Working with Closures

One of the most important, albeit abstract JavaScript concepts is the *closure*. You'll come across different ways of describing closures, but I think it's easiest to think of a closure as a function call with a memory. In other words, a closure is a function tied to the scope in which it was created. This means that a closure function can make use of the variables that existed (in the same scope) when the function was created.

Loosely speaking, you *might* have a closure situation when:

• One function is defined within another

• The inner function references variables that exist in the outer function (including the outer function's parameters)

• The inner function will be called after the outer function has stopped executing

Let's look at an example to better explain this: The `tasks.js` script just defined has a closure in it. The key components of the onload anonymous function are:

```
window.onload = function() {
   // Setup variables, including tasks.
   document.getElementById('theForm').onsubmit = function() {
      tasks.push(t);
   };
};
```

The onload function will only be called once: when the page loads. That function defines some variables, including the `tasks` array. In a normal, non-closure situation, function variables are no longer available once the function execution has completed. This means that without a closure, the `tasks` array will cease to exist after the onload function has executed all its commands.

The closure is created by defining another function within that outer onload function. All closures require that one function be defined within another. The inner function, which handles the form submission, will be called any number of times, but always after the outer function has finished executing (**Figure 14.9**).

Figure 14.9. Form submissions will invoke one function defined within another function that has long since stopped executing.

Because the onsubmit function will be called after the onload function has finished executing, and because the onload function has variables with the same scope as the onsubmit function, JavaScript retains those variables after the onload function has completed, creating a closure. Thanks to the closure, the onsubmit function can make use of `tasks`, because the variable is kept alive. This is the hallmark of a closure: the local variables that were available to the function when the function was defined are still available to that function even when it is called at a later time. The closure creates a persistent but still locally scoped variable.

A variation on this same script created in Chapter 7 also had a closure, this time using an immediately invoked function:

```
(function(){
   var tasks = [];
   function addTask() {
      // Use tasks.
   }
   function init() {
```

```
    document.getElementById('theForm').onsubmit = addTask;
  }
  window.onload = init;
})();
```

The outermost anonymous JavaScript function is executed as soon as JavaScript encounters it. Within that function is a variable, `tasks`, which exists in the local scope. The `addTask()` function will be called every time the form is submitted, which will always come after this immediately invoked anonymous function has stopped running. Again a closure is created, with the `addTask()` function retaining access to the `tasks` variable.

In a moment I'll walk through another closure example, but I want to highlight one common point of confusion first. As explained, a key feature of closure functions is that they have access to the local variables that existed (in the same scope) when the closure was defined. The trick is that the closure will have access to the value of the variable at the time of the closure function *call*, not its definition. For example, this next onload function is a closure with access to the `i` variable:

```
(function() {
  var i = 1;
  window.onload = function() {
    alert(i); // 2, not 1
  }
  i = 2;
})();
```

When the inner function is defined, `i` is initially assigned a value of 1. By the time the inner function is called—in this case, after the outer immediately invoked function has terminated—`i` will have a value of 2, which is what will be alerted.

This problem most commonly arises when closures are created within a loop. For example, say you want to add a click handler to every link in a page. The click handler will then do something with the link that was clicked. You might think you could do this:

```
function someFunction() {
  var links = document.getElementsByTagName('a');
  for (var i = 0, count = links.length; i < count; i++) {
    links[i].onclick = function() {
      // Use links[i] (but this will not work).
      return false;
    }
  }
}
```

What you would find is that `links[i]` within the onclick function always returns `undefined`. To discover why, let's assume there are two links. The `for` loop would be executed twice, successfully adding a click handler to each link. Then the loop terminates when `i` becomes 2, which is greater than the length of the `links` array. When you click on one of the links, because of the closure, the onclick anonymous function will still be able to access `i`, but its value will be 2, which is the last value that the variable had.

There are more advanced uses of closures, most notably involving situations in which one function returns another function. But the topic itself is difficult enough to grasp that I'm choosing to start with the most accessible uses, specifically related to event handling. For another example of a closure, the following script will use a timer. Timers, by their very nature, have functions that are defined at one time but executed at another (see Chapter 9, JavaScript and the Browser, for more on

timers). The following example will use a timer and a closure to create a fader, which changes the opacity of an element from 100 percent to 0 percent to fade it out gradually.

For the HTML page, to be named `fader.html`, you can create any visible element with an `id` value of *target*. It doesn't matter whether the element is a paragraph of text or an image. However, you do need to add this CSS to your page, in order for Internet Explorer to recognize changes in this opacity:

```
#target { zoom: 1; }
```

The HTML page should include the `fader.js` script, to be defined in the following steps.

To use a closure to create a fader:

1. Create a new JavaScript file in your text editor or IDE, to be named `fader.js`.

2. Begin defining an onload anonymous function:

```
window.onload = function() {
   'use strict';
   var target = document.getElementById('target');
```

The outer function first gets a reference to the target element.

3. Set the initial opacity:

```
var opacity = 100;
```

The opacity begins at 100 (i.e., percent), and will be decreased within the fader function.

4. Begin defining the `setInterval()` function:

```
var fader = setInterval(function() {
   opacity -= 10;
```

The `setInterval()` function takes a function as its first argument. This function, which will be called repeatedly after the outer, anonymous onload function terminates, will be a closure, with access to the `target` and `opacity` variables.

Within the function, the opacity is reduced by 10, so that each invocation of the function dims the target even more.

5. If the opacity is greater than or equal to 0, change the opacity style:

```
if (opacity >= 0) {
   if (typeof target.style.opacity == 'string') {
      target.style.opacity = (opacity/100);
   } else {
      target.style.filter = 'alpha(opacity=' + opacity + ')';
   }
```

First, the opacity should only be changed if `opacity` is a positive number. Once the opacity becomes a negative number, the process should stop (in the next step).

To change the opacity, one has to use either the `style.opacity` or the `style.filter` property of the target element, depending upon the browser (see a CSS reference for more details, if needed). For the former, the opacity value needs to be a decimal, so `opacity` is divided by 100.

Note that although closures have access to the last known value of a variable (the common problem explained earlier), the value of `opacity` is changed *within* this closure function, meaning its value is retained from function call to function call.

6. If `opacity` is not greater than or equal to 0, stop the timer:

```
} else {
    clearInterval(fader);
}
```

7. Complete the timer:

```
}, 100);
```

The anonymous function will be called every 100 milliseconds.

8. Complete the onload anonymous function:

```
};
```

9. Save the file as `fader.js`, in a `js` directory next to `fader.html`, and test it in your Web browser.

There's no point to providing an image for this example, since it's quite hard to demonstrate animation in a book!