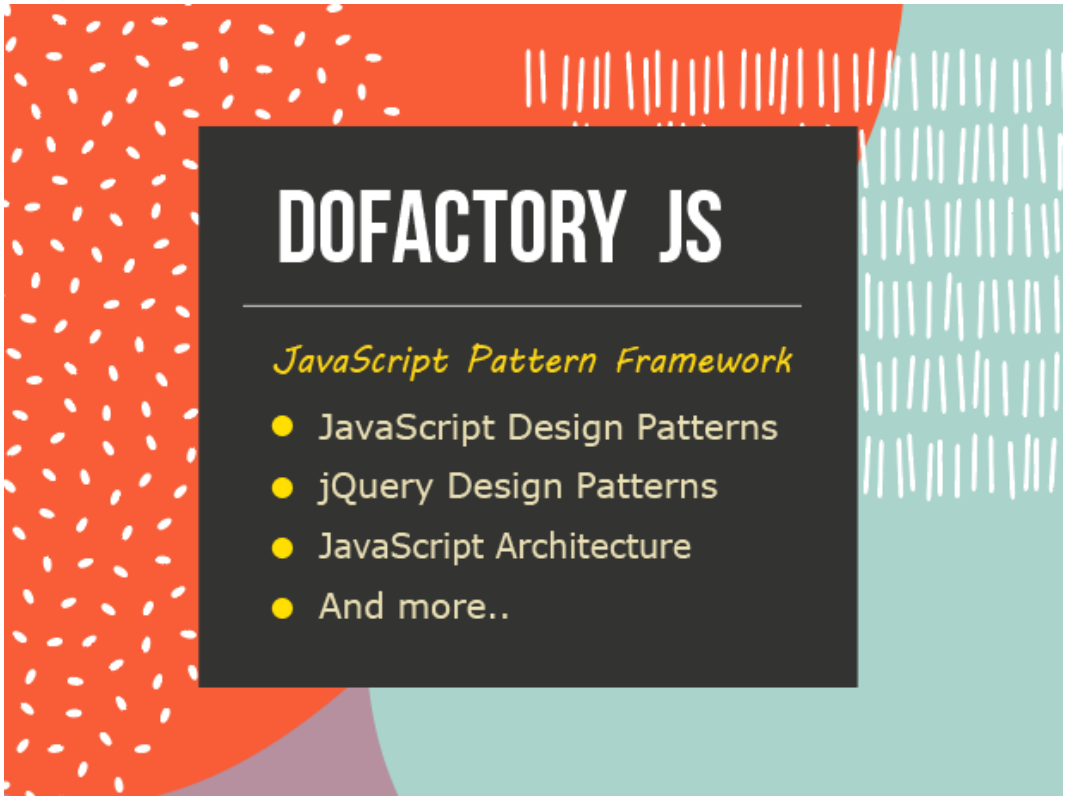
Dofactory JS 6.0

Modern JavaScript Patterns



by

Data & Object Factory, LLC

www.dofactory.com

3. Modern JavaScript Patterns

Index

Introduction

Modern Patterns are more recently developed JavaScript patterns. Several are JavaScript specific meaning they specifically apply to JavaScript and not to any other language. Patterns in this category include Invocation, Monkey Patch and Partial Patterns.

Other Modern Patterns are designed to implement features that exist in other, more mature languages but that are lacking in JavaScript. Examples of this category include Module and Namespace. These patterns and more are presented in this section.

Constructor Pattern

The *Constructor pattern* refers to the process of creating new object instances in JavaScript. There are different ways of doing this and it is important to understand the different options.

If you are coming from a class-based language you're familiar with creating objects by calling a class constructor with the `new` operator. JavaScript does not support classes, but it does have a `new` operator. Calling `new` with a constructor function will return a newly created object instance, like this:

```
var obj = new Object();           // new operator
```

This creates an empty object which doesn't do very much. To create more interesting and useful objects we need to add properties and methods. JavaScript is a dynamic language that allows us to create an empty object and subsequently add object members by using the dot operator:

```
var obj = new Object();
obj.name = "Anthony Miller";
obj.hello = function () { alert("hello"); }
```

Alternatively, we can use the [] operator.

```
var obj = new Object();
obj["name"] = "Anthony Miller";
obj["hello"] = function () { alert("hello"); };
```

For objects with many properties this can become rather tedious. A more compact way of doing this is with object literals which are object definitions surrounded by curly braces. Let's start off with a basic example. Here we create an empty person object:

```
var person = {};      // object literal notation
```

Next we'll create a more interesting object with a property and a method in a single statement:

```
var person = {
   name: "Anthony Miller",
   say: function () { alert(this.name); }
};

person.say();       // => Anthony Miller
```

This object has two members: a property called `name` and a method called `say` (by the way: a method is a function that is part of an object). The `say` method displays the person's name.

Object literal notation is frequently used by JavaScript developers. However, as used above, the object literal is not reusable. Constructor functions were designed to be reusable and are discussed next.

The Object constructor function is native to JavaScript and allows you to create empty objects. It is rarely used, but you can create *custom* constructor functions that are used extensively in JavaScript. Here is an example:

```
function Person() {
    this.name = "Nelly Jones";
```

```
    this.say = function () { alert(this.name); };
}

var person = new Person();
person.say();               // => Nelly Jones
```

This creates a person object with hardcoded property values. A more flexible and reusable approach is to create a constructor function that takes an argument with which the name is initialized:

```
function Person(name) {
    this.name = name;
    this.say = function () { alert(this.name); };
}

var anthony = new Person("Anthony Miller");
var celeste = new Person("Celeste Diaz");

anthony.say();              // => Anthony Miller
celeste.say();              // => Celeste Diaz
```

The '`this`' keyword in the function body is bound to the object that is being created and is called the *context* of the function. We now have a truly reusable constructor function which allows us to create many new instances without much effort.

Alternatively, we can create an *anonymous* constructor function and assign it to a variable, like so:

```
var Person = function (name) {
    this.name = name;
    this.say = function () { alert(this.name); };
}
var anthony = new Person("Anthony Miller");
var celeste = new Person("Celeste Diaz");

anthony.say();              // => Anthony Miller
celeste.say();              // => Celeste Diaz
```

The function does not have a name (it is anonymous) and is assigned to a variable as if it were a primitive value or object type. This is not far off, because, in fact, functions *are* true objects. They come with properties and methods just like objects. This is why they are referred to as *function objects*.

Whether you use a named function or an anonymous function that you assign to a variable makes no difference in how you use it. Either way, you use the `new` operator followed by the function name or the variable name.

However, there is a slight difference when debugging. Since an anonymous function has no name there is no way for the debugger to reference or search for it. For this reason, some developers prefer named functions over anonymous functions.

By convention the name of a constructor function starts with an *uppercase* letter. All other functions start with lowercase. This is merely a convention but a very important one. Here is why. Let's say you forget to include the `new` keyword when creating new persons, like this:

```
var Person = function (name) {
    this.name = name;
    this.say = function () { alert(this.name) };
}

var anthony = Person("Anthony Miller");    // missing new
var celeste = Person("Celeste Diaz");      // missing new
```

There will be no warning and initially the program seems to run fine, but after a while you start to see errors. What happens when you omit the `new` keyword is that the '`this`' reference does not get bound to the current object but to the global object instead (window in browsers).

JavaScript, being a dynamic language, will happily create the `name` property and the `say` function on the global object without warning, but now all persons are starting to overwrite each other's name.

The problem is that these bugs can be very difficult to find. Other than visual inspection (i.e., a developer staring at the code) there is no good way to discover the source of the

problem. Having a constructor function with an upper-case letter is a useful hint to developers that a new prefix is required.

EcmaScript 5 addresses the problem. When in strict mode (`"use strict";`) the keyword this will no longer reference the global object. But what if you are not able to use the ES5 standard yet? Fortunately, there is a simple way to defensively code against forgetting the new prefix:

```
var Person = function (name) {
    if (!(this instanceof Person)) {
       return new Person(name);
    }

    this.name = name;
    this.say = function () {alert(this.name);};
}
var anthony = Person("Anthony Miller");   // missing new
var celeste = Person("Celeste Diaz");     // missing new

anthony.say();       // => Anthony Miller
celeste.say();       // => Celeste Diaz
```

In this example, the constructor function checks if `this` is of type Person. If it is not, then that means that `new` was omitted and it will immediately correct the situation by calling the constructor function prefixed with `new`. The overhead of the if-statement is minimal, and it can avoid late night debugging sessions in the office.

Constructor functions implicitly return the object referenced by this. There is no need to include a return statement. However, you can return your own objects if you prefer:

```
var Person = function (name) {
    var that = {};
    that.name = name;
    that.say = function () {alert(this.name);}
    return that;
};
```

Please note: the variable `that` is not a keyword, just an alternative for `this`. You can write the above code more succinctly by returning an object literal without the need for a `that` variable:

```
var Person = function (name) {
    return {
        name: name,
        say: function () {alert(this.name);}
    };
}
```

This is a frequently used style as it allows the creation of private variables. Classically trained developers are used to access modifiers: private, protected, and public, but JavaScript does not support these. Fortunately, you can create private variables following this approach:

```
var Person = function (name) {

    var privateName = name;    // private

    return {
        say: function () { alert(privateName); }
    };
}

var anthony = new Person("Anthony Miller");
var celeste = new Person("Celeste Diaz");

anthony.say();                                  // Anthony Miller
celeste.say();                                  // Celeste Diaz
```

The incoming `name` argument is assigned to local variable named `privateName`. Notice that the `say` method now displays `privateName`. Although JavaScript does not natively support private members (or any other access modifier), this variable is truly private and not accessible from anywhere outside the function body.

You may be wondering what the value of `privateName` will be after the constructor is called, for example, when the `say` method displays its value. Rest assured it is the original

value when the object was created. The value will be maintained during the lifetime of the object. You can see that this is the case with anthony and celeste.

What is at work here is an important concept in JavaScript called a *closure*. When instantiating an object with a constructor function the internal values are wrapped into a closure and stay into existence until the object goes out of scope and gets garbage collected.

By the way, argument values are also part of a function's closure. In the above case, the closure maintains the values of two variables: `name` and `privateName`. So, if `name` is part of the closure, then there is no need to assign the argument to a local value; correct? Yes, that is indeed correct. Look at the following code:

```
var Person = function (name) {
    return {
        say: function () { alert(name); }
    };
}

var anthony = Person("Anthony Miller");
var celeste = Person("Celeste Diaz");

anthony.say();                                  // Anthony Miller
celeste.say();                                  // Celeste Diaz
```

As you can see this works the same as the prior example that used `privateName`.

You can also change argument values and they will still be maintained by the closure.

```
var Person = function (name) {
    return {
        setName: function (n) {name = n;},
        getName: function () {return name;},
        say: function () {alert(name);}
    };
}

var person = Person("Superman");
person.say();                                   // => Superman
```

```
person.setName("King Kong");
person.say();                                   // => King Kong
```

Works like a charm.

Building private functions is also a breeze with closures:

```
var Person = function (name) {

    var privateName = name;                             // private
    var privateLength = function () {                   // private
        return privateName.length;
    };

    return {
        name: privateName,                              // public
        nameLength: privateLength,                      // public
        say: function () { alert(privateName); }        // public
    };
}
```

The function `privateLength` is private. Everything that is returned is public. Notice that a person's public properties and methods have full access to its private variables and functions (and arguments), even beyond the creation stage; again, this is courtesy of the closure that was constructed for us.

Rather than returning an object using an object literal we can also return a function (actually a *function literal*). Just like the object literal the returned function also has full access to any private variable or function in the surrounding constructor function.

```
var Person = function (name) {

    var privateName = name;                                 // private
    var privateLength = function () {                       // private
        return privateName.length;
    };

    return function (name) {
        this.name = name,                                   // public
        this.nameLength = privateLength,                    // public
```

```
        this.say = function () { alert(privateName); }     // public
    };
}
```

The constructor functions we've looked at so far have both properties and methods defined in their function body. Each object instance that we create has a full set of properties and methods.

We can limit the memory of each instance by having all instances share their methods. We do this by moving the methods out to the function's prototype. Each instance will have full access to these methods through their prototype properties. Here is the original code with methods in the function body.

```
function Person(name) {
    this.name = name;
    this.say = function () { alert(this.name); };
}
```

And here is the same with the say method moved to its prototype

```
function Person(name) {
    this.name = name;
}

Person.prototype.say = function () {
    alert(this.name);
}

var anthony = new Person("Anthony Miller");
var celeste = new Person("Celeste Diaz");

anthony.say();                                  // Anthony Miller
celeste.say();                                  // Celeste Diaz
```

The prototype method displays the name of the person and is shared by both instances. This shows that the prototype function, although shared, has full access to each objects' properties through `this` which is bound to the current object.

Following the construction of an object you can have follow-up step called initialization (also referred to as the Init Pattern). Essentially, o*bject initialization* is a way to complete an object's initialization. Some popular Model-View frameworks use this pattern, notably Backbone and Ember. They let you create new objects by extending their framework objects. Then the framework will call an init or initialize method giving the object the opportunity to complete the initialization process. Here is an example:

```
function Person() {
    this.name = "";
    this.say = function () { alert(this.name); };

    this.age = 0;
    this.profession = "None";

    this.init = function (options) {                // initializer
        this.name = options.name;
        this.age = options.age;
        this.profession = options.profession;
        this.showAge = function () {
            alert(this.age);
        };
    };
}

var person = new Person();
person.init( {name: "Al", age: 33, profession: "JS Geek"} );

person.say();                // John
person.showAge();            // 33
```

This example is a bit contrived but hopefully you get the idea. The init method may accept options that configure the object, or it is a place where relationships are established with other objects in the app.

Namespace Pattern

When you declare variables and functions in your JavaScript file, they end up in the global object space (the global object called `window` in the browser). These variables and

functions become properties and methods on the global object. Built-in functions, such as, `alert`, `confirm`, and `eval` are all functions on the global object.

As you know, using global variables and functions is not a good way to organize your code. Consider the following example:

```
var i, j, k;
var total, name;

function add(x, y) { /* ... */ }

function sort(array) { /* ... */ }
function length(str) { /* ... */ }
```

What happens is that all variables and functions are added to the global object (or global namespace) creating the risk of name collision. Name collision occurs when variables or functions are added to the global object with the same name, essentially overwriting each other. Let's look at an example.

Suppose you have a global variable named `animationRate` in your code and everything works fine. Then you add a third-party animation library on the same page and, unbeknownst to you, this library uses a global variable with the same name. Now your code and this library are reading and writing the same variable resulting in unpredictable results.

If you have animations on your page the name `animationRate` seems pretty common, so that is a name to avoid when making it globally available. Our previous code example is even worse: the names `i`, `j`, `k`, `total`, and `name` are all very common, as are the function names: `add`, `sort`, `length`, etc. The risk of name collision is very high in these cases. This scattering of names into the global object space is called *namespace pollution*.

The trouble with collision bugs like the `animationRate` example is that they are extremely hard to find, especially because you don't know what goes on in the third-party library. Perhaps this library is minified which makes it just about impossible to explore the code.

Other languages have solved the problem of name collision by using namespaces. Namespaces allow variables and functions to exist in their own (named) scope and thus

avoiding collisions. JavaScript does not natively support namespaces, but fortunately we have the Namespace pattern coming to our rescue.

The Namespace pattern allows developers to create a single global object that contains the entire body of code for your application. Usually this global object has a unique name such as a company name, a project name, or a domain name. This will avoid clashes with any other libraries or JavaScript controls that may be used on the same page. In our example we will use `MyFramework`.

Here is the same code as before, but now placed into its own namespace.

```
var MyFramework = {};
var MyFramework.i, MyFramework.j, MyFramework.k;
var MyFramework.total, MyFramework.name;

function MyFramework.add(x, y) { /* ... */ }

function MyFramework.sort(array) { /* ... */ }
function MyFramework.length(str) { /* ... */ }
```

The only name added to the global object is `MyFramework`. Everything is prefixed with our framework's name and there is little risk of anyone stepping on our variables or functions. A whole lot better. Unfortunately, the solution is not perfect because when the project gets large with many properties and method names, we will start polluting our own namespace again.

This is easily corrected by further partitioning `MyFramework` into smaller sub-namespaces, for example `MyFramework.Utils.Dom`, `MyFramework.Utils.Event`, and `MyFramework.Animation`.

We can create these namespaces as object literals, like so:

```
var MyFramework = {};
var MyFramework.Utils = {};
var MyFramework.Utils.Dom = {};
var MyFramework.Utils.Events = {};
var MyFramework.Animation = {};
```

You can do the same in a single statement using object literal notation:

```
var MyFramework = {
    Utils: {
        Dom : { },
        Event: { }
    },
    Animation: { }
};
```

The actual application code for each namespace can be added inline, but this becomes unwieldy very quickly:

```
var MyFramework = {
        Utils: {
            Dom : {
              // Dom code here ...
            },
            Event: {
              // Event code here ...
            }
        },
        Animation: {
            // Animation code here ...
        }
        // MyFramework code here ...
};
```

At some point when a program becomes large and complex you will need to partition your namespace file into smaller files. But then, who manages the namespaces? Simply creating a new namespaces at the top of each file is risky because it may already exist and you would end up overwriting the code.

Consider the following statement:

```
var MyFramework = {};
```

Having this in multiple files will result in errors. You could include it in each file but before creating the namespace you check if it already exists. In JavaScript you can do this like so:

```
var MyFramework = MyFramework || {};
```

If `MyFramework` is undefined it creates a new empty object, otherwise it simply reassigns it to itself. You can do this at the beginning of each file for each namespace:

```
var MyFramework = MyFramework || {};
var MyFramework.Utils = MyFramework.Utils || {};
var MyFramework.Utils.Dom = MyFramework.Utils.Dom || {};
var MyFramework.Utils.Events = MyFramework.Utils.Events || {};
var MyFramework.Animation = MyFramework.Animation || {};
```

This works well, but it seems a bit too repetitive. It can be further improved with a function that builds nested namespaces. It is nondestructive, meaning it is careful not to create namespaces that already exist. The name of this function is: namespace. It exists on the root object of your namespace hierarchy.

Below is the code for the Namespace Pattern:

```
var MyFramework = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};
```

The `namespace` method accepts a single argument which is the fully qualified namespace and returns a reference to the namespace object. Typically, it is used at the beginning of each file declaring its own namespace:

```
var dom = MyFramework.namespace("MyFramework.Utils.Dom");
```

The namespace method is a member of `MyFramework` so `MyFramework` must already exist. This means that the first part of the argument is redundant. You can shorten the passed in name accordingly:

```
var dom = MyFramework.namespace("Utils.Dom");
```

With this reference to you can start building out your code base:

```
var dom = MyFramework.namespace("Utils.Dom");
dom.getFonts = function () { /* ... */ };
dom.count = 0;

alert(dom === MyFramework.Utils.Dom);            // => true
```

In the above example the variable `dom` is an alias for `MyFramework.Util.Dom`. It shortens the code but it gets added to the global object (which is what we tried to avoid with building our own namespace). You can avoid this by adding the code immediately to the namespace returned by the namespace method, like so:

```
MyFramework.namespace("Utils").Dom = {
   getFonts: function () { /* ... */ },
   divCount: 0

};
```

Finally, to add even more flexibility you can assign an *immediate anonymous function* and keep variables and methods private by using the function *closure*, like so:

```
MyFramework.namespace("Utils").Dom = (function () {

   var count = 0;
   var privateVariable = 10;
   var getFonts = function () { /* ... */ };

   return {
```

```
        getFonts: getFonts,
        count: count
    };

})();
```

The details of immediate anonymous functions and closures are discussed in the next section which presents the Module pattern; they will not be discussed here.

The beauty of the approach above is that all variables are private by default. Only the ones that are returned are publicly available.

Just so you are aware: the above pattern is used extensively by the JavaScript optimized Gang-of-Four patterns in our Classic patterns section as well as the Patterns in Action section.

Bookmarklet Pattern

So far, we have seen how to build applications that uses just single global object, the root namespace. Is it possible to bring this down to zero global objects, in other words, leave no global traces at all? The answer is yes, but only in limited circumstances where the code is self-contained and not used by anyone else.

This is the case with *bookmarklets* (the word is a combination of bookmark and applet). Bookmarklets are JavaScript code snippets that are usually designed for one-off functionality on a web page. They have zero global footprint and are implemented as anonymous immediate functions:

```
(function (win) {

    var doc = win.document;
    var nav = win.navigation;

    // ...

})(this);
```

This function executes only once and immediately when the JavaScript engine encounters it. Since no outside references exist, a bookmarklet cannot be used or extended from the outside world. As long as the bookmarklets do not create any global variables or objects they truly have a zero-footprint. Bookmarklets are frequently used to perform one-time initialization of all unobtrusive JavaScript code on a web page.

Here is a simple, but complete, example which can be bookmarked:

```
<a href="javascript:(function (){window.open('http://www.live.com');})();">Live</a>
```

Alternatively, you can separate the JavaScript from HTML, like so:

HTML:

```
<a id="golive">Live</a>
```

JavaScript:

```
<script type="text/javascript">
   $(function () {
       $("#golive").attr("href",
           "javascript:(function (){window.open('http://www.live.com');})();");
   });
</script>
```

The Bookmarklet Pattern looks a lot like the important Module pattern which is discussed next.

The Namespace pattern is used by most JavaScript libraries and frameworks. An example is jQuery which adds jQuery to the global namespace. They also offer an alias for jQuery, which is $, to help shorten the code required to write jQuery selectors. So, both jQuery and $ are added to the global object.

The problem is that other libraries also use $ as their identifier. Examples include Prototype and Mootools. So, if a developer uses both jQuery and Mootools on a single page, they will encounter problems due to a name collision.

Fortunately, most library authors have agreed to a convention that allows you to determine what $ refers to in your code. Each library supports a `noConflict()` method which gives control of the global alias variable ($ in this case) back to anyone who wants to use it. Here is an example:

```
<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">

  // jQuery gives control of $ back to prototype.js
  $.noConflict();

  // Code that uses prototype.js $ can follow here.

  // You can continue using jQuery like so
  jQuery("div").css("color", "red");

</script>
```

This is very helpful as it allows us to use any combination of libraries on a single page.

Module Pattern

We have seen how the Namespace pattern significantly reduces the risk of name collisions. We were even able to reduce the global footprint to zero with the Bookmarklet Pattern. The next step in our progression is the Module pattern which is one of the most significant patterns in JavaScript.

The Module pattern allows developers to organize their applications in modules or functional areas. Most languages allow programs to be organized in logical or functional units called packages, modules, or areas. JavaScript has no special syntax to do so, but the Module pattern is a frequently used technique to organize parts of the program as self-contained units that can be changed independently without affecting any other module.

The Module pattern makes use of immediate anonymous functions and closures. Here is the idea in code:

```
var module = (function () {

    // private area ...

    return {

        // public area ...

    };

}());
```

In its most basic form, the Module is an immediate function whose return value is assigned to a variable (called `module` in our example).

In the private area of the module you define variables and functions which represent the module's private state. It is off-limits to the outside world. This is courtesy of the closure created by the function in which the private state is maintained.

The public area is the returned object literal. It represents the public interface (properties and methods) of the module which is accessible from anywhere in the program.

Let's look at a simple example:

```
var module = (function () {

    // private area

    var count = 0;
    var increment = function () { count++; };
    var decrement = function () { count--; };

    return {

        // public area

        addOne: function () { increment(); },
        subtractOne: function () { decrement(); },
        getCount: function () { return count; }
    };
})();
```

Again, the private area is inaccessible from outside the module. Through closure, the public area *does* have access to all private members. For example, the variable `count` cannot be changed directly, but consumers of the module can indirectly change the value through the `addOne` and `subtractOne` methods on the public interface.

Notice also how a new object is created without the `new` operator by defining an object literal which is returned by the anonymous function.

A variety of the Module pattern is called the Revealing Module pattern. Its purpose is to reveal private variables and/or functions via the public interface. Here is an example:

```
var module = (function () {

    // private area

    var count = 0;
    var increment = function () {count++;};
    var decrement = function () {count--;};

    return {

        // public area

        add: function () { increment(); },
        subtract: function () { decrement(); },
        getCount: function () { return count; },

        inc: increment,                              // revealing method
        dec: decrement                               // revealing method
    };
})();
```

Two new methods were added to the public interface: `inc` and `dec`. They 'reveal' the private functions `increment` and `decrement` to outside consumers of the module.

Here is a slightly modified way to implement the revealing module pattern. First, create all properties and methods as private local variables and functions and, next, only return the ones that should be publicly available. You can use the same or different names for the public members. Here is an example where we use the same names:

```
var module = (function () {

    var count = 0;
    var increment = function () { count++; };
    var decrement = function () { count--; };

    return {
        increment: increment,       // same names
        decrement: decrement
    };
})();
```

In this last example the returned object literal's purpose has been reduced to a mechanism to expose private members as public members. There is no need to have any code in the return value. This is an elegant and tidy way of structuring your immediate functions modules.

There is nothing to prevent you from creating global variables or interact with existing global objects from within the module. However, if you start doing this is becomes difficult to keep track of what is inside and what is outside. Wouldn't it be clearer if we could import the necessary globals into the module and only touch these? The answer is yes, and, in fact, this is common best practice. You import global variables as arguments into the module functions, like so:

```
var module = (function (win, $, undefined) {

    var doc = win.document;
    var nav = win.navigation;

    return {

    };

})(window, jQuery);
```

In this example we are importing 2 global arguments at the bottom of the code: they are window (the global object) and jQuery. Notice that we are also adding a third parameter named undefined at the top, which seems odd. Let's examine each of these arguments.

The first argument named window imports the global object into the module. A couple of properties on the window are assigned to local variables for direct and fast access: they are document and navigation. Following these assignments there is no need to directly access the global object anymore (win or window) and it will be obvious if we do. Also, assigning global properties to local variables will speed up the application as it reduces the need for the JavaScript engine to follow the scope chain to resolve the references.

The second argument jQuery is represented by the $ parameter. As mentioned earlier, there are other libraries that use the same $ global variable name (for example, Prototype) which can cause name conflicts on your page. By including jQuery as an argument and binding it to the parameter named $, then, in your module, you can be sure that $ is in fact an alias for jQuery and nothing else.

The third argument is a variable named undefined. It is included to prevent hackers from overwriting the built-in undefined JavaScript variable (which has a value of undefined by default). Here is how that would work.

As you saw, only two arguments are provided at the end of the module (when invoking the immediate function), whereas the anonymous function accepts three parameters. JavaScript will not complain when too many or too few arguments are provided when calling a function. Any extra arguments will be ignored, and any missing arguments will be undefined. In our module we have a missing argument whose name *and* value will be undefined. So, if someone decided that it would be funny to reassign undefined = true, then this will correct that situation and have no adverse effect on your module; we know that undefined is truly undefined.

The above Module pattern is widely used and is sufficient for most situations. However, when the project starts to grow even larger, you may feel the need to split your module into multiple files. However, you have to be careful because if your JavaScript files are not included in the right order, you run the risk of overwriting all or parts of your own module. This can be solved with the Partial Module pattern. Below is some simple code that shows the idea:

```
var module = (function (self) {

    var top = 10;
```



```
    self.multiply = function (x,y) { return x * y; };
    self.divide = function (x,y) { return x / y; };
    self.getTop = function () { return top; };

    return self;

})(module || {});
```

Notice how module is passed as an argument at the bottom of the module. If no module exists, then an empty object will be passed in. Inside the module the '`self`' reference (that is, the module itself) is enhanced with three methods: `multiply`, `divide`, and `getTop`. The function returns `self` which is then (re-) assigned to a variable called `module`.

A disadvantage of the Partial module is that each file has its own private state which is not shared with the other Partial modules. For example, the top variable in the above code is only visible in this file but not in other files that define the module.

This can be solved by assigning the incoming state to the local state and enhance it with additional properties and methods. Once a module is fully loaded, that is all files are loaded, we remove the public state from the module. If it is not fully loaded, we assign the local state to the self.state which is then carried to the next file. If this sounds complicated, it is, but here is an example of the so-called *private state module*:

```
var module = (function (self) {

    var state = self.state = self.state || {};

    // private state enhancements
    state.index = 9;
    state.getIndex = function () {return this.index;};

    if (isLoaded()) {
        delete self.state;
    } else {
        self.state = state;
    }

    // addional code

    return self;
```

```
})(module || {});
```

We do not recommend this last pattern. The added complexity of reassigning the internal state in each file and the overhead of determining whether all module files have been loaded is too high a price for being able to share private variables.

In reality the limitation of not sharing private state may not be always be bad; it forces you to build self-contained sub-modules each of which has its own responsibility as part of the bigger module.

The Namespace pattern is frequently used in combination with Module pattern. A common approach is as follows: at the beginning of each file a namespace is defined which is then followed by a module that is added to this namespace. Here is an outline of this approach:

```
var events = MyApplication.namespace("Utils.Events");

events.module = (function () {
    // private area ...
    return {
        // public area ...
    };
})();
```

Because the namespace method returns a reference to itself, you can shorten the code a bit by immediately assigning the module to the namespace. An additional benefit is that it also removes the need for a global variable (named `events` in the example above).

```
MyApplication.namespace("Util.Events").module = (function () {

    // private area ...

    return {

        // public area ...

    };
})();
```

This way of chaining code together is another frequently used pattern, called the Chaining pattern which we will discuss in the next section.

Before wrapping up, here are some notable points about the Module pattern.

The Module pattern is JavaScript's manifestation of the Gang-of-Four Singleton pattern. Modules are immediate functions that execute only once, their return value being the module. Only a single instance of the module will ever exist (which is the idea behind Singleton).

As mentioned before, the Module pattern is widely used. You can confirm this by opening up any popular JavaScript library such as jQuery, ExtJs, Backbone, YUI, Ember, Node.js and you'll recognize this pattern right away. Some libraries, such as jQuery and Backbone have their entire code base in a single module. Others, such as ExtJs and YUI are partitioned in namespaces and include numerous modules.

Another reason why Module is so popular in the JavaScript community is because it exhibits many characteristics of good object-oriented design, including: information hiding, encapsulation, high cohesion/loose coupling, open/closed design and extensibility.

Chaining Pattern

The Chaining Pattern enables calling multiple methods on a single object by stringing them together in a single statement, just like a chain. The result is a more concise coding style. Here is an example:

```
var obj = object.method1("arg1").method2("arg2").method3("arg3");
```

The equivalent code, without chaining requires several lines of code with intermediate variables. It would look like this:

```
var obj1 = object.method1("arg1");
var obj2 = obj1.method2("arg2");
```

```
var obj = obj2.method3("arg3");
```

How do you make methods chainable? Actually, it is simple: just return the internal `this` variable, i.e. the current object the method is working on; like so:

```
function func(arg) {

    // do something ...

    return this;
}
```

Let's look at an example:

```
function Calculator(start) {

    var total = start || 0;

    this.add = function (x) { total +=x; return this; };
    this.sub = function (x) { total -=x; return this; };
    this.mul = function (x) { total *=x; return this; };
    this.div = function (x) { total /=x; return this; };

    this.get = function () { return total; };
}
```

Notice that each calculator operation returns `this`. This allows us to chain the operations together in a single statement.

```
var calculator = new Calculator(10).add(8).div(2).mul(4).sub(30).add(100);

alert("value = " + calculator.get());    // => value = 106
```

When a method does not have an obvious return value you should consider returning `this`. If it is never used it will be harmless, but it will allow clients of the method to combine it together with other methods in a single concise (i.e. chained) statement.

A disadvantage of chaining is that it makes debugging harder. If an error occurs in any of your chained methods the debugger will inform you of the line in which the error occurs but *not* the method in which it failed. To dig deeper, you may need to unchain and create temporary variable assignments for each method to be able to pinpoint the failing method.

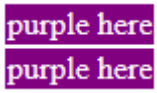
Some developers refer to chaining as *fluent interfaces*, which is just another term for the Chaining Pattern.

Chaining is a widely used pattern. The most popular JavaScript library, which is jQuery, uses it extensively. Consider the following HTML markup and jQuery code:

```
<div id="thediv">
   <table>
      <tr><td class="col"></td></tr>
      <tr><td class="col"></td></tr>
    </table>
</div>

$("#thediv")
    .find("table .col")
    .removeClass(".col")
    .css("background", "purple")
    .css("color", "white")
    .append("<span>purple here</span>");
```

This is an example of jQuery chaining. In a single statement the jQuery selector selects all divs on the page; then for each div it finds tables with elements that have `class='col'`. Next it changes their background to purple, their color to white, and appends text to the element. The results are shown below.



This jQuery example demonstrates that with chaining you can write amazingly compact and concise code. Another benefit to jQuery is that the selector, i.e. the `$("div")` part in the above code, is executed only once. This is significant as the selector is almost always the slowest part of a jQuery statement because it needs to search the entire page.

Adding custom chainable methods to jQuery is remarkably easy. Of course, it must return `this`, which in jQuery is the element that it just worked on. In the code below we are adding a custom turnBlue method to jQuery:

```
$.fn.turnBlue = function () {
     return $(this).css("background", "blue");
};
```

You can now include turnBlue in any jQuery chaining statement. Here is an example:

```
<div id="myDiv" style="background:yellow;width:50px;height:50px;"></div>

$("#myDiv").css("border", "solid 10px red")
           .turnBlue()
           .animate({ "width": "400px" }, 1000);
```

Invocation Pattern

Earlier we had mentioned that each function in JavaScript comes with two bonus arguments: `arguments` and `this`. These are not visible in the formal parameter list but are passed in by the compiler and are available to the entire function scope. The `arguments` parameter holds all arguments that were provided when the function was called. Here we will focus on the second bonus argument: `this`.

All object-oriented programming languages have a similar concept to JavaScript's `this`. It may be called this, me, or self, but they all reference the same thing; the object context that the function executes in. Object context sounds complicated, but an easy way to remember this is that it provides the function an answer to the question where am I, that is, which object instance am I part off?

In JavaScript, the value of this is determined by its Invocation pattern. There are 4 versions: the function invocation pattern, the method invocation pattern, the constructor invocation pattern and the apply invocation pattern. They all differ in how they initialize the value of `this`. We'll look at each of these.

Function Invocation Pattern

First of all, we need to distinguish functions from methods. A standalone function that is not part of an object is called a function or a global function because it resides on the global object. A function that is defined as part of an object is called a method. The '`this`' value in a (global) function is bound to the global object. This is the Function Invocation Pattern. Here is an example:

```
function show() {
    alert(this === window);    // => true
    alert(this)                // => [object Window]
}

show();
```

This confirms that `this` is indeed bound the global object. Note that the above code only works in the browser (not on the server) because window is the name given to the global object in browsers.

Method Invocation Pattern

As mentioned, a method is a function member of an object. When a method is invoked, then the `this` value is bound to the object instance. This is called the Method Invocation Pattern. Consider this code:

```
var calculator = {
    total: 0,
    add: function (x) { this.total += x; },
    sub: function (x) { this.total -= x; },
    show: function () { alert("total = " + this.total); }
}
```

```
calculator.add(10);
calculator.show();       // => total = 10

calculator.sub(6);
calculator.show();       // => total = 4
```

In all three methods (`add`, `sub`, `show`) the value of `this` is bound to the surrounding object instance. These methods have full read-write access to the object's properties (in this example just `this.total`). Outside the object the total property can be accessed by `calculator.total` and inside the object it is accessible as `this.total`.

Constructor Invocation Pattern

Invoking a constructor function with the `new` operator creates a new object. Not surprisingly, the `this` variable will be bound to the newly created object. This is called the Constructor Invocation Pattern.

There are three steps that take place when executing a constructor function. First a new object is created which is then bound to `this`. Next, the function body executes, and properties and methods are added to the object. And, finally, the constructor function implicitly returns the newly created object (you can overwrite the return value by explicitly returning another object but there is rarely a reason to do so).

Here is an example:

```
var Calculator = function () {
    this.total = 0;
    this.add  = function (x) { this.total += x; };
    this.sub = function (x) { this.total -= x; },
    this.show = function () { alert("total = " + this.total); }
}

var calculator = new Calculator();

calculator.add(10);
calculator.show();       // => total = 10

calculator.sub(6);
calculator.show();       // => total = 4
```

Invoking a constructor function without the `new` operator changes the Invocation pattern from a Constructor Invocation pattern to a Function Invocation pattern. Remember that the Function Invocation pattern binds this to the global object, so forgetting the `new` can be a serious problem. We have already discussed this issue and a safeguard in the Constructor pattern.

As an alternative you can move the calculator's methods to its prototype, like so:

```
var Calculator = function () {
    this.total = 0;
}

Calculator.prototype = {
    add: function (x) { this.total += x; },
    sub: function (x) { this.total -= x; },
    show: function () { alert("total = " + this.total); }
}

var calculator = new Calculator();

calculator.add(10);
calculator.show();       // => total = 10

calculator.sub(6);
calculator.show();       // => total = 4
```

In the prototype definition the `this` value is also bound to the calculator object instance.

Apply Invocation Pattern

The Application Invocation pattern offers the most flexibility. You use the `apply` function to invoke a function and then set the `this` value yourself. Two arguments are needed: first the value for `this`, and secondly an array of arguments used to invoke the function. Here is an example:

```
function giveName(name) {
    this.name = name;
};
```

```
function giveName(first, last) {
    this.first = first;
    this.last = last;
};
var person = {};

giveName.apply(person, ["Neil", "Anderson"]);

alert(person.first + " " + person.last);   // => Neil Anderson
```

The `giveName` function assigns a first and last name to an object. We have an empty object named person and then apply the giveName function to it. The first argument is the person object (which is bound to `this`) and a two-element array with the first and last as the second argument. The alert shows that person now has the name Neil Anderson.

In addition to `apply`, JavaScript comes with a very similar built-in function called `call`. It does exactly the same, but the difference is that the second argument is not an array, but a series of individual arguments. Here is an example:

```
function giveName(first, last) {
    this.first = first;
    this.last = last;
};
var person = {};

giveName.call(person, "Neil", "Anderson");

alert(person.first + " " + person.last);   // => Neil Anderson
```

The Apply Invocation pattern is a powerful pattern. You can use it with any object as long as it supports the interface that the function expects. Essentially, you are able to attach 'temporary methods' to all objects that meet a certain interface requirement. Actually, in the above example giveName works for any object as there are no requirements to the object's interface: it creates (or overwrites) a first and last property and assigns the values provided.

If you are coming from .NET, you will recognize the Apply Invocation pattern as .NET's extension methods. Extension methods have a slightly different syntax, but the effects are the same: you get full control over what object `this` is bound to. The name extension

method is very appropriate as you are temporarily extending objects with these methods. Another name you may see in JavaScript for the Apply Invocation pattern is the *borrow method pattern* which also describes the pattern well. Finally, *delegation* has also been used as a name for `apply`/`call` method invocation.

The common thread in the Invocation methods is the binding of `this`. The 4 patterns demonstrate that it can be bound to the global object, the current object instance, a newly created object, or anything we choose.

In object-oriented programming the hidden `this` parameter is an important concept; it is the context (i.e. object) in which the function executes. The value of `this` is bound *just before* the function begins execution. This is called *late binding*, because it takes place at runtime at such a late stage. It makes these functions highly reusable as it allows them to do their work in many different contexts.

Mixin Pattern

The Mixin pattern allows adding functionality to an object. Before getting into mixins lets review which JavaScript patterns exist to extend functionality to objects. They are: 1) Prototypal Inheritance, 2) the Extend Pattern, and 3) the Mixin Pattern. We will discuss each of these.

Prototypal Inheritance

Prototypal inheritance is JavaScript's out-of-the-box inheritance model. If you are coming from a language such as C++, C#, Java, or PHP, you are familiar with class-based inheritance in which classes derive functionality from ancestor classes. As you know, JavaScript does not support classes, but it does offer an object-based inheritance model called prototypal inheritance.

As discussed in the Essentials section, a prototype is an object from which other objects inherit functionality. Each object has a prototype. Prototypes themselves have prototypes, leading to prototype chains. Each object has a built-in internal property, called __proto__ (or [[Prototype]]), which references its prototype object.

In Prototypal inheritance objects inherit from other objects. Suppose you have an object with some properties and methods, and you wish to reuse this functionality. At runtime, you can do this by setting the derived object's prototype property to an instance of the first object, like so:

```
// helper function

function inherit(source) {
    function F() {};
    F.prototype = source;
    return new F();
}

var person = {
    color: "brown",
    say: function () {
        alert("Hi, I have " + this.color + " eyes");
    }
};

var employee = inherit(person);
employee.salary = "$35,000";

employee.say();            // => Hi, I have brown eyes
```

For now, ignore the `inherit` function which will be discussed shortly. First, a person with brown eyes is created. Then a new `employee` object is created that inherits from `person`. This employee has a salary of $35,000. The last line confirms that `employee` indeed inherited both the `color` and `say` members from the `person` object.

The `inherit` function is a helper that creates a new object with its prototype property pointing to the incoming source, or ancestor, object. It creates a new object that is derived from the incoming ancestor object. This function may seem a bit odd, but when you go line by line you will see it is not overly complex. First, an empty constructor function `F` is created (it is capitalized, as all construction functions should be). Its prototype property is assigned the ancestor object. Finally, we return a new instance of `F`, which is empty but with its prototype pointing to the ancestor object. By the way, the `inherit` function is something you may encounter in other libraries where it may have names like `beget` or `object`.

We should mention that this kind of ad hoc prototypal inheritance is unusual because prototypal inheritance relationships are mostly established by constructor functions and their prototype settings. We will see examples of this in this program.

Extend Pattern

The second pattern we examine is the Extend Pattern. This pattern is widely used and is available in many JavaScript frameworks and libraries. It inherits functionality by copying properties from one object into another. Here is how it's done:

```
function extend(source, destination) {
    for (var s in source) {
        if (source.hasOwnProperty(s)) {
            destination[s] = source[s];
        }
    }
}

var person = {
    color: "blue",
    say: function () {
        alert("Hi, I have " + this.color + " eyes");
    }
};

var employee = { salary: "$35,000" };

extend(person, employee);   // the extend pattern

employee.say();             // => Hi, I have blue eyes
```

Skip the `extend` method for now. It is discussed in the next paragraph. A `person` object is created with the `color` blue and a `say` method. Next we create an `employee` object with a `salary` property. The `extend` function copies the members from a source object to a destination object, in this case from a person to an employee. At the end we confirm that `employee` indeed has all members (properties and methods) copied from the `person` object.

The `extend` method gets passed a source object and a destination object. Using a for-in loop it iterates over all properties on the source and copies these over to the destination.

Using the built-in `hasOwnProperty` method we check whether the property is on the object itself or on its prototype. Only properties on the object are copied over.

This version of `extend` performs what is called a *shallow copy*. It does not copy properties that are objects or arrays themselves. Instead, it copies their references and that is usually not what you want.

To make true copies of properties that are objects and arrays we need to recursively call `extend` for each object and array property. This process is called *deep copy*. Here is the code that will do just that:

```
function deepExtend(source, destination) {
    for (var s in source) {
        if (source.hasOwnProperty(s)) {
            if (typeof source[s] === "object") {
                destination[s] = isArray ? [] : {};
                deepExtend(source[s],destination[s]);
            } else {
               destination[s] = source[s];
            }
        }
    }
    function isArray(o) {
        return (typeof o === "[object Array]");
    }
}

var person = {
    name: "Karen",
    address: {
       street: "1 Main St",
       city: "Baltimore"
    },
    scores: [212, 310, 89],
    say: function () {
         alert(this.name + ", " + this.address.street + ", " +
               this.address.city + ", " + this.scores );
    }
};
var employee = { salary: "$45,000" };
deepExtend(person, employee);

employee.say();  // => Karen, 1 Main St, Baltimore, 212, 310, 89
```

Like `extend`, the function `deepExtend` iterates over the properties of the source object. However, `deepExtend` includes a property type check for type object. If true, we have an object or an array. The `isArray` helper method tells us if this is an array or not. If it is an empty array property gets added to destination, otherwise an empty object. The `deepExtend` is then recursively called which will copy the object or array. Note that function objects are still copied by reference which is preferable because it saves memory.

The `person` object is a complex object with multiple member types: a primitive property, an object property, an array property and a function object property (i.e. method). The function `deepExtend` successfully copies all these from `person` to `employee` (source to destination). We confirm this by invoking the `say` method which displays all `person` properties.

Mixin Pattern

The Mixin pattern goes one step further in that it copies properties from one or more source objects. All properties get 'mixed' together in this new object. Here is an example of an array of employees. The mixin function copies properties from two objects into the array object: `sortable` and `enumerable`, which add sorting and enumeration functionality to the array.

```
function mixin(sources, destination) {
    for (var i = 0, len = sources.length; i < len; i++) {
        var source = sources[i];
        for (var s in source) {
            if (source.hasOwnProperty(s)) {
                destination[s] = source[s];
            }
        }
    }
}

var Employee = function (name, gender, age) {
    this.name = name;
    this.age = age;
    this.gender = gender;
```

```
}

var employees = [];

employees.push(new Employee("John", "Male", 33));
employees.push(new Employee("Peter", "Male", 45));
employees.push(new Employee("Ann", "Female", 32));
employees.push(new Employee("Tiffany", "Female", 19));

var sortable = {
    sortAge: function () {
        this.sort(function (a, b) {
            return a.age - b.age;
        });
    }
};

var enumerable = {
    each: function (callback) {
        for (var i = 0, len = this.length; i < len; i++) {
            callback(this[i]);
        }
    }
};

mixin([sortable, enumerable], employees);

employees.sortAge();
employees.each(function (item) {
    alert(item.name + " " + item.age);    // => Tiffany 19, Ann 32,
                                          //    John 33, Peter 45
});
```

This example is very interesting. First off, the mixin function copies properties from multiple sources into a destination object (rather than a single source which was the case in the Extend pattern). The source objects are passed in as the first parameter which is an array of objects.

Our destination object is an array with 4 employees. The sources are named `sortable` and `enumerable` each of which has just a single method. The `mixin` function takes in these sources and copies their methods into the array of employees. By the way, if the source

object names sound like interface names, then you are correct; this is by design because we are adding a `sortable` method and an `enumeration` method to the employees.

Following the mixin action we have an array that can sort its employees by age. The array also has an `each` method that iterates over the employees and invokes a custom callback function that displays each employee's name and age.

A possible variation of the above `mixin` function is `deepMixin`, which just like the earlier `deepExtend` performs deep copies of properties of all source objects provided. If you need a `deepMixin` in your own project, then you can easily create this based on the two prior code snippets.

Functional Mixin Pattern

The Functional Mixin pattern is the logical next step for Mixin. The idea is to group related methods together in terms of functional categories. Then use these functional groups and apply these to the receiver's prototype. In this example our group is called `sortable` and has just one method, but others could have been added.

```
var Employee = function (name, gender, age) {
    this.name = name;
    this.age = age;
    this.gender = gender;
}

var Employees = function () { };
Employees.prototype = [];

var sortable = function () {
    this.sortAge = function () {
        this.sort(function (a, b) {
            return a.age - b.age;
        });
    };
    return this;
};

var enumerable = function () {
    this.each = function (callback) {
        for (var i = 0, len = this.length; i < len; i++) {
            callback(this[i]);
```

```
        }
    };
    return this;
};


sortable.call(Employees.prototype);
enumerable.call(Employees.prototype);

var employees = new Employees();

employees.push(new Employee("John", "Male", 33));
employees.push(new Employee("Peter", "Male", 45));
employees.push(new Employee("Ann", "Female", 32));
employees.push(new Employee("Tiffany", "Female", 19));

employees.sortAge();

employees.each(function (item) {
    alert(item.name + " " + item.age);   // => Tiffany 19, Ann 32,
                                         //    John 33, Peter 45
});
```

In this example there is no `mixin` method. Instead, we are using a `call` method to add a group of methods to the Employee's prototype. This will add all methods to the object's prototype. Note that the `sortable` and `enumerable` objects return `this`, which is important. Notice also that the employee array now derives from Array, rather than being an array instance. This was done to facilitate setting the employee array's prototype.

We've come full circle because the functional mixin pattern involves prototypal inheritance. The functional mixin is an interesting idea because it allows us to change the prototype of a constructor function just before creating a new instance. This facilitates the creation of custom objects that have a certain built-in behavior.

Monkeypatch Pattern

The Monkeypatch pattern allows you to modify code at runtime without changing the original source code. The use of this pattern is somewhat controversial, but it can be very helpful in situations where you need to correct errors in 3rd party controls or libraries until

a fix becomes available (and at that point you can remove your Monkeypatch). Monkeypatch is also referred to as *Duck Punching*.

How is monkey patching implemented? Actually, it is fairly simple: it works by replacing existing properties or methods on an object. Let's look at an example. We have a `person` object that has a `say` method which displays its name and age. The developer wants to make the person appear a few years younger and decides to monkey patch the `say` method:

```
var person = {
    name: "Kevin",
    age: 39,
    say: function () {
        alert(this.name + ", " + this.age + " years old");
    }
};

person.say();   // => Kevin, 39 years old

// monkey patching say

person.say = function () {
    var younger = this.age - 5;
    alert(this.name + ", " + younger + " years old");
};

person.say();   // => Kevin, 34 years old
```

Because of the dynamic nature of JavaScript there is nothing that prevents developers from overwriting or modifying public members on an existing object.

In the above example we wholesale replaced the original `say` with a new definition. What is more common though is that the original method needs a slight adjustment possibly just before and after the method execution without changing the original functionality. Here is one way to handle this situation:

```
var person = {
    name: "Kevin",
    age: 39,
    say: function () {
```

```
        alert(this.name + ", " + this.age + " years old");
    }
};

person.say();   // => Kevin, 39 years old

// monkey patching say

var oldSay = person.say;

person.say = function () {
    this.age -= 5;
    oldSay.call(this);
    this.age += 5;
};

person.say();   // => Kevin, 34 years old
```

First, the original `say` method is saved off into a variable. Next, the `person.say` method is redefined in which the age is adjusted just before calling the original method and then restored to its original value.

The above code adds a variable to the global namespace which we usually try to avoid. We can correct this by wrapping the monkey patch in an immediate function. Its closure will hold a reference to the original method which is accessible to the redefined `person.say` method. Here is what this looks like:

```
var person = {
    name: "Kevin",
    age: 39,
    say: function () {
        alert(this.name + ", " + this.age + " years old");
    }
};

person.say();   // => Kevin, 39 years old

// monkey patching say with immediate function

(function () {

     var oldSay = person.say;
```

```
    person.say = function () {
        this.age -= 5;
        oldSay.call(this);
        this.age += 5;
    };
})();

person.say();   // => Kevin, 34 years old
```

That looks good. The function is redefined within an immediate function and its closure maintains a local variable called `oldSay` which gets assigned to the original `say` method. We were able to inject pre-processing and post-processing functionality surrounding the original `say` method call.

Did you notice that the source code of the `person` object (in this case the object literal at the top of the code snippet) was not touched at all? And, yet, we were able to alter its functionality. This is the Monkeypatch pattern in all its glory. It's a powerful pattern that allows developers to make runtime changes to any object, whether you own it or not.

What if, in the above example, the method was defined on the object's prototype rather than on the object itself? In that case we are not really overwriting the original but masking it instead. Remember that the JavaScript engine searches the prototype chain starting at the object itself until if finds the requested member.

However, you can also monkey patch the prototype object so that the change applies to all instances that share the same prototype object. Here is what this looks like:

```
var Person = function (name, age) {
    this.name = name;
    this.age = age;
};

Person.prototype.say = function () {
    alert(this.name + ", " + this.age + " years old");
}

var kevin = new Person("Kevin", 39);
var sonya = new Person("Sonya", 50);
```

```
kevin.say();   // => Kevin, 39 years old
sonya.say();   // => Sonya, 50 years old

// monkey patching prototype object

(function () {

     var oldSay = Person.prototype.say;

     Person.prototype.say = function () {
         this.age -= 5;
         oldSay.call(this);
         this.age += 5;
     };
})();

kevin.say();   // => Kevin, 34 years old
sonya.say();   // => Sonya, 45 years old
```

Instead of an object literal we now have a `Person` constructor function. We create two persons. Their `say` method is defined on the prototype object. By monkey patching the method on the prototype we affect all person instances. Both Kevin and Sonya appear 5 years younger than they are.

Again, this is a powerful pattern, but it is not without its perils. If you are not clear about the intricacies of the original code, you may unintentionally introduce bugs that affect other parts of the application. It gets worse when multiple patches are applied as they tend to interact in unpredictable, combinatorial ways.

It has been said that with the C language you can shoot yourself in the foot. With C++ you can shoot your leg off. With JavaScript and monkey patching you can blow yourself up and the whole team with it. Clearly with all this power comes great responsibility.

In reality monkey patching is not a widely used practice in JavaScript (at least for now). Remember, monkey patching can be applied to built-in objects as well, so you have the ability to redefine the language's core features turning it into a different language altogether with a its own syntax. In other languages, such as Ruby and Python, dynamic extensions of core classes (i.e. monkey patching) is more commonly used. There are developers in those communities that are pushing back hard, stating that monkey patching is 'destroying the language'.

Some language purists will tell you that monkey patching is a hack that should never be allowed. However, it can be a valuable tool to temporarily correct bugs in 3rd party components or libraries over which you have no direct control. It is better to use it sparingly and when you do, be very careful, document it well, and share this information with your team. Then, as soon as a fix becomes available you immediately remove it. Fortunately, removing monkey patches is very easy.

Lazy Load Pattern

The Lazy Load pattern allows you to load data or code only when absolutely necessary. It is designed to limit the use of resources such as memory, CPU cycles, and/or network traffic. It is a very helpful technique in providing end-users with a better experience, for example, faster display results when multiple items need to be displayed but only a few will do initially. An alternative name for this pattern is *Just-in-Time Pattern*.

There are two variations of this pattern: *Lazy Initialization* and the *Ghost Pattern*. Lazy initialization occurs when an object is lazily loaded. Initially the object will be null or undefined. Upon requesting the object, the code checks if the object exists and if it doesn't then it loads it and returns its reference. Here is an example:

```
var employee = {

    name: "Victor",
    salaryHistory: null,

    getSalaryHistory: function () {
       if (this.salaryHistory === null) {                 // Lazy initializion
           this.salaryHistory = this.initializeHistory();
       }
       return this.salaryHistory;
    },

    initializeHistory: function () {
        var history = {};
        history [2009] = "$34,000";
        history [2010] = "$36,500";
        history [2011] = "$44,000";
        history [2012] = "$45,500";
        history [2013] = "$51,000";
```

```
        return history;
    }
};

var history = employee.getSalaryHistory();

alert("2010: " + history[2010]);   // => 2010: $36,500
```

The `employee` object is initially created without salary history. Perhaps this data is used infrequently and with many employees you can save yourself some significant memory and server access. In case the end-user needs salary data for employee Victor then the `getSalaryHistory` method checks if it is already loaded; if not, it will load it up and returns the array with historical salary data. It loads the data only when absolutely necessary.

The second variety called Ghost Pattern is rather interesting. A Ghost object is one that is only partially loaded and when called upon will load itself completely.

An example where the Ghost Pattern will be helpful is a web page that displays a long list of employees. For each employee you have a partially populated employee object with only the id, name, and thumbnail image loaded. These are 'ghost' versions of the fully loaded employee objects.

To get the full details on an employee, the end-user clicks on the employee image. At that time the ghost object starts loading itself up with the remaining data to turn itself into a fully populated employee object. This loading most likely takes place via an Ajax call in which the full employee record is retrieved from a back-end database. Here is what this looks like:

```
var Employee = function (id, name, thumbnail) {
    this.id = id;
    this.name = name;
    this.thumbnail = thumbnail;

    this.ghost = true;

    this.getDepartment = function () {
        if (this.ghost) this.load();
        return this.department;
    },
    this.getSalary = function () {
```

```
        if (this.ghost) this.load();
        return this.salary;
    }
    this.getBenefits = function () {
        if (this.ghost) this.load();
        return this.benefits;
    }

    this.load = function () {

        // load using an ajax call

        this.department = "Product Development";
        this.salary = "$66,400";
        this.benefits = "401(k); 21 vacation days";

        this.ghost = false;
    };
};

var employee = new Employee(211, "Nicolas Vick", "vick.jpg");

alert(employee.name);

// get more details on employee..

alert(employee.getDepartment());            // this is when it loads
alert(employee.getSalary());                // fully loaded
alert(employee.getBenefits());              // fully loaded
```

This code does not account for the asynchronous nature of Ajax. It requires some sort of callback mechanism. But you get the idea. The beauty of the Ghost Pattern is that it is usually totally transparent to the client programmer.

Of course, you have to balance the initial performance of loading the entire page versus a slower display of employee details. There are smart techniques that predict which employees are likely to require details and data for these can be pre-fetched, so some sort of balance is found between initial fast performance (for entire employee list) and later fast performance (for employee detail). All this depends on your particular situation.

Multiton Pattern

The Multiton pattern is from the same lineage as the Singleton pattern. Singleton limits the number of object instances to one and Multiton limits it to any given number. To get a better appreciation for Multiton you may want to review the Singleton pattern which is listed in the Classic Patterns section.

Two varieties of Multiton exist. The first one is similar to Singleton, but instead of limiting the number of concurrent instances to 1 it limits it to n, where n is a number you specify. We will call this *N-ton*. The second variety of Multiton is a map (or dictionary) of Singletons, where each can be retrieved by a key. Essentially these are name/value pairs where named instances of Singletons are stored. This last construct is sometimes referred to as a *Registry of Singletons*. We will look at each one.

The N-ton model is helpful when, for whatever reason, you need to limit the number of instances. This is usually related to some limited resource (such as a server or a database) that you are trying to manage. Here is the code for the N-ton model:

```
var Multiton = function (limit) {
    var limit = limit;
    var instances = [];

    var Instance = function () {
        // instance code here
        this.say = function () {
             alert("I am an instance" );
        };
    };

    return {
        getInstance: function () {
            if (instances.length === 0) {
                for (var i = 0; i < limit; i++) {
                    instances.push(new Instance());
                }
            }
            var random = Math.floor(Math.random() * limit);
            return instances[random];
        }
    };
```

```
};

var multiton = new Multiton(6);

var m1 = multiton.getInstance();
var m2 = multiton.getInstance();
var m3 = multiton.getInstance();
var m4 = multiton.getInstance();

m1.say();       // => I am an instance
m2.say();       // => I am an instance
```

In this example the Multiton manages a maximum of 6 instances. When an instance is requested with `getInstance` a randomly selected instance is returned. Notice that the constructor function named `Instance` is privately scoped within Multiton and can only be called from within Multiton. The `Instance` constructor is where you specify instance specific properties and methods.

An example of the Registry of Singletons variety of the Multiton pattern is demonstrated next.

```
var Multiton = function () {
    var instances = {};

    var Instance = function (id) {
        // instance code here
        this.id = id;
        this.say = function () {
             alert("Instance for Server " + this.id );
        };
    };

    return {
        getInstance: function (id) {
            if (Object.keys(instances).length === 0) {
                for (var i = 1; i < 5; i++) {
                    instances[i] = new Instance(i);
                }
            }
            return instances[id];
        }
    };
```

```
};

var multiton = new Multiton();

var server1 = multiton.getInstance(1);
var server2 = multiton.getInstance(2);
var server3 = multiton.getInstance(3);
var server4 = multiton.getInstance(4);

server1.say();       // => Instance for Server 1
server3.say();       // => Instance for Server 3
```

This code is quite similar to the previous variety only that we ask for specific instances that are keyed by a unique id (hardcoded from 1 to 4). Each singleton represents a server. The `instances` private object maintains the name/value pairs (in the N-ton implementation this was an array).

Partial Pattern

The Partial pattern in JavaScript comes in 2 flavors: Partial Functions and Partial Objects. Partial Functions allow you to pre-run parts of a function by returning a so-called partial function. This partial function offers reusability and increased performance. The Partial Object Pattern is about organizing your object code over separate files so that 1) multiple developers can work on an object at the same time, or 2) to allow for code generation and have an object part (i.e. file) that cannot be changed and a part that allows customization. We also have Partial Modules, but these have already been discussed under the Module pattern.

Partial Function Pattern

The Partial Function pattern is also referred to as Partial Application (as in applying, or invoking, a function). A related term is *currying* which we will also look at.

In JavaScript you can invoke, meaning execute, a function. Another term used in functional languages is you *apply* a function. As you know, JavaScript does have a built-in `apply` method which invokes a function with two parameters: the first one will be

bound to `this` and the second one is an array of parameters that will be 'applied' to the function.

Function Application simply means that you are applying parameters to a function. Partial Application occurs when you apply a partial set of parameters to a function. In the latter scenario the function remembers the first set of arguments and when called a second time it appends the new ones to the first ones. This is implemented by a function returning a function and using closures. Let's look at an example:

```
function concat(first, last) {
    if (typeof last === "undefined") {
        return function (last) {             // partial function
            return first + " " + last;
        };
    }
    return first + " " + last;
}

var mary = concat("Mary", "Milliken");      // full application
alert(mary);                                // => Mary Milliken

var john = concat("John");                  // partial application
var jones = john("Jones");
var sellers = john("Watson");

alert(jones);                               // => John Jones
alert(sellers);                             // => John Watson
```

The function `concat` concatenates first and last names together. It can be called with one or two parameters. When called with two parameters it returns first and last concatenated. However, if only the first name is provided it returns a 'transformed' function that remembers the first name by storing the argument in its closure (recall that local variables *and* arguments are saved in a function's closure). This transformed function is the partial function.

Running `concat` with two arguments executes as expected, returning the string "Mary Milliken". When running concat with one argument a partial function is returned which remembers the first name "John". We then call the partial function, named `john`, with

only a last name. This returns the original first name concatenated to the last name, i.e. John Jones and John Watson in the example.

This process of transforming one function to another is called *currying* (named after a mathematician by name of Haskell Curry). Partial functions and currying are useful when you find yourself calling the same function over and over with the same arguments. It is also useful if the first half of the function requires heavy duty processing of which the results can be saved for all subsequent calls. Here is an example of the last situation where we cache a dataset that is retrieved from the server only once:

```
function retrieve(query, id) {

    if (typeof id === "undefined") {
        var cache = server(query);
        return function (id) {            // partial function
            return cache[id];
        };
    }
    return server(query)[id];             // return for 'full' function

    function server(query) {              // helper, simulates database access
       var items = {};
       items[0] = "Moe";
       items[1] = "Larry";
       items[2] = "Curly";
       return items;
    }
}

alert(retrieve("stooges", 2));      // => Curly (no caching)

var partial = retrieve("stooges");

alert(partial(0));         // => Moe
alert(partial(1));         // => Larry
alert(partial(2));         // => Curly
```

The function named `retrieve` returns a partial function if called with just one argument (the query). If so, it calls the `server` method, which simulates a server call that retrieves data from a database and stores the results in a local cache. The `cache` variable is an object

which holds name/value pairs of the 3 Stooges. The cache is stored in the closure. The partial function returns the name of the stooge that matches the id value.

Immediately after the `retrieve` function declaration we call retrieve with 2 parameters. This will perform a regular retrieve, meaning it retrieves the data from the (simulated) server and returns the requested stooge. This is rather inefficient because for each call the server needs to be accessed and a full set of stooge records is retrieved. Only one of those is returned. The partial approach is far more efficient and effective.

When initially called with just the `query` argument, the function retrieves the data from the server, stores it in the cache, and returns a partial function that uses the cached values. Any subsequent call on the partial method simply pulls the appropriate record out of cache and returns it. We could do this trick for multiple data sets by creating partial functions for different queries.

The examples above are custom partial functions that work well for a particular scenario. Is there a way that partial functionality can be made more generic, one that remembers a first batch of parameters and then merges it with a second batch in the transformed function? The answer is yes, and here is how:

```
function makePartial(fn) {
    return partial.apply(this,
        [].concat.apply([fn, this],
            [].slice.call(arguments, 1)));            // 1st batch of arguments
    function partial (fn, scope) {
        var args = [].slice.call(arguments, 2);       // 2nd batch of arguments
        return function () {
            return fn.apply(scope,
                [].concat.apply(args, arguments));    // all arguments
        };
    }
}
```

Notice that we use [] in several places. It is an array literal which creates an array instance when called. Replacing [] with Array.prototype would make it a bit more efficient but at the cost of readability which is why we choose to use [].

Granted the above code is dense, but here it goes. The `makePartial` function accepts a function (`fn`) and any number of additional arguments (the 1st batch). Note that the additional arguments are not shown as parameters in the function declaration but are visible in the built-in arguments object. Next, `makePartial` returns a partial function that will accept additional arguments (the 2nd batch). The first batch and function (`fn`) is stored in the closure. When the user calls this partial function, it merges the first batch of arguments with the second batch and invokes the function passed in originally.

Let's see an example of how it is used.

```
function add(v, w, x, y, z) {
    return v + w + x + y + z;
}

var add3 = makePartial(add, 2, 2, 10);

var total = add3(10, 6);
alert(total);                  // => 30
```

The function `add` accepts 5 values and adds these together. The variable `add3` is a partial function that already knows 3 of the 5 argument values. When invoking `add3` you pass the 2 remaining arguments and the total gets returned.

Partial Object Pattern

Next, let's look at partial objects, which, by the way, is unrelated to partial functions. A common problem with larger JavaScript projects is that multiple developers will work on the same object concurrently. The Partial object pattern alleviates this problem by allowing the object's code to be split over multiple files, each file holding only part of the object (i.e. partial object).

There are different options to build Partial Objects. Here is one way:

File 1

```
function Customer(name, street, /* ... */ ) {
    this.name = name;
    this.street = street;
```

```
    //...
}
```

File 2

```
Customer.prototype.validate( /* ... */ )
   // ...
}

Customer.prototype.insert( /* ... */ )
   // ...
}

Customer.prototype.update( /* ... */ )
   // ...
}
```

File 3

```
Customer.prototype.checkCredit( /* ... */ )
   // ...
}

Customer.prototype.sendInvoice( /* ... */ )
   // ...
}

Customer.prototype.acceptPayment( /* ... */ )
   // ...
}
```

The customer object has been spread over 3 files, organized by functional area. At runtime, when all files are brought together and are loaded you can use the customer object like any regular object:

```
var customer = new Customer("IBM", "123 Main Highway", ...);

customer.sendInvoice("$4,000", ...);
```

Another area where partial objects may be helpful is in code generation scenarios. This technique is frequently used in .NET which supports true partial classes, that is, a single class can be spread out by multiple partial classes in different files. These code-generators often use classes that are made available as two partial classes. One partial class contains core functionality and should not be touched by developers. The other partial class offers extension points where developers can add their custom code. The first one is created by the code generator and overwritten when the code generator runs again (for example to reflect data model changes). However, the second one is never overwritten by the code-generator and should only be touched by the developers.

These extension points in .NET can be in the form of event handlers or partial methods (not related to JavaScript Partial Functions). At compile time the two partials are merged to form a single class which runs and behaves like a normal class.

This approach works really well. The question is can we do something similar in JavaScript? The answer is yes. Let's see how.

File 1. -- code generated

```
var ClickableControl = function () {

    var handlers = [];

    this.register = function (handler) {
        handlers.push(handler);
    }

    this.click = function (x, y) {
        for (var i = 0, len = handlers.length; i < len; i++) {
            handlers[i].apply(this,[x,y]);
        }
    }
}

var control = new ClickableControl();
```

File 2. -- hand coded

```
function showClick (x,y) {
    alert("Location: " + x + ", " + y);
}

control.register(showClick);
```

Here is a user or browser generated event

```
control.click(250, 120);    // => Location 250, 120
control.click(248, 123);    // => Location 248, 123
```

If you have already read the Classic Patterns section, you may recognize the Observer Pattern or callback pattern in this example. File 1 defines a clickable control that allows registration for click handlers. The click handlers are provided in File 2 which is hand coded. When the app generated clicks, all registered click handlers are invoked one after the other. This example shows we are able to make a clean separation of the 'code-generated' control (File 1) and the custom event handlers (File 2).

As far as code-generation this example is a bit contrived, but it conveys the general idea of separating objects in non-changeable and changeable files. This separation is not limited to events; it is perfectly fine to use what almost seems like 'abstract' methods. An example of where this is used is Backbone, a popular open source MVC framework (discussed in the Model View Patterns section). Backbone provides a base Model object that can be extended with a custom object like so:

```
var car = Backbone.Model.extend({

    urlRoot: "cars",

    initialize: function () {
        alert("I am a car");
    },

    defaults: {
        id: 0,
        model: "",
        make: "",
        year: "",
        description: ""
```

```
    }
});
```

Backbone.Model provides the base functionality (in this case Model behavior in a MVC context) and offers a series of properties and methods that can be overridden with custom code. For example, it has an empty `initialize` method on its prototype, but it will invoke the overridden one if present. It checks whether `urlRoot` is defined and has a value, and similarly for `defaults`.

Backbone is not code generated but it provides predefined hooks to extend the standard object. It is not far-fetched to think of the Model being code generated which then provides the hooks that can be added. These hooks are added in a different file, that is, the partial object.