CHAPTER 2

Object-Oriented JavaScript

Objects are the fundamental units of JavaScript. Virtually everything in JavaScript is an object and takes advantage of that fact. However, to build up a solid object-oriented language, JavaScript includes a vast arsenal of features that make it an incredibly unique language, both in possibilities and in style.

In this chapter I'm going to begin by covering some of the most important aspects of the JavaScript language, such as references, scope, closures, and context, that you will find sorely lacking in other JavaScript books. After the important groundwork has been laid, we'll begin to explore the important aspects of object-oriented JavaScript, including exactly how objects behave and how to create new ones and set up methods with specific permissions. This is quite possibly the most important chapter in this book if taken to heart, as it will completely change the way you look at JavaScript as a language.

Language Features

JavaScript has a number of language features that are fundamental to making the language what it is. There are very few other languages like it. Personally, I find the combination of features to fit just right, contributing to a deceptively powerful language.

References

A fundamental aspect of JavaScript is the concept of reference*s*. A *reference* is a pointer to an actual location of an object. This is an incredibly powerful feature The premise is that a physical object is never a reference. A string is always a string; an array is always an array. However, multiple variables can refer to that same object. It is this system of references that JavaScript is based around. By maintaining sets of references to other objects, the language affords you much more flexibility.

Additionally, an object can contain a set of properties, all of which are simply references to other objects (such as strings, numbers, arrays, etc.). When multiple variables point to the same object, modifying the underlying type of that object will be reflected in all variables. An example of this is shown in Listing 2-1, where two variables point to the same object, but the modification of the object's contents is reflected globally.

Listing 2-1. *Example of Multiple Variables Referring to a Single Object*

```
// Set obj to an empty object
var obj = new Object();

// objRef now refers to the other object
var objRef = obj;

// Modify a property in the original object
obj.oneProperty = true;

// We now see that that change is represented in both variables
// (Since they both refer to the same object)
alert( obj.oneProperty === objRef.oneProperty );
```

I mentioned before that self-modifying objects are very rare in JavaScript. Let's look at one popular instance where this occurs. The array object is able to add additional items to itself using the push() method. Since, at the core of an Array object, the values are stored as object properties, the result is a situation similar to that shown in Listing 2-1, where an object becomes globally modified (resulting in multiple variables' contents being simultaneously changed). An example of this situation can be found in Listing 2-2.

Listing 2-2. *Example of a Self-Modifying Object*

```
// Create an array of items
var items = new Array( "one", "two", "three" );

// Create a reference to the array of items
var itemsRef = items;

// Add an item to the original array
items.push( "four" );

// The length of each array should be the same,
// since they both point to the same array object
alert( items.length == itemsRef.length );
```

It's important to remember that references only point to the final referred object, not a reference itself. In Perl, for example, it's possible to have a reference point to another variable, which also is a reference. In JavaScript, however, it traverses down the reference chain and only points to the core object. An example of this situation can be seen in Listing 2-3, where the physical object is changed but the reference continues to point back at the old object.

Listing 2-3. *Changing the Reference of an Object While Maintaining Integrity*

```
// Set items to an array (object) of strings
var items = new Array( "one", "two", "three" );
```

```
// Set itemsRef to a reference to items
var itemsRef = items;

// Set items to equal a new object
items = new Array( "new", "array" );

// items and itemsRef now point to different objects.
// items points to new Array( "new", "array" )
// itemsRef points to new Array( "one", "two", "three" )
alert( items !== itemsRef );
```

Finally, let's look at a strange instance that appears to be one of object self-modification, but results in a new nonreferential object. When performing string concatenation the result is always a new string object rather than a modified version of the original string. This can be seen in Listing 2-4.

Listing 2-4. *Example of Object Modification Resulting in a New Object, Not a Self-Modified Object*

```
// Set item equal to a new string object
var item = "test";

// itemRef now refers to the same string object
var itemRef = item;

// Concatenate some new text onto the string object
// NOTE: This creates a new object, and does not modify
// the original object.
item += "ing";

// The values of item and itemRef are NOT equal, as a whole
// new string object has been created
alert( item != itemRef );
```

References can be a tricky subject to wrap your mind around, if you're new to them. Although, understanding how references work is paramount to writing good, clean JavaScript code. In the next couple sections we're going to look at a couple features that aren't necessarily new or exciting but are important to writing good, clean code.

Function Overloading and Type-Checking

A common feature in other object-oriented languages, such as Java, is the ability to "overload" functions to perform different behaviors when different numbers or types of arguments are passed to them. While this ability isn't immediately available in JavaScript, a number of tools are provided that make this quest entirely possible.

Function overloading requires two things: the ability to determine how many arguments are provided, and the ability to determine the type of the arguments that are provided. Let's start by looking at the number of arguments provided.

Inside of every function in JavaScript there exists a contextual variable named *arguments* that acts as a pseudo-array containing all the arguments passed into the function. Arguments isn't a true array (meaning that you can't modify it, or call .push() to add new items), but you can access items in the array, and it does have a .length property. There are two examples of this in Listing 2-5.

Listing 2-5. *Two Examples of Function Overloading in JavaScript*

```
// A simple function for sending a message
function sendMessage( msg, obj ) {
    // If both a message and an object are provided
    if ( arguments.length == 2 )
        // Send the message to the object
        obj.handleMsg( msg );

    // Otherwise, assume that only a message was provided
    else
        // So just display the default error message
        alert( msg );
}

// Call the function with one argument - displaying the message using an alert
sendMessage( "Hello, World!" );

// Otherwise, we can pass in our own object that handles
// a different way of displaying  information
sendMessage( "How are you?", {
    handleMsg: function( msg ) {
        alert( "This is a custom message: " + msg );
    }
});

// A function that takes any number of arguments and makes
// an array out of them
function makeArray() {
    // The temporary array
    var arr = [];

    // Go through each of the submitted arguments
    for ( var i = 0; i < arguments.length; i++ ) {
        arr.push( arguments[i] );
    }

    // Return the resulting array
    return arr;
}
```

Additionally, there exists another method for determining the number of arguments passed to a function. This particular method uses a little more trickiness to get the job done, however. We take advantage of the fact that any argument that isn't provided has a value of *undefined*. Listing 2-6 shows a simple function for displaying an error message and providing a default message if one is not provided.

Listing 2-6. *Displaying an Error Message and a Default Message*

```
function displayError( msg ) {
    // Check and make sure that msg is not undefined
    if ( typeof msg == 'undefined' ) {
        // If it is, set a default message
        msg = "An error occurred.";
    }

    // Display the message
    alert( msg );
}
```

The use of the typeof statement helps to lead us into the topic of type-checking. Since JavaScript is (currently) a dynamically typed language, this proves to be a very useful and important topic. There are a number of different ways to check the type of a variable; we're going to look at two that are particularly useful.

The first way of checking the type of an object is by using the obvious-sounding typeof operator. This utility gives us a string name representing the type of the contents of a variable. This would be the perfect solution except that for variables of type object or array, or a custom object such as user, it only returns object, making it hard to differentiate between all objects. An example of this method can be seen in Listing 2-7.

Listing 2-7. *Example of Using Typeof to Determine the Type of an Object*

```
// Check to see if our number is actually a string
if ( typeof num  == "string" )
    // If it is, then parse a number out of it
    num = parseInt( num );

// Check to see if our array is actually a string
if ( typeof arr == "string" )
    // If that's the case, make an array, splitting on commas
    arr = arr.split(",");
```

The second way of checking the type of an object is by referencing a property of all JavaScript objects called constructor. This property is a reference to the function used to originally construct this object. An example of this method can be seen in Listing 2-8.

Listing 2-8. *Example of Using the Constructor Property to Determine the Type of an Object*

```
// Check to see if our number is actually a string
if ( num.constructor == String )
    // If it is, then parse a number out of it
    num = parseInt( num );

// Check to see if our string is actually an array
if ( str.constructor == Array )
    // If that's the case, make a string by joining the array using commas
    str = str.join(',');
```

Table 2-1 shows the results of type-checking different object types using the two different methods that I've described. The first column in the table shows the object that we're trying to find the type of. The second column is the result of running typeof Variable (where *Variable* is the value contained in the first column). The result of everything in this column is a string. Finally, the third column shows the result of running Variable.constructor against the objects contained in the first column. The result of everything in this column is an object.

Table 2-1. *Type-Checking Variables*

Variable	typeof Variable	Variable.constructor
{ an: "object" }	object	Object
["an", "array"]	object	Array
function(){}	function	Function
"a string"	string	String
55	number	Number
true	boolean	Boolean
new User()	object	User

Using the information in Table 2-1 you can now build a generic function for doing type-checking within a function. As may be apparent by now, using a variable's constructor as an object-type reference is probably the most foolproof way of valid type-checking. Strict type-checking can help in instances where you want to make sure that exactly the right number of arguments of exactly the right type are being passed into your functions. We can see an example of this in action in Listing 2-9.

Listing 2-9. *A Function That Can Be Used to Strictly Maintain All the Arguments Passed into a Function*

```
// Strictly check a list of variable types against a list of arguments
function strict( types, args ) {

    // Make sure that the number of types and args matches
    if (  types.length != args.length ) {
```

```
        // If they do not, throw a useful exception
        throw "Invalid number of arguments. Expected " + types.length +
            ", received " + args.length + " instead.";
    }

    // Go through each of the arguments and check their types
    for ( var i = 0; i < args.length; i++ ) {
        //
        if ( args[i].constructor != types[i] ) {
            throw "Invalid argument type. Expected " + types[i].name +
                ", received " + args[i].constructor.name + " instead.";
        }
    }
}

// A simple function for printing out a list of users
function userList( prefix, num, users ) {
    // Make sure that the prefix is a string, num is a number,
    // and users is an array
    strict( [ String, Number, Array ], arguments );

    // Iterate up to 'num' users
    for ( var i = 0; i < num; i++ ) {
        // Displaying a message about each user
        print( prefix + ": " + users[i] );
    }
}
```

Type-checking variables and verifying the length of argument arrays are simple concepts at heart but can be used to provide complex methods that can adapt and provide a better experience to the developer and users of your code. Next, we're going to look at scope within JavaScript and how to better control it.

Scope

Scope is a tricky feature of JavaScript. All object-oriented programming languages have some form of scope; it just depends on what context a scope is kept within. In JavaScript, scope is kept within functions, but not within blocks (such as while, if, and for statements). The end result could be some code whose results are seemingly strange (if you're coming from a block-scoped language). Listing 2-10 shows an example of the implications of function-scoped code.

Listing 2-10. *Example of How the Variable Scope in JavaScript Works*

```
// Set a global variable, foo, equal to test
var foo = "test";
```

```
// Within an if block
if ( true ) {
    // Set foo equal to 'new test'
    // NOTE: This is still within the global scope!
    var foo = "new test";
}

// As we can see here, as foo is now equal to 'new test'
alert( foo == "new test" );

// Create a function that will modify the variable foo
function test() {
    var foo = "old test";
}

// However, when called, 'foo' remains within the scope
// of the function
test();

// Which is confirmed, as foo is still equal to 'new test'
alert( foo == "new test" );
```

You'll notice that in Listing 2-10, the variables are within the global scope. An interesting aspect of browser-based JavaScript is that all globally scoped variables are actually just properties of the window object. Though some old versions of Opera and Safari don't, it's generally a good rule of thumb to assume a browser behaves this way. Listing 2-11 shows an example of this global scoping occurring.

Listing 2-11. *Example of Global Scope in JavaScript and the Window Object*

```
// A globally-scoped variable, containing the string 'test'
var test = "test";

// You'll notice that our 'global' variable and the test
// property of the the window object are identical
alert( window.test == test );
```

Finally, let's see what happens when a variable declaration is misdefined. In Listing 2-12 a value is assigned to a variable (foo) within the scope of the test() function. However, nowhere in Listing 2-12 is the scope of the variable actually declared (using var foo). When the foo variable isn't explicitly defined, it will become defined globally, even though it is only used within the context of the function scope.

Listing 2-12. *Example of Implicit Globally Scoped Variable Declaration*

```
// A function in which the value of foo is set
function test() {
    foo = "test";
}

// Call the function to set the value of foo
test();

// We see that foo is now globally scoped
alert( window.foo == "test" );
```

As should be apparent by now, even though the scoping in JavaScript is not as strict as a block-scoped language, it is still quite powerful and featureful. Especially when combined with the concept of closures, discussed in the next section, JavaScript reveals itself as a powerful scripting language.

Closures

Closures are means through which inner functions can refer to the variables present in their outer enclosing function after their parent functions have already terminated. This particular topic can be very powerful and very complex. I highly recommend referring to the site mentioned at the end of this section, as it has some excellent information on the topic of closures.

Let's begin by looking at two simple examples of closures, shown in Listing 2-13.

Listing 2-13. *Two Examples of How Closures Can Improve the Clarity of Your Code*

```
// Find the element with an ID of 'main'
var obj = document.getElementById("main");

// Change it's border styling
obj.style.border = "1px solid red";

// Initialize a callback that will occur in one second
setTimeout(function(){
    // Which will hide the object
    obj.style.display = 'none';
}, 1000);

// A generic function for displaying a delayed alert message
function delayedAlert( msg, time ) {
    // Initialize an enclosed callback
    setTimeout(function(){
        // Which utilizes the msg passed in from the enclosing function
        alert( msg );
    }, time );
}
```

```
// Call the delayedAlert function with two arguments
delayedAlert( "Welcome!", 2000 );
```

The first function call to setTimeout shows a popular instance where new JavaScript developers have problems. It's not uncommon to see code like this in a new developer's program:

```
setTimeout("otherFunction()", 1000);

// or even…
setTimeout("otherFunction(" + num + "," + num2 + ")", 1000);
```

Using the concept of closures, it's entirely possible to circumnavigate this mess of code. The first example is simple; there is a setTimeout callback being called 1,000 milliseconds after when it's first called, but still referring to the obj variable (which is defined globally as the element with an ID of main). The second function defined, delayedAlert, shows a solution to the setTimeout mess that occurs, along with the ability to have closures within function scopes.

You should be able to find that when using simple closures such as these in your code, the clarity of what you're writing should increase instead of turning into a syntactical soup.

Let's look at a fun side effect of what's possible with closures. In some functional programming languages, there's the concept of currying. *Currying* is a way to, essentially, pre–fill in a number of arguments to a function, creating a new, simpler function. Listing 2-14 has a simple example of currying, creating a new function that pre–fills in an argument to another function.

Listing 2-14. *Example of Function Currying Using Closures*

```
// A function that generates a new function for adding numbers
function addGenerator( num ) {

    // Return a simple function for adding two numbers
    // with the first number borrowed from the generator
    return function( toAdd ) {
        return num + toAdd
    };

}

// addFive now contains a function that takes one argument,
// adds five to it, and returns the resulting number
var addFive = addGenerator( 5 );

// We can see here that the result of the addFive function is 9,
// when passed an argument of 4
alert( addFive( 4 ) == 9 );
```

There's another, common, JavaScript-coding problem that closures can solve. New JavaScript developers tend to accidentally leave a lot of extra variables sitting in the global

scope. This is generally considered to be bad practice, as those extra variables could quietly interfere with other libraries, causing confusing problems to occur. Using a self-executing, anonymous function you can essentially hide all normally global variables from being seen by other code, as shown in Listing 2-15.

Listing 2-15. *Example of Using Anonymous Functions to Hide Variables from the Global Scope*

```
// Create a new anonymous function, to use as a wrapper
(function(){
    // The variable that would, normally, be global
    var msg = "Thanks for visiting!";

    // Binding a new function to a global object
    window.onunload = function(){
        // Which uses the 'hidden' variable
        alert( msg );
    };

// Close off the anonymous function and execute it
})();
```

Finally, let's look at one problem that occurs when using closures. Remember that closures allow you to reference variables that exist within the parent function. However, it does not provide the value of the variable at the time it is created; it provides the last value of the variable within the parent function. The most common issue under which you'll see this occur is during a for loop. There is one variable being used as the iterator (e.g., *i*). Inside of the for loop, new functions are being created that utilize the closure to reference the iterator again. The problem is that by the time the new closured functions are called, they will reference the last value of the iterator (i.e., the last position in an array), not the value that you would expect. Listing 2-16 shows an example of using anonymous functions to induce scope, to create an instance where expected closure is possible.

Listing 2-16. *Example of Using Anonymous Functions to Induce the Scope Needed to Create Multiple Closure-Using Functions*

```
// An element with an ID of main
var obj = document.getElementById("main");

// An array of items to bind to
var items = [ "click", "keypress" ];

// Iterate through each of the items
for ( var i = 0; i < items.length; i++ ) {
    // Use a self-executed anonymous function to induce scope
    (function(){
        // Remember the value within this scope
        var item = items[i];
```

```
        // Bind a function to the elment
        obj[ "on" + item ] = function() {
            // item refers to a parent variable that has been successfully
            // scoped within the context of this for loop
            alert( "Thanks for your " + item );
        };
    })();
}
```

The concept of closures is not a simple one to grasp; it took me a lot of time and effort to truly wrap my mind around how powerful closures are. Luckily, there exists an excellent resource for explaining how closures work in JavaScript: "JavaScript Closures" by Jim Jey at `http://jibbering.com/faq/faq_notes/closures.html`.

Finally, we're going to look at the concept of *context*, which is the building block upon which much of JavaScript's object-oriented functionality is built.

Context

Within JavaScript your code will always have some form on context (an object within which it is operating). This is a common feature of other object-oriented languages too, but without the extreme in which JavaScript takes it.

The way context works is through the this variable. The this variable will always refer to the object that the code is currently inside of. Remember that global objects are actually properties of the window object. This means that even in a global context, the this variable will still refer to an object. Context can be a powerful tool and is an essential one for object-oriented code. Listing 2-17 shows some simple examples of context.

Listing 2-17. *Examples of Using Functions Within Context and Then Switching Its Context to Another Variable*

```
var obj = {
    yes: function(){
        // this == obj
        this.val = true;
    },
    no: function(){
        this.val = false;
    }
};

// We see that there is no val property in the 'obj' object
alert( obj.val == null );

// We run the yes function and it changes the val property
// associated with the 'obj' object
obj.yes();
alert( obj.val == true );
```

```
// However, we now point window.no to the obj.no method and run it
window.no = obj.no;
window.no();

// This results in the obj object staying the same (as the context was
// switched to the window object)
alert( obj.val == true );

// and window val property getting updated.
alert( window.val == false );
```

You may have noticed in Listing 2-17 when we switched the context of the obj.no method to the window variable the clunky code needed to switch the context of a function. Luckily, JavaScript provides a couple methods that make this process much easier to understand and implement. Listing 2-18 shows two different methods, call and apply, that can be used to achieve just that.

Listing 2-18. *Examples of Changing the Context of Functions*

```
// A simple function that sets the color style of its context
function changeColor( color ) {
    this.style.color = color;
}

// Calling it on the window object, which fails, since it doesn't
// have a style object
changeColor( "white" );

// Find the element with an ID of main
var main = document.getElementById("main");

// Set its color to black, using the call method
// The call method sets the context with the first argument
// and passes all the other arguments as arguments to the function
changeColor.call( main, "black" );

// A function that sets the color on  the body element
function setBodyColor() {
    // The apply method sets the context to the body element
    // with the first argument, the second argument is an array
    // of arguments that gets passed to the function
    changeColor.apply( document.body, arguments );
}

// Set the background color of the body to black
setBodyColor( "black" );
```

While the usefulness of context may not be immediately apparent, it will become more visible when we look at object-oriented JavaScript in the next section.

Object-Oriented Basics

The phrase *object-oriented JavaScript* is somewhat redundant, as the JavaScript language is completely object-oriented and is impossible to use otherwise. However, a common shortcoming of most new programmers (JavaScript programmers included) is to write their code functionally without any context or grouping. To fully understand how to write optimal JavaScript code, you must understand how JavaScript objects work, how they're different from other languages, and how to use that to your advantage.

In the rest of this chapter we will go through the basics of writing object-oriented code in JavaScript, and then in upcoming chapters look at the practicality of writing code this way.

Objects

Objects are the foundation of JavaScript. Virtually everything within the language is an object. Much of the power of the language is derived from this fact. At their most basic level, objects exist as a collection of properties, almost like a hash construct that you see in other languages. Listing 2-19 shows two basic examples of the creation of an object with a set of properties.

Listing 2-19. *Two Examples of Creating a Simple Object and Setting Properties*

```
// Creates a new Object object and stores it in 'obj'
var obj = new Object();

// Set some properties of the object to different values
obj.val = 5;
obj.click = function(){
    alert( "hello" );
};

// Here is some equivalent code, using the {…} shorthand
// along with key-value pairs for defining properties
var obj = {

    // Set the property names and values use key/value pairs
    val: 5,
    click: function(){
        alert( "hello" );
    }

};
```

In reality there isn't much more to objects than that. Where things get tricky, however, is in the creation of new objects, especially ones that inherit the properties of other objects.

Object Creation

Unlike most other object-oriented languages, JavaScript doesn't actually have a concept of classes. In most other object-oriented languages you would instantiate an instance of

a particular class, but that is not the case in JavaScript. In JavaScript, objects can create new objects, and objects can inherit from other objects. This whole concept is called *prototypal inheritance* and will be discussed more later in the "Public Methods" section.

Fundamentally, though, there still needs to be a way to create a new object, no matter what type of object scheme JavaScript uses. JavaScript makes it so that any function can also be instantiated as an object. In reality, it sounds a lot more confusing than it is. It's a lot like having a piece of dough (which is a raw object) that is molded using a cookie cutter (which is an object constructor, using an object's prototype).

Let's look at Listing 2-20 for an example of how this works.

Listing 2-20. *Creation and Usage of a Simple Object*

```
// A simple function which takes a name and saves
// it to the current context
function User( name ) {
    this.name = name;
}

// Create a new instance of that function, with the specified name
var me = new User( "My Name" );

// We can see that its name has been set as a property of itself
alert( me.name == "My Name" );

// And that it is an instance of the User object
alert( me.constructor == User );

// Now, since User() is just a function, what happens
// when we treat it as such?
User( "Test" );

// Since its 'this' context wasn't set, it defaults to the global 'window'
// object, meaning that window.name is equal to the name provided
alert( window.name == "Test" );
```

Listing 2-20 shows the use of the constructor property. This property exists on every object and will always point back to the function that created it. This way, you should be able to effectively duplicate the object, creating a new one of the same base class but not with the same properties. An example of this can be seen in Listing 2-21.

Listing 2-21. *An Example of Using the Constructor Property*

```
// Create a new, simple, User object
function User() {}

// Create a new User object
var me = new User();
```

```
// Also creates a new User object (from the
// constructor reference of  the first)
var you = new me.constructor();

// We can see that the constructors are, in fact, the same
alert( me.constructor == you.constructor );
```

Now that we know how to create simple objects, it's time to add on what makes objects so useful: contextual methods and properties.

Public Methods

Public methods are completely accessible by the end user within the context of the object. To achieve these public methods, which are available on every instance of a particular object, you need to learn about a property called *prototype*, which simply contains an object that will act as a base reference for all new copies of its parent object. Essentially, any property of the prototype will be available on every instance of that object. This creation/reference process gives us a cheap version of inheritance, which I discuss in Chapter 3.

Since an object prototype is just an object, you can attach new properties to them, just like any other object. Attaching new properties to a prototype will make them a part of every object instantiated from the original prototype, effectively making all the properties public (and accessible by all). Listing 2-22 shows an example of this.

Listing 2-22. *Example of an Object with Methods Attached Via the Prototype Object*

```
// Create a new User constructor
function User( name, age ){
    this.name = name;
    this.age = age;
}

// Add a new function to the object prototype
User.prototype.getName = function(){
    return this.name;
};

// And add another function to the prototype
// Notice that the context is going to be within
// the instantiated object
User.prototype.getAge = function(){
    return this.age;
};

// Instantiate a new User object
var user = new User( "Bob", 44 );
```

```
// We can see that the two methods we attached are with the
// object, with proper contexts
alert( user.getName() == "Bob" );
alert( user.getAge() == 44 );
```

Simple constructors and simple manipulation of the prototype object is as far as most JavaScript developers get when building new applications. In the rest of this section I'm going to explain a couple other techniques that you can use to get the most out of your object-oriented code.

Private Methods

Private methods and variables are only accessible to other private methods, private variables, and privileged methods (discussed in the next section). This is a way to define code that will only be accessible within the object itself, and not outside of it. This technique is based on the work of Douglas Crockford, whose web site provides numerous documents detailing how object-oriented JavaScript works and how it should be used:

- List of JavaScript articles: `http://javascript.crockford.com/`
- "Private Members in JavaScript" article: `http://javascript.crockford.com/private.html`

Let's now look at an example of how a private method could be used within an application, as shown in Listing 2-23.

Listing 2-23. *Example of a Private Method Only Usable by the Constructor Function*

```
// An Object constructor that represents a classroom
function Classroom( students, teacher ) {
    // A private method used for displaying all the students in the class
    function disp() {
        alert( this.names.join(", ") );
    }

    // Store the class data as public object properties
    this.students = students;
    this.teacher = teacher;

    // Call the private method to display the error
    disp();
}

// Create a new classroom object
var class = new Classroom( [ "John", "Bob" ], "Mr. Smith" );

// Fails, as disp is not a public property of the object
class.disp();
```

While simple, private methods and variables are important for keeping your code free of collisions while allowing greater control over what your users are able to see and use. Next, we're going to take a look at privileged methods, which are a combination of private and public methods that you can use in your objects.

Privileged Methods

Privileged methods is a term coined by Douglas Crockford to refer to methods that are able to view and manipulate private variables (within an object) while still being accessible to users as a public method. Listing 2-24 shows an example of using privileged methods.

Listing 2-24. *Example of Using Privileged Methods*

```
// Create a new User object constructor
function User( name, age ) {
    // Attempt to figure out the year that the user was born
    var year = (new Date()).getFullYear() - age;

    // Create a new Privileged method that has access to
    // the year variable, but is still publically available
    this.getYearBorn = function(){
        return year;
    };
}

// Create a new instance of the user object
var user = new User( "Bob", 44 );

// Verify that the year returned is correct
alert( user.getYearBorn() == 1962 );

// And notice that we're not able to access the private year
// property of the object

alert( user.year == null );
```

In essence, privileged methods are dynamically generated methods, because they're added to the object at runtime, rather than when the code is first compiled. While this technique is computationally more expensive than binding a simple method to the object prototype, it is also much more powerful and flexible. Listing 2-25 is an example of what can be accomplished using dynamically generated methods.

Listing 2-25. *Example of Dynamically Generated Methods That Are Created When a New Object Is Instantiated*

```
// Create a new user object that accepts an object of properties
function User( properties ) {
    // Iterate through the properties of the object, and make sure
    // that it's properly scoped (as discussed previously)
    for ( var i in properties ) { (function(){
        // Create a new getter for the property
        this[ "get" + i ] = function() {
            return properties[i];
        };

        // Create a new setter for the property
        this[ "set" + i ] = function(val) {
            properties[i] = val;
        };
    })(); }
}

// Create a new user object instance and pass in an object of
// properties to seed it with
var user = new User({
    name: "Bob",
    age: 44
});

// Just note that the name property does not exist, as it's private
// within the properties object
alert( user.name == null );

// However, we're able to access its value using the new getname()
// method, that was dynamically generated
alert( user.getname() == "Bob" );

// Finally, we can see that it's possible to set and get the age using
// the newly generated functions
user.setage( 22 );
alert( user.getage() == 22 );
```

The power of dynamically generated code cannot be understated. Being able to build code based on live variables is incredibly useful; it's what makes macros in other languages (such as Lisp) so powerful, but within the context of a modern programming language. Next we'll look at a method type that is useful purely for its organizational benefits.

Static Methods

The premise behind static methods is virtually identical to that of any other normal function. The primary difference, however, is that the functions exist as static properties of an object. As a property, they are not accessible within the context of an instance of that object; they are only available in the same context as the main object itself. For those familiar with traditional classlike inheritance, this is sort of like a static class method.

In reality, the only advantage to writing code this way is to keep object namespaces clean, a concept that I discuss more in Chapter 3. Listing 2-26 shows an example of a static method attached to an object.

Listing 2-26. *A Simple Example of a Static Method*

```
// A static method attached to the User object
User.cloneUser = function( user ) {
    // Create, and return, a new user
    return new User(
        // that is a clone of the other user object
        user.getName(),
        user.getAge()
    );
};
```

Static methods are the first methods that we've encountered whose purpose is purely organizationally related. This is an important segue to what we'll be discussing in the next chapter. A fundamental aspect of developing professional quality JavaScript is its ability to quickly, and quietly, interface with other pieces of code, while still being understandably accessible. This is an important goal to strive for, and one that we will look to achieve in the next chapter.

Summary

The importance of understanding the concepts outlined in this chapter cannot be understated. The first half of the chapter, giving you a good understanding of how the JavaScript language behaves and how it can be best used, is the starting point for fully grasping how to use JavaScript professionally. Simply understanding how objects act, references are handled, and scope is decided can unquestionably change how you write JavaScript code.

With the skill of knowledgeable JavaScript coding, the importance of writing clean object-oriented JavaScript code should become all the more apparent. In the second half of this chapter I covered how to go about writing a variety of object-oriented code to suit anyone coming from another programming language. It is this skill that much of modern JavaScript is based upon, giving you a substantial edge when developing new and innovative applications.