Functions as Objects

Functions in JavaScript have a very unique quality in that *functions are also themselves objects*. This makes functions “first-class” citizens in JavaScript: they can be used and manipulated as you would any other value type. This probably sounds rather abstract to you now, but the end result is that you can do things with JavaScript functions that you cannot do with functions in many other languages. Moreover, although the implications are complicated, understanding functions as objects will help you to appreciate some of the things commonly done in JavaScript, including many pieces of code you’ve already seen.

Looking back at what you already know, say you create a new (and unnecessary) function in JavaScript like so:

```
function getOne() {
   return 1;
}
```

You understand, certainly, that `getOne()` is a function, and that it can be invoked:

```
getOne();
```

However, in JavaScript, a function is an object, specifically of type `Function`. By declaring that function you’ve also created an object variable, with an identifier of *getOne*, whose value is the function definition (**Figure 7.13**).

Figure 7.13. A function variable’s value is the function definition.

Because of this quality, you can test for the presence of a function using code like:

```
if (Date.now) {
```

That code verifies that there is a definition for `now` as part of the `Date` object. This is different than the `Date.now()` function call.

More precisely, you could check that the property is a function:

```
if (typeof Date.now == 'function') {
```

Once you understand that a function definition is just another type of value in JavaScript, you might realize that you can do with a function definition what you can do with any value type, such as a number or string, including: assign the function definition to a variable, use it as a value to be passed to another function, or even return a function from another function. I’ll explain...

Functions as Variable Values

The syntax used thus far for declaring a function constitutes a JavaScript *statement*. You can also create a function using an *expression*, whereby the creation of the function as a value of a variable is overt:

```
var getTwo = function() {
   return 2;
}
```

This syntax probably seems strange, but the end result is the same: an object of type `Function` has been created. Because it’s a `Function` object, it can be invoked, unlike other objects (**Figure 7.14**):

```
getTwo();
```

Figure 7.14. A function definition can be a value assigned to a variable.

Any value that can be assigned to a variable can also be assigned to an object property, as an object property is just a variable associated with an object. This is code that's been used many times over in this book:

```
window.onload = init;
```

That code assigns to the `unload` property of the `window` object the value of the `init` variable, which is to say the `init()` function definition.

Note that the code does not invoke the function—it's lacking the invocation parentheses:

```
window.onload = init(); // No!
```

Doing the above would call the `init()` function and assign the value returned by it to the `window.onload` property, which is not the intent.

Taking this further, you can skip the step of naming the function and/or creating a function variable, and just assign a function expression to an object property directly:

```
window.onload = function() {
   // Function body goes here.
}
```

In that code, the function itself is called an *anonymous* function, as it has no name. You'll use anonymous functions frequently in JavaScript.

Functions as Argument Values

A second way you can use a function as an object is to pass a function definition to another function, as you would any other argument value. This only makes sense, of course, in situations where the function being called expects one of its arguments to be a function. To do this, you can create the function and assign it to a variable, then pass that variable to the other function:

```
var someFunction = function() {
};
someOtherFunction(someFunction);
```

Or, you can also simplify this and write the function definition within the other function's invocation:

```
someOtherFunction(function() {
});
```

When you do this, just be mindful of the syntax so that you don't create a syntactical error. (In both cases, these are also anonymous functions.)

As an example, in Chapter 6, it's said that arrays have a `sort()` method, but that the method is of limited use without knowing how to define your own functions. This is because the built-in `sort()` method can only reliably be used to sort array elements alphabetically. This is fine if you have an array of strings (**Figure 7.15**):

```
var people = ['Mac', 'Dennis', 'Dee', 'Frank', 'Charlie'];
people.sort();
```

Figure 7.15. The `sort()` method will perform a proper, case-sensitive sorting of strings.

In current browsers, `sort()` will properly sort numbers, but in older browsers, sorting a list of numbers was done alphabetically, too:

```
var numbers = [1, 4, 3, 2];
numbers.sort(); // 4, 1, 3, 2
```

The solution (again, for the older browsers) was to create a function that will perform the comparison needed, and then to tell the `sort()` method to use that function instead of its default mechanism. The comparison function needs to take two arguments—the two values being compared—and return:

• A negative value if the first argument comes before the second

• 0, if the two arguments are the same

• A positive value if the second argument comes before the first

Conventionally, the returned values are -1, 0, and 1.

Thus, to sort an array of numbers, the code to use is (**Figure 7.16**):

```
function compareNumbers(x, y) {
   return x-y;
}
var numbers = [1, 4, 3, 2];
numbers.sort(compareNumbers);
```

Figure 7.16. To change how array elements are sorted, provide the method with your own function definition.

First, note that the function *identifier* is being used as the argument value to `sort()`, not a function *call*. Within the function, a little shortcut is being used to determine what value is returned: the second argument is subtracted from the first. If the result of the subtraction is positive, then `x` must be bigger (e.g., 8-7); if the result is negative, then `x` must be smaller (e.g., 7-8); if the numbers are the same, 0 will be returned.

As another example, if you wanted to perform a case-insensitive string sort, you can write a function to do that:

```
function caseInsensitiveCompare(x, y) {
   x = x.toLowerCase();
   y = y.toLowerCase();
   if (x > y) {
      return 1;
   } else if (y > x) {
      return -1;
   } else {
      return 0;
   }
}
```

Putting It Together

To practice providing functions as arguments to other functions, let’s look at some of the new array functions added in ECMAScript 5. Each of these requires a user-defined function in order to work:

• `forEach()` loops through an array, one element at a time.

• `every()` tests each array element against a condition and returns a Boolean if every element passes.

• `some()` tests each array element against a condition and returns a Boolean if at least one element passes.

• `map()` provides each array element to a function where it will be modified and returned, creating a new array.

• `filter()` tests each array element against a condition and only returns those that pass, creating a new array in the process.

• `reduce()` can be used to group an array's elements into a single value.

For example, to confirm that an array contains nothing but strings (e.g., prior to sorting the array), you can use `every()`. It returns a Boolean value indicating if every element in the array passes the condition set in the user-defined function (**Figure 7.17**):

```
var mix = [1, true, 'test'];
mix.every(function (value) {
   return (typeof value == 'string');
});
```

Figure 7.17. The `every()` method returns `false` because not every element in the array is a string.

As a reminder, these are newer functions, and may not be supported by all browsers (**Figure 7.18**). To test for support, and perform the same task regardless, you could use code like this:

```
// Function that returns a Boolean indicating a String:
function testForString(value) {
   return (typeof value == 'string');
}
// Array to be tested:
var mix = [1, true, 'test'];
if (mix.every) { // Can use every()!
   var result = mix.every(testForString);
} else { // Must write every() equivalent.
   var result = true; // Assume truth.
   for (var i = 0, count = mix.length; i++) { // Loop through array.
      if (!testForString(mix[i])) { // Is it not a String?
         result = false; // Change result to false.
         break; // Terminate the loop.
      } // IF
   } // FOR
}
```

Figure 7.18. IE9 does not support the `every()` method.

As another example, this next script will take a list of words from the user, then perform a case-insensitive sort of the words, and output the result (**Figure 7.19**). The relevant HTML, in a page named `words.html`, is:

```
<div><label for="words">Words</label><input type="text" name="words" id="words"
required></div>
<input type="submit" value="Sort!" id="submit">
<h2>Sorted Words</h2>
```

```
<div id="output"></div>
```

Figure 7.19. A sentence of words is quickly parsed, sorted, and redisplayed by this script.

The form uses one text input for the list of words. Just below the submit button is an empty `DIV` that will be updated by the JavaScript code, providing the sorted output.

The HTML page includes the `words.js` JavaScript file, to be written in the subsequent steps.

To sort an array with a user-defined function:

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

2. Define the `$()` function:

```
function $(id) {
   'use strict';
   if (typeof id != 'undefined') {
      return document.getElementById(id);
   }
} // End of $ function.
```

This function, explained earlier, will be used to get references to form elements.

3. Define the `setText()` function:

```
function setText(elementId, message) {
   'use strict';
   if ( (typeof elementId == 'string')
   && (typeof message == 'string') ) {
      var output = $(elementId);
      if (output.textContent !== undefined) {
         output.textContent = numbers;
      } else {
          output.innerText = numbers;
      }
   } // End of main IF.
} // End of setText() function.
```

This function will also be used by the script.

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

```
function sortWords(max) {
   'use strict';
   var words = $('words').value;
```

The `sortWords()` function does the work when the form is submitted. It starts by getting a reference to the form value.

5. Convert the string to an array:

```
words = words.split(' ');
```

The `split()` function returns an array of pieces from a string, using the provided argument as the delineator. It was explained in Chapter 6. The result of the operation is assigned back to `words`, changing that string into an array.

6. Perform a case-insensitive sort of the words:

```
var sorted = words.map(function(value) {
    return value.toLowerCase();
}).sort();
```

That code looks a bit complicated because it has two chained method calls and an anonymous function, but here's what is happening: To the `words` array, the `map()` method is applied. The `map()` method takes a function as its argument, and `map()` will pass to that function each array element, one at a time. The anonymous function used as the `map()` argument therefore has to be written to accept a value as an argument. This value can be manipulated and returned: in this case, the value is converted to all lowercase letters. The result of using `map()` is a new array. To this array, the `sort()` method is applied. The result of that action is then assigned to the new `sorted` variable.

You could write this out more overtly as:

```
var changeToLowerCase(value) {
    return value.toLowerCase();
}
var sorted = words.map(changeToLowerCase);

sorted = sorted.sort();
```

You could also combine the code in Steps 5 and 6 to make it more complicated, but a single step.

To save space, this code does not check if the browser supports the `map()` method. That can be a challenge for you to pursue, using the code already explained as a starting point.

7. Send the output to the page:

```
setText('output', sorted.join(', '));
```

Finally, the HTML page is updated using the `setText()` function. For the text itself, a function call to `join()` provides that value. It returns a string using the provided argument as the glue (it was also discussed in the last chapter).

8. Complete the `sortWords()` function:

```
    return false;
} // End of sortWords() function.
```

9. Add an event listener to the form's submission:

```
function init() {
    'use strict';
    $('theForm').onsubmit = sortWords;
} // End of init() function.
window.onload = init;
```

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