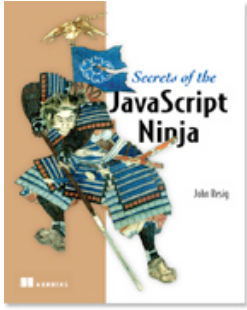


Using (function(){})()

	<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 construct `(function(){})()` and describes its use in relationship to closures. For the table of contents, source code, the Author Forum, and other resources, go to www.manning.com/resig.*

So much of advanced JavaScript, and the use of closures, centers around one simple construct: `(function(){})()`. This single piece of code is incredibly versatile and ends up giving the JavaScript language a ton of unforeseen power. Since the syntax is a little strange, let's deconstruct what's going on.

First, let's examine the use of `(...)()`. We know that we can call a function using the `functionName()` syntax. In this case the extra set of parentheses sort of hides us from whatever is inside of it. In the end we just assume that whatever is in there will be a reference to a function and executed. For example, the following is also valid: `(functionName)()`. Thus, instead of providing a reference to an existing function, we provide an anonymous function, creating: `(function(){})()`.

The result of this code is a code block which is instantly created, executed, and discarded. Additionally, since we're dealing with a function that can have a closure, we also have access to all outside variables. As it turns out, this simple construct ends up becoming immensely useful, as we'll see in the following sections.

Temporary Scope and Private Variables

Using the executed anonymous function we can start to build up interesting enclosures for our work. Since the function is executed immediately, and all the variables inside of it are kept inside of it, we can use this to create a temporary scope within which our state can be maintained, like in listing 1.

Listing 1: Creating a temporary enclosure for persisting a variable

```
(function(){
  var numClicks = 0;

  document.addEventListener("click", function(){
    alert( ++numClicks );
  }, false);
}) ();
```

Since the above anonymous function is executed immediately the click handler is also bound right away. Additionally, a closure is created allowing the `numClicks` variable to persist with the handler. This is the most common way in which executed anonymous functions are used, just as a simple wrapper. However it's important to remember that since they are functions they can be used in interesting ways, like in listing 2.

Listing 2: An alternative to the example in Listing 1, returning a value from the enclosure

```
document.addEventListener("click", (function(){
  var numClicks = 0;
```

```

return function(){
    alert( ++numClicks );
};
})( ), false);

```

This is a, debatably, more complicated version of our first example. In this case we're, again, creating an executed anonymous function but this time we return a value from it. Since this is just like any other function that value is returned and passed along to the `addEventListener` method. However, this function that we've created also gets the necessary `numClicks` variable via its closure.

This technique is a very different way of looking at scope. In most languages you can scope things based upon the block which they're in. In JavaScript variables are scope based upon the function they're in. However, with this simple construct, we can now scope variables to block, and sub-block, levels. The ability to scope some code to a unit as small as an argument within a function call is incredibly powerful and truly shows the flexibility of the language.

Listing 3 is a quick example from the Prototype JavaScript library.

Listing 3: Using the anonymous function wrapper as a variable shortcut

```

(function(v) {
    Object.extend(v, {
        href: v._getAttr,
        src: v._getAttr,
        type: v._getAttr,
        action: v._getAttrNode,
        disabled: v._flag,
        checked: v._flag,
        readonly: v._flag,
        multiple: v._flag,
        onload: v._getEv,
        onunload: v._getEv,
        onclick: v._getEv
        // ...
    });
})(Element._attributeTranslations.read.values);

```

In this case they're extending an object with a number of new properties and methods. In that code they could've created a temporary variable for `Element._attributeTranslations.read.values` but, instead, they chose to pass it in as the first argument to the executed anonymous function. This means that the first argument is now a reference to this data structure and is contained within this scope. This ability to create temporary variables within a scope is especially useful once we start to examine looping.

Loops

One of the most useful applications of executed anonymous functions is the ability to solve a nasty issue with loops and closures. Let's take a quick look at a common piece of problematic code (listing 4)

Listing 4: A problematic piece of code in which the iterator is not maintained in the closure

```

var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
    div[i].addEventListener("click", function(){
        alert( "div #" + i + " was clicked." );
    }, false);
}

```

In this example we encounter a common issue with closures and looping, namely that the variable that's being enclosed (`i`) is being updated after the function is bound. This means that every bound function handler will always alert the last value stored in `i` (in this case, '2'). This is due to the fact that closures only remember references to variables - not their actual values at the time at which they were called. This is an important distinction and one that trips up a lot of people.

Not to fear, though, as we can combat this closure craziness with another closure, as shown in listing 5.

Listing 5: Using an anonymous function wrapper to persist the iterator properly

```

var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) (function(i){
    div[i].addEventListener("click", function(){

```

```
    alert( "div #" + i + " was clicked." );
  }, false);
})(i);
```

Using this executed anonymous function as a wrapper for the for loop (replacing the existing `{...}` braces) we can now enforce that the correct value will be enclosed by these handlers. Note that, in order to achieve this, we pass in the iterator value to the anonymous function and then re-declare it in the arguments. This means that within the scope of each step of the for loop the `i` variable is defined anew, giving our click handler closure the value that it expects.

Library Wrapping

The final concept that closures, and executed anonymous functions, encapsulate is an important one to JavaScript library development. When developing a library it's incredibly important that you don't pollute the global namespace with unnecessary variables, especially ones that are only temporarily used. This is a point at which the executed anonymous functions can be especially useful: Keep as much of the library private as possible and only selectively introduce variables into the global namespace. The jQuery library actually does a good job of this, completely enclosing all of its functionality and only introducing the variables it needs, like jQuery (see listing 6).

Listing 6: Placing a variable outside of a function wrapper

```
(function(){
  var jQuery = window.jQuery = function(){
    // Initialize
  };
  // ...
})();
```

Note that there are two assignments done here, this is intentional. First, the jQuery constructor is assigned to `window.jQuery` in an attempt to introduce it as a global variable. However, that does not guarantee that that variable will be the only named jQuery within our scope, thus we assign it to a local variable, `jQuery`, to enforce it as such. That means that we can use the jQuery function name continuously within our library while, externally, someone could've re-named the global jQuery object to something else. Since all of the functions and variables that are required by the library are nicely encapsulated it ends up giving the end user a lot of flexibility in how they wish to use it.

However, that isn't the only way in which that type of definition could be done; here's another, in listing 7.

Listing 7: An alternative means of putting a variable in the outer scope

```
var jQuery = (function(){
  function jQuery(){
    // Initialize
  }
  // ...
  return jQuery;
})();
```

This would have the same effect as what was shown in the first example, but structured in a different manner. Here we define a `jQuery` function within our anonymous scope, use it and define it, then return it back out the global scope where it is assigned to a variable, also named `jQuery`. Oftentimes this particular technique is preferred, if you're only exporting a single variable, as it's clearer as to what the result of the function is.

In the end a lot of this is left to developer preference, which is good, considering that the JavaScript language gives you all the power you'll need to make any particular application structure happen.