3.10 Variable Scope

The *scope* of a variable is the region of your program source code in which it is defined. A *global* variable has global scope; it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are *local* variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable:

```
var scope = "global";         // Declare a global variable
function checkscope() {
    var scope = "local";      // Declare a local variable with the same name
    return scope;             // Return the local value, not the global one
}
checkscope()                  // => "local"
```

Although you can get away with not using the `var` statement when you write code in the global scope, you must always use `var` to declare local variables. Consider what happens if you don't:

```
scope = "global";            // Declare a global variable, even without var.
function checkscope2() {
    scope = "local";         // Oops! We just changed the global variable.
    myscope = "local";       // This implicitly declares a new global variable.
    return [scope, myscope]; // Return two values.
}
checkscope2()                // => ["local", "local"]: has side effects!
scope                        // => "local": global variable has changed.
myscope                      // => "local": global namespace cluttered up.
```

Function definitions can be nested. Each function has its own local scope, so it is possible to have several nested layers of local scope. For example:

```
var scope = "global scope";          // A global variable
function checkscope() {
    var scope = "local scope";       // A local variable
    function nested() {
        var scope = "nested scope";  // A nested scope of local variables
        return scope;                // Return the value in scope here
    }
    return nested();
}
checkscope()                         // => "nested scope"
```

3.10.1 Function Scope and Hoisting

In some C-like programming languages, each block of code within curly braces has its own scope, and variables are not visible outside of the block in which they are declared. This is called *block scope*, and JavaScript does *not* have it. Instead, JavaScript uses

function scope: variables are visible within the function in which they are defined and within any functions that are nested within that function.

In the following code, the variables `i`, `j`, and `k` are declared in different spots, but all have the same scope—all three are defined throughout the body of the function:

```
function test(o) {
    var i = 0;                      // i is defined throughout function
    if (typeof o == "object") {
        var j = 0;                  // j is defined everywhere, not just block
        for(var k=0; k < 10; k++) { // k is defined everywhere, not just loop
            console.log(k);         // print numbers 0 through 9
        }
        console.log(k);             // k is still defined: prints 10
    }
    console.log(j);                 // j is defined, but may not be initialized
}
```

JavaScript's function scope means that all variables declared within a function are visible *throughout* the body of the function. Curiously, this means that variables are even visible before they are declared. This feature of JavaScript is informally known as *hoisting*: JavaScript code behaves as if all variable declarations in a function (but not any associated assignments) are "hoisted" to the top of the function. Consider the following code:

```
var scope = "global";
function f() {
    console.log(scope);  // Prints "undefined", not "global"
    var scope = "local"; // Variable initialized here, but defined everywhere
    console.log(scope);  // Prints "local"
}
```

You might think that the first line of the function would print "global", because the `var` statement declaring the local variable has not yet been executed. Because of the rules of function scope, however, this is not what happens. The local variable is defined throughout the body of the function, which means the global variable by the same name is hidden throughout the function. Although the local variable is defined throughout, it is not actually initialized until the `var` statement is executed. Thus, the function above is equivalent to the following, in which the variable declaration is "hoisted" to the top and the variable initialization is left where it is:

```
function f() {
    var scope;          // Local variable is declared at the top of the function
    console.log(scope); // It exists here, but still has "undefined" value
    scope = "local";    // Now we initialize it and give it a value
    console.log(scope); // And here it has the value we expect
}
```

In programming languages with block scope, it is generally good programming practice to declare variables as close as possible to where they are used and with the narrowest possible scope. Since JavaScript does not have block scope, some programmers make a point of declaring all their variables at the top of the function, rather than trying to

declare them closer to the point at which they are used. This technique makes their source code accurately reflect the true scope of the variables.

3.10.2 Variables As Properties

When you declare a global JavaScript variable, what you are actually doing is defining a property of the global object (§3.5). If you use `var` to declare the variable, the property that is created is nonconfigurable (see §6.7), which means that it cannot be deleted with the `delete` operator. We've already noted that if you're not using strict mode and you assign a value to an undeclared variable, JavaScript automatically creates a global variable for you. Variables created in this way are regular, configurable properties of the global object and they can be deleted:

```
var truevar = 1;     // A properly declared global variable, nondeletable.
fakevar = 2;         // Creates a deletable property of the global object.
this.fakevar2 = 3;   // This does the same thing.
delete truevar       // => false: variable not deleted
delete fakevar       // => true: variable deleted
delete this.fakevar2 // => true: variable deleted
```

JavaScript global variables are properties of the global object, and this is mandated by the ECMAScript specification. There is no such requirement for local variables, but you can imagine local variables as the properties of an object associated with each function invocation. The ECMAScript 3 specification referred to this object as the "call object," and the ECMAScript 5 specification calls it a "declarative environment record." JavaScript allows us to refer to the global object with the `this` keyword, but it does not give us any way to refer to the object in which local variables are stored. The precise nature of these objects that hold local variables is an implementation detail that need not concern us. The notion that these local variable objects exist, however, is an important one, and it is developed further in the next section.

3.10.3 The Scope Chain

JavaScript is a *lexically scoped* language: the scope of a variable can be thought of as the set of source code lines for which the variable is defined. Global variables are defined throughout the program. Local variables are defined throughout the function in which they are declared, and also within any functions nested within that function.

If we think of local variables as properties of some kind of implementation-defined object, then there is another way to think about variable scope. Every chunk of JavaScript code (global code or functions) has a *scope chain* associated with it. This scope chain is a list or chain of objects that defines the variables that are "in scope" for that code. When JavaScript needs to look up the value of a variable `x` (a process called *variable resolution*), it starts by looking at the first object in the chain. If that object has a property named `x`, the value of that property is used. If the first object does not have a property named `x`, JavaScript continues the search with the next object in the chain. If the second object does not have a property named `x`, the search moves on to the next

object, and so on. If `x` is not a property of any of the objects in the scope chain, then `x` is not in scope for that code, and a ReferenceError occurs.

In top-level JavaScript code (i.e., code not contained within any function definitions), the scope chain consists of a single object, the global object. In a non-nested function, the scope chain consists of two objects. The first is the object that defines the function's parameters and local variables, and the second is the global object. In a nested function, the scope chain has three or more objects. It is important to understand how this chain of objects is created. When a function is defined, it stores the scope chain then in effect. When that function is invoked, it creates a new object to store its local variables, and adds that new object to the stored scope chain to create a new, longer, chain that represents the scope for that function invocation. This becomes more interesting for nested functions because each time the outer function is called, the inner function is defined again. Since the scope chain differs on each invocation of the outer function, the inner function will be subtly different each time it is defined—the code of the inner function will be identical on each invocation of the outer function, but the scope chain associated with that code will be different.

This notion of a scope chain is helpful for understanding the `with` statement (§5.7.1) and is crucial for understanding closures (§8.6).