JavaScript Closures

❮ Previous Next ❯

JavaScript variables can belong to the **local** or **global** scope.

Global variables can be made local (private) with **closures**.

Global Variables

A `function` can access all variables defined **inside** the function, like this:

Example

```
function myFunction() {
  let a = 4;
  return a * a;
}
```

Try it Yourself »

But a `function` can also access variables defined **outside** the function, like this:

Example

```
let a = 4;
function myFunction() {
```

```
```

Try it Yourself »

In the last example, **a** is a **global** variable.

In a web page, global variables belong to the page.

Global variables can be used (and changed) by all other scripts in the page.

In the first example, **a** is a **local** variable.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Note

Variables created **without** a declaration keyword (`var` , `let` , or `const`) are always global, even if they are created inside a function.

Example

```
function myFunction() {
  a = 4;
}
```

Try it Yourself »

Global variables live until the page is discarded, like when you navigate to another page or close the window.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

A Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a `function` to increase the counter:

Example

```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

// The counter should now be 3
```

Try it Yourself »

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

Example

```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 0
```

Try it Yourself »

It did not work because we display the global counter instead of the local counter.

We can remove the global counter and access the local counter by letting the function return it:

Example

```
// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
  return counter;
}

// Call add() 3 times
add();
add();
```

```
//The counter should now be 3. But it is 1.
```

Try it Yourself »

It did not work because we reset the local counter every time we call the function.

A JavaScript inner function can solve this.

JavaScript Nested Functions

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function `plus()` has access to the `counter` variable in the parent function:

Example

```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}
```

Try it Yourself »

This could have solved the counter dilemma, if we could reach the `plus()` function from the outside.

We also need to find a way to execute `counter = 0` only once.

JavaScript Closures

Remember self-invoking functions? What does this function do?

Example

```
const add = (function () {
  let counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();

// the counter is now 3
```

Try it Yourself »

Example Explained

The variable `add` is assigned to the return value of a self-invoking function.

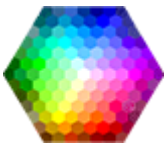
The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

COLOR PICKER



HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
Angular Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate

C# Certificate
XML Certificate