log()** statement on line # 6 outputs **50**. I'll just note here that it may seem a little odd at first to declare a variable anywhere other than at the top of the function, but this actually is the correct syntax; if we want a block-level scope variable, we use the let keyword inside of a set of curly braces.

Take a look at Example # 1 B. Notice how, in the second **console.log()** statement, the output is **100**. This is because that second **console.log()** statement is in the global scope, and in that scope the **"i"** variable is equal to **100**. So, there we have it: two different scopes without even having used a function.

Example # 2 A

```
1   var i = 0,
2       j = 100;
3
4   for (; i < 10; i++) {
5       //j is now private to this block
6       let j = 50;
7
8       //increment j
9       j++;
10
11      //j will always be 51 in the console
12      console.log('i = ' + i + ' | j = ' + j);
13  }
```

let-example-2.js hosted with ❤ by GitHub view raw

Example # 2 B

```
i = 0 | j = 51
i = 1 | j = 51
i = 2 | j = 51
i = 3 | j = 51
i = 4 | j = 51
i = 5 | j = 51
i = 6 | j = 51
i = 7 | j = 51
i = 8 | j = 51
i = 9 | j = 51
```

let-example-2-output.js hosted with ❤ by GitHub view raw

Now, in Example # 2 A, there are two **"j"** variables.
The first **"j"** variable is a global, equal to **100**, and the second is defined inside of the for loop. And because it's defined inside of a block, it has block-level scope. Now look at example # 2 B. Because **"i"** is global, the **"i"** variable increments, just as we would expect. But notice that the **"j"** variable is always **50** in each **console.log()** statement, even though there is a global **"j"** variable. This is because on each iteration of the for loop, a block-level **"j"** variable is declared using the let keyword, and it is incremented (just to demonstrate that with let, we can re-assign a variable value). So in this case, with each iteration of the for loop we have a block-scoped **"j"** variable and it is always **51**. Note that the global **"j"** variable is ignored on line # 12.

Example # 3 A

```
var i = 100;

{
    const i = 50;

    console.log('(A) i = ' + i); //50
}

console.log('(B) i = ' + i); //100
```

const-example-1.js hosted with ❤ by GitHub view raw

Example # 3 B

```
// const-example-1: output
(A) i = 50
(B) i = 100
```

const-example-1-output.js hosted with ❤ by GitHub view raw

In Examples # 3 A and # 3 B you'll see a similarity to Examples # 1 A and # 1 B, the only difference being the use of the use of the const keyword instead of let when declaring our block-level version of the **"i"** variable.

Example # 4 A

```
 1  var i = 0,
 2  j = 100;
 3
 4  for (; i < 10; i++) {
 5      //j is now private to this block
 6      const j = 50;
 7
 8      //attempt to increment j (throws a TypeError)
 9      j++;
10
11      //this code never even has a chance to execute
12      console.log('i =' + i + ' | j = ' + j);
13  }
```

const-example-2.js hosted with ❤ by GitHub view raw

Example # 4 B

```
// const-example-2: output
Uncaught TypeError: Assignment to constant variable.
```

const-example-2-output.js hosted with ❤ by GitHub view raw

Now here in Example # 4 A, we've run into a problem. We tried to take the same approach as Example # 2 A, that is, we tried to increment the **"j"** variable declared on line # 6. The problem, though, is that when you use the JavaScript const keyword, you cannot re-assign a new value to a variable. So when you look at Example # 4 B, you'll see that we never see the full output of the for loop that we expected, because line # 9 of Example # 4A throws a TypeError. This is because when we try to change the value of **"j"**, we find that this is not possible because it was created using the const keyword. In other words: it's a constant.

Example # 5 A

```
var i = 0,
    j = 100;
```

```

for (; i < 10; i++) {
    //j is now private to this block
    const j = 50;

    //j will always be 50 in the console
    console.log('i = ' + i + ' | j = ' + j);
}
```

const-example-3.js hosted with ❤ by GitHub view raw

Example # 5 B

```
i = 0 | j = 50
i = 1 | j = 50
i = 2 | j = 50
i = 3 | j = 50
i = 4 | j = 50
i = 5 | j = 50
i = 6 | j = 50
i = 7 | j = 50
i = 8 | j = 50
i = 9 | j = 50
```

const-example-3-output.js hosted with ❤ by GitHub view raw

Now Example # 5 A is virtually identical to Example # 4 A, except that we have not tried to increment the **"j"** variable. And when you look at Example # 5 B, you'll see that we no longer have an error. In the console, the value of **"j"** is **50** every time.

Summary

So to recap, we now know that the JavaScript **let** and **const** keywords allow you to create block-level scope for variables, which, in turn, negates the need to always use functions to create scope. With block-level scope, all you need are the curly braces **"{ }"**, and within that block, any variable created using let or const is private (or local) to that block. This is particularly helpful with for-loops. And a very important thing to keep in mind: with **const**, you cannot re-assign values to a variable. In other words, any variable created with the **const** keyword is a constant and the assignment cannot be changed.

A lot to take in here, but I think it's worth keeping on your radar, given this very functional block-level scope now increasingly available in browsers.