OO Techniques and JavaScript

Now that the basics of the language have been presented, it is time to dig into the creative side of JavaScript. If you are new to object-oriented (OO)programming languages, you can rest assured that you do not need to be a rocket scientist to use JavaScript. In fact, it is a great language to introduce OO programming, even though it isn't an OO language in a strict sense. If you come from a language such as C++ or Java, you will find JavaScript to be simplistic and flexible compared to what you are used to, but you can still apply most of what you already know. In this chapter, we'll look at:

- What objects are
- OOP techniques and JavaScript
- Object inheritance
- Adding protected data members to JavaScript

What are Objects?

Everything in JavaScript is an object or a primitive value that can be coerced into acting like an object (an example of primitive coercion is when we use `"STRING".toLowerCase()` to obtain `"string"`). Why did the creators of JavaScript make objects such an integral part of JavaScript? The quick answer is that manipulating objects in code is easier to understand, because it is closer to the way we see the world around us.

Look around the room you are in right now. You will most likely see at least a table, a chair, a light, and many other objects. You are definitely holding an object right now – this book. The human brain is geared to classify our surroundings into classifications. When we look at a kitchen chair and an office chair, we think of each as a chair, yet they have different shapes, different number of legs, one chair has wheels and armrests, and the other does not. We see the similarities and classify both types as a chair. We take our surroundings and develop associations of the objects we see around us. Objects bring this power to the world of programming.

We'll start with a short explanation of the theory behind object-orientated languages in general. Many of the terms used throughout this chapter will be defined within this section. If you are familiar with the theory, or if you wish to see how it applies to JavaScript, you can jump ahead to the section *OOP Techniques and JavaScript*.

Data Abstraction

One of the key concepts of OOP is data **abstraction**. Abstraction is the representation of a concept or information in a symbolic manner that can be manipulated. All programming languages use abstraction, but the difference is in what the language abstracts.

Assembly language abstracts the way a computer processor works. It is the programming language that is human readable for generating the computer binary instructions called machine language. This is a powerful abstraction in itself, as it allows the programmer to more readily understand and manipulate the fine details of every operation. The benefits of a low-level language like this are speed and control, but they also suffer from long development times and complexity.

High-level procedural programming languages, such as Basic and Pascal concentrate on the steps needed to solve a task. This style of programming vastly improved the development time over low-level programming languages. A disadvantage of procedural language, however, is that the program code describes the path to the solution, not the problem itself.

Along come OOP languages, which display the problem as objects with characteristics (called **properties**) and perform actions (called **methods**). These objects have interactions with other objects (such as an engine propelling a car) and associations with similar objects (a Mustang and a Celica are both cars). This type of abstraction allows the programmer to concentrate upon the problem, or subject, instead of concentrating upon the task.

Another benefit of using objects, is in the use of naming conventions. By giving the object a descriptive name, then the property names describe the object's characteristics and the methods describe its abilities. For example, if we have an object called `car` with a `color` property (`car.color`) and a `stop()` method (`car.stop()`), we can assume that the `color` property describes the color, and the `stop()`method makes the car stop.

Object Members

Objects are a unit of information together with the code for manipulating that information. Data is contained within properties. In JavaScript, this data can be any valid JavaScript value, including primitive values (strings and numbers) or an object reference. Code is contained within methods. Methods are functions which can use the `this` value in order to access other properties and methods of the object. An object's properties and methods are collectively referred to as the object's **data members**.

Interfaces

An **interface**, at its simplest level, is a mapping of member names (and any arguments that they may take) to their corresponding internal functions. It is a layer of code that separates an object from the rest of the computer environment, providing a means of I/O for communication. The major benefit behind using interfaces is **encapsulation**.

The many reasons for implementing interfaces for objects are beyond the scope of this book. For more information dealing with interfaces, any good OOP book should suffice.

JavaScript does NOT implement user-defined interfaces. Care should be taken by the reader to understand that when the term interface is being used in relation to JavaScript user-defined objects, it is referring to a pre-determined set of properties and methods defined for that object class, not for interfaces as defined within this section.

Extensibility

Extensibility is the ability to expand upon or add to something. If you go to buy a car, you are not simply buying any car. The make, model, and color, are all properties of a car and these can be used to distinguish between different cars of the same type. However, cars come in a variety of types. The concept of a car (you get in, start it, and drive wherever you want to go) is expanded to include a specific type of car (a Ford Mustang that is souped up and goes very fast). The ability to expand the concept of a car to a specific type of car without re-creating the whole concept is called extensibility.

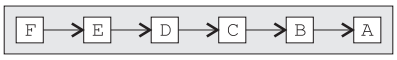
In order to implement extensibility in a flexible manner, we must be able to define and classify these concepts. A vehicle can be described as a mechanical means of transportation. An automobile is a vehicle that has four wheels, and is propelled by an engine. A Mustang is an automobile with a fast engine manufactured by Ford. Each of these concepts builds upon the concepts of the previous classification. In OO programming, this is called **inheritance**.

In order to use inheritance, we must have a way of defining the concepts that are to be inherited from. In the real world, the concepts are classifications; in OOP languages, we call them **classes** of objects. Every instance of an object has a class associated with it. This class describes what properties and methods the object has available to use and what, if any, class it inherits properties and methods from. There are two major advantages to extensibility: **code re-use** and **polymorphism**.

Class Hierarchy

There are a few terms that describe where in this inheritance chain a particular class belongs: **base** class, **derived** class, **child** class, **parent** class, **ancestor** class, and **descendant** class. What classes these refer to depend upon where in the inheritance chain, or inheritance hierarchy, the two classes being compared are.

In order to describe this hierarchy, let us assume that we have a chain of inheritance where class B inherits from class A, class C inherits from class B, and so on, to class F. The following can represent this:



A base class does not inherit from any other class. This is class A in the above diagram. A derived class is any class that inherits from another class. B, C, D, E, and F are derived classes.

In JavaScript, all native objects inherit from the Object class. The Object class is the base class for all built-in object and user-defined objects, and conversely, all built-in and user defined objects are derived from the Object class.

The terms child class and parent class describe the relationship between two classes in which one class directly inherits from the other. In the diagram above, class D inherits from class C, so D is a child of class C, and C the parent of class D. Just because class D inherits from class C, this does not mean that another class cannot directly inherit from class C, but it does mean that class D *only* directly inherits from class C. This is called a one-to-many relationship.

The terms ancestor class and descendant class relate to inheritance anywhere along the inheritance chain. Class F in the diagram has five ancestor classes, classes E, D, C, B and A. Conversely, A has five descendant classes: B, C, D, E, and F. Therefore, an ancestor class is the parent class, the parent of a parent class, and so on until we return to the base class. Conversely, a descendant class is any class that is derived either directly or indirectly from another class.

Overriding Data Members

What happens when an ancestor class has a property or method with the same name as one of its ancestor classes? When we request that property or method, which one is used? The answer is the lowest descendant that declared that member is used. This process of a descendant class property superseding an ancestor class property is called **overriding**.

In a true OO language, not all properties and methods can be overridden. When a member is defined within the class, it must be explicitly set so that it can be overridden. In JavaScript, *any* property or method can be overridden.

Code Reuse

Code re-use is not copying and pasting code from a previous program into a current program, and then modifying the code to suit new needs. Code re-use in this case is including a script into your current program, and then extending its functionality without modifying the original file. Creating a new class and having that class inherit from the previous class accomplishes this. From our car example, the new class would be the Mustang class, and the previous class would be the automobile class.

There are two advantages of code re-use, which are a reduction in development time, and a reduction in bugs. Since we are using a lot of previously developed code, we are speeding up our development time. Less time writing new code means less development costs. In addition, since we are writing less code, by using old code that already work, we will be introducing fewer bugs.

Polymorphism

An advantage of using inheritance is that many classes of objects can inherit from the same class. This allows us to perform certain actions on all of the descendant classes of this class. Under some circumstances, we can treat all of the descendant classes as if they were of a common class. We can treat both a Mustang and a Celica as an automobile, since they both have wheels, can be steered, and can accelerate and decelerate. The ability to treat a class as if it were one of its ancestor classes is called **polymorphism**.

This is a very powerful tool. Let us assume that we have created the class `automobile` and have a library of routines for manipulating `automobile` objects, such as routines on how to drive, clean, and maintain automobiles. Polymorphism allows us to keep using these same routines on any class of object derived from `automobile`, without having to modify our code. As you can see, we have fallen back into the previous advantage, code reuse.

Encapsulation

An OOP language reduces the amount needed for the end user to be able to properly implement the object class to just its exposed properties and methods Since all of the data and code is contained *inside* of the object, it is said to encapsulate its data and implementation.

Data Encapsulation

There are three basic ways of exposing data and functions (object members) to JavaScript code. There are **public members**, **private members**, and **protected members**:

- Public members are the properties and methods of the object. These members are exposed to all code that has access to the object itself.
- Private members are variables and functions contained within the class's implementation code (code defined within the constructor function in JavaScript).
- Protected members are variables and functions that are accessible to all of the ancestor and descendant classes' implementation code, but not to other code.

JavaScript natively supports public members, and has its own version of private members, but does not support protected members. Later in this chapter, we will introduce a method to add support for protected members.

> *In this text, the term "object implementation code", when applied to JavaScript, is referring to constructor function code. Any code that is defined within the constructor, including function definitions, is considered part of the object's implementation code.*

When we use private variables to hold all of the data for an object, and require all data access to be through the object's implementation code, we are encapsulating the data. This has two major benefits: **security** and **data integrity**. Data security simply means that the data is where we expect it to be. If the data was not encapsulated, it could be removed or deleted from the object by outside code, and the object's implementation code would not know where to find it. Data integrity, on the other hand, means making sure that the data is type checked and validated before storing. The chance of our code failing is reduced by making sure that the object's data is valid and always accessible through the correct channels, thus reducing bugs.

Implementation Hiding

Let us assume that we have created a class and have written the implementation code for it. We have distributed this code and a number of programs depend upon it. Time goes by, and we find that the code is unable to handle increased load demands or has a bug in it. It would be useful to be able to change our code without breaking the programs that depend upon it. In OO languages, we can readily do this.

The reason why it is easier to modify our code without worrying about breaking programs that depend upon it, is because the program is limited to the interface (properties and methods) that we expose for that object class. To the application, the object is a black box that we send information into and retrieve information from. How the object's code works is not important to the application, as long as it can find the properties and methods that it expects. In fact, the code is not even accessible to the program at all unless we make it so. This is called implementation hiding. Implementation hiding is not an integral part of JavaScript, since it does not implement formally defined interfaces, but the concepts of implementation hiding are just as valid as they are for other OO languages.

Why Are Objects so Important?

OO programming has the following major advantages over traditional programming languages:

- Reduced program development time and cost
- Ease of code reuse
- The code is easier to read, understand, maintain, and upgrade

JavaScript

Up until now, we have only been discussing the theory of OO languages, so now it is time to start applying this theory to JavaScript. In order to accomplish this, a few topics first need to be covered. These subjects include:

- JavaScript Objects
- Execution Contexts and Function Code
- Object Constructors
- Code Reuse

JavaScript Objects

JavaScript supports three types of objects: native, host, and user-defined. Native objects are objects supplied by the JavaScript language. Examples of these objects are `Object`, `Math`, `String`, and so on Host objects are supplied to JavaScript from the browser implementation, for interaction with the browser environment and the loaded document. Examples of these are `window`, `document`, `frames`, and so on User-defined objects are objects that you, the programmer, write implementation code for.

`Object`, with a capital "O", is a function that is used to create an object. Care should be taken when reading in that all object class names and constructor function names begin with a capital letter, and instances of an object all begin with a lower case letter. In most implementations host objects do not inherit from the `Object` class.

The Object Object

Object is the name of the base object class for all native objects, as well as the constructor function used to create an instance of the `Object` class. When we call the `Object` function as a constructor (`var x = new Object();`) without any arguments, it creates an object with the following properties:

- `constructor` – Contains a reference to constructor function used to create the object.
- `toString()` – A method that returns "`[object Class]`", where the substring `Class` is replaced by the class name of the object (its constructor function's name).
- `toLocaleString()` – A method that acts similarly as `toString()` above.
- `valueOf()` – A method that returns the primitive value of the object. If the object does not have a primitive value associated with it then this method returns a reference to itself.
- `hasOwnProperty(`*`propertyName`*`)` – A method that returns `true` if the object has a property with the name `propertyName`, and `false` if it does not. Example: if an instance of an object named `myObject` has a property `myProperty`, then `myObject.hasOwnProperty("myProperty")` would return true.
- `isPrototypeOf(`*`objectRef`*`)` – If `objectRef` is an instance of this object, then this method returns `true`, else returns `false`.
- `propertyIsEnumerable(propertyName)` – If this object has a property with the name `propertyName`, and that property is able to be enumerated in a `for...in` loop, then this function returns `true`, else returns `false`.

There is another way of creating an object of the `Object` class. You can do so using an object constant. There is no difference between the two objects being created below in the example. They have the same properties, are of the same class (`Object`), and can be used for the same purpose:

```
var obj1 = new Object();
obj1.prop1 = "This is property 1";
obj1.prop2 = 2;

var obj2 = {prop1: "this is property 1", prop2:2};
```

The `Object` constructor is a special constructor that can create objects of another class, depending on what is passed into the constructor as its only argument. If the argument is a primitive value, then an object is created that corresponds to that primitive value. The string primitive value corresponds to the `String` class, the number primitive value corresponds to the `Number` class, and so on. Finally, if the argument passed in is already an object, then a reference to that object is returned unchanged.

The built-in JavaScript constructors can be called for type conversion. The same is true for the `Object` constructor. Calling `Object` as a function, will convert whatever is passed in into an object of the appropriate type (if it is not already an object), as described in the above paragraph, or creates a new object if no arguments are passed in.

Functions Are Objects Too

Functions are the basic unit of callable code in JavaScript. As such, they are considered a datatype and have a class associated with them, the `Function` class. They are objects and have properties and methods.

There are three ways of creating a `Function` object: with a named function, with an anonymous function, or using the `Function` constructor:

```
function functionName(arg1, arg2, arg3, ..., argN)
{
   //function body goes here
}
var functionName = function(arg1, arg2, arg3, ..., argN)
{
   //function body goes here
}
var functionName = new Function("arg1Name", "arg2Name", ..., "argNName",
   "function body as a string");
```

The first method shown above is the most commonly used and the easiest to recognize, read, and understand. The second method is commonly used for creating a method for an object. This can be hard to read at times and should be used with care. The last method of is the one that most readily demonstrates that a function is an object, since a constructor is explicitly being called to create one. This is the least commonly used method, but a good application for this is for adding code that cannot be compiled in older browsers when it is detected that the browser does support the syntax needed. For more information on functions and the `Function` constructor, see Chapter 2.

Execution Contexts, Function Code, and Scope

There are three types of executable code, and their related execution contexts: Global code, Eval code, and Function code. Global code is any code that is outside of a function. Eval code is code passed in as a string to the `eval()` function (such as `eval("alert('this is an Eval code generated alert')");`). Function code is code that is in the body of a function. The term execution context is just a formal name for the process of setting the variable scope and the `this` value to reflect our current position in code.

There is only one global execution context, but whenever a function is called or the `eval()` function is called, a new function or `eval` execution context is created. Understanding how the execution context is created when calling a function, specifically when calling a function with the `new` keyword, is important for object creation.

Entering a Function Execution Context

Every function call enters a new execution context, which ends when the function returns or is terminated by an exception (error). The execution context can be thought of as resolving the scope chain and setting two objects to contain values depending upon where in the code we are: the variable's object, and the `this` value.

Every execution context has a variable object associated with it. We cannot access this object; you can only access the variables that it contains as its properties. When we enter a function execution context, the variable object is initialized to contain only the formally declared arguments of the function, and is destroyed when the function finishes executing. Variables are added to the variable object, while the function code is executed by using the `var` keyword. These are the private variables for the function, and are accessible to the code within the function block. Variables lifetimes may be extended by the use of nested functions as described later in the next section.

The scope chain can be thought of as an object that contains references to all of the variables and objects that the currently executing code can directly see and manipulate. When first entering the function, the scope contains all of the defined function arguments, the arguments array, the global variables and, if the function is nested within other functions, each of the parent functions variables. When the function code is being executed, any variable declared, is added to the scope. You can think of a function as a block of code that can see any variable declared within its block of code and any variable declared in any block of code that contains the function's declaration.

The `this` value (or object) corresponds to the object that the currently executing code is attached too. The `this` value is set to the value of the global object (in browsers the global object is the window object) except under the following conditions. If the function is a method of an object, the `this` value is set to the object. If the function is called using the `call()` or `apply()` methods of a function object, then the `this` value is determined by the first argument passed into these methods.

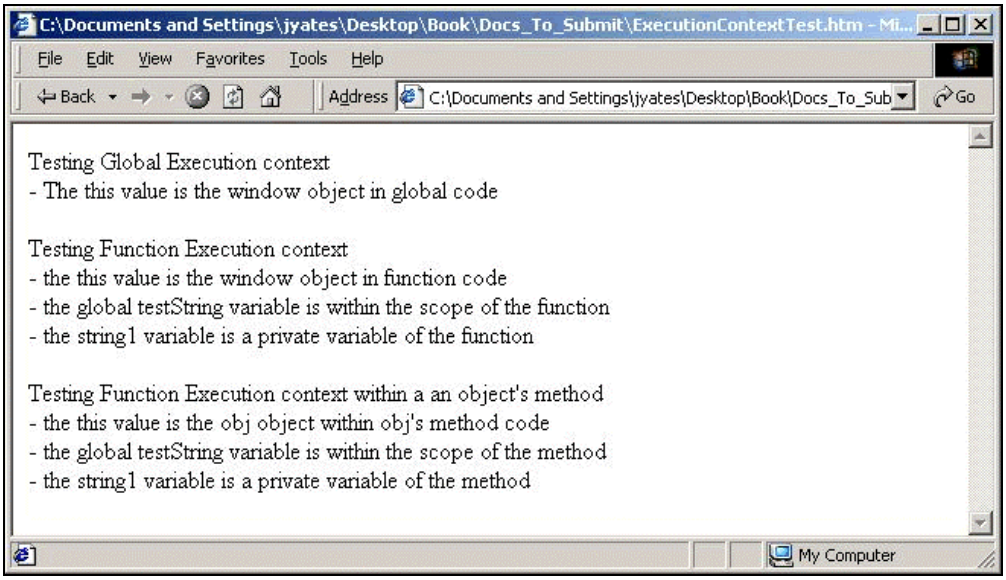
Finally, if the function is called using the `new` keyword, then the `this` value is a newly created object that inherits all of the properties of this functions prototype property. No matter where we are in the code, there is a specific `this` value associated with it.

Below is an example that shows the `this` value and variable scope in practice within JavaScript:

```
// Extract from ExecutionContextTest.htm
var testString = "this string is defined in the global code";
var string1 = "a global string";

document.write( "<P>Testing Global Execution context<BR>");
document.write( "- The this value is " );
if( this != window ) document.write( "NOT " );
document.write( "the window object in global code <\/P>" );

function testScope(){
   document.write( "<P>Testing Function Execution context<BR>");

   var string1 = "a private string";

   document.write( "- the this value is " );
   if( this != window ) document.write( "NOT " );
```

```
    document.write( "the window object in function code <br>" );

    document.write( "- the global testString variable is ");
    if( !testString ) document.write( "NOT ");
    document.write( "within the scope of the function<br>");

    document.write( "- the string1 variable is ");
    if( string1 != "a private string" ) document.write( "NOT ");
    document.write( "a private variable of the function<\/P>");
}
testScope();


function testMethod(){
    document.write( "<P>Testing Function Execution context within a an object's
method<BR>");

    var string1 = "a private string";

    document.write( "- the this value is " );
    if( this != obj ) document.write( "NOT " );
    document.write( " the obj object within obj's method code <br>" );

    document.write( "- the global testString variable is ");
    if( !testString ) document.write( "NOT ");
    document.write( "within the scope of the method<br>");

    document.write( "- the string1 variable is ");
    if( string1 != "a private string" ) document.write( "NOT ");
    document.write( "a private variable of the method<\/P>");
}
var obj = new Object();
obj.method = testMethod;
obj.method();
```

The output for the above code is as follows:



This example tests the value of the `this` value within three execution contexts by comparing it to the expected value of `this` following the steps outlined above. Within the global code and function code, it should be (and is) the window object, and from within an object's method code, it should be the object the method is attached too (and is). This example also illustrates that each of the three contexts have the `testString` global variable within their scope, but not the `string1` global variable. The reason why the function and the method do not have the `string1` variable within their scope is that they both declare a local variable with that name that supersedes it. For more information on variable scope, see Chapter 2.

The Power of Nested Functions

There are times when we will want to extend the lifetime of a function's private variables beyond the time when the function code terminates. Maybe the function calculates a value that is to be used later, but you want to restrict access to the value so that only private code will be able to access it. At this time, the reason is not important, but later on in this chapter, its importance will be made clear.

There are two methods of extending the lifetime of a function's variable. You can make it a property of a global object (such as window or the function itself), or you can use nested functions. Functions are objects of the `Function` class. They can be defined anywhere in code, and where they are defined determines their scope. If we define a function from *within* a function, the newly created function has access to the parent functions private variables.

Say we have two functions, one that performs a calculation on the arguments passed into it, and the other that recalls the results of the last calculation. This can be accomplished using nested functions with the benefit that there is no code that can modify the stored calculation, other than the two functions that we are creating. For example:

```
// Extract from NestedFunctions.htm
var doCalc, recallCalc;
function initCalculations(){
   var lastCalc;
   doCalc = function (x, y){
      lastCalc = x + y;
      return lastCalc;
   }
   recallCalc = function (){
      return lastCalc;
   }
}
initCalculations();

document.write( "<P>The sum of 3 + 7 is " + doCalc(3,7) + "</P>" );
document.write( "<P>The sum of 5 + 19 is " + doCalc(5,19) + "</P>" );
document.write( "<P>The last calculation had a result of " + recallCalc()
   + "</P>");
```

The output for this is as follows:

```
The sum of 3 + 7 is 10
The sum of 5 + 19 is 24
The last calculation had a result of 24
```

When the `initCalculations()` function is called, a local variable called `lastCalc` is created, and two functions are created and assigned to two global variables (`doCalc` and `recallCalc`). These two functions reference the `lastCalc` variable, thus keeping it from being destroyed when the `initCalculations()` function terminates. This can be seen when we call the `recallCalc()` function after calling the `doCalc()` function, which returns the value that `doCalc()` stored within it when last executed.

Notice that the only reason why the `initCalculations()` private variables are not destroyed is *only* because other code that they are within the scope of has not been terminated. Until the `doCalc()` and `recallCalc()` functions are destroyed (by deleting them or overwriting them), then the private variables will not be destroyed.

Object Constructors

Objects are constructed in code by calling a constructor function to create and initialize them. There are two types of constructor code: built-in constructors (those supplied by the implementation of JavaScript), and user-defined constructors.

Built-In Constructors

JavaScript comes with a several built in constructors. ECMAScript v3 defines a number of them, and individual implementations may define some additional ones. The objects that built-in constructors as defined by ECMAScript are `Object`, `Function`, `Array`, `String`, `Boolean`, `Number`, `Date`, `RegExp`, `Error` (with sub-classes of `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, and `TypeError`). For more information on how to use these constructors, refer to Chapter 2.

User-Defined Constructors

Just like native JavaScript objects, user-defined objects have constructor functions. The difference is that you, the author, write the constructor. The constructor can be very simple or complex depending upon your needs. The constructor will both define a class of object and initialize it. The next section will introduce the mechanics of writing your own object constructors, and the different techniques available.

Code Reuse

In JavaScript, there are two methods of code reuse (with shades of gray in between). Of the two, the copy-and-paste method is most widely used (or abused). The other method is that each object implementation is stored in its own file to be included in a web page with an external `<SCRIPT>` tag. Each method has its strengths and weaknesses.

The copy-and-paste method is very simple. We can add new code or modify the existing code to suit. The main advantage is that the web page loads faster since it does not have to retrieve any external files. If our web page has many external files, whether script, style, or image, then the time that the page takes to load (upon a first visit) can be greatly extended. When an external script file is parsed in an HTML page, some parsers wait until the script file is loaded, parsed, and executed before rendering the rest of the page, inducing a significant delay to its loading time.

On the flip side, the upkeep of a site will be more difficult. Suppose we have an OO navigation menu system script on our site. In order to add a menu item to the site menu, we will then have to edit every page on the site to include this menu item. You may be thinking about storing the whole menu in one file that all of the pages access. This is the middle ground between the two methods. We are still cutting and pasting into one file, but sharing the file between different pages, which is a good compromise. We get pages that load relatively fast and decrease the upkeep requirements to help keep our site current.

What happens though if we add a sub-web to our site that uses the same menu code, but has different menu entries? The answer to this is to split the code into two files. The first file contains the code for generating the menu, and the second file holds the information for making the entries to the first file. The site is still easy to update and the cost is minor; our page takes just a little longer to load.

Now we decide that we want to change the way the second menu looks, or how it functions. Should we go back to storing it all in one file? No, instead we create a new menu object that inherits from the first menu object. This new menu object only modifies the original menu as needed for the desired effect. Now we have three files that we are loading.

If we follow this to its extreme, we have the second method of code reuse, one file for every constructor. The cost is that our page loads slower, and the benefit is that we do not have the same code floating around in different files. So, which method should be used?

Firstly, if you combine all of the constructors that are most commonly used throughout our site into one file, commonly called a script library, then one slightly larger file takes less time to load than a few smaller files. In addition, once the script is downloaded, it is stored in the browser's cache and can be quickly retrieved the next time it is needed. You will need to use some judgment. If the script is only used on one page, keep it inside the page, not as an external file. If multiple pages use the script, store it within an external file. If most of your pages use it, then store it in a library script file, with all of the other scripts that your site depends on. Only you can decide which is best for your needs.

OOP Techniques and JavaScript

In this section, we will be covering the basics of OO programming techniques as they apply to JavaScript, specifically constructors, classes, and inheritance.

Object Constructors, Classes and Instances

We'll start by clarifying the terms instance, class, and constructor. Let's look at an example. You want to buy a car, so you go to a car dealer and pick out a model that you want, and the dealer orders it from the factory. When the car arrives, the dealer calls you to collect it. In this example, the make and model of car is the **class,** the factory that makes it is the **constructor**, and the specific car that you drive away in is the **instance**.

Every object that we use in JavaScript is a class instance. They can be manipulated, modified, and used to perform their designed task. Whenever we create an object, we are creating an object that is an instance of a class. Object classes in JavaScript are defined as the constructor function used to create the instance of that object, and can be referenced via its constructor property.

Simple Object Creation

Now that we know the difference between objects, classes, and constructors, it is time to create our own instance of a class. We can do this by writing a function to initialize the properties of the objects that will be created by this class. The simplest constructor possible is shown below. The constructor does nothing at all; it does not initialize any property values. The objects created by this constructor will only have the properties and methods of the `Object` class, since all objects in JavaScript inherit from this. The only use for a constructor like this is to define it as a class of object for type checking using the `instanceof` operator:

```
function MyClass()
{
}
```

In order to create an instance of `MyClass`, we would use code similar to:

```
var myObject = new MyClass();
```

When we execute this code, a number of steps will be taken by the JavaScript engine. When the `MyClass()` function is called with the `new` keyword, a new object is created as if we called the native `Object` constructor using the `new` keyword, and assigned it to the `this` value. Its constructor property is set to `MyClass` and the newly created object is linked to the `MyClass.prototype` properties (see Chapter 2 for a description of the `prototype` property, and *Execution Contexts, Function Code, and Scope* in this chapter for the `this` value). Then the code within the `MyClass()` function is executed, which in this case is none. When the function code completes, the newly created object is returned and assigned to the `myObject` variable.

Let's create a more useful object that shows how the constructor can be used to initialize the object's values:

```
function Car(){
   this.speed = 0;
   this.accelerate = Car_Accelerate;
   this.decelerate = Car_Decelerate;
   this.getSpeed = Car_GetSpeed;
   this.purchased = false;
}
function Car_Accelerate ( ){
   this.speed++;
}
function Car_Decelerate ( ){
   this.speed--;
}
function Car_GetSpeed ( ){
   return this.speed;
}

var myCar = new Car();
```

The example above shows one of the many ways that you can design a constructor function for your class. This constructor creates a `Car` class of object that contains two properties and three methods. The properties are `speed` (how fast the car is traveling), and `purchased` (Boolean value indicating whether the car has been bought). The three methods are `accelerate()` (increase the speed property), `decelerate()` (decrease the speed property), and `getSpeed()` (returns the speed the car is traveling). All of these properties and methods are attached to the newly created object via the use of the `this` value from within the constructor.

One problem with this code is that the three functions that are to be assigned to the property methods are within the global scope. If someone writes a function with the same name somewhere else in code, it is possible that the constructor will use the wrong function(s) for its object's methods. In order to prevent this, we will move the functions out of the global scope as follows:

```
function Car()
{
   this.speed = 0;
   this.accelerate = Accelerate;
   this.decelerate = Decelerate;
   this.getSpeed = function(){ return this.speed };
   this.purchased = false;
   return;

   function Accelerate(){
      this.speed++;
   }
```

```
      function Decelerate(){
         this.speed--;
      }
   }

   var myCar = new Car();
```

This updated example shows how the same constructor can be written using private functions and anonymous functions. The private functions are the `Accelerate()` and `Decelerate()` functions that are located within the constructor. These functions are not within the global scope, so are safe from other functions or variables overwriting them. The anonymous function is the one assigned to the `getSpeed()` method of the newly created object. This is accomplished as shown by declaring a function without a function name. The same effect could have been accomplished using the `Function` constructor.

Those familiar with JavaScript may wonder why the functions used as methods were defined within the constructor. The first reason is that code outside of the constructor cannot access the function directly. Since the functions are designed to be methods, if they are called directly as functions instead of methods of an object, the `this` value will be an incorrect value. Another reason is that when reading the code, it is easier to visually associate the function with the constructor, which can help prevent coding errors. A third reason is that the function names are not placed in the global scope, thus more than one function can have the same name, so long as they are defined within different scopes.

An interesting side effect of the way JavaScript compiles its code is that you can declare a function within another function and use that function before its declaration. Notice how the `Accelerate()` function is assigned to the `accelerate` property before it is defined. In fact, in the above example the constructor's code *always* terminates before the nested functions are defined, due to the unconditional `return` statement. The code works because when the outer function is compiled the nested functions are compiled as well, thus the `Accelerate()` and `Decelerate()` functions are created within the constructors scope. This has the added benefit of separating the initialization code, the code before the `return` statement, from the implementation code, that after the return statement.

These two methods of object construction work, and meet our needs, can be slow and use up a lot of memory. There is a faster and a less memory intensive way of creating objects. Every function has a property called the `prototype` property, where if you add a property or method to it, then any object created by that function (when called as a constructor) will have access to these same properties and methods. Thus, every object will "get" the property and method if you attach them to it. An example of this is as follows (see *Extending a Class via the Prototype Object* later in this chapter for more information on the prototype property):

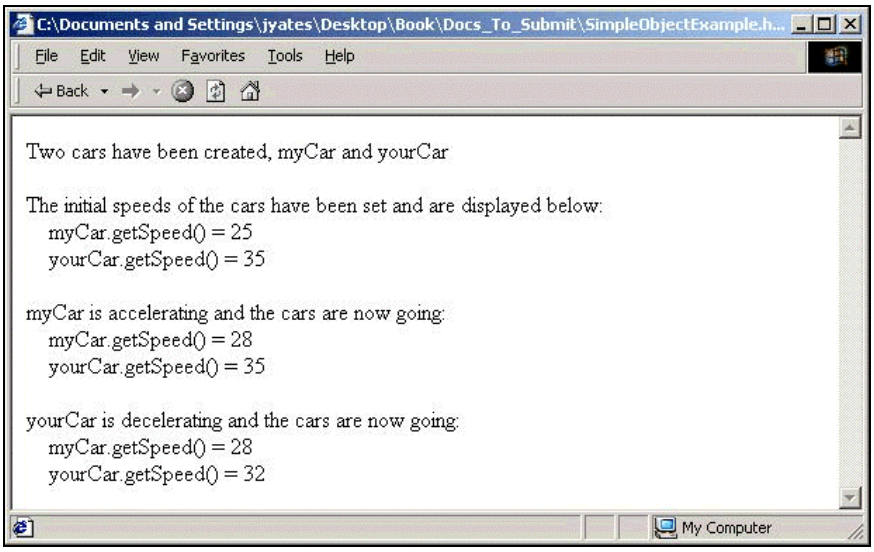
```
// Extract from SimpleObjectExample.htm
function Car() { };
Car.prototype.speed = 0;
Car.prototype.purchased = false;
Car.prototype.accelerate = function(){
   this.speed++;
}
Car.prototype.decelerate = function(){
   this.speed--;
}
Car.prototype.getSpeed = function(){
   return this.speed;
}

var myCar = new Car();
```

We have now seen three means of writing a constructor for creating the same type of object. It will receive the same properties and methods, and have the same functionality. The objects created by these three different constructors are almost indistinguishable for all practical purposes. Each of the three methods shown have their uses, which will become more apparent later on in this chapter, and you can also combine the different ways depending upon your needs.

The following example assumes that one of the above three versions of the `Car()` constructor has been defined, and shows how the objects created can be used:

```
var myCar = new Car();
var yourCar = new Car();

document.write( "<p>Two cars have been created, myCar and yourCar<\/p>");

myCar.speed = 25;
yourCar.speed = 35;

document.write( "<p>The initial speeds of the cars have been set" +
                " and are displayed below: <br>");
document.write("    myCar.getSpeed() = " +
                myCar.getSpeed() + "<BR>");
document.write("    yourCar.getSpeed() = " +
                yourCar.getSpeed()+ "<\/p>");

myCar.accelerate();
myCar.accelerate();
myCar.accelerate();

document.write("<p>myCar is accelerating and the cars are now going:<br>");
document.write("    myCar.getSpeed() = " +
               myCar.getSpeed() + "<BR>");
document.write("    yourCar.getSpeed() = " +
               yourCar.getSpeed()+ "<\/p>");

yourCar.decelerate();
yourCar.decelerate();
yourCar.decelerate();

document.write("<p>yourCar is decelerating and the cars are" +
               " now going:<br>");
document.write("    myCar.getSpeed() = " +
               myCar.getSpeed() + "<BR>");
document.write("    yourCar.getSpeed() = " +
               yourCar.getSpeed()+ "<\/p>");
```

When the code is run, you will get the following output:



This code creates two objects of the `Car` class, `myCar` and `yourCar`. `myCar` is initialized with a speed of 25, and `yourCar` with a speed of 35. After displaying the speeds of the cars, `myCar` is then accelerated three times, and the speeds are displayed again. `yourCar` is decelerated three times and the speeds again displayed. This example shows that the instances of the `Car` class are independent of each other. When one car accelerates or decelerates, the other instances are not affected.

Note that all the properties and methods used so far are public members. There is no way to enforce the design on their actual use in other code. If the programmer wishes to, they could change the `speed` property to `fast`, although this means that afterwards, the `accelerate()` and `decelerate()` methods would not work properly.

Creating Objects That Implement Private Members

So far, the objects that we have created implement all data storage within public members (the properties of the object), which is how most JavaScript programmers store data. It is easy to get to and to modify. It also requires them to have intimate knowledge of how the object works, how it performs its calculations, and the types of data that it understands to be stored within its properties. This is perfectly acceptable if we are writing constructors for our own immediate use. However, the situation changes if the code is to be used by others (say, as part of a site library of objects), or if we expect there to be large gaps in times between our using the object and when we wrote the code for it, thus loosing the "feel" of the object.

The solution for this is to implement private members for data storage. The advantage here is the ability to protect your data from malicious code by preventing casual modification of the data by limiting its modification to public methods. These public methods can perform data typing and other validation upon the value before storing it within your data structure. This same method can throw an error if it is passed invalid data, thus aiding in the programmer in troubleshooting their code.

JavaScript does not have formal private members for objects, but it does have something that works just as well for our uses, private variables, and private functions. Remember that a function's private variables are not destroyed after the function terminates execution, as long as a nested function from within it still exists (say, as a method of an object). This enables us to be able to use the private variables to hold data that is accessible via the use of nested functions attached to the object as methods. It must be noted here that this precludes the programmer from attaching any of the methods that access the "private members" to the constructor's prototype property, but must instead attach the method from within the constructor's code.

The following constructor shows the how to implement private members (both a function and variable) with the use of public members attached within the constructor:

```
function MyClass()
{
   var privVar = "a private member";

   //the following methods are public members
   //that use private members
   this.setValue = function( value ){ return SetValue(value) };
   this.getValue = function (){ return privVar; }
   return;

   function SetValue(aNewValue)  //a private member
   {
      if (Object(aNewValue) instanceof String)
      {
         privVar = aNewValue;
      }
      return privVar;
   }
}

var myObject = new MyClass();
```

Here there are two public methods, `setValue()` and `getValue()`, which set and retrieve the value of the private variable `privVar`. `getValue()` directly accesses `privVar` and returns its value, while `setValue()` calls the private function `SetValue()` to perform data validation before the new value is placed in the `privVar` variable.

Each object created has its own copy of each private variable. This allows an object's methods to share information, but prevent access to these variables to code outside of the object's implementation code. Here, the value being set is limited to string values (either primitive or object) only. If nested functions were not used, then the value would have to be a property of the object and any code with access to the object could modify the property, thus bypassing this type checking. This is an example of implementing data encapsulation.

> **Make sure to document any methods that are dependent upon each other to prevent yourself or another author from overriding the methods; thus breaking the logic of your code. In the previous example, if the `setValue()` method is overridden, you have no means of modifying the `privVar` variable; thus breaking the logic of the `getValue()` method.**

Let's implement the `Car` class of objects using a private member for the speed of the car and rerun the acceleration and deceleration tests as we did before:

```
// Extract from, PrivateMemberExample.htm
function Car(initialSpeed){
   var speed = 0;
   if( !isNaN(Number(initialSpeed)) )
      speed = Number(initialSpeed);

   this.accelerate = function( ){ speed++; }
   this.decelerate = function( ){ speed--; }
   this.getSpeed = function( ){ return speed; }
}
```

```
Car.prototype.purchased = false;

var myCar = new Car(25);
var yourCar = new Car(35);

document.write( "<P>Two cars have been created, myCar and yourCar</P>");
document.write( "<P>The initial speeds of the cars have been set " +
                "and are displayed below: <BR>");
document.write("    myCar.getSpeed() = " +
               myCar.getSpeed() + "<BR>");
document.write("    yourCar.getSpeed() = " +
              yourCar.getSpeed()+ "</P>");

myCar.accelerate();
myCar.accelerate();
myCar.accelerate();

document.write("<P>myCar is accelerating and the cars are now going:<BR>");
document.write("    myCar.getSpeed() = " +
               myCar.getSpeed() + "<BR>");
document.write("    yourCar.getSpeed() = " +
               yourCar.getSpeed()+ "</P>");

yourCar.decelerate();
yourCar.decelerate();
yourCar.decelerate();

document.write("<P>yourCar is decelerating and the cars are " +
               "now going:<BR>");
document.write("    myCar.getSpeed() = " +
               myCar.getSpeed() + "<BR>");
document.write("    yourCar.getSpeed() = " +
               yourCar.getSpeed()+ "</P>");
```

The only major difference for using the `Car` constructor is that you are to now pass in the initial speed of the car for when you create it. The external code now does not have access to the speed data, thus it must be set from within the constructor. The only other way that you could explicitly set the speed would be to create a new method for doing so (since the accelerate and decelerate only modify the current value, not to set it to a specific value). After creating the two objects, and setting their initial speed during creation, we then run the same test on the objects. When you run this code, you will find that you will get the exact same output as you did in the previous example, because the same logic is being used, just two different means of implementing it.

JavaScript is designed to be a co-operative programming environment. This means that the author of a script is expected to understand the restrictions imposed by code upon arguments to functions and methods, as well as those for properties of the objects. *The language does not impose these requirements.* The method for data encapsulation presented here requires the programmer to send all values to a method of the object and to use (possibly) another method to retrieve it. This method is available for critical operations but, overall, we should just document a property's requirements and depend upon the programmer to only set the property's value to one that is valid.

Making Sure Constructors Always Create Objects

Nothing. A constructor *is* a function. A constructor can be called as a function by *not* using the `new` keyword. This is a bug that can, at times, be hard to find. In order to prevent this, we need a method of guaranteeing that a constructor always creates a new object. We can accomplish this by adding one line of code that detects if the `this` value within the constructor is an instance of the constructor's class using the `instanceof` operator. If it is not an instance of the constructor's class, then the constructor needs to be called using the `new` keyword, and the newly created object is returned:

```
function MyClass(arg1, arg2, arg3)
{
   if (! (this instanceof MyClass)) return new MyClass(arg1, arg2, arg3);
   //constructor code goes here
}

var myObject = MyClass();
```

The above code is not cross-browser compatible. Only use the `instanceof` operator when you are sure that the end user is running JavaScript version 1.4+ or JScript 5.5+ (Netscape 6+ or IE 5.5+). There are programming techniques that can be used for testing the browser to see what version it is, and running different code as necessary. The following example shows how the above code can be made backward compatible, but with a loss in functionality. This code assumes that a global variable called ECMA3 has been defined to true if the browser is IE5.5+ or Netscape 6+, false otherwise:

```
function MyClass(arg1, arg2, arg3)
{
   if (ECMA3 && !eval("this instanceof MyClass"))
      return new MyClass(arg1, arg2, arg3);
   //constructor code goes here
}

var myObject = MyClass();
```

This will run in older browsers as well as the newer browsers, but the `eval()` function is very slow to use since the string supplied must be compiled. The `eval()` function is necessary in this instance though, because without it, the whole script will not run due to a compiler error, because the older browser will not recognize the `instanceof` operator.

This covers the basics for creating user-defined objects. Next, we get into more complex object creation.

Object Inheritance

Recall that an object inherits all of the properties and methods of the `Object` class. The mechanism of this inheritance is the object's internal (not accessible) `prototype` property, which is set by its constructor's `prototype` property (which is accessible). The internal `prototype` property is an object, it also has an internal `prototype` property, whose members are inherited. This regression of prototypes continues until we get to the prototype of the `Object` class, the base class for all user-defined objects. The series of prototypes that an object inherits from is called the **prototype chain**.

You may have received the impression that when an object inherits a property it receives a copy of that property or in some way a separate value all of its own. This is not the case. The inherited property is just added to the lookup path (the prototype chain) for when the code requests the value of a property.

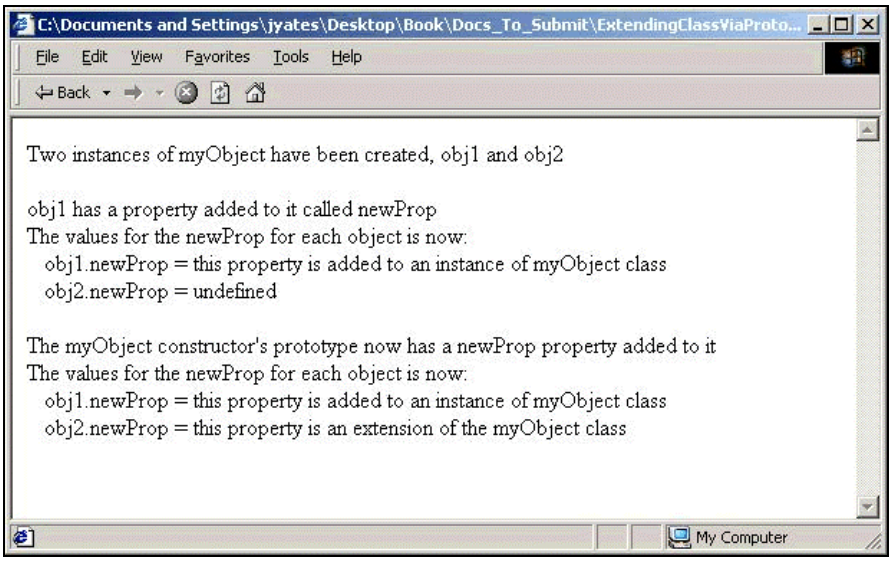
Extending a Class via the Prototype Object

When the value of a property is requested, the code first looks to the object to see if it has a property of that name that has been explicitly set (not inherited). If it does not, it then follows the prototype chain until it finds a property with that name and returns that value. If it does not find a property with the specified name, then the value returned for that property is `undefined`. Due to this mechanism, we can add a property to a constructor's prototype object and *all* objects derived from that constructor's prototype immediately inherit the added property, as long as a descendant class does not contain a property of that name.

Extending an object via its constructor's prototype can be very powerful. The following example shows how an object automatically acquires a property using this technique:

```
// Extract from ExtendingClassViaPrototypeExample.htm
function myObject()
{
   //this is just a bare constructor for this example
}
var obj1 = new myObject();
var obj2 = new myObject();

document.write("<P>Two instances of myObject have been "
   + "created, obj1 and obj2</P>");

// Test the adding of a property to an instance of the class

obj1.newProp = "This property is added to an instance of myObject class";

document.write("<P>obj1 has a property added to it called newProp<BR>");
document.write("The values for the newProp for each object is now:<BR>");
document.write("   obj1.newProp = "
                + obj1.newProp + "<BR>");
document.write("   obj2.newProp = "
                 + obj2.newProp + "</P>");

//Test the adding of a property to the class prototype

myObject.prototype.newProp =
   "This property is an extension of the myObject class";

document.write("<P>The myObject constructor's prototype now has "
               + "a newProp property added to it<br>");
document.write("The values for the newProp for each object is now:<BR>");
document.write("   obj1.newProp = "
               + obj1.newProp + "<BR>");
document.write("   obj2.newProp = "
               + obj2.newProp + "</P>");
```

This example creates two instances of an object and adds a property to one of them. After displaying the value of that property for both instances, the code then adds a property to the class prototype with the same name, and displays the value of the property for both objects. The output of this code follows:



This illustrates two aspects of extending a class via its prototype object. Firstly, the property is immediately added to the prototype chain for all instances of the class. Secondly, if the property is overridden for a particular instance, then the overridden value is returned, not the value just added.

This technique is very useful for correcting a browser's implementation of a method. For example, Netscape's JavaScript 1.2 implemention of the `Array.push()` method is incorrectly per the ECMA standard. Chapter 19, *Extending JavaScript Objects*, shows numerous examples of extending classes via their prototype property. The next section shows how to add the `call()` and `apply()` methods to the `Function` object for older browsers, since they are used extensively in this chapter.

Upgrading Native Objects for Older Browsers

JavaScript 1.3 (Netscape Navigator 4.06+) introduced two very useful methods to the `Function` object, which are used frequently in this chapter: the `call()` and `apply()` methods, which are both used to call the function.

These two methods were including in the ECMAScript v3 standard, and Microsoft included them in Internet Explorer 5.5 (Jscript 5.5). They are very important to the implementation of inheritance as presented in this chapter. A means of providing these methods to older versions of JavaScript is presented below through the example of extending a class of objects via properties of the prototype of the `Function` constructor:

```
//Patch the apply method of the Function class of objects if needed
function _Apply_( thisObj, argArray )
{
   var str, i, len, retValue;
   if (thisObj + "" == "null" || thisObj + "" == "undefined"
      thisObj = window;
   if (argArray + "" == "null" || argArray + "" == "undefined")
      argArray = new Array();
   //make sure that the property getting the method is unique
   var index = 0;
   while( thisObj["temp" + index] + "" != "undefined" ) index++;

   thisObj["temp" + index] = this;
```

```
      str = "thisObj.temp" + index + "(";
      len = argArray.length;
      for(i=0; i<len; i++ )
      {
         str += "argArray[" + i + "]";
         if (i + 1 < len) str += ", ";
      }
      str += ");";
      retValue = eval(str);
      thisObj["temp" + index] = undefined;

      return retValue;
   }
   if (!Function.prototype.apply) Function.prototype.apply = _Apply_;

   //Patch the call method of the Function class of objects if needed
   function _Call_( thisObj, arg1, arg2, argN )
   {
      return this.apply(thisObj, Array.apply(null, arguments).slice(1));
   }
   if (!Function.prototype.call) Function.prototype.call = _Call_;

   //patch the undefined value for compatablility for older browsers
   var undefined;
```

The `apply()` method takes two arguments: an object reference to be used as the `this` value for the function call (`thisObj`) and an array containing the values to be passed to the function as its arguments (`argArray`). It creates a function call on the fly, as if the function were a method of the `thisObj` object. If the `thisObj` argument is `null` or `undefined`, then the `this` value for the function call is the global object (in browsers, the the `window` object). If the `argArray` argument is not provided (is `null` or `undefined`), then no arguments are passed to the function.

Creating the function (more specifically a method) call on the fly takes two steps. The first step is to find a property that is not being used. The following line of code accomplishes this:

```
while( thisObj["temp" + index] + "" != "undefined" ) index++;
```

Notice how the return value of the property is converted to a string by adding a null string to it. This is so that it can be tested against "`undefined`", the string value of what is returned by a property request for a property that has not yet been defined. If the property is not `undefined`, then `index` is incremented, and the test is performed again. This is repeated until an unused property is found, at which point the `this` value (a `Function` object) is assigned to that property; making it a method of the object.

The next step is to construct the method call, using the `eval()` function, because we do not know how many arguments to pass into the method in advance. The following lines of code accomplish this:

```
str = "thisObj.temp" + index + "(";
len = argArray.length;
for( i=0; i<len; i++ )
{
   str += "argArray[" + i + "]";
   if (i + 1 < len) str += ", ";
}
str += ");";
retValue = eval( str );
```

If `argArray` had three elements, and the `index` variable contains 3, then the `str` variable being constructed would contain the following code:

```
"thisObj.temp3(argArray[0], argArray[1], argArray[2]);"
```

This is the code string to be evaluated by the `eval()` function, which is the method call generated on the fly. The last step, before terminating the method, is to reset the value of the property back to `undefined`. Since some browsers do not define a global variable or property of `undefined`, we define it explicitly. using `var undefined`.

The `call()` method is very similar to `apply()`, so in the above code, `call()` uses `apply()` for its implementation. This method takes one or more arguments: the first argument is an object reference to be used as the `this` value for the function call, while the rest of the arguments are passed to the function as arguments for the function call.

Note how the arguments are passed to `apply()` from within this method. All of the arguments passed into `call()` are to be passed as the second argument to the apply method with the exception of the first argument, `thisObj`. A copy of the arguments array needs to be created, minus the first argument. The easiest way of doing this is to use the `Array.slice()` method, but this cannot be done directly, since the arguments array is not a native `Array` class of object. To get around this, we call the `Array`, using the already defined `apply()` method, to create an array that contains all of the elements within the arguments array, and then perform the `slice()` method.

Below are the two lines of code that attach the methods to the `Function` object:

```
if (!Function.prototype.apply)
   Function.prototype.apply = _Apply_;
...
if (!Function.prototype.call)
   Function.prototype.call = _Call_;
```

They first test to see if the `Function` object already has the method. If it does not, then the method is attached. The code above will work properly in version 4 and greater browsers (IE4+ and NN4+) and we recommend their inclusion in all OO scripts presented in this chapter.

Inheriting from an Instance

The simplest method of implementing class inheritance in JavaScript is to set the derived class constructor's prototype property to an instance of the parent class. The prototype will then contain all of the properties and methods of the parent class, because it is an instance of that parent class. The benefit of using this method is that the `instanceof` operator will return `true` for both the object class and all of its ancestor classes.

The following code shows inheriting via instance in a three-class inheritance chain. The `Mustang` class inherits from the `Ford` class, which inherits from the `Car` class. Each of these three constructors implements only public members in this example:

```
// Extract from InheritFromInstancePublicMembers.htm
function Car(initSpeed) {
   this.speed = 0;
   this.accelerate = Accelerate;
   this.decelerate = Decelerate;
   this.getSpeed = function(){ return this.speed };
   this.purchased = false;
```

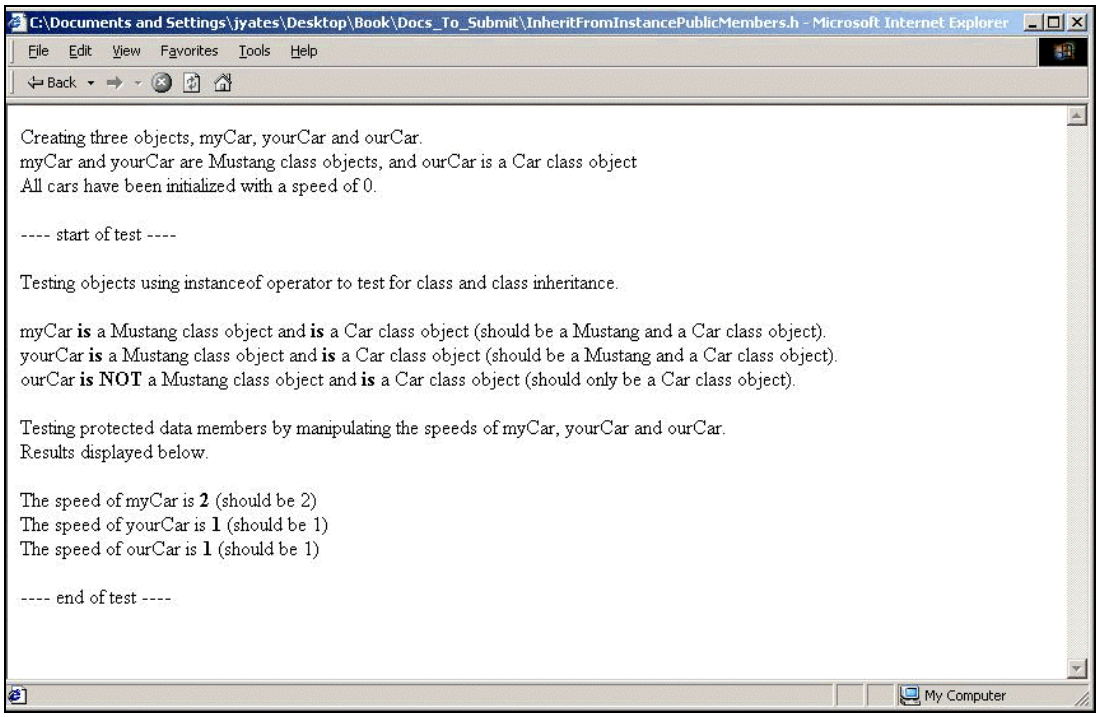
```
      return;

      function Accelerate ( ){
         this.speed++;
      }
      function Decelerate ( ) {
         this.speed--;
      }
   }

   function Ford(initSpeed){
      this.make = "Ford";
   }
   Ford.prototype = new Car();

   function Mustang(initSpeed){
      this.year = 2001;
      this.doors = 2;
   }

   Mustang.prototype = new Ford();

   var myCar = new Mustang( );
   var yourCar = new Mustang( );
   var ourCar = new Car( );
```

Note that the `Car` constructor presented here is the same that was introduced earlier in this chapter. This constructor, and the other two presented here, will be used throughout the rest of the chapter to show the different inheritance techniques with modifications in order to properly discuss each of the techniques. Similarly, the test code below will also be used throughout the chapter. The only part of this code that will be modified from example to example will be the portion of the code that displays the expected values returned by the `getSpeed()` method of the objects, and thus will not be repeated throughout this chapter:

```
   // Further extract from InheritFromInstancePublicMembers.htm
   //Note: ECMA3 is true if the browser is IE5.5+ or Netscape 6+

   document.write("<p>Creating three objects, myCar, yourCar "
      + "and ourCar.<br>");
   document.write("myCar and yourCar are Mustang class objects, and ourCar "
      + "is a Car class object<br>");
   document.write("All cars have been initialized with a speed of 0.<\/p>");

   //start of test
   document.write("<P>---- start of test ----</P>");

   //display info about the objects created
   document.write("<P>Testing objects using instanceof operator to test "
      + "for class and class inheritance.</P>");
   if( ECMA3 )
   {
      document.write( "<P>myCar <B>"
         + (eval("myCar instanceof Mustang")?"is":"is NOT")
         + "</B> a Mustang class object and <B>"
         + (eval("myCar instanceof Car")?"is":"is NOT")
         + "</B> a Car class object "
         + "(should be a Mustang and a Car class object).<BR>" );
```

```
      document.write("yourCar <B>"
         + (eval("yourCar instanceof Mustang")?"is":"is NOT")
         + "</B> a Mustang class object and <B>"
         + (eval("yourCar instanceof Car")?"is":"is NOT")
         + "</B> a Car class object "
         + "(should be a Mustang and a Car class object).<BR>" );
      document.write( "    ourCar <B>"
         + (eval("ourCar instanceof Mustang")?"is":"is NOT")
         + "</B> a Mustang class object and <B>"
         + (eval("ourCar instanceof Car")?"is":"is NOT")
         + "</B> a Car class object "
         + "(should only be a Car class object).</P>" );
   }
   else
   {
      document.write( "<P>   your browser does not support "
      + "the instanceof operator</P>");
   }

   //change car speeds (manipulate the objects)
   myCar.accelerate();
   myCar.accelerate();
   yourCar.accelerate();
   ourCar.accelerate();

   document.write("<P>Testing protected data members by manipulating "
      + "the speeds of myCar, yourCar and ourCar.<BR>");
   document.write("Results displayed below.</P>");

   //display the results of the object manipulation
   document.write("<P>The speed of myCar is <B>"
      + myCar.getSpeed() + "</B> (should be 2)<BR>");

   document.write("The speed of yourCar is <B>"
      + yourCar.getSpeed() + "</B> (should be 1)<BR>");

   document.write("The speed of ourCar is <B>"
      + ourCar.getSpeed() + "</B> (should be 1)</P>");

   //end of test
   document.write("<P>---- end of test ----</P>");
```

When the `InheritFromInstancePublicMembers.htm` file is viewed in a browser, you will see the following output:



In this example, the `Mustang` class inherits from the `Ford` class by setting the `Mustang`'s prototype property to an instance of the `Ford` class. The `Ford` class inherits from the `Car` class by setting the `Ford`'s prototype property to an instance of the `Car` class. Because `Ford` inherits from `Car`, `Mustang` also inherits from `Car`. This is inheriting by object instance. Three objects are created: `myCar`, `yourCar`, and `ourCar`. `myCar` is a `Mustang`, as is `yourCar`, but `ourCar` is just of `Car` class. These three objects are tested for proper inheritance and the class implementation works properly.

As seen by the output above, `myCar` and `yourCar` are both a `Mustang` and a `Car` (which implies that they are both `Ford` class as well even though they were not tested as such). This first test is just to make sure that the internal prototype chain is verified, and that the `instanceof` operator works as expected.

The second test is to call their `accelerate()` method to show that the inherited methods are working properly. In the output the result of the method call, is displayed in bold, and the expected value follows within the parenthesis. Again, the output above shows that the results of this operation came back as expected.

Inheriting from an instance of a class requires only one line of code to be added per constructor that is inheriting from another class:

```
function Ford( initSpeed ){
   this.make = "Ford";
}
Ford.prototype = new Car();

function Mustang( initSpeed ){
   this.year = 2001;
   this.doors = 2;
}
Mustang.prototype = new Ford();
```

Most JavaScript programmers use this method of implementing inheritance, but there are requirements that must be met in order to use this technique. Specifically, all of the properties and methods of each of the inherited classes *must* be public. If any of the inherited classes use private methods that depend upon each other, such as a pair of set and get methods, then this means of inheritance will fail.

Let's look at an example. Note that this code is provided as an example showing that inheriting from an instance does not work when the class being inherited implements private members. This code will not work as expected, and it should not be expected to:

```
// Extract from InheritFromInstancePrivateMembers.htm
function Car()
{
   var speed = 0;
   this.accelerate = function (){ speed++ };
   this.decelerate = function (){ speed-- };
   this.getSpeed = function (){ return speed };
   this.purchased = false;
}
```

Here we can see that inheriting from an instance fails when one of the classes being inherited implements private members. The output indicates that the speed of `myCar` is 3 when it should be 2, and `yourCar`'s speed is also 3 when it should be 1. This is because both of these objects are "sharing" the same instance of a `Car` class object within their prototype chain. The `accelerate()` method is called twice for `myCar`, and once for `yourCar`, for 3 calls that would set an instance of `Car` to have a speed of 3.

Whenever any instance of a `Ford` class (as well as `Mustang`, since it inherits from it) accelerates, *all* instances of the `Ford` class will accelerate. This not what we want, as we assume that each instance is independent of the others. In the above example, the only object that produced the expected output is `ourCar`, which is not an instance of a `Ford` class, and so is independent of that class.

There are two solutions for this problem. We can make all the members (properties and methods) of the classes public, or we can implement inheriting via class constructors, as described in the next section.

Inheriting from a Class Constructor

The above style of inheriting is fine for most applications, but not when one of the ancestor classes implements data hiding. The problem arises when we create more than one object that derives from a class implementing data hiding, since all instances of the class inherit from the *same* object instance. In the previous example, if we accelerate one car, we accelerate all cars. This is not the intended effect, so the code must be modified so that each instance has its own copy of the ancestors class's properties. The following code expands upon inheriting from an object instance to include inheriting from the class constructor as well:

```
// Extract from InheritFromConstructor.htm - with highlighted changes
function Ford(){
   Car.call(this);
   this.make = "Ford";
}
Ford.prototype = new Car();

function Mustang(){
   Ford.call( this );
   this.year = 2001;
   this.doors = 2;
}
...
```

Now when this modified code is run, we get the expected output. When `myCar` accelerates, `yourCar` does not. The key is that from within the `Ford` constructor, the `Car` constructor is called using the constructor function's `call()` method in order to re-initialize the private variables and the methods that access them. The same technique is used for the `Mustang` constructor. We must still set the prototype properties to an instance of the parent class (in order to keep the prototype chain intact), so that the `instanceof` operator operates properly.

> **Always call the parent class constructor before adding any properties to the object being constructed. If you do not then the parent class constructor may overwrite your property.**

This technique does have its disadvantages though: speed and memory consumption. Every call to the constructor causes a call to every one of its ancestor constructors. This can cause a dramatic increase in the amount of code run per object instance. In addition, each instance receives its own copy of each property and method, instead of inheriting them via the prototype property, which will increase the amount of memory used. In most cases, the speed and memory consumption difference is minor, but when a large number of objects are created, it may become significant.

Inheriting from a Class Constructor – Private Members and Overriding

In our examples so far, we have not overridden any of the private member access methods. The reason for this is that if any single method that uses the private data member is overridden, then all of the methods that access that data member must be overridden. Suppose we want our Mustang (a sports car) to accelerate faster than a normal car. To achieve this, we need to override the `accelerate()` method of the `Car` class within the `Mustang` classIn the following code, the `speed` is increased by 3 every time `accelerate()`is called, instead of by one:

```
/* Extract from InheritFromConstructorWithOverride.htm with highlighted
   changes */
function Mustang( initSpeed )
{
   var speed;
   Ford.call(this);
   this.year = 2001;
   this.doors = 2;
   this.accelerate = function (){ speed += 3; };
}
Mustang.prototype = new Ford();
```

Here we would expect the speed of `myCar` and `yourCar` to be 6 and 3 respectively. However, if you run this code, you'll see that they are both 0. This is because their `accelerate()` method is modifying the private `speed` variable within the `Mustang` constructor, but the `getSpeed()` method, which has not been overridden, is retrieving the value of the private `speed` variable from within the `Car` constructor. These two different variables have two different values.

There are two solutions to this dilemma. One solution is to do away with private data members and revert to using public data members (using only the properties and methods of the object for storing data), and so lose control over the value of our date. The other solution is to implement protected data members, which we'll look at next.

So far, we have covered how JavaScript implements objects, constructors, classes, inheritance, private variables, and public properties. The techniques you've learned should be sufficient for most of your user-defined object needs. The next section of this chapter is for those who wish to learn how to push JavaScript to its limits and see how much we can get away with by adding protected data member support for our class constructors.

Adding Protected Data Members to JavaScript

Within an object's code, JavaScript supports two types of data members, public and private. Public data members are the object's properties and methods. Any code that has access to the object can publicly access these data members. Private data members are also supported in the form of variables and functions defined within the object's constructor function, and are only accessible to methods defined within the constructor as nested functions.

In other OOP languages, there is another datatype, protected data members, which are variables and functions shared between a class implementation code and its derived class's implementation code. In JavaScript, this correlates to variables of one constructor being made accessible to another constructor, where one of the constructors inherits from the other. In the previous example, the `Car` class has three methods for manipulating the speed of the car. If any one of the methods is to be overridden, then they all must be overridden to prevent the other methods from being broken. If the private `speed` variable was made into a protected variable though, then each individual method could be overridden by different constructors, while retaining their functionality. Therefore, the use of protected data members can be used in order to overcome the limitations of using private members.

Since JavaScript does not directly support protected members, a mechanism must be created for doing so. There are two obstacles to be overcome when sharing a variable. Variables cannot be passed by reference – they are passed by value. This means that if code in one scope changes the value of the variable, code within another scope does not see this change. This problem can be overcome by storing the variables as properties of a private object. The other obstacle is security. A solution must be devised that will allow a constructor to retrieve the variables, but not outside of the code. Overcoming the limitations of JavaScript in order to use protected data members may add complexity to our code, and should be limited to only those variables that absolutely need to be shared between constructor codes.

In this section, we will cover a method of extending JavaScript to include protected data members, so that class implementation code can privately share information between them without exposing said information to outside code.

In this section, we will learn:

- What the protected method is
- How to use the protected method
- What the protected method does
- The internal workings of the protected method

The Protected Method

The goal of implementing protected members is to create a means for constructors to pass around a private object that contains all of the protected data members (the container object). One method of doing this is to extend the `Object` class to provide a means of sharing the private member container object.

It was discovered during the development of the protected method that any class implementing protected members needed to implement inheritance from a class constructor as well. Due to this, as well as timing issues, the protected member was designed to incorporate inheritance from a class constructor within itself. This has the added benefit of adding one line of code to our constructors, while at the same time eliminating one, with a net change of no added lines of code to our constructors. This can be a very attractive solution, since it does not add much more complexity to the code.

The syntax for implementing the protected method within a constructor is as follows:

```
var protect=this._protected_([ParentClass[, arg1, arg2, argN]]);
```

All tokens in the above should be replaced by names of your own choice. They are just placeholders for this description. All arguments within the square brackets are optional. A description of each part of the syntax follows:

- The *protect* variable is the variable that your constructor will use as it's private member container object.
- The *_protected_* method of the `this` value is the `protect()` method inherited from the `Object` class that this section is presenting.
- The *ParentClass* argument is a reference to the parent class constructor. This is the same constructor that we would normally use the `call()` method on when implementing inheritance from a class constructor; as was presented earlier in this chapter. If the current constructor does not inherit from another user-defined class, then the protected method should be called without any arguments.
- The optional arguments *arg1*, *arg2*, and *argN* represent the arguments to be sent to the parent class constructor from within the protected method.

The return value of the protected method is an object for your constructor to attach as properties all of its protected data members. All constructors of the current object will receive a reference to the same object, which means that any property you add to it will be available to all of the objects constructors.

For better understanding of how to use the protected method from within your constructor, the following section provides templates which you should follow when designing your constructors.

Constructor Templates for Implementing Protected Members

Below are the two templates to use for your constructors when implementing the protected members. Everything in italics is to be replaced by token names (function, variable, and argument names) of your own choice. The number and name of each argument is left to the discretion of the reader, depending upon the needs of your code.

This template, the `AncientClass` template that follows, is for use when the class constructor does NOT inherit from another user-defined class. Calling the protected method in this type of constructor is done in order to enable private members only:

```
// The below function is a template for a constructor for a class
// that DOES NOT inherit from another class other than the Object class.
function AncientClass(arg1, arg2, argN)
{
   var protect = this._protected_();

   /*Note: All protected variables and functions should be
            attached to the protect object */

   ////////////////////////////////////////////////
   // TODO:  constructor code goes here
}
```

The following template, the `DescendantClass` template, is for use when the class constructor inherits from another user-defined class. Calling the protected method in this type of constructor happens in order to enable private members and to implement inheritance from class constructors:

```
// The below function is a template for a constructor for a class
// that DOES inherit from another class.
function DescendantClass(arg1, arg2, argN)
{
   var protect = this._protected_(ParentClass, arg1, arg2, argN);

   /* Note:  All protected variables and functions should be
             attached to the protect object */

   ////////////////////////////////////////////////
   // TODO:  constructor code goes here
}
DescendantClass.prototype = new ParentClass();
```

By studying these two templates, you will see that protected members are implemented by using only one line of code per constructor:

```
var protect = this._protected_();
```

The above line is for the `AncientClass` constructor, and the following line:

```
var protect = this._protected_( ParentClass, arg1, arg2, argN );
```

is for the `DescendantClass` constructor.

For a better understanding of the implementation of the protected method, an example is presented below. This example modifies the example presented in the *Inheriting from a Class Constructor – Private Members and Overriding* section that did not work properly. This example corrects the prior example by implementing private members. The code used for implementing protected members is presented in bold:

```
// Extract from ProtectedExample.htm

//Car class constructor
function Car(initSpeed)
{
   var protect = this._protected_();
   protect.speed = 0;

   if(!isNaN(Number(initSpeed)))
      protect.speed = Number(initSpeed);

   this.accelerate = function() {protect.speed++};
   this.decelerate = function() {protect.speed--};
   this.getSpeed = function() {return protect.speed};
   this.purchased = false;
}

//Ford class constructor (extends Car class)
function Ford(initSpeed)
{
   var protect = this._protected_(Car, initSpeed);
   this.make = "Ford";
};
Ford.prototype = new Car();

//Mustang class constructor (extends Ford class)
```

```
function Mustang(initSpeed)
{
   var protect = this._protected_(Ford, initSpeed);
   this.year = 2001;
   this.doors = 2;
   this.accelerate = function() {protect.speed += 3};
}
Mustang.prototype = new Ford();

var myCar = new Mustang();
var yourCar = new Mustang();
var ourCar = new Car();
```

The `Mustang` class is overriding the `Car` class's `accelerate()` method with a method that increases the protected `speed` variable by 3. The original `accelerate()` method as implemented by the `Car` class, only increases it by 1.

The example creates three objects, two of which are of the `Mustang` class (`myCar` and `yourCar`). The code then calls `myCar`'s `accelerate()` method twice, which should give it a speed of 6, and calls `yourCar`'s `accelerate()` method once, which should give it a speed of 3. From viewing the output of the example, we can see that the expected values are returned, therefore protected data member access methods are shown to be overridable.

Source Code for the Protected Method

Now that we know to use the protected method, it is time to dig into the code behind it. The protected member attempts to perform three actions:

- Provide a means of security, so that the protected method can only be called from within an object's constructor
- Provide a means of creating and sharing an object to be used as the protected data member container.
- Provide support for inheriting from a class constructor

Below is the implementation of protected data members that was developed:

```
// Extract from ProtectedExample.htm

//extend the Object class to proved protected data members
function _Protected_(ParentClass, ParentClassConstructorArguments)
{
   /*make sure this function is called from within the objects
     constructor */
   if (ECMA3 && _Protected_.caller &&
       !eval("this instanceof _Protected_.caller"))
      eval('throw new Error("Protected Data Members can only be shared '
         + 'between related classes.");');
   if (ParentClass)
   {
      //the base class has to be pre-defined as a base class of this
      //object
      if (ECMA3 && !eval("this instanceof ParentClass"))
         eval('throw new Error("The base class constructor supplied'
            + ' was not pre-defined as the base class for this class.");');
      //reset the _protectedMembers_ property. It may have been modified.
```

```
        this._protectedMembers_ = Object.prototype._protectedMembers_;

        //re-initialize the base object of this class for this instance
        ParentClass.apply( this, Array.apply(null, arguments).slice(1));
     }
     var protect;
     var ACCESS_KEY = "Type here whatever you want to be the access key";
     function GetProtectedMembers(key)
     {
        var returnValue=this;
        if (key == ACCESS_KEY)
           returnValue=protected
        return returnValue;
     }
     protect = this._protectedMembers_(ACCESS_KEY);
     if (protect == this)
     {
        protect = new Object();
        this._protectedMembers_ = GetProtectedMembers;
     }
     return protect;
  }
  Object.prototype._protected_ = _Protected_;
  Object.prototype._protectedMembers_ = function(){return this};
```

The above code was designed for backward compatablility. The `ECMA3` variable was supplied by a browser detection script, which sets this variable to true if the browser supports the `instanceof` operator and the `throw...catch` statement.

This section contains a line-by-line description of the protected method whose source is provided above:

```
function _Protected_(ParentClass, ParentClassConstructorArguments)
```

This is the function declaration. There are two formally declared arguments, `ParentClass` and `ParentClassConstructorArguments`. `ParentClass` is an optionally supplied constructor reference to use when inheriting via an object constructor, to re-initialize the parent class properties of this object. If the constructor inherits, then this argument should always be supplied. `ParentClassConstructorArguments` is not a true argument, since it is not used within the code. Instead, it is a reminder to the programmer that any arguments the parent class requires for its constructor should be individually included there. This means that this function can be called with zero, one, or many arguments:

```
if (ECMA3 && _Protected_.caller &&
    !eval("this instanceof _Protected_.caller"))
   eval('throw new Error("Protected Data Members can only be shared '
      + 'between related classes.");');
```

Here is the first level of security for protecting our protected variables from external code. This code checks to make sure that the function only works when it is called from within one of the object's constructors.

The first check ensures that this line of code will run in the browser by testing the ECMA3 variable. The ECMA3 variable is supplied by a browser detection script discussed earlier in this chapter, which sets this variable to `true` if the browser supports the `instanceof` operator and the `throw...catch` statement. If this variable is `false`, that security is disabled.

If `ECMA3` is `true`, then the caller property of the function is tested to see if it is a constructor of the object. The `caller` property of a function holds a reference to the function that called it. If the calling function is not a constructor of the current object (tested using the `instanceof` operator), then an exception is thrown, stopping the execution of the code.

In order for this implementation to work properly, there must be a mechanism for calling the parent class constructors and to allow them to initialize the object before the current constructor does its initialization. When the function is called with a `ParentClass` argument, the following lines perform this action:

```
if(ParentClass)
{
   //the base class has to be pre-defined as a base class of this
   //object
   if (ECMA3 && !eval("this instanceof ParentClass"))
      eval('throw new Error("The base class constructor supplied'
         + ' was not pre-defined as the base class for this class.");');

   //reset the _protectedMembers_ property. It may have been modified.
   this._protectedMembers_ = Object.prototype._protectedMembers_;

   //re-initialize the base object of this class for this instance
   ParentClass.apply( this, Array.apply(null, arguments).slice(1));
}
```

Calling the `_Protected_` function with an invalid `ParentClass` argument can cause bugs in your code that are hard to trace. The above lines of code are designed to help you prevent this from happening, and to give you an idea of where the problem lies.

Part of the implementation of the protected method is expanding the `Object` class to have a method called `_protectedMembers_`, which returns a reference to the object to which it is attached. This method will be overridden later on in the code, but at this point in the code, it *must* be set to the original function. The following line of code guarantees this:

```
this._protectedMembers_ = Object.prototype._protectedMembers_;
```

This following line is for implementing inheritance via constructors. The parent class is called using the `apply` method of the constructor to give it its chance for initialization of the object:

```
ParentClass.apply(this, Array.apply(null, arguments).slice(1));
```

This code is not backward compliant due to the `apply()` method being used, but this can be overcome by implementing the `apply()` method yourself, using the `eval()` function. An example of this was provided earlier in the chapter in section: "Upgrading Native Objects for Older Browsers".

The next variable declared is the `ACCESS_KEY` variable.

```
var ACCESS_KEY = "Type in whatever you want here to be the access key";
```

This variable holds the security key (a string that you can modify to whatever you wish) in order to limit the retrieval of the protected variable container to calls from within this function. This security can be easily bypassed by anyone understanding this code and is intended as a deterrent, not as a true security measure:

```
function GetProtectedMembers(key)
{
   var returnValue=this;
   if (key == ACCESS_KEY)
      returnValue=protect;
   return returnValue;
}
```

The `GetProtectedMembers()` nested function will become a method of the object being created. Its purpose is to pass the protected variable container between the constructors. This function references the variable `protect` that is in the `_Protected_` function, but since this function has only made a method of the object once, it only references *one* instance of this variable. This is the key to the functionality of protected members. The rest of this code is just to support these four lines:

```
protect = this._protectedMembers_(ACCESS_KEY);
```

This line of code retrieves the protected variable container. If this is the first time this code has been executed on the current object, then a reference to the current object will be returned:

```
if (protect == this)
{
   protect = new Object();
   this._protectedMembers_ = GetProtectedMembers;
}
```

If this is the first time this code has been run on the current object, then a new object is initialized to act as the protected members container. After a new container is made, a means of retrieving this container is implemented by overriding the original `_protectedMembers_` method (the one that returns a reference to the object) with the `GetProtectedMembers` function. This is only done once per object instance:

```
return protect;
```

Return the protected members container to the constructor:

```
Object.prototype._protected_ = _Protected_;
```

This line of code enables protected members for any object that wishes to implement it:

```
Object.prototype._protectedMembers_ = function(){ return this };
```

This line of code sets the `_protectedMembers_` method to its initial value, a function that returns a reference to its object.

The key to understanding this code is to realize that each constructor for each class for the object is calling this code with a reference to the next constructor to be called. The only constructor that does not pass in a constructor to be called is the lower-most user-defined class (the class that directly extends the `Object` class), and here is where the chain of function calls end. When the lower-most user defined class constructor calls this code, the protected member container is created and initialized. Then, in reverse order from the above, each constructor retrieves this container and keeps a reference to it for its own use.

This concludes the discussion of protected members. We have seen that there are three types of data used by object code

- Public data is stored as a property of the object itself and can be modified without restriction by any code that can get access to the object.
- Private data is stored as a private variable within the object constructor and can only be accessed via methods. If any of these methods are to be overridden, then all of the methods must be overridden.
- Protected data is stored in a container object that only the constructors of the object and their methods have access. Unlike using private data members, using protected data members eliminates the need for overriding all of the methods that access the data when only one of the methods are overridden. Which method we use to store and retrieve our object's data is dependent upon our needs when writing object code.

Summary

Here is a short review the features that most OO languages support:

- Interfaces – This is a cross language method for creating and accessing object members at run-time (a technique known as late binding). JavaScript does *not* support user-defined interfaces.
- Data Abstraction – The representation of information in a symbolic manner
- Extensibility – The ability to expand upon previously developed code
- Polymorphism – The ability to use and manipulate an object with only a partial understanding of the properties and methods that it supports
- Encapsulation – This is the ability to group data and code within a structure or entity. This is also referred to as information hiding.

JavaScript supports all of the above features, with the exception of interfaces. Interfaces allow objects created in one language to be accessible to programs written in almost any other language. This is a very powerful feature, but the nature of JavaScript as a loosely typed scripting language limits how useful this feature could be.

Data abstraction means concentrating upon the problem being solved, instead of the steps it takes to solve the problem. Creating an entity called an object and populating it with data, and methods for manipulating the data, help to accomplish this. In JavaScript, an object is seen as a unit of data and code for manipulating that data, thus data abstraction is a fundamental part of the JavaScript language.

Extensibility is the ability to readily expand upon existing code. In OOP languages, this is accomplished through inheritance. JavaScript natively implements a type of inheritance called prototype inheritance, or object-based inheritance. In this chapter, we covered the weaknesses of prototype inheritance and ways to overcome these weaknesses.

Polymorphism is the ability to use or manipulate an object as if it were an object of one of its ancestor classes. In JavaScript, all objects inherit from the base object class, `Object`. In this sense, we can say that JavaScript supports polymorphism to an extreme. With the addition of the `instanceof` operator, which allows the programmer to perform type checking, a more realistic application of polymorphism can be realized. Therefore, the answer to "does JavaScript support polymorphism" is yes, but the complete implementation of polymorphism requires the programmers to test and/or reject the object manually.

Encapsulation is the ability to contain data and code within an entity. Natively, JavaScript supports a form of encapsulation that we refer to as co-operative encapsulation. All of the object data and functionality is available for inspection and modification. This data and functionality is found in **public members**. Data encapsulation is the isolation or protection of data from casual modification from code external to the object code (information hiding). In JavaScript, this is not natively supported. With a little work by the programmer, using nested functions, we can use private and/or protected members to implement a more realistic form of data encapsulation. In conclusion, JavaScript does support a primitive form of encapsulation, and with a little work by the programmer, it can fully support all of the major forms of encapsulation.

In conclusion, we can say that although JavaScript is not a true OO language, it has, or can be made to act as though it has most of the features that identify an OO language. JavaScript is often referred to as an object-based language, because of the features it is lacking.