Ryan Morr

Web developer based in Barrie, Ontario, Canada. I love to create dynamic web applications with a particular passion for JavaScript, Node.js, CSS, and HTML5.

Understanding Scope and Context in JavaScript

August 16, 2013 JavaScript

JavaScript's implementation of scope and context is a unique feature of the language, in part because it is so flexible. Functions can be adopted for various contexts and scope can be encapsulated and preserved. These concepts lend to some of the most powerful design patterns JavaScript has to offer. However, this is also a tremendous source of confusion amongst developers, and for good reason. The following is a comprehensive explanation of scope and context, the difference between them, and how various design patterns make use of them.

Context vs. Scope

The first important thing to clear up is that context and scope are not the same. I have noticed many developers over the years often confuse the two terms (myself included), incorrectly describing one for the other. To be fair, the terminology has become quite muddled over the years.

Every function invocation has both a scope and a context associated with it. Fundamentally, scope is function-based while context is object-based. In other words, scope pertains to the variable access of a function when it is invoked and is unique to each invocation. Context is always the value of the `this` keyword which is a reference to the object that "owns" the currently executing code.

Variable Scope

A variable can be defined in either local or global scope, which establishes the variables' accessibility from different scopes during runtime. Any defined global variable, meaning any variable declared outside of a function body will live throughout runtime and can be accessed and altered in any scope. Local variables exist only within the function body of which they are defined and will have a different scope for every call of that function. There it is subject for value assignment, retrieval, and manipulation only within that call and is not accessible outside of that scope.

ECMAScript 6 (ES6/ES2015) introduced the `let` and `const` keywords that support the declaration of block scope local variables. This means the variable will be confined to the scope of a block that it is defined in, such as an `if` statement or `for` loop and will not be accessible outside of the opening and closing curly braces of the block. This is contrary to `var` declarations which are accessible outside blocks they are defined in. The difference between `let` and `const` is that a `const` declaration is, as the name implies, constant - a read-only reference to a value. This does not mean the value is immutable, just that the variable identifier cannot be reassigned.

What is “this” Context

Context is most often determined by how a function is invoked. When a function is called as a method of an object, `this` is set to the object the method is called on:

```
var obj = {
    foo: function() {
        return this;
    }
};

obj.foo() === obj; // true
```

The same principle applies when invoking a function with the `new` operator to create an instance of an object. When invoked in this manner, the value of `this` within the scope of the function will be set to the newly created instance:

```
function foo() {
    alert(this);
}

foo() // window
new foo() // foo
```

When called as an unbound function, `this` will default to the global context or window object in the browser. However, if the function is executed in *strict mode*, the context will default to undefined.

Execution Context

JavaScript is a single threaded language, meaning only one task can be executed at a time. When the JavaScript interpreter initially executes code, it first enters into a global execution context by default. Each invocation of a function from this point on will result in the creation of a new execution context.

This is where confusion often sets in, the term “execution context” is actually for all intents and purposes referring more to scope and not context as previously discussed. It is an unfortunate naming convention, however it is the terminology as defined by the ECMAScript specification, so we’re kinda stuck with it.

Each time a new execution context is created it is appended to the top of the *execution stack*. The browser will always execute the current execution context that is atop the execution stack. Once completed, it will be removed from the top of the stack and control will return to the execution context below.

An execution context can be divided into a creation and execution phase. In the creation phase, the interpreter will first create a *variable object* (also called an *activation object*) that is composed of all the variables, function declarations, and arguments defined inside the execution context. From there the *scope chain* is initialized next, and the value of `this` is determined last. Then in the execution phase, code is interpreted and executed.

The Scope Chain

For each execution context there is a scope chain coupled with it. The scope chain contains the variable object for every execution context in the execution stack. It is used for determining variable access and identifier resolution. For example:

```
function first() {
    second();
    function second() {
        third();
        function third() {
```

```
            fourth();
            function fourth() {
                // do something
            }
        }
    }
}
first();
```

Running the preceding code will result in the nested functions being executed all the way down to the `fourth` function. At this point the scope chain would be, from top to bottom: fourth, third, second, first, global. The `fourth` function would have access to global variables and any variables defined within the `first`, `second`, and `third` functions as well as the functions themselves.

Name conflicts amongst variables between different execution contexts are resolved by climbing up the scope chain, moving locally to globally. This means that local variables with the same name as variables higher up the scope chain take precedence.

To put it simply, each time you attempt to access a variable within a function's execution context, the look-up process will always begin with its own variable object. If the identifier is not found in the variable object, the search continues into the scope chain. It will climb up the scope chain examining the variable object of every execution context looking for a match to the variable name.

Closures

Accessing variables outside of the immediate lexical scope creates a closure. In other words, a closure is formed when a nested function is defined inside of another function, allowing access to the outer functions variables. Returning the nested function allows you to maintain access to the local variables, arguments, and inner function declarations of its outer function. This encapsulation allows us to hide and preserve the execution context from outside scopes while exposing a public interface and thus is subject to further manipulation. A simple example of this looks like the following:

```
function foo() {
    var localVariable = 'private variable';
    return function() {
        return localVariable;
    }
}

var getLocalVariable = foo();
getLocalVariable() // "private variable"
```

One of the most popular types of closures is what is widely known as the *module pattern*; it allows you to emulate public, private, and privileged members:

```
var Module = (function() {
    var privateProperty = 'foo';

    function privateMethod(args) {
        // do something
    }

    return {

        publicProperty: '',

        publicMethod: function(args) {
            // do something
        },

        privilegedMethod: function(args) {
            return privateMethod(args);
        }
    };
})();
```

The module acts as if it were a singleton, executed as soon as the compiler interprets it, hence

the opening and closing parenthesis at the end of the function. The only available members outside of the execution context of the closure are your public methods and properties located in the return object (`Module.publicMethod` for example). However, all private properties and methods will live throughout the life of the application as the execution context is preserved, meaning variables are subject to further interaction via the public methods.

Another type of closure is what is called an immediately-invoked function expression (IIFE) which is nothing more than a self-invoked anonymous function executed in the context of the window:

```
(function(window) {

    var foo, bar;

    function private() {
        // do something
    }

    window.Module = {

        public: function() {
            // do something
        }
    };

})(this);
```

This expression is most useful when attempting to preserve the global namespace as any variables declared within the function body will be local to the closure but will still live throughout runtime. This is a popular means of encapsulating source code for applications and frameworks, typically exposing a single global interface in which to interact with.

Call and Apply

These two methods inherent to all functions allow you to execute any function in any desired context. This makes for incredibly powerful capabilities. The `call` function requires the arguments to be listed explicitly while the `apply` function allows you to provide the arguments as an array:

```
function user(firstName, lastName, age) {
    // do something
}

user.call(window, 'John', 'Doe', 30);
user.apply(window, ['John', 'Doe', 30]);
```

The result of both calls is exactly the same, the `user` function is invoked in the context of the window and provided the same three arguments.

ECMAScript 5 (ES5) introduced the `Function.prototype.bind` method that is used for manipulating context. It returns a new function which is permanently bound to the first argument of `bind` regardless of how the function is being used. It works by using a closure that is responsible for redirecting the call in the appropriate context. See the following polyfill for unsupported browsers:

```
if(!('bind' in Function.prototype)){
    Function.prototype.bind = function() {
        var fn = this;
        var context = arguments[0];
        var args = Array.prototype.slice.call(arguments, 1);
        return function() {
            return fn.apply(context, args.concat([].slice.call(arguments)));
        }
    }
}
```

It is commonly used where context is commonly lost: object-orientation and event handling.

This is necessary because the `addEventListener` method of a node will always execute the callback in the context of the node the event handler is bound to, which is the way it should be. However if your employing advanced object-oriented techniques and require your callback to be a method of an instance, you will be required to manually adjust the context, this is where `bind` comes in handy:

```
function Widget() {
    this.element = document.createElement('div');
    this.element.addEventListener('click', this.onClick.bind(this), false);
}

Widget.prototype.onClick = function(e) {
    // do something
};
```

While reviewing the source of the polyfill for `Function.prototype.bind` function, you may have noticed 2 invocations involving the `slice` method of an `Array`:

```
Array.prototype.slice.call(arguments, 1);
[].slice.call(arguments);
```

What is interesting to note here is that the `arguments` object is not actually an array at all, however it is often described as an array-like object much like a nodelist (anything returned by `element.childNodes`). They contain a length property and indexed values but they are still not arrays, and subsequently don't support any of the native methods of arrays such as `slice` and `push`. However, because of their similar behavior, the methods of `Array` can be adopted or hijacked, if you will, and executed in the context of an array-like object as is the case above.

This technique of adopting another object's methods also applies to object-orientation when emulating classical based inheritance in JavaScript:

```
SubClass.prototype.init = function(){
    // call the superclass init method in the context of the "SubClass" instance
    SuperClass.prototype.init.apply(this, arguments);
}
```

By invoking the method of the superclass (`SuperClass`) in the context of an instance of a subclass (`SubClass`), we can mimic the ability of calling a method's super to fully exploit this powerful design pattern.

Conclusion

It is important to understand these concepts before you begin to approach advanced design patterns, as scope and context play a fundamental role in modern JavaScript. Whether we're talking about closures, object-orientation and inheritance, or various native implementations, context and scope play a significant role in all of them. If your goal is to master the JavaScript language and better understand all it encompasses then scope and context should be one of your starting points.