Closures

## Definition

Per [1, p 25]:

Closures are critical to successful application development.

In a nutshell, a *closure* stores function state, even after the function has returned. To create a closure, simply define a function inside another function and expose it. TO expose a function, return it or pass it to another function. The inner function will have access to the variables declared in the outer function. This technique is commonly used to give objects data privacy.

Because the closure variable in the outer function are only in scope within the containing function, you can’t get at the data except through its *privileged methods*. In other languages, a privileged method is an exposed method that has access to private data. In JavaScript, any exposed method defined within the closure scope is privileged.

Per [2, p 90]:

Succinctly put, a *closure* is the scope created when a function is declared that allows the function to access and manipulate variables that are external to that function. Put another way, closures allow a function to access all the variables, as well as other functions, that are in scope wen the function itself is declared.

That may seem rather intuitive until you remember that a declared function can be called at any later time, *even* after the scope in which it was declared has gone away.

Per [5, p 160]:

The concept of a closure is actually very simple: a closure is the combination of a function and the environment (the set of available variables) in which it was defined.

That’s it. See, it wasn’t complicated, was it?

In JavaScript, a closure is created when a function is defined within another function. If the inner function is accessed from outside of its containing function, it still has access to the scope in which it was originally defined – even though the variables in that scope are not accessible to any other functions or variables.

Per [1, p 26]:

In addition to the data privacy benefits, closures are an essential ingredient in languages that support first-class functions, because they give you access to outer scope variables from inside your lambdas.

Closures are commonly used to feed data to event handlers or callbacks, which might be triggered long after the containing function has finished.

Per [2, p 89]:

Closely tied to the functions that we learned about in the previous chapters, closures are a defining feature of JavaScript. While scores of page authors get along writing on-page script without understanding the benefits of closures, the use of closures can not only help us reduce the amount and complexity of the script necessary to add advanced features to our pages, they allow us to do things that would simply not be possible, or would simply be too complex to be feasible, without them. The landscape of the language, and how we write our code using it, is forever shaped by the inclusion of closures.

Traditionally, closures have been a feature of purely functional programming languages. Having them cross over into mainstream development has been particularly encouraging, and it’s not uncommon to find closures permeating JavaScript libraries, along with other advanced code bases, due to their ability to drastically simplify complex operations.

# Properties

Per [2, p 93]:

This shows three more interesting concepts regarding closures:

* Function parameters are included in the closure of that function. (Seems obvious, but now we’ve said it for sure.)
* All variables in an outer scope, even those declared after function declaration, are included.
* Within the same scope, variables not yet defined cannot be forward-referenced.

Per [2, p 93 – 94]:

It’s important to note that while all of this structure isn’t readily visible anywhere (there’s no “closure” object holding all of tis information that you can inspect), there’s a direct cost to storing and referencing information in this manner. It’s important to remember that each function that accesses information via closure has a “ball and chain,” if you will, attached to it carrying this information around. So while closures are incredibly useful, they certainly aren’t free of overhead. All that information needs to be held in memory until it’s absolutely clear to the JavaScript engine that it will no longer be needed (and is safe to garbage-collect), or until the page unloads.

# Applications

## Memoization

## Data Hiding

## Callbacks and Timers

Per [2, p 96]:

Another one of the most common areas in which we can use closures is when dealing with callbacks or timers. In both cases, a function is being asynchronously called at an unspecified later time, and within such functions we frequently need to access outside data.

Closures act as an intuitive way of accessing that data, especially when we wish to avoid creating extra top-level variables just to store that information.

## Module Pattern

Per [3, p 80 – 81]:

The module pattern is an object-creation pattern designed to create singleton objects with private data. The basic approach is to use an immediately invoked function expression (IIFE) that returns an object. An IIFE is a function expression that is defined and then called immediately to produce a result. That function expression can contain any number of local variables that aren’t accessible from outside that function. Because the returned object is defined within that function, the object’s methods have access to the data. (All objects defined within the IIFE have access to the same local variables.) Methods that access private data in this way are called privileged methods. Here’s the basic format for the module pattern:

var yourObject = (function() {

// private data variables

return {

// public methods and properties

};

}());

In this pattern, an anonymous function is created and executed immediately. (Note the extra parentheses at the end of the function. You can execute anonymous functions immediately using this syntax.) That means the function exists for a moment, is executed, and then is destroyed. IIFEs are a very popular pattern in JavaScript, partially for their use in the module pattern.

The module pattern allows you to use regular variables as defacto object properties that aren’t exposed publicly. You accomplish this by creating *closure* functions as object methods. Closures are simply functions that access data outside their own scope. For example, whenever you access a global object in a function, such as window in a web browser, that function is accessing a variable outside its own scope. The difference with the module function is that the variables are declared within the IIFE, and a function that is also declared inside the IIFE accesses those variables.

Per [4, p 111 - 112]:

Functions are also referred to as closures, particularly when focusing on the fact that functions create new scopes. An IIFE is a function that you execute immediately. The term IIFE stands for Immediately-Invoked Function Expression. Using an IIFE is useful when all you want is a closure. The following code is an example IIFE:

(function () }

})();

Note the parentheses wrapping the function. These tell the interpreter you’re not only declaring an anonymous function, but also using it as a value. These expressions can also be used in assignments, which are useful if you need variables accessible by the exported return value. This is commonly referred to as the module pattern, as shown in the following code:

var api = (function () {

var local = 0; // private and in-place!

var publicInterface = {

counter: function () {

return ++local;

}

};

return publicInterface;

})();

api.counter();

// <- 1

A common variant to the previous code doesn’t rely on anything outside of the closure, but instead imports the variables it’s going to use. If it wants to expose a public API, then it imports the global object. I tend to favor this approach because everything is nicely wrapped by a closure, and you can instruct JSHint to blow up on issues due to undeclared variables. Without a closure and JSHint, these would inadvertently become globals. To illustrate, look at the following code:

(function (window) {

var privateThing;

var privateMethod () {

}

window.api = {

// public interface

}

})(window);

Pe [5, p 163 – 154]:

Speaking of YUI, another interesting use of closures (and a useful pattern in itself) is the module pattern that the YUI team developed as a means to arrange code into independent modules, and to manage method and property visibility. It is, in fact, a close cousin of the self-executing function, and works in exactly the same way. …

Let’s dig into some example code:

Menu = (function() {

// a private variable

var active = false;

// a private method

var reset = function() {

⁞

}

return {

// a public property accessible via Menu.instances

instances: [],

// a public method accessible via Menu.create()

create: function(element) {

⁞

}

}

})();

As you can see, this example is almost the same as the self-executing function pattern, although it returns an object with public properties and methods that, in this case, are assigned to Menu, our module. The interesting thing about this solution is that the public methods have access to all of the private functions and properties through the closure. That peculiarity aside, the private methods are truly private – there’s no way to access them from outside the module at all.

# References

[1] Programming JavaScript Applications, by Eric Elliot, published by O’Reilly Media, Inc., 2014

[2] Secrets of the JavaScript Ninja, by John Resig and Bear Bibeault, published by Manning Publications Co., 2013

[3] The Principles of Object-Oriented JavaScript, by Nicholas C. Zakas, published by No Starch Press, Inc., 2014

[4] JavaScript Application Design: A Build First Approach, by Nicolas Bevacqua, published by Manning Publications Co., 2015

[5] The Art & Science of JavaScipt, by Cameron Adams, James Edwards, Christian Heilmann, Michael Mahemoff, Ara Pehlivanian, Dan Webb, and Simon Willison, published by SitePoint Pty. Ltd., 2007