If you haven't heard of JavaScript Closures, then you're in for some good news and some good news and some bad news. Roughly speaking, closures describe how JavaScript variables are inherited in nested functions. The good news about closures is that JavaScript will automatically retain a variable and its value as long as it's needed, even if the original function has long disappeared. The bad news is that it may not be the variable of value that you had in mind.

Variable Scope

Variable scope is the section of code for which a variable is defined and retains its value. You can think of scope as the variable's context. A more detailed discussion of variable scope will appear elsewhere, but the short version is new variables are created either in the context of a new function, or, by default in the global context.

It is generally received wisdom that global variables are to be avoided, or, at least minimised. In JavaScript, it is all to easy to create a new global variable, either by design or my accident. For this reason, it is always a good idea to declare new variables with the var keyword, and to use strict mode to enforce this.

JavaScript Closures

In JavaScript, there are essentially two types of scope: Global and Function. Since functions can be nested, you can have function scopes within function scopes.

A JavaScript Closure is a sort of mini global environment. Since JavaScript only defines a new scope inside a function, the containing function becomes a closure for this purpose. It goes a little like this:

```
var date=new Date();    //  Create a date object with the current date.
var then=new Date();

setTimeout(test,2000);      //  Run the test() function after 2 seconds.
function test() {
    var date=new Date();    //  A new variable with the same name
    alert(date);            //  Show the LOCAL date, just created
    alert(then)             //  Show the GLOBAL then, created earlier
}
```

Of course, you already know that local variables created inside the fuction will take precedence over global variables with the same name; if there is no local variable, JavaScript will fall through to the global variable.

Here is another example, using a nested function:

```
//  … as before …

function test() {
    var date=new Date();
    doit();
    function doit() {
        alert(date);    //  Show the LOCAL date, just created
        alert(then)     //  Show the GLOBAL then, created earlier
    }
}
```

The doit function is nested inside the test function. In this case, neither variable is local, so the function will use the values it inherits from its parent scope.

Now, let's make things worse:

```
//  No Global date variable now
var then=new Date();
setTimeout(test,2000);      //  Run the test() function after 2 seconds.
function test() {
    var date=new Date();
    setTimeout(doit,2000);  //  call doit after 2 seconds
    function doit() {
        alert(date);        //  Show the LOCAL date, just created
        alert(then)         //  Show the GLOBAL then, created earlier
    }
    //  return
}
```

This does the same job as before, but the actual messages are delayed by a further 2 seconds. The comment at the end of the function is simply there to remind you that the function will finish at this point, having put the doit function on the queue for later.

The important thing to note is that by the time the queued doit function is called, you would have expected the date variable to have long gone, together with the function in which it was defined; and in this example we have not defined a global date variable either. The good news, depending on what you were hoping for, is that the date variable inside the test function is retained until it is no longer needed.

A closure, then, is the persistent variable scope which is retained, even when the containing function has finished. In this case, the variables inside the test function are retained for the doit function.

Postponing a function isn't the only way to run a function when the creating context is gone. You will also see the same situation when assigning an event handler. For example:

```
function init() {
    var date = new Date();
    document.getElementByID('doit').onclick=function() {
        alert(date);
    };
}
```

The init function, possibly called when the window loads, defines a local variable to be displayed when you click on an element. Again, the variable and its value are retained long after the init function has finished.

All this is useful, but it comes, of course, with a trap.

Backfiring Closures

The fact that the variable itself persists is easily misunderstood when using the data in the nested function. Take the following simple example:

```
function test() {
    var thing=3;
    setTimeout(doit,2000);
    function doit() {
        alert(thing);
    }
    thing=4;
}
```

What is the value of thing when the function is called? Note that the alert function has as its parameter the variable thing, not a particular value. Note also that thing is not local to doit, but is inherited by it. As a result, when the time comes to display it, it will be the current value, now 4, which will be used. That is, thing is very much a live variable, and is subject to change after the event.

The Infamous Loop Problem

The problem is clearly demonstrated in a very common operation: assigning event handlers to multiple elements. Suppose, for example, you want to assign event handlers to a number of radio buttons inside a form.

HTML

```
<form id="contact">
    …
    <label><input type="radio" name="choice" value="…" …> … </label>
    <label><input type="radio" name="choice" value="…" …> … </label>
</form>
```

Here the form has an id of contact, and the buttons share a name of choice. We attempt to assign event handlers as follows:

JavaScript

```
var form = document.getElementById('contact');  // Get the Form Element
var buttons = form.getElementsByName('choice'); //  A collection of elements named “choice”
for(var i=0;i<buttons.length;i++) {     //  For each button: 0, 1, 2
    buttons[i].onclick=function() {     //  Assign function as onclick handlers
        alert(i);
    }
}
```

The bad news is that each button will respond with 3! This is because you are displaying the current value of i, which finished at 3 when we ran out of buttons.

There is no simple way to pass the actual value rather than the variable to the nested function, but there is a method which looks rather indirect, and somewhat befuddling, but is full of all of the goodness of closures.

First, we need to remember (or if you didn’t know, understand) that JavaScript functions are objects, and can be assigned to variables, and can be returned as data from other functions. The first step is to create and additional function which will, in turn, return the required function as a return value:

```
function newFunction(data) {
    return function() {
        alert(data);    //  Desired value
        //  alert(i);   //  Reference to inherited variable
    }
}
```

In this case the newFunction returns a copy of the function we are trying to assign in the previous example. Note this generates a new function. The important thing is that the data to be displayed is passed as a parameter rather than inherited (you can still use the inherited value as above, but it will be just as before).

Secondly, rather than assigning the original function to the event handler, we assign the return result of the

new function:

```
for(var i=0;i<buttons.length;i++) {     //  For each button: 0, 1, 2
    buttons[i].onclick=newFunction(i);  //  Assign function as onclick handle
rs
}
```

Rather than actually create a new named function, we can involve an anonymous function:

```
for(var i=0;i<buttons.length;i++) {     //  For each button: 0, 1, 2
    buttons[i].onclick=function(data) { //  Assign function as onclick handle
rs
        return function() {
            alert(data);    //  Desired value
            //  alert(i);   //  Reference to inherited variable
        }
    }(i);
}
```

Note (a) we dispense with the name of the anonymous function; and (b) we invoke the anonymous function directly with the parentheses and the variable i.

Anonymous Functions

Anonymous function, as the name suggests, is a function without a name.

```
function(…) {
    …
}
```

By itself, it is useless, but it can be assigned to a variable:

```
var fn=function(…) {
    …
}
```

This is nearly the same as

```
function fn(…) {
    …
}
```

but, as discussed elsewhere has some subtle differences.

You can also assign an anonymous function to an object property.

Finally, you can run an anonymous function immediately. In the first case, you might assign the return value of the function to a variable:

```
var thing=function(…) {
    …
}(…);
```

Note that we invoke the function in the usual way by appending parentheses with possible arguments.

If the function has no return value, or none we want to keep, we can also simple invoke it as follows:

```
(function(…) {
    …
})(…);
```

Undoubtedly you noticed the additional parentheses. It's a precedence thing: JavaScript will misinterpret your intentions here, and must be told to define the function before trying to run it.

For some reason, assigning the whole thing to a variable doesn't require this.

A Generalised Closure Function

The following simply wraps the above discussion inside a function which will allow you to attach any function complete with closure variables.

```
/*  Closure Function
    ========================================================= */

    function cloneFunction(fn) {
        //  usage: … = cloneFunction(fn[,…,…,…]);           //  optional addi
tional arguments
        var args=Array.prototype.slice.call(arguments,1);   //  get additiona
l arguments
        return function() {
            fn.apply(fn,args);                              //  run fn with a
dditional arguments
        }
    }
```

The function is simple enough, but does involve some advanced function techniques, such as the function arguments object and borrowing object methods, which will be discussed in another article.

Here is how it might be used:

```
//  Sample Function

function sample(i) {        //  additional parameters will work
    alert(i);
}

for(var i=0;i<buttons.length;i++) { //  For each button: 0, 1, 2
    buttons[i].onclick=cloneFunction(sample,i);
}
```

Update: `bind` ing a Function

Modern JavaScript offers a more natural way of generatiing a copy of a function. Each function has a built-in `bind` method, which creates a clone of the function.

`bind` also offers an additional advantage: it allows you to set the value of `this`. Normally, a function can be called by one or other object; if nothing else, it's the `window` object. Using `bind`, you can predetermine which object will be used.

The general syntax is:

```
fn.bind([object][,arguments]);
```

Typically it's used to attach the object which is creating the function:

```
fn.bind(this[,arguments]);      //  bind current object
```

The above example may now be re-written as this:

```
//  Sample Function

function sample(i) {        //  additional parameters will work
    alert(i);
}

for(var i=0;i<buttons.length;i++) { //  For each button: 0, 1, 2
    buttons[i].onclick=sample.bind(this,i);
}
```

Note that in this example, we're not actually using `this`, but we need to fill it in.

For those coping with Legacy™ Browsers, this simple patch will do the job:

```
if(!Function.prototype.bind)    //  what follows is only for old Browsers:
Function.prototype.bind=function(context) {
    var args = Array.prototype.slice.call(arguments, 1);
    var thing = this;
    return function() {
        return thing.apply(context, args.concat([].slice.call(arguments)));
    }
}
```

This will certainly do the job, but it presupposes that you're not trying to do anything stupid with the code. More detain on the `bind` function, as well as a more comprehensive patch is available from:

Function.prototype.bind() - JavaScript | MDN

Share and Enjoy …