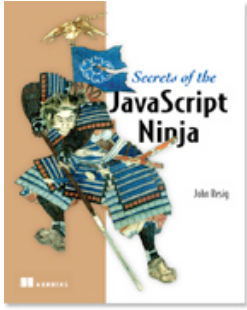


How closures work

| | |
|---|--|
|  | <p>RELATED READING</p> <p>Secrets of the JavaScript Ninja John Resig</p> <p>MEAP Release: March 2008 Paper book: January 2009 (est.) 300 pages ISBN: 193398869X</p> |
|---|--|

*This article is excerpted from the upcoming title *The Secrets of JavaScript Ninjas* by John Resig and published by Manning Publications. This article introduces the closure, an important aspect of JavaScript, and describes its use. For the table of contents, source code, the Author Forum, and other resources, go to www.manning.com/resig.*

Closures are one of the defining features of JavaScript, differentiating it from most other scripting languages. Simply: A closure is a way to access and manipulate external variables from within a function. Another way of imagining it is the fact that a function is able to access all the variables, and functions, declared in the same scope as itself. The result is rather intuitive and is best explained through code, like in listing 1.

Listing 1: A few examples of closures

```
var stuff = true;

function a(arg1){
  var b = true;
  assert( a && stuff, "These come from the closure." );

  function c(arg2){
    assert( a && stuff && b && c && arg1, "All from a closure, as well." );
  }

  c(true);
}
a(true);

assert( stuff && a, "Globally-accessible variables and functions." );
```

Note that the above example contains two closures: function `a()` includes a closed reference to itself and the variable `stuff`. function `c()` includes a closed reference to the variables `stuff`, `b`, and `arg1` and references to the functions `b()` and `c()`.

It's important to note that while all of this reference information isn't immediately visible (there's no "closure" object holding all of this information, for example) there is a direct cost to storing and referencing your information in this manner. It's important to remember that each function that accesses information via a closure immediately has at "ball and chain," if you will, attached to them carrying this information. So while closures are incredibly useful, they certainly aren't free of overhead.

Private Variables

One of the most common uses of closures is to encapsulate some information as a "private variable," of sorts. Object-oriented code, written in JavaScript, is unable to have traditional private variables (properties of the object that are hidden from outside uses). However, using the concept of a closure, we can arrive at an acceptable result, as shown in listing 2.

Listing 2: An example of keeping a variable private but accessible via a closure

```
function Ninja(){
```

```

var slices = 0;

this.getSlices = function(){
    return slices;
};

this.slice = function(){
    slices++;
};
}

var ninja = new Ninja();
ninja.slice();
assert( ninja.getSlices() == 1, "We're able to access the internal slice data." );
assert( ninja.slices === undefined, "And the private data is inaccessible to us." );

```

In this example we create a variable to hold our state, `slices`. This variable is only accessible from within the `Ninja()` function (including the methods `getSlices()` and `slice()`). This means that we can maintain the state, within the function, without letting it be directly accessed by the user.

Callbacks and Timers

Another one of the most beneficial places for using closures is when you're dealing with callbacks or timers. In both cases a function is being called at a later time and within the function you have to deal with some, specific, outside data. Closures act as an intuitive way of accessing data, especially when you wish to avoid creating extra variables just to store that information. Let's look at a simple example of an Ajax request, using the jQuery JavaScript Library (Listing 3).

Listing 3: Using a closure from a callback in an Ajax request

```

var elem = jQuery("div").html("Loading...");

jQuery.ajax({
    url: "test.html",
    success: function(html){
        assert( elem, "The element to append to, via a closure." );
        elem.html( html );
    }
});

```

There are a couple things occurring in this example. To start, we're placing a loading message into the div to indicate that we're about to start an Ajax request. We have to do the query, for the div, once to edit its contents - and would typically have to do it again to inject the contents when the Ajax request completed. However, we can make good use of a closure to save a reference to the original jQuery object (containing a reference to the div element), saving us from that effort.

Listing 4 is a slightly more complicated example, creating a simple animation.

Listing 4: Using a closure from a timer interval

```

var elem = document.getElementById("box");
var count = 0;

var timer = setInterval(function(){
    if ( count <= 100 ) {
        elem.style.left = count + "px";
        count++;
    } else {
        assert( count == 100, "Count came via a closure, accessed each step." );
        assert( timer, "The timer reference is also via a closure." );
        clearInterval( timer );
    }
}, 10);

```

What's especially interesting about the above example is that it only uses a single (anonymous) function to accomplish the animation, and three variables, accessed via a closure. This structure is particularly important as the three variables (the DOM element, the pixel counter, and the timer reference) all must be maintained across steps of the animation. This example is a particularly good one in demonstrating how the concept of closures is capable of producing some surprisingly intuitive, and concise, code.