Example 1:

function sayHello2(name) {

var text = 'Hello ' + name; // Local variable

var say = function() { console.log(text); }

return say;

}

var say2 = sayHello2('Bob');

say2(); // logs "Hello Bob"

The above code has a closure because the anonymous function function() { console.log(text); } is declared inside another function, sayHello2() in this example. In JavaScript, if you use the function keyword inside another function, you are creating a closure.

In C and most other common languages, after a function returns, all the local variables are no longer accessible because the stack-frame is destroyed.

In JavaScript, if you declare a function within another function, then the local variables of the outer function can remain accessible after returning from it. This is demonstrated above, because we call the function say2() after we have returned from sayHello2(). Notice that the code that we call references the variable text, which was a local variable of the function sayHello2().

Looking at the output of say2.toString(), we can see that the code refers to the variable text. The anonymous function can reference text which holds the value 'Hello Bob' because the local variables of sayHello2() have been secretly kept alive in a closure.

Example 2:

This example shows that the local variables are not copied — they are kept by reference. It is as though the stack-frame stays alive in memory even after the outer function exits!

function say667() {

// Local variable that ends up within closure

var num = 42;

var say = function() { console.log(num); }

num++;

return say;

}

var sayNumber = say667();

sayNumber(); // logs 43

Example 3:

All three global functions have a common reference to the same closure because they are all declared within a single call to setupSomeGlobals().

var gLogNumber, gIncreaseNumber, gSetNumber;

function setupSomeGlobals() {

// Local variable that ends up within closure

var num = 42;

// Store some references to functions as global variables

gLogNumber = function() { console.log(num); }

gIncreaseNumber = function() { num++; }

gSetNumber = function(x) { num = x; }

}

setupSomeGlobals();

gIncreaseNumber();

gLogNumber(); // 43

gSetNumber(5);

gLogNumber(); // 5

var oldLog = gLogNumber; //oldlog = console.log(5); but oldLog does not have access to . //var num because it is not defined in setupSomeGlobals.

setupSomeGlobals();

gLogNumber(); // 42

oldLog() // 5

The three functions have shared access to the same closure — the local variables of setupSomeGlobals() when the three functions were defined.

Note that in the above example, if you call setupSomeGlobals() again, then a new closure (stack-frame!) is created.

The old gLogNumber, gIncreaseNumber, gSetNumber variables are overwritten with new functions that have the new closure. (In JavaScript, whenever you declare a function inside another function, the inside function(s) is/are recreated again each time the outside function is called.)

Example 4

This example shows that the closure contains any local variables that were declared inside the outer function before it exited. Note that the variable alice is actually declared after the anonymous function. The anonymous function is declared first and when that function is called it can access the alice variable because alice is in the same scope (JavaScript does variable hoisting). Also sayAlice()() just directly calls the function reference returned from sayAlice() — it is exactly the same as what was done previously but without the temporary variable.

function sayAlice() {

var say = function() { console.log(alice); }

// Local variable that ends up within closure

var alice = 'Hello Alice';

return say;

}

sayAlice()();// logs "Hello Alice"

JavaScript Hoisting

Source: <http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html>

“JavaScript has **function-level scope**. This is radically different from the C family. Blocks, such as if statements, **do not** create a new scope. Only functions create a new scope.”

**JavaScript Scoping and Hoisting**

Do you know what value will be alerted if the following is executed as a JavaScript program?

var foo = 1;

function bar() {

if (!foo) {

var foo = 10;

}

alert(foo);

}

bar();

If it surprises you that the answer is “10”, then this one will probably really throw you for a loop:

var a = 1;

function b() {

a = 10;

return;

function a() {}

}

b();

alert(a);

Here, of course, the browser will alert “1”. So what’s going on here? While it might seem strange, dangerous, and confusing, this is actually a powerful and expressive feature of the language. I don’t know if there is a standard name for this specific behavior, but I’ve come to like the term “hoisting”. This article will try to shed some light on this mechanism, but first lets take a necessary detour to understand JavaScript’s scoping.

Scoping in JavaScript

One of the sources of most confusion for JavaScript beginners is scoping. Actually, it’s not just beginners. I’ve met a lot of experienced JavaScript programmers who don’t fully understand scoping. The reason scoping is so confusing in JavaScript is because it looks like a C-family language. Consider the following C program:

#include <stdio.h>

int main() {

int x = 1;

printf("%d, ", x); // 1

if (1) {

int x = 2;

printf("%d, ", x); // 2

}

printf("%d\n", x); // 1

}

The output from this program will be 1, 2, 1. This is because C, and the rest of the C family, has block-level scope. When control enters a block, such as the if statement, new variables can be declared within that scope, without affecting the outer scope. This is not the case in JavaScript. Try the following in Firebug:

var x = 1;

console.log(x); // 1

if (true) {

var x = 2;

console.log(x); // 2

}

console.log(x); // 2

In this case, Firebug will show 1, 2, 2. This is because JavaScript has function-level scope. This is radically different from the C family. Blocks, such as if statements, do not create a new scope. Only functions create a new scope.

To a lot of programmers who are used to languages like C, C++, C#, or Java, this is unexpected and unwelcome. Luckily, because of the flexibility of JavaScript functions, there is a workaround. If you must create temporary scopes within a function, do the following:

function foo() {

var x = 1;

if (x) {

(function () {

var x = 2;

// some other code

}());

}

// x is still 1.

}

This method is actually quite flexible, and can be used anywhere you need a temporary scope, not just within block statements. However, I strongly recommend that you take the time to really understand and appreciate JavaScript scoping. It’s quite powerful, and one of my favorite features of the language. If you understand scoping, hoisting will make a lot more sense to you.

**Declarations, Names, and Hoisting**

In JavaScript, a name enters a scope in one of four basic ways:

Language-defined: All scopes are, by default, given the names this and arguments.

Formal parameters: Functions can have named formal parameters, which are scoped to the body of that function.

Function declarations: These are of the form function foo() {}.

Variable declarations: These take the form var foo;.

Function declarations and variable declarations are always moved (“hoisted”) invisibly to the top of their containing scope by the JavaScript interpreter. Function parameters and language-defined names are, obviously, already there. This means that code like this:

function foo() {

bar();

var x = 1;

}

is actually interpreted like this:

function foo() {

var x;

bar();

x = 1;

}

It turns out that it doesn’t matter whether the line that contains the declaration would ever be executed. The following two functions are equivalent:

function foo() {

if (false) {

var x = 1;

}

return;

var y = 1;

}

function foo() {

var x, y;

if (false) {

x = 1;

}

return;

y = 1;

}

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted. This is not the case with function declarations, where the entire function body will be hoisted as well. But remember that there are two normal ways to declare functions. Consider the following JavaScript:

function test() {

foo(); // TypeError "foo is not a function"

bar(); // "this will run!"

var foo = function () { // function expression assigned to local variable 'foo'

alert("this won't run!");

}

function bar() { // function declaration, given the name 'bar'

alert("this will run!");

}

}

test();

In this case, only the function declaration has its body hoisted to the top. The name ‘foo’ is hoisted, but the body is left behind, to be assigned during execution.

That covers the basics of hoisting, which is not as complex or confusing as it seems. Of course, this being JavaScript, there is a little more complexity in certain special cases.

Name Resolution Order

The most important special case to keep in mind is name resolution order. Remember that there are four ways for names to enter a given scope. The order I listed them above is the order they are resolved in. In general, if a name has already been defined, it is never overridden by another property of the same name. This means that a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored. There are a few exceptions:

The built-in name arguments behaves oddly. It seems to be declared following the formal parameters, but before function declarations. This means that a formal parameter with the name arguments will take precedence over the built-in, even if it is undefined. This is a bad feature. Don’t use the name arguments as a formal parameter.

Trying to use the name this as an identifier anywhere will cause a SyntaxError. This is a good feature.

If multiple formal parameters have the same name, the one occurring latest in the list will take precedence, even if it is undefined.

Named Function Expressions

You can give names to functions defined in function expressions, with syntax like a function declaration. This does not make it a function declaration, and the name is not brought into scope, nor is the body hoisted. Here’s some code to illustrate what I mean:

foo(); // TypeError "foo is not a function"

bar(); // valid

baz(); // TypeError "baz is not a function"

spam(); // ReferenceError "spam is not defined"

var foo = function () {}; // anonymous function expression ('foo' gets hoisted)

function bar() {}; // function declaration ('bar' and the function body get hoisted)

var baz = function spam() {}; // named function expression (only 'baz' gets hoisted)

foo(); // valid

bar(); // valid

baz(); // valid

spam(); // ReferenceError "spam is not defined"

How to Code With This Knowledge

Now that you understand scoping and hoisting, what does that mean for coding in JavaScript? The most important thing is to always declare your variables with a var statement. I strongly recommend that you have exactly one var statement per scope, and that it be at the top. If you force yourself to do this, you will never have hoisting-related confusion. However, doing this can make it hard to keep track of which variables have actually been declared in the current scope. I recommend using JSLint with the onevar option to enforce this. If you’ve done all of this, your code should look something like this:

/\*jslint onevar: true [...] \*/

function foo(a, b, c) {

var x = 1,

bar,

baz = "something";

}