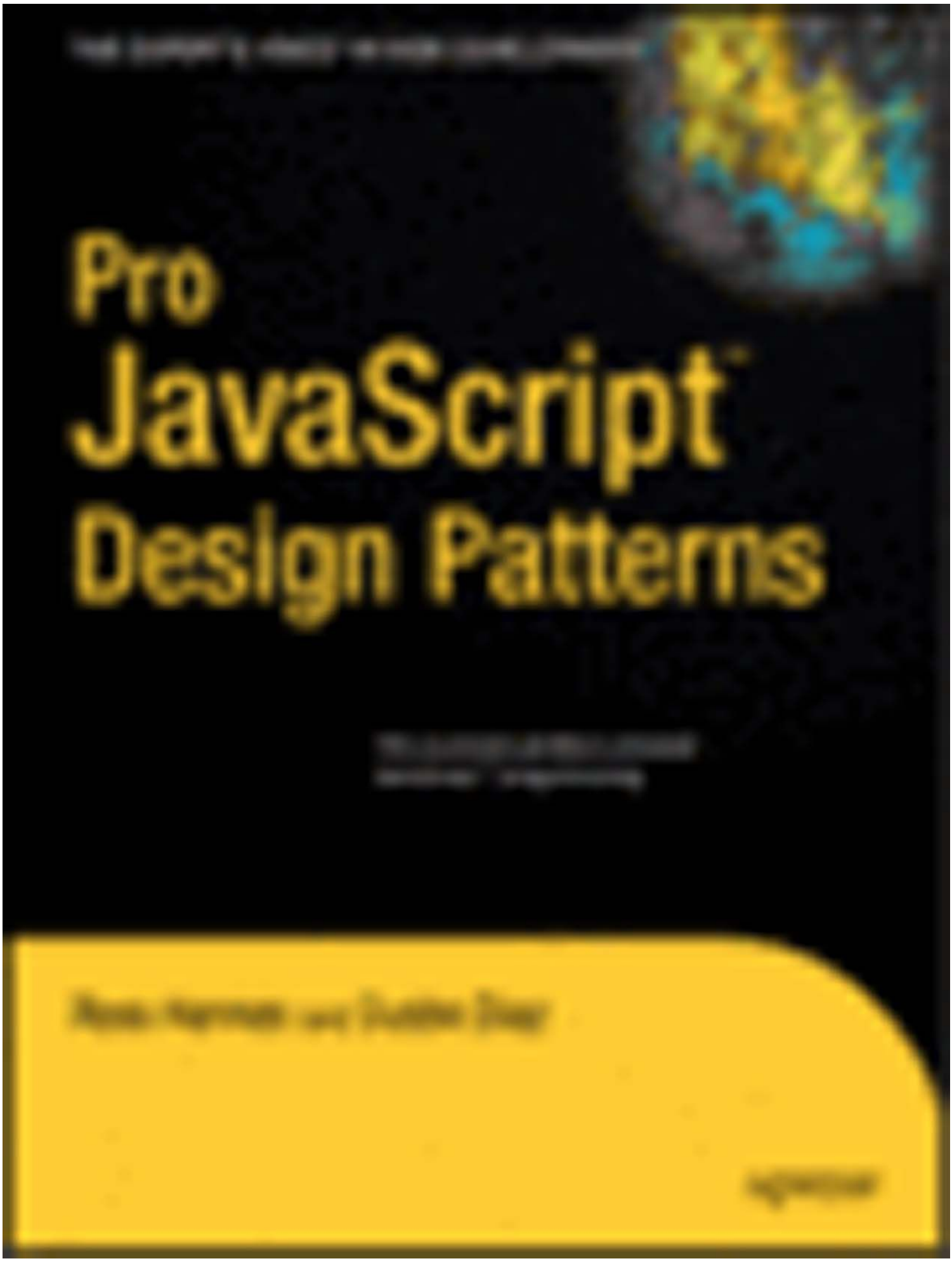
Pro
JavaScript
Design Patterns

Table of Contents

CHAPTER 5

The Singleton Pattern

The *singleton* is one of the most basic, but useful, patterns in JavaScript, and one that you will probably use more than any other. It provides a way to group code into a logical unit that can be accessed through a single variable. By ensuring that there is only one copy of a singleton object, you know that all of your code makes use of the same global resource.

Singleton classes have many uses in JavaScript. They can be used for namespacing, which reduces the number of global variables in your pages. They can be used to encapsulate browser differences through a technique known as *branching*, which allows you to use common utility functions without worrying about browser sniffing. Most importantly, they can be used to organize your code in a consistent manner, which increases the readability and maintainability of your pages.

This pattern is extremely important in JavaScript, maybe more so than in any other language. Using global variables in your pages presents a huge risk, and a namespace created with a singleton is one of the best ways to remove those global variables. This alone would make the singleton worth knowing, but this pattern can be used for many different purposes. We cover the most useful ones in this chapter.

The Basic Structure of the Singleton

Later in this chapter we get into some of the more advanced singleton patterns, but for right now, let's focus on the most basic type. It is essentially an object literal containing methods and attributes that have been grouped together because they are somehow related:

```
/* Basic Singleton. */

var Singleton = {
  attribute1: true,
  attribute2: 10,

  method1: function() {

  },
  method2: function(arg) {

  }
};
```

In this example, all of the members are now accessible through the `Singleton` variable. You can access them using the dot notation:

```
Singleton.attribute1 = false;
var total = Singleton.attribute2 + 5;
var result = Singleton.method1();
```

The singleton object shown here can be modified. You can add new members to it, just as you can with any other object literal. You can also remove members, using the `delete` keyword. This violates one of the principles of object-oriented design: classes should be open to extension but closed for modification. This is true of any object in JavaScript, and it is just one of the ways in which it differs from some other object-oriented languages, such as C++ and Java. It isn't anything to worry about (Python, Ruby, and Smalltalk also allow modification of classes after they have been defined), but you should be aware that there is nothing in the language to prevent object modification from happening. If you do need to protect certain variables, you can always define them within a closure, as discussed in Chapter 3.

It may not be clear so far how a singleton differs from a normal object literal. The traditional definition of the singleton pattern describes a class that can only be instantiated once and is accessible through a well-known access point. Following that definition strictly, the previous example isn't a singleton because it is not an instantiable class. We choose to define the singleton pattern a little more broadly: it is an object that is used to namespace and group together a related set of methods and attributes, and if it is instantiable at all, it can only be instantiated once.

Using an object literal is only one technique for creating a singleton. The others that we cover later in the chapter look more like singleton classes from other object-oriented languages. Also, not all object literals are singletons. If it is used simply to simulate an associative array, or to hold data, it obviously isn't a singleton. But if it is used to group together a related set of methods and attributes, it probably is. The difference lies mostly in intent.

Namespacing

A singleton object consists of two parts: the object itself, containing the members (both methods and attributes) within it, and the variable used to access it. The variable is usually global, so that the singleton object can be accessed anywhere in the page. This is a key feature of the singleton pattern. It needn't be global by definition, but it should be widely accessible. Because all of the members are contained within the singleton, they are not global. In a sense, they can be said to be namespaced within the singleton because they can only be accessed through the singleton's variable.

Namespacing is a large part of responsible programming in JavaScript. Because everything can be overwritten, it is very easy to wipe out a variable, a function, or even a complete class without even knowing it. These types of errors are extremely time-consuming to find:

```
/* Declared globally. */

function findProduct(id) {
  ...
}

...
```

```
// Later in your page, another programmer adds...
var resetProduct = $('reset-product-button');
var findProduct = $('find-product-button'); // The findProduct function just got
                                            // overwritten.
```

Although it doesn't apply directly to this example, it's worth noting how important it is to use `var` to declare variables within a function. If you don't use `var`, the variable will be declared globally and will be much more likely to interfere with other code in the global namespace.

Back to the example: one of the best ways to prevent accidentally overwriting variables is to namespace your code within a singleton object. Here is the previous example rewritten using a singleton:

```
/* Using a namespace. */

var MyNamespace = {
  findProduct: function(id) {
  ...
  },
  // Other methods can go here as well.
}
...

// Later in your page, another programmer adds...
var resetProduct = $('reset-product-button');
var findProduct = $('find-product-button'); // Nothing was overwritten.
```

The `findProduct` function is now a method under `MyNamespace`, and is protected from any new variables that are declared in the global namespace. It is important to note that the method is still accessible globally. Instead of calling `findProduct(id)`, you now call `MyNamespace.findProduct(id)`. This has the added benefit of letting other programmers know generally where this method was declared, as well as what it does. Namespaces can help document your code by allowing you to group like methods together.

■ **Note** `MyNamespace` is a poor choice for the name for a singleton and is used here only to illustrate that the object literal is being used as a namespace. A namespace should always describe the purpose of the code contained within it. In this example, a better name would be `ProductTools`.

You can take this one step further. In a lot of pages today, there is code from more than one source. There may be library code, advertiser code, and badge code in addition to anything you write. All of these variables exist within the global namespace of the page. In order to prevent collisions, you can put *all* of your code under a single variable:

```
/* GiantCorp namespace. */
var GiantCorp = {};
```

You can then group all of your code and data within objects or singletons under that single global variable:

```
GiantCorp.Common = {
  // A singleton with common methods used by all objects and modules.
};

GiantCorp.ErrorCodes = {
  // An object literal used to store data.
};

GiantCorp.PageHandler = {
  // A singleton with page specific methods and attributes.
};
```

The odds of some externally produced code colliding with the `GiantCorp` variable are small. If it does happen, the results are catastrophic and easily detectable. You can also sleep well knowing that you acted responsibly and didn't litter the global namespace. You only added a single variable, which is as small a footprint as any JavaScript programmer can hope to have.

A Singleton As a Wrapper for Page-Specific Code

Now that you've seen how to use a singleton object as a namespace, let's look at one particular use for the singleton pattern. In a multipage site, you will often have code that is used on all of the pages, usually stored in an external file. You also have code that is specific to a single page and isn't used anywhere else. It can be a good idea to separate these two into their own singleton objects.

The singleton that wraps the page-specific code usually looks similar from page to page. It needs to encapsulate some data (perhaps as constants), contain some methods for page-specific activities, and have an initialization method. Most of the code that involves specific elements in the DOM, such as event attachment, can only be done once those elements are loaded. By creating an `init` method and attaching it to the window's `load` event (or something similar, such as the derived `DOMContentLoaded` or `DOMLoaded` events[1]), you can group all of this initialization code in one place.

Here is a skeleton for a singleton that wraps page-specific code:

```
/* Generic Page Object. */

Namespace.PageName = {

  // Page constants.
  CONSTANT_1: true,
  CONSTANT_2: 10,

  // Page methods.
  method1: function() {
```

1. For more information, go to `http://peter.michaux.ca/article/553`.

```
  },
  method2: function() {

  },

  // Initialization method.
  init: function() {

  }
}

// Invoke the initialization method after the page loads.
addLoadEvent(Namespace.PageName.init);
```

To explain how this can be used, let's take a fairly common task in web development and walk through it. Often it is desirable to add functionality to a form with JavaScript. In order to degrade gracefully, the page is usually created first as a normally submitting, JavaScript-free, HTML-only experience. Then the form action is hijacked using JavaScript to provide additional features.

Here is a singleton that will look for a specific form and hijack it:

```
/* RegPage singleton, page handler object. */

GiantCorp.RegPage = {

  // Constants.
  FORM_ID: 'reg-form',
  OUTPUT_ID: 'reg-results',

  // Form handling methods.
  handleSubmit: function(e) {
    e.preventDefault(); // Stop the normal form submission.

    var data = {};
    var inputs = GiantCorp.RegPage.formEl.getElementsByTagName('input');

    // Collect the values of the input fields in the form.
    for(var i = 0, len = inputs.length; i < len; i++) {
      data[inputs[i].name] = inputs[i].value;
    }

    // Send the form values back to the server.
    GiantCorp.RegPage.sendRegistration(data);
  },
  sendRegistration: function(data) {
    // Make an XHR request and call displayResult() when the response is
    // received.
    ...
  },
```

```
  displayResult: function(response) {
    // Output the response directly into the output element. We are
    // assuming the server will send back formatted HTML.
    GiantCorp.RegPage.outputEl.innerHTML = response;
  },

  // Initialization method.
  init: function() {
    // Get the form and output elements.
    GiantCorp.RegPage.formEl = $(GiantCorp.RegPage.FORM_ID);
    GiantCorp.RegPage.outputEl = $(GiantCorp.RegPage.OUTPUT_ID);

    // Hijack the form submission.
    addEvent(GiantCorp.RegPage.formEl, 'submit', GiantCorp.RegPage.handleSubmit);
  }
};

// Invoke the initialization method after the page loads.
addLoadEvent(GiantCorp.RegPage.init);
```

We are first assuming that the `GiantCorp` namespace has already been created as an empty object literal. If it hasn't, this first line will cause an error. This error can be prevented with a line of code that defines `GiantCorp` if it doesn't already exist, using the boolean OR operator to provide a default value if one isn't found:

```
var GiantCorp = window.GiantCorp || {};
```

In this example, we put the IDs for the two HTML elements that we care about in constants since these won't change in the execution of the program.

The initialization method gets the two HTML elements and stores them as new attributes within the singleton. This is fine; you can add or remove members from the singleton at runtime. This method also attaches a method to the form's submit event. Now when the form is submitted, the normal behavior will be stopped (with `e.preventDefault()`) and instead all of the form data will be collected and sent back to the server using Ajax.

A Singleton with Private Members

In Chapter 3 we discussed several different ways to create private members in classes. One of the drawbacks of having true private methods is that they are very memory-inefficient because a new copy of the method would be created for each instance. But because singleton objects are only instantiated once, you can use true private methods without having to worry about memory. That being said, it is still easier to create pseudoprivate members, so we will cover those first.

Using the Underscore Notation

The easiest and most straightforward way to create the appearance of private members within a singleton object is to use the underscore notation. This lets other programmers know that

the method or attribute is intended to be private and is used in the internal workings of the object. Using the underscore notations within singleton objects is a straightforward way of telling other programmers that certain members shouldn't be accessed directly:

```
/* DataParser singleton, converts character delimited strings into arrays. */

GiantCorp.DataParser = {
  // Private methods.
  _stripWhitespace: function(str) {
    return str.replace(/\s+/, '');
  },
  _stringSplit: function(str, delimiter) {
    return str.split(delimiter);
  },

  // Public method.
  stringToArray: function(str, delimiter, stripWS) {
    if(stripWS) {
      str = this._stripWhitespace(str);
    }
    var outputArray = this._stringSplit(str, delimiter);
    return outputArray;
  }
};
```

In this example, there is a singleton object with one public method, `stringToArray`. This method takes as arguments a string, a delimiter, and an optional boolean that tells the method whether to remove all white space. This method uses two private methods to do most of the work: `_stripWhitespace` and `_stringSplit`. These methods should not be public because they aren't part of the singleton's documented interface and aren't guaranteed to be there in the next update. Keeping these methods private allows you to refactor all of the internal code without worrying about breaking someone else's program. Let's say that later on you take a look at this object and realize that `_stringSplit` doesn't really need to be a separate function. You can remove it completely, and because it is marked as private with the underscore, you can be fairly confident that no one else is calling it directly (and if they are, they deserve whatever errors they get).

In the `stringToArray` method, `this` was used to access other methods within the singleton. It is the shortest and most convenient way to access other members of the singleton, but it is also slightly risky. It isn't always guaranteed that `this` will point to `GiantCorp.DataParser`. For instance, if you are using a method as an event listener, `this` may instead point to the `window` object, which means the methods `_stripWhitespace` and `_stringSplit` will not be found. Most JavaScript libraries do scope correction for event attachment, but it is safer to access other members within the singleton by using the full name, `GiantCorp.DataParser`.

Using Closures

The second way to get private members within a singleton object is to create a closure. This will look very similar to how we created true private members in Chapter 3, but with one major difference. Before, we added variables and functions to the body of the constructor (without

the this keyword) to make them private. We also declared all privileged methods within the constructor but used this to make them publicly accessible. All of the methods and attributes declared within the constructor are recreated for each instance of the class. This has the potential to be very inefficient.

Because a singleton is only instantiated once, you don't have to worry about how many members you declare within the constructor. Each method and attribute is only created once, so you can declare *all* of them within the constructor (and thus, within the same closure). Up to this point, all of the singletons have been object literals, like this:

```
/* Singleton as an Object Literal. */

MyNamespace.Singleton = {};
```

You will now use a function, executed immediately, to provide the same thing:

```
/* Singleton with Private Members, step 1. */

MyNamespace.Singleton = function() {
  return {};
}();
```

In these two examples, the two versions of MyNamespace.Singleton that are created are completely identical. It is important to note that in the second example you are not assigning a function to MyNamespace.Singleton. You are using an anonymous function to return an object. The object is what gets assigned to the MyNamespace.Singleton variable. To execute this anonymous function immediately, simply put a pair of parentheses next to the closing bracket.

Some programmers find it useful to add another pair of parentheses around the function to denote that it is being executed as soon as it is declared. This is especially useful if the singleton is large. You can then see at a glance that the function is used only to create a closure. This is what the previous singleton would look like with this extra set of parentheses:

```
/* Singleton with Private Members, step 1. */

MyNamespace.Singleton = (function() {
  return {};
})();
```

You can add public members to that singleton in the same manner as before by adding them to the object literal that gets returned:

```
/* Singleton with Private Members, step 2. */

MyNamespace.Singleton = (function() {
  return { // Public members.
    publicAttribute1: true,
    publicAttribute2: 10,

    publicMethod1: function() {
      ...
    },
```

```
    publicMethod2: function(args) {
      ...
    }
  };
})();
```

So why bother adding a function wrapper if it produces the same object that you can create using nothing more than an object literal? Because that function wrapper creates a closure to add true private members. Any variable or function declared within the anonymous function (but not within the object literal) is accessible only to other functions declared within that same closure. The closure is maintained even after the anonymous function has returned, so the functions and variables declared within it are always accessible within (and only within) the returned object.

Here is how to add private members to the anonymous function:

```
/* Singleton with Private Members, step 3. */

MyNamespace.Singleton = (function() {
  // Private members.
  var privateAttribute1 = false;
  var privateAttribute2 = [1, 2, 3];

  function privateMethod1() {
    ...
  }
  function privateMethod2(args) {
    ...
  }

  return { // Public members.
    publicAttribute1: true,
    publicAttribute2: 10,

    publicMethod1: function() {
      ...
    },
    publicMethod2: function(args) {
      ...
    }
  };
})();
```

This particular singleton pattern is also known as the *module pattern,*[2] referring to the fact that it modularizes and namespaces a set of related methods and attributes.

2. For more information, go to http://yuiblog.com/blog/2007/06/12/module-pattern/.

Comparing the Two Techniques

Now let's return to our `DataParser` example to see how to implement it using true private members. Instead of appending an underscore to the beginning of each private method, put these methods in the closure:

```
/* DataParser singleton, converts character delimited strings into arrays. */
/*   Now using true private methods. */

GiantCorp.DataParser = (function() {
  // Private attributes.
  var whitespaceRegex = /\s+/;

  // Private methods.
  function stripWhitespace(str) {
    return str.replace(whitespaceRegex, '');
  }
  function stringSplit(str, delimiter) {
    return str.split(delimiter);
  }

  // Everything returned in the object literal is public, but can access the
  // members in the closure created above.
  return {
    // Public method.
    stringToArray: function(str, delimiter, stripWS) {
      if(stripWS) {
        str = stripWhitespace(str);
      }
      var outputArray = stringSplit(str, delimiter);
      return outputArray;
    }
  };
})(); // Invoke the function and assign the returned object literal to
      // GiantCorp.DataParser.
```

You call the private methods and attributes by just using their names. You don't need to add `this.` or `GiantCorp.DataParser.` before their names; that is only used for the public members.

This pattern has several advantages over the underscore notation. By putting the private members in a closure, you are ensuring that they will never be used outside of the object. You have complete freedom to change the implementation details without breaking anyone else's code. This also allows you to protect and encapsulate data, although singletons are rarely used in this way unless the data needs to exist in only one place.

Using this pattern, you get all of the advantages of true private members with none of the drawbacks because this class is only instantiated once. That is what makes the singleton pattern one of the most popular and widely used in JavaScript.

■ **Caution** It is important to remember that public members and private members within a singleton are declared using a different syntax, due to the fact that the public members are in an object literal and the private members are not. Private attributes must be declared using `var`, or else they will be made global. Private methods are declared as `function funcName(args) { ... }`, with no semicolon needed after the closing bracket. Public attributes and methods are declared as `attributeName: attributeValue` and `methodName: function(args) { ... }`, respectively, with a comma following if there are more members declared after.

Lazy Instantiation

All of the implementations of the singleton pattern that we have discussed so far share one thing in common: they are all created as soon as the script loads. If you have a singleton that is expensive to configure, or resource-intensive, it might make more sense to defer instantiation until it is needed. Known as *lazy loading*, this technique is used most often for singletons that must load large amounts of data. If you are using a singleton as a namespace, a page wrapper, or as a way to group related utility methods, they probably should be instantiated immediately.

These lazy loading singletons differ in that they must be accessed through a static method. Instead of calling `Singleton.methodName()`, you would call `Singleton.getInstance().methodName()`. The `getInstance` method checks to see whether the singleton has been instantiated. If it hasn't, it is instantiated and returned. If it has, a stored copy is returned instead. To illustrate how to convert a singleton to a lazy loading singleton, let's start with our skeleton for a singleton with true private members:

```
/* Singleton with Private Members, step 3. */

MyNamespace.Singleton = (function() {
  // Private members.
  var privateAttribute1 = false;
  var privateAttribute2 = [1, 2, 3];

  function privateMethod1() {
    ...
  }
  function privateMethod2(args) {
    ...
  }

  return { // Public members.
    publicAttribute1: true,
    publicAttribute2: 10,

    publicMethod1: function() {
      ...
    },
```

```
    publicMethod2: function(args) {
      ...
    }
  };
})();
```

So far, nothing has changed. The first step is to move all of the code within the singleton into a `constructor` method:

```
/* General skeleton for a lazy loading singleton, step 1. */

MyNamespace.Singleton = (function() {

  function constructor() { // All of the normal singleton code goes here.
    // Private members.
    var privateAttribute1 = false;
    var privateAttribute2 = [1, 2, 3];

    function privateMethod1() {
      ...
    }
    function privateMethod2(args) {
      ...
    }

    return { // Public members.
      publicAttribute1: true,
      publicAttribute2: 10,

      publicMethod1: function() {
        ...
      },
      publicMethod2: function(args) {
        ...
      }
    }
  }

})();
```

This method is inaccessible from outside of the closure, which is a good thing. You want to be in full control of when it gets called. The public method `getInstance` is used to implement this control. To make it publicly accessible, simply put it in an object literal and return it:

```
/* General skeleton for a lazy loading singleton, step 2. */

MyNamespace.Singleton = (function() {
```

```
  function constructor() { // All of the normal singleton code goes here.
    ...
  }

  return {
    getInstance: function() {
      // Control code goes here.
    }
  }
})();
```

Now you are ready to write the code that controls when the class gets instantiated. It needs to do two things. First, it must know whether the class has been instantiated before. Second, it needs to keep track of that instance so it can return it if it has been instantiated. To do both of these things, use a private attribute and the existing private method `constructor`:

```
/* General skeleton for a lazy loading singleton, step 3. */

MyNamespace.Singleton = (function() {

  var uniqueInstance; // Private attribute that holds the single instance.

  function constructor() { // All of the normal singleton code goes here.
    ...
  }

  return {
    getInstance: function() {
      if(!uniqueInstance) { // Instantiate only if the instance doesn't exist.
        uniqueInstance = constructor();
      }
      return uniqueInstance;
    }
  }
})();
```

Once the singleton itself has been converted to a lazy loading singleton, you must also convert all calls made to it. In this example, you would replace all method calls like this:

```
MyNamespace.Singleton.publicMethod1();
```

In their place, we would write method calls like this:

```
MyNamespace.Singleton.getInstance().publicMethod1();
```

Part of the downside of a lazy loading singleton is the added complexity. The code used to create this type of singleton is unintuitive and can be difficult to understand (though good documentation can help). If you need to create a singleton with deferred instantiation, it's helpful to leave a comment stating why it was done, so that someone else doesn't come along and simplify it to just a normal singleton.

It may also be useful to note that long namespaces can be shortened by creating an *alias*. An alias is nothing more than a variable that holds a reference to a particular object. In this case, `MyNamespace.Singleton` could be shortened to `MNS`:

```
var MNS = MyNamespace.Singleton;
```

This creates another global variable, so it might be best to declare it within a page wrapper singleton instead. When singletons are wrapped in singletons, issues of scope start to arise. This would be a good place to use fully qualified names (such as `GiantCorp.SingletonName`) instead of `this` when accessing other members.

Branching

Branching is a technique that allows you to encapsulate browser differences into dynamic methods that get set at run-time. As an example, let's create a method that returns an XHR object. This XHR object is an instance of the `XMLHttpRequest` class for most browsers and an instance of one of the various ActiveX classes for older versions of Internet Explorer. A method like this usually incorporates some type of browser sniffing or object detection. If branching isn't used, each time this method is called, all of the browser sniffing code must be run again. This can be very inefficient if the method is called often.

A more efficient way is to assign the browser-specific code only once, when the script loads. That way, once the initialization is complete, each browser only executes the code specific to its implementation of JavaScript. The ability to dynamically assign code to a function at run-time is one of the reasons that JavaScript is such a flexible and expressive language. This kind of optimization is easy to understand and makes each of these function calls more efficient.

It may not be immediately clear how the topic of branching is related to the singleton pattern. In each of the three patterns described previously, all of the code is assigned to the singleton object at run-time. This can be seen most clearly with the pattern that uses a closure to create private members:

```
MyNamespace.Singleton = (function() {
  return {};
})();
```

At run-time, the anonymous function is executed and the returned object literal is assigned to the `MyNamespace.Singleton` variable. It would be easy to create two different object literals and assign one of them to the variable based on some condition:

```
/* Branching Singleton (skeleton). */

MyNamespace.Singleton = (function() {
  var objectA = {
    method1: function() {
      ...
    },
    method2: function() {
      ...
    }
  };
```

```
  var objectB = {
    method1: function() {
      ...
    },
    method2: function() {
      ...
    }
  };

  return (someCondition) ? objectA : objectB;
})();
```

Here, two object literals are created, both with the exact same methods. To the programmer using this singleton, it doesn't matter which one gets assigned because they both implement the same interface and perform the same task; only the specific code used has changed. This isn't limited to just two branches; you could just as easily create a singleton with three or four branches, if you had a reason to. The condition used to choose among these branches is determined at run-time. This condition is often some form of capability checking, to ensure that the JavaScript environment running the code implements the needed features. If it doesn't, fallback code is used instead.

Branching isn't always more efficient. In the previous example, two objects (`objectA` and `objectB`) are created and maintained in memory, even though only one is ever used. When deciding whether to use this technique, you must weigh the benefit of reduced computation time (since the code that decides which object to use is only executed once) versus the drawback of higher memory usage. The next example shows a case when branching should be used, as the branch objects are small and the cost of deciding which to use is large.

Example: Creating XHR Objects with Branching

Let's walk through the example of creating a singleton with a method that instantiates an XHR object. There is a more advanced version of this in Chapter 7. First determine how many branches you need. Since there are three different types of objects that can be instantiated, you need three branches. Name each branch by the type of XHR object that it returns:

```
/* SimpleXhrFactory singleton, step 1. */

var SimpleXhrFactory = (function() {

  // The three branches.
  var standard = {
    createXhrObject: function() {
      return new XMLHttpRequest();
    }
  };
  var activeXNew = {
    createXhrObject: function() {
      return new ActiveXObject('Msxml2.XMLHTTP');
    }
```

```
  };
  var activeXOld = {
    createXhrObject: function() {
      return new ActiveXObject('Microsoft.XMLHTTP');
    }
  };

})();
```

Each of the three branches contains an object literal with one method, `createXhrObject`. This method simply returns a new object that can be used to make an asynchronous request.

The second part to creating a branching singleton is to use the condition to assign one of these branches to the variable. To do that, test each of the XHR objects until you find one that the given JavaScript environment supports:

```
/* SimpleXhrFactory singleton, step 2. */

var SimpleXhrFactory = (function() {

  // The three branches.
  var standard = {
    createXhrObject: function() {
      return new XMLHttpRequest();
    }
  };
  var activeXNew = {
    createXhrObject: function() {
      return new ActiveXObject('Msxml2.XMLHTTP');
    }
  };
  var activeXOld = {
    createXhrObject: function() {
      return new ActiveXObject('Microsoft.XMLHTTP');
    }
  };

  // To assign the branch, try each method; return whatever doesn't fail.
  var testObject;
  try {
    testObject = standard.createXhrObject();
    return standard; // Return this if no error was thrown.
  }
  catch(e) {
    try {
      testObject = activeXNew.createXhrObject();
      return activeXNew; // Return this if no error was thrown.
    }
    catch(e) {
```

```
      try {
        testObject = activeXOld.createXhrObject();
        return activeXOld; // Return this if no error was thrown.
      }
      catch(e) {
        throw new Error('No XHR object found in this environment.');
      }
    }
  }

})();
```

This singleton can now be used to instantiate an XHR object. The programmer that uses this API need only call `SimpleXhrFactory.createXhrObject()` to get the correct object for the particular run-time environment. Branching allows all of the feature sniffing code to be executed only once ever, instead of once for each object that is instantiated.

This is a powerful technique that can be used in any situation where the particular implementation can only be chosen at run-time. We cover this topic in more depth when we discuss the factory pattern in Chapter 7.

When Should the Singleton Pattern Be Used?

When used for namespacing and modularizing your code, the singleton pattern should be used as often as possible. It is one of the most useful patterns in JavaScript and has its place in almost every project, no matter how large or small. In quick and simple projects, a singleton can be used simply as a namespace to contain all of your code under a single global variable. On larger, more complex projects, it can be used to group related code together for easier maintainability later on, or to house data or code in a single well-known location. In big or complicated projects, it can be used as an optimizing pattern: expensive and rarely used components can be put into a lazy loading singleton, while environment-specific code can be put into a branching singleton.

It is rare to find a project that can't benefit from some form of the singleton pattern. JavaScript's flexibility allows a singleton to be used for many different tasks. We would even go as far as to call it a much more important pattern in this language than in any other. This is mostly because it can be used to create namespaces, reducing the number of global variables. This is a very important thing in JavaScript, where global variables are more dangerous than in other languages; the fact that code from any number of sources and programmers is often combined in a single page means variables and functions can be very easily overwritten, effectively killing your code. That a singleton can prevent this makes it a huge asset to any programmer's toolbox.

Benefits of the Singleton Pattern

The main benefit of the singleton pattern is the way it organizes your code. By grouping related methods and attributes together in a single location, which can't be instantiated multiple times, you can make it easier to debug and maintain your code. Using descriptive namespaces

also makes your code self-documenting and easier for novices to read and understand. Sandboxing your methods within a singleton shields them from being accidentally overwritten by other programmers and prevents the global namespace from becoming cluttered with variables. It separates your code from third-party library or ad-serving code, allowing greater stability to the page as a whole.

The more advanced versions of the singleton pattern can be used later in the development cycle to optimize your scripts and improve performance to the end user. Lazy instantiation allows you to create objects only as needed, reducing memory (and potentially bandwidth) consumption for users who don't need them. Branching allows you to create efficient methods, regardless of browser or environment incompatibilities. By assigning object literals based on the conditions at run-time, you can create methods tailored to a particular environment without having to waste cycles checking the environment again each time a method is called.

Drawbacks of the Singleton Pattern

By providing a single point of access, the singleton pattern has the potential to tightly couple modules together. This is the main complaint leveraged at this pattern, and it is a valid one. There are times when it is better to create an instantiable class, even if it is only ever instantiated once. It also makes your code harder to unit test because it has the potential to tightly couple classes together. You can't independently test a class that calls methods from a singleton; instead, you must test the class and the singleton together as one unit. Singletons are best reserved for namespacing and for implementing branching methods. In these cases, coupling isn't as much of an issue.

There are times when a more advanced pattern is better suited to the task. A virtual proxy can be used instead of a lazy loading singleton when you want a little more control over how the class gets instantiated. A true object factory can be used instead of a branching singleton (although that factory may *also* be a singleton). Don't be afraid to investigate the more specific patterns in this book, and don't settle on using a singleton just because it is "good enough." Make sure that the pattern you choose is right for the job.

Summary

The singleton pattern is one of the most fundamental patterns in JavaScript. Not only is it useful by itself, as we have seen in this chapter, but it can be used in some form or another with most of the patterns in this book. For instance, you can create object factories as singletons, or you can encapsulate all of the sub-objects of a composite within a singleton namespace. This book is about creating reusable and modular code. Finding ways to organize and document that code is one of the biggest steps toward accomplishing that goal. Singletons can help out enormously in that regard. By putting your code within a singleton, you are going a long way toward creating an API that can be used by others without fear of having their global variables overwritten. It is the first step to becoming an advanced and responsible JavaScript programmer.