JavaScript let: Declaring Block-Scoped Variables

Summary: in this tutorial, you will learn how to use the JavaScript `let` keyword to declare block-scoped variables.

Introduction to the JavaScript let keyword

In ES5, when you declare a variable using the `var` keyword, the scope of the variable is either global or local. If you declare a variable outside of a function, the scope of the variable is global. When you declare a variable inside a function, the scope of the variable is local.

ES6 provides a new way of declaring a variable by using the `let` keyword. The `let` keyword is similar to the `var` keyword, except that these variables are blocked-scope. For example:

```
let variable_name;
```

In JavaScript, blocks are denoted by curly braces `{}`, for example, the `if else`, `for`, `do while`, `while`, `try catch` and so on:

```
if(condition) {
   // inside a block
}
```

See the following example:

```
let x = 10;
if (x == 10) {
    let x = 20;
    console.log(x); // 20:  reference x inside the block
}
console.log(x); // 10: reference at the begining of the script
```

How the script works:

- First, declare a variable `x` and initialize its value to 10.
- Second, declare a new variable with the same name `x` inside the `if` block but with an initial value of 20.
- Third, output the value of the variable `x` inside and after the `if` block.

Because the `let` keyword declares a block-scoped variable, the `x` variable inside the `if` block is a **new variable** and it shadows the `x` variable declared at the top of the script. Therefore, the value of `x` in the console is `20`.

When the JavaScript engine completes executing the `if` block, the `x` variable inside the `if` block is out of scope. Therefore, the value of the `x` variable that following the `if` block is 10.

JavaScript let and global object

When you declare a global variable using the `var` keyword, you add that variable to the property list of the global object. In the case of the web browser, the global object is the `window`. For example:

```
var a = 10;
console.log(window.a); // 10
```

However, when you use the `let` keyword to declare a variable, that variable is **not** attached to the global object as a property. For example:

```
let b = 20;
console.log(window.b); // undefined
```

JavaScript let and callback function in a for loop

See the following example.

```
for (var i = 0; i < 5; i++) {
    setTimeout(function () {
```

```
        console.log(i);
    }, 1000);
}
```

The intention of the code is to output numbers from 0 to 4 to the console every second. However, it outputs the number `5` five times:

```
5
5
5
5
5
```

In this example, the variable `i` is a global variable. After the loop, its value is 5. When the callback functions are passed to the `setTimeout()` function executes, they reference the same variable `i` with the value 5.

In ES5, you can fix this issue by creating another scope so that each callback function references a new variable. And to create a new scope, you need to create a function. Typically, you use the IIFE pattern as follows:

```
for (var i = 0; i < 5; i++) {
    (function (j) {
        setTimeout(function () {
            console.log(j);
        }, 1000);
    })(i);
}
```

Output:

```
0
1
2
3
4
```

In ES6, the `let` keyword declares a new variable in each loop iteration. Therefore, you just need to replace the `var` keyword with the `let` keyword to fix the issue:

```
for (let i = 0; i < 5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
}
```

To make the code completely ES6 style, you can use an arrow function as follows:

```
for (let i = 0; i < 5; i++) {
    setTimeout(() => console.log(i), 1000);
}
```

Note that you'll learn more about the arrow functions in the later tutorial.

Redeclaration

The `var` keyword allows you to redeclare a variable without any issue:

```
var counter = 0;
var counter;
console.log(counter); // 0
```

However, redeclaring a variable using the `let` keyword will result in an error:

```
let counter = 0;
let counter;
console.log(counter);
```

Here's the error message:

```
Uncaught SyntaxError: Identifier 'counter' has already been declared
```

JavaScript let variables and hoisting

Let's examine the following example:

```
{
    console.log(counter); //
    let counter = 10;
}
```

This code causes an error:

```
Uncaught ReferenceError: Cannot access 'counter' before initialization
```

In this example, accessing the `counter` variable before declaring it causes a `ReferenceError`. You may think that a variable declaration using the `let` keyword does not **hoist,** but it does**.**

In fact, the JavaScript engine will hoist a variable declared by the `let` keyword to the top of the block. However, the JavaScript engine does not initialize the variable. Therefore, when you reference an uninitialized variable, you'll get a `ReferenceError`.

Temporal death zone (TDZ)

A variable declared by the `let` keyword has a so-called temporal dead zone (TDZ). The TDZ is the time from the start of the block until the variable declaration is processed.

The following example illustrates that the temporal dead zone is time-based, not location-based.

```
{ // enter new scope, TDZ starts
    let log = function () {
        console.log(message); // messagedeclared later
    };

    // This is the TDZ and accessing log
    // would cause a ReferenceError

    let message= 'Hello'; // TDZ ends
```

```
    log(); // called outside TDZ
}
```

In this example:

First, the curly brace starts a new block scope, therefore, the TDZ starts.

Second, the `log()` function expression accesses the `message` variable. However, the `log()` function has not been executed yet.

Third, declare the `message` variable and initialize its value to `'Hello'`. The time from the start of the block scope to the time that the `message` variable is accessed is called a *temporal death zone*. When the JavaScript engine processes the declaration, the TDZ ends.

Finally, call the `log()` function that accesses the `message` variable outside of the TDZ.

Note that if you access a variable declared by the `let` keyword in the TDZ, you'll get a `ReferenceError` as illustrated in the following example.

```
{ // TDZ starts
    console.log(typeof myVar); // undefined
    console.log(typeof message); // ReferenceError
    let message; // TDZ ends
}
```

Notice that `myVar` variable is a non-existing variable, therefore, its type is undefined.

The temporal death zone prevents you from accidentally referencing a variable before its declaration.

Summary

- Variables declared using the `let` keyword are block-scoped, are not initialized to any value, and are not attached to the global object.
- Redeclaring a variable using the `let` keyword will cause an error.
- A temporal dead zone of a variable is declared using the `let` keyword starts from the block until the initialization is evaluated.