Javascript Closures

- Introduction
- The Resolution of Property Names on Objects
  - Assignment of Values
  - Reading of Values
- Identifier Resolution, Execution Contexts and Scope Chains
  - The Execution Context
  - Scope chains and [[scope]]
  - Identifier Resolution
- Closures
  - Automatic Garbage Collection
  - Forming Closures
- What can be done with Closures?
  - Example 1: setTimeout with Function References
  - Example 2: Associating Functions with Object Instance Methods
  - Example 3: Encapsulating Related Functionality
  - Other Examples
- Accidental Closures
- The Internet Explorer Memory Leak Problem

Introduction

Closure

A "closure" is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Closures are one of the most powerful features of ECMAScript (javascript) but they cannot be property exploited without understanding them. They are, however, relatively easy to create, even accidentally, and their creation has potentially harmful consequences, particularly in some relatively common web browser environments. To avoid accidentally encountering the drawbacks and to take advantage of the benefits they offer it is necessary to understand their mechanism. This depends heavily on the role of scope chains in identifier resolution and so on the resolution of property names on objects.

The simple explanation of a Closure is that ECMAScript allows inner functions; function definitions and function expressions that are inside the function bodes of other functions. And that those inner functions are allowed access to all of the local variables, parameters and declared inner functions within their outer function(s). A closure is formed when one of those inner functions is made accessible outside of the function in which it was contained, so that it may be executed after the outer function has returned. At which point it still has access to the local variables, parameters and inner function declarations of its outer function. Those local variables, parameter and function declarations (initially) have the values that they had when the outer function returned and may be interacted with by the inner function.

Unfortunately, properly understanding closures requires an understanding of the mechanism behind them, and quite a bit of technical detail. While some of the ECMA 262 specified algorithms have been brushed over in the early part of the following explanation, much cannot be omitted or easily simplified. Individuals familiar with object property name resolution may skip that section but only people already familiar with closures can afford to skip the following sections, and they can stop reading now and get back to exploiting them.

The Resolution of Property Names on Objects

ECMAScript recognises two categories of object, "Native Object" and "Host Object" with a sub-category of native objects called "Built-in Object" (ECMA 262 3rd Ed Section 4.3). Native objects belong to the language and host objects are provided by the environment, and may be, for example, document objects, DOM nodes and the like.

Native objects are loose and dynamic bags of named properties (some implementations are not that dynamic when it comes to the built in object sub-category, though usually that doesn't matter). The defined named properties of an object will hold a value, which may be a reference to another Object (functions are also Objects in this sense) or a primitive value: String, Number, Boolean, Null or Undefined. The Undefined primitive type is a bit odd in that it is possible to assign a value of Undefined to a property of an object but doing so does not remove that property from the object; it remains a defined named property, it just holds the value `undefined`.

The following is a simplified description of how property values are read and set on objects with the internal details brushed over to the greatest extent possible.

Assignment of Values

Named properties of objects can be created, or values set on existing named properties, by assigning a value to that named property. So given:-

```
var objectRef = new Object(); //create a generic javascript object.
```

A property with the name "testNumber" can be created as:-

```
objectRef.testNumber = 5;
/* - or:- */
```

```
objectRef["testNumber"] = 5;
```

The object had no "testNumber" property prior to the assignment but one is created when the assignment is made. Any subsequent assignment does not need to create the property, it just re-sets its value:-

```
objectRef.testNumber = 8;
/* - or:- */
objectRef["testNumber"] = 8;
```

Javascript objects have prototypes that can themselves be objects, as will be described shortly, and that prototype may have named properties. But this has no role in assignment. If a value is assigned and the actual object does not have a property with the corresponding name a property of that name is created and the value is assigned to it. If it has the property then its value is re-set.

Reading of Values

It is in reading values from object properties that prototypes come into play. If an object has a property with the property name used in the property accessor then the value of that property is returned:-

```
/* Assign a value to a named property. If the object does not have a
   property with the corresponding name prior to the assignment it
   will have one after it:-
*/
objectRef.testNumber = 8;

/* Read the value back from the property:- */

var val = objectRef.testNumber;
/* and  - val - now holds the value 8 that was just assigned to the
   named property of the object. */
```

But all objects may have prototypes, and prototypes are objects so they, in turn, may have prototypes, which may have prototypes, and so on forming what is called the prototype chain. The prototype chain ends when one of the objects in the chain has a null prototype. The default prototype for the `Object` constructor has a null prototype so:-

```
var objectRef = new Object(); //create a generic javascript object.
```

Creates an object with the prototype `Object.prototype` that itself has a null prototype. So the prototype chain for `objectRef` contains only one object: `Object.prototype`. However:-

```
/* A "constructor" function for creating objects of a -
   MyObject1 - type.
*/
function MyObject1(formalParameter){
    /* Give the constructed object a property called - testNumber - and
```

```
        assign it the value passed to the constructor as its first
        argument:-
     */
     this.testNumber = formalParameter;
}

/* A "constructor" function for creating objects of a -
   MyObject2 - type:-
*/
function MyObject2(formalParameter){
   /* Give the constructed object a property called - testString -
      and assign it the value passed to the constructor as its first
      argument:-
     */
     this.testString = formalParameter;
}

/* The next operation replaces the default prototype associated with
   all MyObject2 instances with an instance of MyObject1, passing the
   argument - 8 - to the MyObject1 constructor so that its -
   testNumber - property will be set to that value:-
*/
MyObject2.prototype = new MyObject1( 8 );

/* Finally, create an instance of - MyObject2 - and assign a reference
   to that object to the variable - objectRef - passing a string as the
   first argument for the constructor:-
*/

var objectRef = new MyObject2( "String_Value" );
```

The instance of `MyObject2` referred to by the `objectRef` variable has a prototype chain. The first object in that chain is the instance of `MyObject1` that was created and assigned to the prototype property of the `MyObject2` constructor. The instance of `MyObject1` has a prototype, the default Object prototype that corresponds with the object referred to by `Object.prototype`. `Object.prototype` has a null prototype so the prototype chain comes to an end at this point.

When a property accessor attempts to read a named property form the object referred to by the variable `objectRef` the whole prototype chain can enter into the process. In the simple case:-

```
var val = objectRef.testString;
```

- the instance of `MyObject2` referred to by `objectRef` has a property with the name "testString" so it is the value of that property, set to "String_Value", that is assigned to the variable `val`. However:-

```
var val = objectRef.testNumber;
```

- cannot read a named property form the instance of `MyObject2` itself as it has no such property but the variable `val` is set to the value of `8` rather than undefined because having failed to find a corresponding named property on the object itself the interpreter then examines the object that is its prototype. Its prototype is the instance of `MyObject1` and it was created with a property named "testNumber" with the value `8` assigned to that property, so the property accessor evaluates as the value `8`. Neither `MyObject1` or `MyObject2` have defined a `toString` method, but if a property accessor attempts to read the value of a `toString` property from `objectRef`:-

```
var val = objectRef.toString;
```

- the `val` variable is assigned a reference to a function. That function is the `toString` property of `Object.prototype` and is returned because the process of examining the prototype of `objectRef`, when `objectRef` turns out not to have a "toString" property, is acting on an object, so when that prototype is found to lack the property its prototype is examined in turn. Its prototype is `Object.prototype`, which does have a `toString` method so it is a reference to that function object that is returned.

Finally:-

```
var val = objectRef.madeUpProperty;
```

- returns `undefined`, because as the process of working up the prototype chain finds no properties on any of the object with the name "madeUpPeoperty" it eventually gets to the prototype of `Object.prototype`, which is null, and the process ends returning `undefined`.

The reading of named properties returns the first value found, on the object or then from its prototype chain. The assigning of a value to a named property on an object will create a property on the object itself if no corresponding property already exists.

This means that if a value was assigned as `objectRef.testNumber = 3` a "testNumber" property will be created on the instance of `MyObject2` itself, and any subsequent attempts to read that value will retrieve that value as set on the object. The prototype chain no longer needs to be examined to resolve the property accessor, but the instance of `MyObject1` with the value of `8` assigned to its "testNumber" property is unaltered. The assignment to the `objectRef` object masks the corresponding property in its prototype chain.

Note: ECMAScript defines an internal `[[prototype]]` property of the internal Object type. This property is not directly accessible with scripts, but it is the chain of objects referred to with the internal `[[prototype]]` property that is used in property accessor resolution; the object's prototype chain. A public `prototype` property exists to allow the assignment, definition and manipulation of prototypes in association with the internal `[[prototype]]` property. The details of the relationship between to two are described in ECMA 262 (3rd edition) and are beyond the scope of this discussion.

Identifier Resolution, Execution Contexts and Scope Chains

The Execution Context

An execution context is an abstract concept used by the ECMSScript specification (ECMA 262 3rd edition) to define the behaviour required of ECMAScript implementations. The specification does not say anything about how execution contexts should be implemented but execution contexts have associated attributes that refer to specification defined structures so they might be conceived (and even implemented) as objects with properties, though not public properties.

All javascript code is executed in an execution context. Global code (code executed inline, normally as a JS file, or HTML page, loads) gets executed in what I will be calling a global execution context, and each invocation of a function (possibly as a constructor) has an associated execution context. Code executed with the `eval` function also gets a distinct execution context but as `eval` is never normally used by javascript programmers it will not be discussed here. The specified details of execution contexts are to be found in section 10.2 of ECMA 262 (3rd edition).

When a javascript function is called it enters an execution context, if another function is called (or the same function recursively) a new execution context is created and execution enters that context for the duration of the function call. Returning to the original execution context when that called function returns. Thus running javascript code forms a stack of execution contexts.

When an execution context is created a number of things happen in a defined order. First, in the execution context of a function, an "Activation" object is created. The activation object is another specification mechanism. It can be considered as an object because it ends up having accessible named properties, but it is not a normal object as it has no prototype (at least not a defined prototype) and it cannot be directly referenced by javascript code.

The next step in the creation of the execution context for a function call is the creation of an `arguments` object, which is an array-like object with integer indexed members corresponding with the arguments passed to the function call, in order. It also has `length` and `callee` properties (which are not relevant to this discussion, see the spec for details). A property of the Activation object is created with the name "arguments" and a reference to the `arguments` object is assigned to that property.

Next the execution context is assigned a scope. A scope consists of a list (or chain) of objects. Each function object has an internal `[[scope]]` property (which we will go into more detail about shortly) that also consists of a list (or chain) of objects. The scope that is assigned to the execution context of a function call consists of the list referred to by the `[[scope]]` property of the corresponding function object with the Activation object added at the front of the chain (or the top of the list).

Then the process of "variable instantiation" takes place using an object that ECMA 262 refers to as the "Variable" object. However, the Activation object is used as the Variable object (note this, it is important: they are the same object). Named properties of the Variable object are created for each of the function's formal parameters, and if arguments to the function call correspond with those parameters the values of those arguments are assigned to the properties (otherwise the assigned value is `undefined`). Inner function definitions are used to create function objects which are assigned to properties of the Variable object with names that correspond to the function name used in the function declaration. The last stage of variable instantiation is to create named properties of the Variable object that correspond with all the local variables declared within the function.

The properties created on the Variable object that correspond with declared local variables are initially assigned `undefined` values during variable instantiation, the actual initialisation of local variables does not happen until the evaluation of the corresponding assignment expressions during the execution of the function body code.

It is the fact that the Activation object, with its `arguments` property, and the Variable object, with named properties corresponding with function local variables, are the same object, that allows the identifier `arguments` to be treated as if it was a function local variable.

Finally a value is assigned for use with the `this` keyword. If the value assigned refers to an object then property accessors prefixed with the `this` keyword reference properties of that object. If the value assigned (internally) is null then the `this` keyword will refer to the global object.

The global execution context gets some slightly different handling as it does not have arguments so it does not need a defined Activation object to refer to them. The global execution context does need a scope and its scope chain consists of exactly one object, the global object. The global execution context does go through variable instantiation, its inner functions are the normal top level function declarations that make up the bulk of javascript code. The global object is used as the Variable object, which is why globally declared functions become properties of the global object. As do globally declared variables.

The global execution context also uses a reference to the global object for the `this` object.

Scope chains and [[scope]]

The scope chain of the execution context for a function call is constructed by adding the execution context's Activation/Variable object to the front of the scope chain held in the function object's `[[scope]]` property, so it is important to understand how the internal `[[scope]]` property is defined.

In ECMAScript functions are objects, they are created during variable instantiation from function declarations, during the evaluation of function expressions or by invoking the `Function` constructor.

Function objects created with the `Function` constructor always have a `[[scope]]` property referring to a scope chain that only contains the global object.

Function objects created with function declarations or function expressions have the scope chain of the execution context in which they are created assigned to their internal `[[scope]]` property.

In the simplest case of a global function declaration such as:-

```
function exampleFunction(formalParameter){
    ...   // function body code
}
```

- the corresponding function object is created during the variable instantiation for the global execution context. The global execution context has a scope chain consisting of only the global object. Thus the function object that is created and referred to by the property of the global object with the name "exampleFunction" is assigned an internal `[[scope]]` property referring to a scope chain containing only the global object.

A similar scope chain is assigned when a function expression is executed in the global context:-

```
var exampleFuncRef = function(){
    ...   // function body code
}
```

- except in this case a named property of the global object is created during variable instantiation for the global execution context but the function object is not created, and a reference to it assigned to the named property of the global object, until the assignment expression is evaluated. But the creation of the function object still happens in the global execution context so the `[[scope]]` property of the created function object still only contains the global object in the assigned scope chain.

Inner function declarations and expressions result in function objects being created within the execution context of a function so they get more elaborate scope chains. Consider the following code, which defines a function with an inner function declaration and then executes the outer function:-

```
function exampleOuterFunction(formalParameter){
    function exampleInnerFuncitonDec(){
        ... // inner function body
    }
    ...  // the rest of the outer function body.
}

exampleOuterFunction( 5 );
```

The function object corresponding with the outer function declaration is created during variable instantiation in the global execution context so its `[[scope]]` property contains the one item scope chain with only the global object in it.

When the global code executes the call to the `exampleOuterFunction` a new execution context is created for that function call and an Activation/Variable object along with it. The scope of that new execution context becomes the chain consisting of the new Activation object followed by the chain refereed to by the outer function object's `[[scope]]` property (just the global object). Variable instantiation for that new execution context results in the creation of a function object that corresponds with the inner function definition and the `[[scope]]` property of that function object is assigned the value of the scope from the execution context in which it was created. A scope chain that contains the Activation object followed by the global object.

So far this is all automatic and controlled by the structure and execution of the source code. The scope chain of the execution context defines the `[[scope]]` properties of the function objects created and the `[[scope]]` properties of the function objects define the scope for their execution contexts (along with the corresponding Activation object). But ECMAScript provides the `with` statement as a means of modifying the scope chain.

The `with` statement evaluates an expression and if that expression is an object it is added to the scope chain of the current execution context (in front of the Activation/Variable object). The `with` statement then executes another statement (that may itself be a block statement) and then restores the execution context's scope chain to what it was before.

A function declaration could not be affected by a `with` statement as they result in the creation of function objects during variable instantiation, but a function expression can be evaluated inside a `with` statement:-

```
/* create a global variable - y - that refers to an object:- */
var y = {x:5}; // object literal with an - x - property
function exampleFuncWith(){
    var z;
    /* Add the object referred to by the global variable - y - to the
       front of he scope chain:-
    */
    with(y){
        /* evaluate a function expression to create a function object
           and assign a reference to that function object to the local
           variable - z - :-
        */
        z = function(){
            ... // inner function expression body;
        }
    }
    ...
}

/* execute the - exampleFuncWith - function:- */
exampleFuncWith();
```

When the `exampleFuncWith` function is called the resulting execution context has a scope chain consisting of its Activation object followed by the global object. The execution of the `with` statement adds the object referred to by the global variable `y` to the front of that scope chain during the evaluation of the function expression. The function object created by the evaluation of the function expression is assigned a `[[scope]]` property that corresponds with the scope of the execution context in which it is created. A scope chain consisting of object `y` followed by the Activation object from the execution context of the outer function call, followed by the global object.

When the block statement associated with the `with` statement terminates the scope of the execution context is restored (the `y` object is removed), but the function object has been created at that point and its `[[scope]]` property assigned a reference to a scope chain with the `y` object at its head.

Identifier Resolution

Identifiers are resolved against the scope chain. ECMA 262 categorises `this` as a keyword rather than an identifier, which is not unreasonable as it is always resolved dependent on the `this` value in the execution context in which it is used, without reference to the scope chain.

Identifier resolution starts with the first object in the scope chain. It is checked to see if it has a property with a name that corresponds with the identifier. Because the scope chain is a chain of objects this checking encompasses the prototype chain of that object (if it has one). If no corresponding value can be found on the first object in the scope chain the search progresses to the next object. And so on until one of the objects in the chain (or one of its prototypes) has a property with a name that corresponds with the identifier or the scope chain is exhausted.

The operation on the identifier happens in the same way as the use of property accessors on objects described above. The object identified in the scope chain as having the corresponding property takes the place of the object in the property accessor and the identifier acts as a property name for that object. The global object is always at the end of the scope chain.

As execution contexts associated with function calls will have the Activation/Variable object at the front of the chain, identifiers used in function bodies are effectively first checked to see whether they correspond with formal parameters, inner function declaration names or local variables. Those would be resolved as named properties of the Activation/Variable object.

Closures

Automatic Garbage Collection

ECMAScript uses automatic garbage collection. The specification does not define the details, leaving that to the implementers to sort out, and some implementations are known to give a very low priority to their garbage collection operations. But the general idea is that if an object becomes un-referable (by having no remaining references to it left accessible to executing code) it becomes available for garbage collection and will at some future point be destroyed and any resources it is consuming freed and returned to the system for re-use.

This would normally be the case upon exiting an execution context. The scope chain structure, the Activation/Variable object and any objects created within the execution context, including function objects, would no longer be accessible and so would become available for garbage collection.

Forming Closures

A closure is formed by returning a function object that was created within an execution context of a function call from that function call and assigning a reference to that inner function to a property of another object. Or by directly assigning a reference to such a function object to, for example, a global variable, a property of a globally accessible object or an object passed by reference as an argument to the outer function call. e.g:-

```
function exampleClosureForm(arg1, arg2){
    var localVar = 8;
    function exampleReturned(innerArg){
        return ((arg1 + arg2)/(innerArg + localVar));
    }
    /* return a reference to the inner function defined as -
       exampleReturned -:-
    */
    return exampleReturned;
}

var globalVar = exampleClosureForm(2, 4);
```

Now the function object created within the execution context of the call to `exampleClosureForm` cannot be garbage collected because it is referred to by a global variable and is still accessible, it can even be executed with `globalVar(n)`.

But something a little more complicated has happened because the function object now referred to by `globalVar` was created with a `[[scope]]` property referring to a scope chain containing the Activation/Variable object belonging to the execution context in which it was created (and the global object). Now the Activation/Variable object cannot be garbage collected either as the execution of the function object referred to by `globalVar` will need to add the whole scope chain from its `[[scope]]` property to the scope of the execution context created for each call to it.

A closure is formed. The inner function object has the free variables and the Activation/Variable object on the function's scope chain is the environment that binds them.

The Activation/Variable object is trapped by being referred to in the scope chain assigned to the internal `[[scope]]` property of the function object now referred to by the `globalVar` variable. The Activation/Variable object is preserved along with its state; the values of its properties. Scope resolution in the execution context of calls to the inner function will resolve identifiers that correspond with named properties of that Activation/Variable object as properties of that object. The value of those properties can still be read and set even though the execution context for which it was created has exited.

In the example above that Activation/Variable object has a state that represents the values of formal parameters, inner function definitions and local variables, at the time when the outer function returned (exited its execution context). The `arg1` property has the value `2`,the `arg2` property the value `4`, `localVar` the value `8` and an `exampleReturned` property that is a reference to the inner function object that was returned form the outer function. (We will be referring to this Activation/Variable object as "ActOuter1" in later discussion, for convenience.)

If the `exampleClosureForm` function was called again as:-

```
var secondGlobalVar = exampleClosureForm(12, 3);
```

- a new execution context would be created, along with a new Activation object. And a new function object would be returned, with its own distinct `[[scope]]` property referring to a scope chain containing the Activation object form this second execution context, with `arg1` being `12` and `arg2` being `3`. (We will be referring to this Activation/Variable object as "ActOuter2" in later discussion, for convenience.)

A second and distinct closure has been formed by the second execution of `exampleClosureForm`.

The two function objects created by the execution of `exampleClosureForm` to which references have been assigned to the global variable `globalVar` and `secondGlobalVar` respectively, return the expression `((arg1 + arg2)/(innerArg + localVar))`. Which applies various operators to four identifiers. How these identifiers are resolved is critical to the use and value of closures.

Consider the execution of the function object referred to by `globalVar`, as `globalVar(2)`. A new execution context is created and an Activation object (we will call it "ActInner1"), which is added to the head of the scope chain referred to the `[[scope]]` property of the executed function object. ActInner1 is given a property named `innerArg`, after its formal parameter and the argument value `2` assigned to it. The scope chain for this new execution context is: `ActInner1->` `ActOuter1->` `global object`.

Identifier resolution is done against the scope chain so in order to return the value of the expression `((arg1 + arg2)/(innerArg + localVar))` the values of the identifiers will be determined by looking for properties, with names corresponding with the identifiers, on each object in the scope chain in turn.

The first object in the chain is ActInner1 and it has a property named `innerArg` with the value `2`. All of the other 3 identifiers correspond with named properties of ActOuter1; `arg1` is `2`, `arg2` is `4` and `localVar` is `8`. The function call returns `((2 + 4)/(2 + 8))`.

Compare that with the execution of the otherwise identical function object referred to by `secondGlobalVar`, as `secondGlobalVar(5)`. Calling the Activation object for this new execution context "ActInner2", the scope chain becomes: `ActInner2->` `ActOuter2->` `global object`. ActInner2 returns `innerArg` as `5` and ActOuter2 returns `arg1`, `arg2` and `localVar` as `12`, `3` and `8` respectively. The value returned is `((12 + 3)/(5 + 8))`.

Execute `secondGlobalVar` again and a new Activation object will appear at the front of the scope chain but ActOuter2 will still be next object in the chain and the value of its named properties will again be used in the resolution of the identifiers `arg1`, `arg2` and `localVar`.

This is how ECMAScript inner functions gain, and maintain, access to the formal parameters, declared inner functions and local variables of the execution context in which they were created. And it is how the forming of a closure allows such a function object to keep referring to those values, reading and writing to them, for as long as it continues to exist. The Activation/Variable object from the execution context in which the inner function was created remains on the scope chain referred to by the function object's `[[scope]]` property, until all references to the inner function are freed and the function object is made available for garbage collection (along with any now unneeded objects on its scope chain).

Inner function may themselves have inner functions, and the inner functions returned from the execution of functions to form closures may themselves return inner functions and form closures of their own. With each nesting the scope chain gains extra Activation objects originating with the execution contexts in which the inner function objects were created. The ECMAScript specification requires a scope chain to be finite, but imposes no limits on their length. Implementations probably do impose some practical limitation but no specific magnitude has yet been reported. The potential for nesting inner functions seems so far to have exceeded anyone's desire to code them.

What can be done with Closures?

Strangely the answer to that appears to be anything and everything. I am told that closures enable ECMAScript to emulate anything, so the limitation is the ability to conceive and implement the emulation. That is a bit esoteric and it is probably better to start with something a little more practical.

Example 1: setTimeout with Function References

A common use for a closure is to provide parameters for the execution of a function prior to the execution of that function. For example, when a function is to be provided as the first argument to the `setTimout` function that is common in web browser environments.

`setTimeout` schedules the execution of a function (or a string of javascript source code, but not in this context), provided as its first argument, after an interval expressed in milliseconds (as its second argument). If a piece of code wants to use `setTimeout` it calls the `setTimeout` function and passes a reference to a function object as the first argument and the millisecond interval as the second, but a reference to a function object cannot provide parameters for the scheduled execution of that function.

However, code could call another function that returned a reference to an inner function object, with that inner function object being passed by reference to the `setTimeout` function. The parameters to be used for the execution of the inner function are passed with the call to the function that returns it. `setTimout` executes the inner function without passing arguments but that inner function can still access the parameters provided by the call to the outer function that returned it:-

```
function callLater(paramA, paramB, paramC){
    /* Return a reference to an anonymous inner function created
       with a function expression:-
    */
    return (function(){
        /* This inner function is to be executed with - setTimeout
           - and when it is executed it can read, and act upon, the
           parameters passed to the outer function:-
        */
        paramA[paramB] = paramC;
    });
}

...

/* Call the function that will return a reference to the inner function
   object created in its execution context. Passing the parameters that
   the inner function will use when it is eventually executed as
   arguments to the outer function. The returned reference to the inner
   function object is assigned to a local variable:-
*/
var functRef = callLater(elStyle, "display", "none");
/* Call the setTimeout function, passing the reference to the inner
   function assigned to the - functRef - variable as the first argument:-
*/
hideMenu=setTimeout(functRef, 500);
```

Example 2: Associating Functions with Object Instance Methods

There are many other circumstances when a reference to a function object is assigned so that it would be executed at some future time where it is useful to provide parameters for the execution of that function that would not be easily available at the time of execution but cannot be known until the moment of assignment.

One example might be a javascript object that is designed to encapsulate the interactions with a particular DOM element. It has `doOnClick`, `doMouseOver` and `doMouseOut` methods and wants to execute those methods when the corresponding events are triggered on the DOM element, but there may be any number of instances of the javascript object created associated with different DOM elements and the individual object instances do not know how they will be employed by the code that instantiated them. The object instances do not know how to reference themselves globally because they do not know which global variables (if any) will be assigned references to their instances.

So the problem is to execute an event handling function that has an association with a particular instance of the javascript object, and knows which method of that object to call.

The following example uses a small generalised closure based function that associates object instances with element event handlers. Arranging that the execution of the event handler calls the specified method of the object instance, passing the event object and a reference to the associated element on to the object method and returning the method's return value.

```
/* A general function that associates an object instance with an event
   handler. The returned inner function is used as the event handler.
   The object instance is passed as the - obj - parameter and the name
   of the method that is to be called on that object is passed as the -
   methodName - (string) parameter.
*/
function associateObjWithEvent(obj, methodName){
    /* The returned inner function is intended to act as an event
       handler for a DOM element:-
    */
    return (function(e){
        /* The event object that will have been parsed as the - e -
           parameter on DOM standard browsers is normalised to the IE
           event object if it has not been passed as an argument to the
           event handling inner function:-
        */
        e = e||window.event;
        /* The event handler calls a method of the object - obj - with
           the name held in the string - methodName - passing the now
           normalised event object and a reference to the element to
           which the event handler has been assigned using the - this -
           (which works because the inner function is executed as a
           method of that element because it has been assigned as an
           event handler):-
        */
        return obj[methodName](e, this);
    });
}

/* This constructor function creates objects that associates themselves
   with DOM elements whose IDs are passed to the constructor as a
   string. The object instances want to arrange than when the
   corresponding element triggers onclick, onmouseover and onmouseout
   events corresponding methods are called on their object instance.
*/
function DhtmlObject(elementId){
    /* A function is called that retrieves a reference to the DOM
       element (or null if it cannot be found) with the ID of the
       required element passed as its argument. The returned value
       is assigned to the local variable - el -:-
    */
    var el = getElementWithId(elementId);
    /* The value of - el - is internally type-converted to boolean for
       the - if - statement so that if it refers to an object the
       result will be true, and if it is null the result false. So that
       the following block is only executed if the - el - variable
       refers to a DOM element:-
    */
    if(el){
        /* To assign a function as the element's event handler this
           object calls the - associateObjWithEvent - function
           specifying itself (with the - this - keyword) as the object
           on which a method is to be called and providing the name of
           the method that is to be called. The - associateObjWithEvent
           - function will return a reference to an inner function that
           is assigned to the event handler of the DOM element. That
           inner function will call the required method on the
           javascript object when it is executed in response to
           events:-
        */
        el.onclick = associateObjWithEvent(this, "doOnClick");
        el.onmouseover = associateObjWithEvent(this, "doMouseOver");
        el.onmouseout = associateObjWithEvent(this, "doMouseOut");
        ...
    }
}
DhtmlObject.prototype.doOnClick = function(event, element){
    ... // doOnClick method body.
}
DhtmlObject.prototype.doMouseOver = function(event, element){
    ... // doMouseOver method body.
}
DhtmlObject.prototype.doMouseOut = function(event, element){
    ... // doMouseOut method body.
}
```

And so any instances of the `DhtmlObject` can associate themselves with the DOM element that they are interested in without any need to know anything about how they are being employed by other code, impacting on the global namespace or risking clashes with other instances of the `DhtmlObject`.

Example 3: Encapsulating Related Functionality

Closures can be used to create additional scopes that can be used to group interrelated and dependent code in a way that minimises the risk of accidental interaction. Suppose a function is to build a string and to avoid the repeated concatenation operations (and the creation of numerous intermediate strings) the desire is to use an array to store the parts of the string in sequence and then output the results using the `Array.prototype.join` method (with an empty string as its argument). The array is going to act as a buffer for the output, but defining it locally to the function will result in its re-creation on each execution of the function, which may not be necessary if the only variable content of that array will be re-assigned on each function call.

One approach might make the array a global variable so that it can be re-used without being re-created. But the consequences of that will be that, in addition to the global variable that refers to the function that will use the buffer array, there will be a second global property that refers to the array itself. The effect is to render the code less manageable, as, if it is to be used elsewhere, its author has to remember to include both the function definition and the array definition. It also makes the code less easy to integrate with other code because instead of just ensuring that the function name is unique within the global namespace it is necessary to ensure that the Array on which it is dependent is using a name that is unique within the global namespace.

A Closure allows the buffer array to be associated (and neatly packaged) with the function that is dependent upon it and simultaneously keep the property name to which the buffer array as assigned out of the global namespace and free of the risk of name conflicts and accidental interactions.

The trick here is to create one additional execution context by executing a function expression in-line and have that function expression return an inner function that will be the function that is used by external code. The buffer array is then defined as a local variable of the function expression that is executed in-line. That only happens once so the Array is only created once, but is available to the function that depends on it for repeated use.

The following code creates a function that will return a string of HTML, much of which is constant, but those constant character sequences need to be interspersed with variable information provided as parameter to the function call.

A reference to an inner function object is returned from the in-line execution of a function expression and assigned to a global variable so that it can be called as a global function. The buffer array is defined as a local variable in the outer function expression. It is not exposed in the global namespace and does not need to be re-created whenever the function that uses it is called.

```
/* A global variable - getImgInPositionedDivHtml - is declared and
   assigned the value of an inner function expression returned from
   a one-time call to an outer function expression.

   That inner function returns a string of HTML that represents an
   absolutely positioned DIV wrapped round an IMG element, such that
   all of the variable attribute values are provided as parameters
   to the function call:-
*/
var getImgInPositionedDivHtml = (function(){
    /* The - buffAr - Array is assigned to a local variable of the
       outer function expression. It is only created once and that one
       instance of the array is available to the inner function so that
       it can be used on each execution of that inner function.

       Empty strings are used as placeholders for the date that is to
       be inserted into the Array by the inner function:-
    */
    var buffAr = [
        '<div id="',
        '',   //index 1, DIV ID attribute
        '" style="position:absolute;top:',
        '',   //index 3, DIV top position
        'px;left:',
        '',   //index 5, DIV left position
        'px;width:',
        '',   //index 7, DIV width
        'px;height:',
        '',   //index 9, DIV height
        'px;overflow:hidden;\"><img src=\"',
        '',   //index 11, IMG URL
        '\" width=\"',
        '',   //index 13, IMG width
        '\" height=\"',
        '',   //index 15, IMG height
        '\" alt=\"',
        '',   //index 17, IMG alt text
        '\"><\/div>'
    ];
    /* Return the inner function object that is the result of the
       evaluation of a function expression. It is this inner function
       object that will be executed on each call to -
       getImgInPositionedDivHtml( ... ) -:-
    */
    return (function(url, id, width, height, top, left, altText){
        /* Assign the various parameters to the corresponding
           locations in the buffer array:-
        */
        buffAr[1] = id;
        buffAr[3] = top;
        buffAr[5] = left;
        buffAr[13] = (buffAr[7] = width);
        buffAr[15] = (buffAr[9] = height);
        buffAr[11] = url;
        buffAr[17] = altText;
        /* Return the string created by joining each element in the
           array using an empty string (which is the same as just
           joining the elements together):-
        */
        return buffAr.join('');
    }); //:End of inner function expression.
})();
/*^^- :The inline execution of the outer function expression. */
```

If one function was dependent on one (or several) other functions, but those other functions were not expected to be directly employed by any other code, then the same technique could be used to group those functions with the one that was to be publicly exposed. Making a complex multi-function process into an easily portable and encapsulated unit of code.

Other Examples

Probably one of the best known applications of closures is Douglas Crockford's technique for the emulation of private instance variables in ECMAScript objects. Which can be extended to all sorts of structures of scope contained nested accessibility/visibility, including the emulation of private static members for ECMAScript objects.

The possible application of closures are endless, understanding how they work is probably the best guide to realising how they can be used.

Accidental Closures

Rendering any inner function accessible outside of the body of the function in which it was created will form a closure. That makes closures very easy to create and one of the consequences is that javascript authors who do not appreciate closures as a language feature can observe the use of inner functions for various tasks and employ inner functions, with no apparent consequences, not realising that closures are being created or what the implications of doing that are.

Accidentally creating closures can have harmful side effects as the following section on the IE memory leak problem describes, but they can also impact of the efficiency of code. It is not the closures themselves, indeed carefully used they can contribute significantly towards the creation of efficient code. It is the use of inner functions that can impact on efficiency.

A common situation is where inner functions are used is as event handlers for DOM elements. For example the following code might be used to add an onclick handler to a link element:-

```
/* Define the global variable that is to have its value added to the
   - href - of a link as a query string by the following function:-
*/
var quantaty = 5;
/* When a link passed to this function (as the argument to the function
   call - linkRef -) an onclick event handler is added to the link that
   will add the value of a global variable - quantaty - to the - href -
   of that link as a query string, then return true so that the link
   will navigate to the resource specified by the - href - which will
   by then include the assigned query string:-
*/
function addGlobalQueryOnClick(linkRef){
    /* If the - linkRef - parameter can be type converted to true
       (which it will if it refers to an object):-
    */
    if(linkRef){
        /* Evaluate a function expression and assign a reference to the
           function object that is created by the evaluation of the
           function expression to the onclick handler of the link
           element:-
        */
        linkRef.onclick = function(){
            /* This inner function expression adds the query string to
               the - href - of the element to which it is attached as
               an event handler:-
            */
            this.href += ('?quantaty='+escape(quantaty));
            return true;
        };
    }
}
```

Whenever the `addGlobalQueryOnClick` function is called a new inner function is created (and a closure formed by its assignment). From the efficiency point of view that would not be significant if the `addGlobalQueryOnClick` function was only called once or twice, but if the function was heavily employed many distinct function objects would be created (one for each evaluation of the inner function expression).

The above code is not taking advantage of the fact that inner functions are becoming accessible outside of the function in which they are being created (or the resulting closures). As a result exactly the same effect could be achieved by defining the function that is to be used as the event handler separately and then assigning a reference to that function to the event handling property. Only one function object would be created and all of the elements that use that event handler would share a reference to that one function:-

```
/* Define the global variable that is to have its value added to the
   - href - of a link as a query string by the following function:-
*/
var quantaty = 5;

/* When a link passed to this function (as the argument to the function
   call - linkRef -) an onclick event handler is added to the link that
   will add the value of a global variable - quantaty - to the - href -
   of that link as a query string, then return true so that the link
   will navigate to the resource specified by the - href - which will
   by then include the assigned query string:-
*/
function addGlobalQueryOnClick(linkRef){
    /* If the - linkRef - parameter can be type converted to true
       (which it will if it refers to an object):-
    */
    if(linkRef){
        /* Assign a reference to a global function to the event
           handling property of the link so that it becomes the
           element's event handler:-
        */
        linkRef.onclick = forAddQueryOnClick;
    }
}
/* A global function declaration for a function that is intended to act
   as an event handler for a link element, adding the value of a global
   variable to the - href - of an element as an event handler:-
*/
function forAddQueryOnClick(){
    this.href += ('?quantaty='+escape(quantaty));
    return true;
}
```

As the inner function in the first version is not being used to exploit the closures produced by its use, it would be more efficient not to use an inner function, and thus not repeat the process of creating many essentially identical function objects.

A similar consideration applies to object constructor functions. It is not uncommon to see code similar to the following skeleton constructor:-

```
function ExampleConst(param){
    /* Create methods of the object by evaluating function expressions
       and assigning references to the resulting function objects
       to the properties of the object being created:-
    */
    this.method1 = function(){
        ... // method body.
    };
    this.method2 = function(){
        ... // method body.
    };
    this.method3 = function(){
        ... // method body.
    };
    /* Assign the constructor's parameter to a property of the object:-
    */
    this.publicProp = param;
}
```

Each time the constructor is used to create an object, with `new ExampleConst(n)`, a new set of function objects are created to act as its methods. So the more object instances that are created the more function objects are created to go with them.

Douglas Crockford's technique for emulating private members on javascript objects exploits the closure resulting form assigning references to inner function objects to the public properties of a constructed object from within its constructor. But if the methods of an object are not taking advantage of the closure that they will form within the constructor the creation of multiple function objects for each object instantiation will make the instantiation process slower and more resources will be consumed to accommodate the extra function objects created.

In that case it would be more efficient to create the function object once and assign references to them to the corresponding properties of the constructor's `prototype` so they may be shared by all of the objects created with that constructor:-

```
function ExampleConst(param){
    /* Assign the constructor's parameter to a property of the object:-
    */
    this.publicProp = param;
}
/* Create methods for the objects by evaluating function expressions
   and assigning references to the resulting function objects to the
   properties of the constructor's prototype:-
*/
ExampleConst.prototype.method1 = function(){
    ... // method body.
};
ExampleConst.prototype.method2 = function(){
    ... // method body.
};
ExampleConst.prototype.method3 = function(){
    ... // method body.
};
```

The Internet Explorer Memory Leak Problem

The Internet Explorer web browser (verified on versions 4 to 6 (6 is current at the time of writing)) has a fault in its garbage collection system that prevents it from garbage collecting ECMAScript and some host objects if those host objects form part of a "circular" reference. The host objects in question are any DOM Nodes (including the document object and its descendants) and ActiveX objects. If a circular reference is formed including one or more of them, then none of the objects involved will be freed until the browser is closed down, and the memory that they consume will be unavailable to the system until that happens.

A circular reference is when two or more objects refer to each other in a way that can be followed and lead back to the starting point. Such as object 1 has a property that refers to object 2, object 2 has a property that refers to object 3 and object 3 has a property that refers back to object 1. With pure ECMAScript objects as soon as no other objects refer to any of objects 1, 2 or 3 the fact that they only refer to each other is recognised and they are made available for garbage collection. But on Internet Explorer, if any of those objects happen to be a DOM Node or ActiveX object, the garbage collection cannot see that the circular relationship between them is isolated from the rest of the system and free them. Instead they all stay in memory until the browser is closed.

Closures are extremely good at forming circular references. If a function object that forms a closure is assigned as, for example, and event handler on a DOM Node, and a reference to that Node is assigned to one of the Activation/Variable objects in its scope chain then a circular reference exists. `DOM_Node.onevent -> function_object.[[scope]] -> scope_chain -> Activation_object.nodeRef -> DOM_Node`. It is very easy to do, and a bit of browsing around a site that forms such a reference in a piece of code common to each page can consume most of the systems memory (possibly all).

Care can be taken to avoid forming circular references and remedial action can be taken when they cannot otherwise be avoided, such as using IE's onunload event to null event handling function references. Recognising the problem and understanding closures (and their mechanism) is the key to avoiding this problem with IE.

comp.lang.javascript FAQ notes T.O.C.

Written by **Richard Cornford**. March 2004.
With corrections and suggestions by:-
Martin Honnen.
Yann-Erwan Perio (Yep).
Lasse Reichstein Nielsen. (definition of closure)
Mike Scirocco.
Dr John Stockton.