

Unearthing the Excellence in JavaScript



JavaScript: The Good Parts

O'REILLY®

YAHOO! PRESS

Douglas Crockford

JavaScript: The Good Parts

by Douglas Crockford

Copyright © 2008 Yahoo! Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Simon St.Laurent

Production Editor: Sumita Mukherji

Copyeditor: Genevieve d'Entremont

Proofreader: Sumita Mukherji

Indexer: Julie Hawks

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Printing History:

May 2008: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *JavaScript: The Good Parts*, the image of a Plain Tiger butterfly, and related trade dress are trademarks of O'Reilly Media, Inc.

Java™ is a trademark of Sun Microsystems, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51774-8

[M]

This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.

CHAPTER 4

Functions

*Why, every fault's condemn'd ere it be done:
Mine were the very cipher of a function...*
—William Shakespeare, *Measure for Measure*

The best thing about JavaScript is its implementation of functions. It got almost everything right. But, as you should expect with JavaScript, it didn't get everything right.

A function encloses a set of statements. Functions are the fundamental modular unit of JavaScript. They are used for code reuse, information hiding, and composition. Functions are used to specify the behavior of objects. Generally, the craft of programming is the factoring of a set of requirements into a set of functions and data structures.

Function Objects

Functions in JavaScript are objects. Objects are collections of name/value pairs having a hidden link to a prototype object. Objects produced from object literals are linked to `Object.prototype`. Function objects are linked to `Function.prototype` (which is itself linked to `Object.prototype`). Every function is also created with two additional hidden properties: the function's context and the code that implements the function's behavior.

Every function object is also created with a `prototype` property. Its value is an object with a `constructor` property whose value is the function. This is distinct from the hidden link to `Function.prototype`. The meaning of this convoluted construction will be revealed in the next chapter.

Since functions are objects, they can be used like any other value. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and functions can be returned from functions. Also, since functions are objects, functions can have methods.

The thing that is special about functions is that they can be invoked.

Function Literal

Function objects are created with function literals:

```
// Create a variable called add and store a function
// in it that adds two numbers.

var add = function (a, b) {
    return a + b;
};
```

A function literal has four parts. The first part is the reserved word `function`.

The optional second part is the function's name. The function can use its name to call itself recursively. The name can also be used by debuggers and development tools to identify the function. If a function is not given a name, as shown in the previous example, it is said to be *anonymous*.

The third part is the set of parameters of the function, wrapped in parentheses. Within the parentheses is a set of zero or more parameter names, separated by commas. These names will be defined as variables in the function. Unlike ordinary variables, instead of being initialized to `undefined`, they will be initialized to the arguments supplied when the function is invoked.

The fourth part is a set of statements wrapped in curly braces. These statements are the body of the function. They are executed when the function is invoked.

A function literal can appear anywhere that an expression can appear. Functions can be defined inside of other functions. An inner function of course has access to its parameters and variables. An inner function also enjoys access to the parameters and variables of the functions it is nested within. The function object created by a function literal contains a link to that outer context. This is called *closure*. This is the source of enormous expressive power.

Invocation

Invoking a function suspends the execution of the current function, passing control and parameters to the new function. In addition to the declared parameters, every function receives two additional parameters: `this` and `arguments`. The `this` parameter is very important in object oriented programming, and its value is determined by the *invocation pattern*. There are four patterns of invocation in JavaScript: the method invocation pattern, the function invocation pattern, the constructor invocation pattern, and the apply invocation pattern. The patterns differ in how the bonus parameter `this` is initialized.

The invocation operator is a pair of parentheses that follow any expression that produces a function value. The parentheses can contain zero or more expressions, separated by commas. Each expression produces one argument value. Each of the argument values will be assigned to the function's parameter names. There is no runtime error when the number of arguments and the number of parameters do not match. If there are too many argument values, the extra argument values will be ignored. If there are too few argument values, the `undefined` value will be substituted for the missing values. There is no type checking on the argument values: any type of value can be passed to any parameter.

The Method Invocation Pattern

When a function is stored as a property of an object, we call it a *method*. When a method is invoked, this is bound to that object. If an invocation expression contains a refinement (that is, a `.` dot expression or [*subscript*] expression), it is invoked as a method:

```
// Create myObject. It has a value and an increment
// method. The increment method takes an optional
// parameter. If the argument is not a number, then 1
// is used as the default.

var myObject = {
  value: 0;
  increment: function (inc) {
    this.value += typeof inc === 'number' ? inc : 1;
  }
};

myObject.increment();
document.writeln(myObject.value);    // 1

myObject.increment(2);
document.writeln(myObject.value);    // 3
```

A method can use `this` to access the object so that it can retrieve values from the object or modify the object. The binding of `this` to the object happens at invocation time. This very late binding makes functions that use `this` highly reusable. Methods that get their object context from `this` are called *public methods*.

The Function Invocation Pattern

When a function is not the property of an object, then it is invoked as a function:

```
var sum = add(3, 4);    // sum is 7
```

When a function is invoked with this pattern, `this` is bound to the global object. This was a mistake in the design of the language. Had the language been designed correctly, when the inner function is invoked, `this` would still be bound to the `this`

variable of the outer function. A consequence of this error is that a method cannot employ an inner function to help it do its work because the inner function does not share the method's access to the object as its `this` is bound to the wrong value. Fortunately, there is an easy workaround. If the method defines a variable and assigns it the value of `this`, the inner function will have access to `this` through that variable. By convention, the name of that variable is `that`:

```
// Augment myObject with a double method.

myObject.double = function () {
    var that = this;    // Workaround.

    var helper = function () {
        that.value = add(that.value, that.value)
    };

    helper();    // Invoke helper as a function.
};

// Invoke double as a method.

myObject.double();
document.writeln(myObject.getValue());    // 6
```

The Constructor Invocation Pattern

JavaScript is a *prototypal* inheritance language. That means that objects can inherit properties directly from other objects. The language is class-free.

This is a radical departure from the current fashion. Most languages today are *classical*. Prototypal inheritance is powerfully expressive, but is not widely understood. JavaScript itself is not confident in its prototypal nature, so it offers an object-making syntax that is reminiscent of the classical languages. Few classical programmers found prototypal inheritance to be acceptable, and classically inspired syntax obscures the language's true prototypal nature. It is the worst of both worlds.

If a function is invoked with the `new` prefix, then a new object will be created with a hidden link to the value of the function's prototype member, and `this` will be bound to that new object.

The `new` prefix also changes the behavior of the return statement. We will see more about that next.

```
// Create a constructor function called Quo.
// It makes an object with a status property.

var Quo = function (string) {
    this.status = string;
};

// Give all instances of Quo a public method
```

```

// called get_status.

Quo.prototype.get_status = function () {
    return this.status;
};

// Make an instance of Quo.

var myQuo = new Quo("confused");

document.writeln(myQuo.get_status()); // confused

```

Functions that are intended to be used with the `new` prefix are called *constructors*. By convention, they are kept in variables with a capitalized name. If a constructor is called without the `new` prefix, very bad things can happen without a compile-time or runtime warning, so the capitalization convention is really important.

Use of this style of constructor functions is not recommended. We will see better alternatives in the next chapter.

The Apply Invocation Pattern

Because JavaScript is a functional object-oriented language, functions can have methods.

The `apply` method lets us construct an array of arguments to use to invoke a function. It also lets us choose the value of `this`. The `apply` method takes two parameters. The first is the value that should be bound to `this`. The second is an array of parameters.

```

// Make an array of 2 numbers and add them.

var array = [3, 4];
var sum = add.apply(null, array); // sum is 7

// Make an object with a status member.

var statusObject = {
    status: 'A-OK'
};

// statusObject does not inherit from Quo.prototype,
// but we can invoke the get_status method on
// statusObject even though statusObject does not have
// a get_status method.

var status = Quo.prototype.get_status.apply(statusObject);
// status is 'A-OK'

```

Arguments

A bonus parameter that is available to functions when they are invoked is the `arguments` array. It gives the function access to all of the arguments that were supplied with the invocation, including excess arguments that were not assigned to parameters. This makes it possible to write functions that take an unspecified number of parameters:

```
// Make a function that adds a lot of stuff.

// Note that defining the variable sum inside of
// the function does not interfere with the sum
// defined outside of the function. The function
// only sees the inner one.

var sum = function () {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};

document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

This is not a particularly useful pattern. In Chapter 6, we will see how we can add a similar method to an array.

Because of a design error, `arguments` is not really an array. It is an array-like object. `arguments` has a `length` property, but it lacks all of the array methods. We will see a consequence of that design error at the end of this chapter.

Return

When a function is invoked, it begins execution with the first statement, and ends when it hits the `}` that closes the function body. That causes the function to return control to the part of the program that invoked the function.

The `return` statement can be used to cause the function to return early. When `return` is executed, the function returns immediately without executing the remaining statements.

A function always returns a value. If the `return` value is not specified, then `undefined` is returned.

If the function was invoked with the `new` prefix and the `return` value is not an object, then `this` (the new object) is returned instead.

Exceptions

JavaScript provides an exception handling mechanism. Exceptions are unusual (but not completely unexpected) mishaps that interfere with the normal flow of a program. When such a mishap is detected, your program should throw an exception:

```
var add = function (a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw {
      name: 'TypeError',
      message: 'add needs numbers'
    }
  }
  return a + b;
}
```

The `throw` statement interrupts execution of the function. It should be given an exception object containing a `name` property that identifies the type of the exception, and a descriptive `message` property. You can also add other properties.

The exception object will be delivered to the `catch` clause of a `try` statement:

```
// Make a try_it function that calls the new add
// function incorrectly.

var try_it = function () {
  try {
    add("seven");
  } catch (e) {
    document.writeln(e.name + ': ' + e.message);
  }
}

tryIt();
```

If an exception is thrown within a `try` block, control will go to its `catch` clause.

A `try` statement has a single `catch` block that will catch all exceptions. If your handling depends on the type of the exception, then the exception handler will have to inspect the `name` to determine the type of the exception.

Augmenting Types

JavaScript allows the basic types of the language to be *augmented*. In Chapter 3, we saw that adding a method to `Object.prototype` makes that method available to all objects. This also works for functions, arrays, strings, numbers, regular expressions, and booleans.

For example, by augmenting `Function.prototype`, we can make a method available to all functions:

```

Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};

```

By augmenting `Function.prototype` with a method `method`, we no longer have to type the name of the prototype property. That bit of ugliness can now be hidden.

JavaScript does not have a separate integer type, so it is sometimes necessary to extract just the integer part of a number. The method JavaScript provides to do that is ugly. We can fix it by adding an `integer` method to `Number.prototype`. It uses either `Math.ceil` or `Math.floor`, depending on the sign of the number:

```

Number.method('integer', function () {
    return Math[this < 0 ? 'ceil' : 'floor'](this);
});

document.writeln((-10 / 3).integer()); // -3

```

JavaScript lacks a method that removes spaces from the ends of a string. That is an easy oversight to fix:

```

String.method('trim', function () {
    return this.replace(/^\s+|\s+$/g, '');
});

document.writeln("'" + "  neat  ".trim() + "'");

```

Our `trim` method uses a regular expression. We will see much more about regular expressions in Chapter 7.

By augmenting the basic types, we can make significant improvements to the expressiveness of the language. Because of the dynamic nature of JavaScript's prototypal inheritance, all values are immediately endowed with the new methods, even values that were created before the methods were created.

The prototypes of the basic types are public structures, so care must be taken when mixing libraries. One defensive technique is to add a method only if the method is known to be missing:

```

// Add a method conditionally.

Function.prototype.method = function (name, func) {
    if (!this.prototype[name]) {
        this.prototype[name] = func;
    }
};

```

Another concern is that the `for in` statement interacts badly with prototypes. We saw a couple of ways to mitigate that in Chapter 3: we can use the `hasOwnProperty` method to screen out inherited properties, and we can look for specific types.

Recursion

A *recursive* function is a function that calls itself, either directly or indirectly. Recursion is a powerful programming technique in which a problem is divided into a set of similar subproblems, each solved with a trivial solution. Generally, a recursive function calls itself to solve its subproblems.

The Towers of Hanoi is a famous puzzle. The equipment includes three posts and a set of discs of various diameters with holes in their centers. The setup stacks all of the discs on the source post with smaller discs on top of larger discs. The goal is to move the stack to the destination post by moving one disc at a time to another post, never placing a larger disc on a smaller disc. This puzzle has a trivial recursive solution:

```
var hanoi = function (disc, src, aux, dst) {  
  if (disc > 0) {  
    hanoi(disc - 1, src, dst, aux);  
    document.writeln('Move disc ' + disc +  
      ' from ' + src + ' to ' + dst);  
    hanoi(disc - 1, aux, src, dst);  
  }  
}  
  
hanoi(3, 'Src', 'Aux', 'Dst');
```

It produces this solution for three discs:

```
Move disc 1 from Src to Dst  
Move disc 2 from Src to Aux  
Move disc 1 from Dst to Aux  
Move disc 3 from Src to Dst  
Move disc 1 from Aux to Src  
Move disc 2 from Aux to Dst  
Move disc 1 from Src to Dst
```

The `hanoi` function moves a stack of discs from one post to another, using the auxiliary post if necessary. It breaks the problem into three subproblems. First, it uncovers the bottom disc by moving the substack above it to the auxiliary post. It can then move the bottom disc to the destination post. Finally, it can move the substack from the auxiliary post to the destination post. The movement of the substack is handled by calling itself recursively to work out those subproblems.

The `hanoi` function is passed the number of the disc it is to move and the three posts it is to use. When it calls itself, it is to deal with the disc that is above the disc it is currently working on. Eventually, it will be called with a nonexistent disc number. In that case, it does nothing. That act of nothingness gives us confidence that the function does not recurse forever.

Recursive functions can be very effective in manipulating tree structures such as the browser's Document Object Model (DOM). Each recursive call is given a smaller piece of the tree to work on:

```

// Define a walk_the_DOM function that visits every
// node of the tree in HTML source order, starting
// from some given node. It invokes a function,
// passing it each node in turn. walk_the_DOM calls
// itself to process each of the child nodes.

var walk_the_DOM = function walk(node, func) {
    func(node);
    node = node.firstChild;
    while (node) {
        walk(node, func);
        node = node.nextSibling;
    }
};

// Define a getElementsByAttribute function. It
// takes an attribute name string and an optional
// matching value. It calls walk_the_DOM, passing it a
// function that looks for an attribute name in the
// node. The matching nodes are accumulated in a
// results array.

var getElementsByAttribute = function (att, value) {
    var results = [];

    walk_the_DOM(document.body, function (node) {
        var actual = node.nodeType === 1 && node.getAttribute(att);
        if (typeof actual === 'string' &&
            (actual === value || typeof value !== 'string')) {
            results.push(node);
        }
    });

    return results;
};

```

Some languages offer the *tail recursion optimization*. This means that if a function returns the result of invoking itself recursively, then the invocation is replaced with a loop, which can significantly speed things up. Unfortunately, JavaScript does not currently provide tail recursion optimization. Functions that recurse very deeply can fail by exhausting the return stack:

```

// Make a factorial function with tail
// recursion. It is tail recursive because
// it returns the result of calling itself.

// JavaScript does not currently optimize this form.

var factorial = function factorial(i, a) {
    a = a || 1;
    if (i < 2) {
        return a;
    }
    return factorial(i - 1, a * i);
};

```

```
};  
  
document.writeln(factorial(4)); // 24
```

Scope

Scope in a programming language controls the visibility and lifetimes of variables and parameters. This is an important service to the programmer because it reduces naming collisions and provides automatic memory management:

```
var foo = function () {  
    var a = 3, b = 5;  
  
    var bar = function () {  
        var b = 7, c = 11;  
  
        // At this point, a is 3, b is 7, and c is 11  
  
        a += b + c;  
  
        // At this point, a is 21, b is 7, and c is 11  
  
    };  
  
    // At this point, a is 3, b is 5, and c is not defined  
  
    bar();  
  
    // At this point, a is 21, b is 5  
  
};
```

Most languages with C syntax have block scope. All variables defined in a block (a list of statements wrapped with curly braces) are not visible from outside of the block. The variables defined in a block can be released when execution of the block is finished. This is a good thing.

Unfortunately, JavaScript does not have block scope even though its block syntax suggests that it does. This confusion can be a source of errors.

JavaScript does have function scope. That means that the parameters and variables defined in a function are not visible outside of the function, and that a variable defined anywhere within a function is visible everywhere within the function.

In many modern languages, it is recommended that variables be declared as late as possible, at the first point of use. That turns out to be bad advice for JavaScript because it lacks block scope. So instead, it is best to declare all of the variables used in a function at the top of the function body.

Closure

The good news about scope is that inner functions get access to the parameters and variables of the functions they are defined within (with the exception of this and arguments). This is a very good thing.

Our `getElementsByClassName` function worked because it declared a `results` variable, and the inner function that it passed to `walk_the_DOM` also had access to the `results` variable.

A more interesting case is when the inner function has a longer lifetime than its outer function.

Earlier, we made a `myObject` that had a `value` and an `increment` method. Suppose we wanted to protect the `value` from unauthorized changes.

Instead of initializing `myObject` with an object literal, we will initialize `myObject` by calling a function that returns an object literal. That function defines a `value` variable. That variable is always available to the `increment` and `getValue` methods, but the function's scope keeps it hidden from the rest of the program:

```
var myObject = function () {
  var value = 0;

  return {
    increment: function (inc) {
      value += typeof inc === 'number' ? inc : 1;
    },
    getValue: function () {
      return value;
    }
  }
}();
```

We are not assigning a function to `myObject`. We are assigning the result of invoking that function. Notice the `()` on the last line. The function returns an object containing two methods, and those methods continue to enjoy the privilege of access to the `value` variable.

The `Quo` constructor from earlier in this chapter produced an object with a `status` property and a `get_status` method. But that doesn't seem very interesting. Why would you call a getter method on a property you could access directly? It would be more useful if the `status` property were private. So, let's define a different kind of `quo` function to do that:

```
// Create a maker function called quo. It makes an
// object with a get_status method and a private
// status property.
```

```

var quo = function (status) {
  return {
    get_status: function () {
      return status;
    }
  };
};

// Make an instance of quo.

var myQuo = quo("amazed");

document.writeln(myQuo.get_status());

```

This quo function is designed to be used without the new prefix, so the name is not capitalized. When we call quo, it returns a new object containing a get_status method. A reference to that object is stored in myQuo. The get_status method still has privileged access to quo's status property even though quo has already returned. get_status does not have access to a copy of the parameter; it has access to the parameter itself. This is possible because the function has access to the context in which it was created. This is called *closure*.

Let's look at a more useful example:

```

// Define a function that sets a DOM node's color
// to yellow and then fades it to white.

var fade = function (node) {
  var level = 1;
  var step = function () {
    var hex = level.toString(16);
    node.style.backgroundColor = '#FFFF' + hex + hex;
    if (level < 15) {
      level += 1;
      setTimeout(step, 100);
    }
  };
  setTimeout(step, 100);
};

fade(document.body);

```

We call fade, passing it document.body (the node created by the HTML <body> tag). fade sets level to 1. It defines a step function. It calls setTimeout, passing it the step function and a time (100 milliseconds). It then returns—fade has finished.

Suddenly, about a 10th of a second later, the step function gets invoked. It makes a base 16 character from fade's level. It then modifies the background color of fade's node. It then looks at fade's level. If it hasn't gotten to white yet, it then increments fade's level and uses setTimeout to schedule itself to run again.

Suddenly, the step function gets invoked again. But this time, fade's level is 2. fade returned a while ago, but its variables continue to live as long as they are needed by one or more of fade's inner functions.

It is important to understand that the inner function has access to the actual variables of the outer functions and not copies in order to avoid the following problem:

```
// BAD EXAMPLE

// Make a function that assigns event handler functions to an array of nodes the
// wrong way.
// When you click on a node, an alert box is supposed to display the ordinal of the
// node.
// But it always displays the number of nodes instead.

var add_the_handlers = function (nodes) {
  var i;
  for (i = 0; i < nodes.length; i += 1) {
    nodes[i].onclick = function (e) {
      alert(i);
    }
  }
};

// END BAD EXAMPLE
```

The `add_the_handlers` function was intended to give each handler a unique number (`i`). It fails because the handler functions are bound to the variable `i`, not the value of the variable `i` at the time the function was made:

```
// BETTER EXAMPLE

// Make a function that assigns event handler functions to an array of nodes the
// right way.
// When you click on a node, an alert box will display the ordinal of the node.

var add_the_handlers = function (nodes) {
  var i;
  for (i = 0; i < nodes.length; i += 1) {
    nodes[i].onclick = function (i) {
      return function (e) {
        alert(i);
      };
    }(i);
  }
};
```

Now, instead of assigning a function to `onclick`, we define a function and immediately invoke it, passing in `i`. That function will return an event handler function that is bound to the value of `i` that was passed in, not to the `i` defined in `add_the_handlers`. That returned function is assigned to `onclick`.

Callbacks

Functions can make it easier to deal with discontinuous events. For example, suppose there is a sequence that begins with a user interaction, making a request of the server, and finally displaying the server's response. The naïve way to write that would be:

```
request = prepare_the_request();
response = send_request_synchronously(request);
display(response);
```

The problem with this approach is that a synchronous request over the network will leave the client in a frozen state. If either the network or the server is slow, the degradation in responsiveness will be unacceptable.

A better approach is to make an asynchronous request, providing a callback function that will be invoked when the server's response is received. An asynchronous function returns immediately, so the client isn't blocked:

```
request = prepare_the_request();
send_request_asynchronously(request, function (response) {
    display(response);
});
```

We pass a function parameter to the `send_request_asynchronously` function that will be called when the response is available.

Module

We can use functions and closure to make modules. A module is a function or object that presents an interface but that hides its state and implementation. By using functions to produce modules, we can almost completely eliminate our use of global variables, thereby mitigating one of JavaScript's worst features.

For example, suppose we want to augment `String` with a `deentityify` method. Its job is to look for HTML entities in a string and replace them with their equivalents. It makes sense to keep the names of the entities and their equivalents in an object. But where should we keep the object? We could put it in a global variable, but global variables are evil. We could define it in the function itself, but that has a runtime cost because the literal must be evaluated every time the function is invoked. The ideal approach is to put it in a closure, and perhaps provide an extra method that can add additional entities:

```
String.method('deentityify', function () {

    // The entity table. It maps entity names to
    // characters.

    var entity = {
        quot: '"',
```

```

        lt: '<',
        gt: '>'
    };

    // Return the deentityify method.

    return function () {

        // This is the deentityify method. It calls the string
        // replace method, looking for substrings that start
        // with '&' and end with ';' . If the characters in
        // between are in the entity table, then replace the
        // entity with the character from the table. It uses
        // a regular expression (Chapter 7).

        return this.replace(/&([^\&];+);/g,
            function (a, b) {
                var r = entity[b];
                return typeof r === 'string' ? r : a;
            }
        );
    };
}());

```

Notice the last line. We immediately invoke the function we just made with the `()` operator. That invocation creates and returns the function that becomes the `deentityify` method.

```

document.writeln(
    '&quot;&gt;'.deentityify()); // <>

```

The module pattern takes advantage of function scope and closure to create relationships that are binding and private. In this example, only the `deentityify` method has access to the entity data structure.

The general pattern of a module is a function that defines private variables and functions; creates privileged functions which, through closure, will have access to the private variables and functions; and that returns the privileged functions or stores them in an accessible place.

Use of the module pattern can eliminate the use of global variables. It promotes information hiding and other good design practices. It is very effective in encapsulating applications and other singletons.

It can also be used to produce objects that are secure. Let's suppose we want to make an object that produces a serial number:

```

var serial_maker = function () {

    // Produce an object that produces unique strings. A
    // unique string is made up of two parts: a prefix
    // and a sequence number. The object comes with
    // methods for setting the prefix and sequence

```

```

// number, and a gensym method that produces unique
// strings.

var prefix = '';
var seq = 0;
return {
  set_prefix: function (p) {
    prefix = String(p);
  },
  set_seq: function (s) {
    seq = s;
  },
  gensym: function () {
    var result = prefix + seq;
    seq += 1;
    return result;
  }
};
})();

var sequer = serial_maker();
sequer.set_prefix = 'Q';
sequer.set_seq = 1000;
var unique = sequer.gensym(); // unique is "Q1000"

```

The methods do not make use of this or that. As a result, there is no way to compromise the `sequer`. It isn't possible to get or change the prefix or seq except as permitted by the methods. The `sequer` object is mutable, so the methods could be replaced, but that still does not give access to its secrets. `sequer` is simply a collection of functions, and those functions are capabilities that grant specific powers to use or modify the secret state.

If we passed `sequer.gensym` to a third party's function, that function would be able to generate unique strings, but would be unable to change the prefix or seq.

Cascade

Some methods do not have a return value. For example, it is typical for methods that set or change the state of an object to return nothing. If we have those methods return `this` instead of `undefined`, we can enable *cascades*. In a cascade, we can call many methods on the same object in sequence in a single statement. An Ajax library that enables cascades would allow us to write in a style like this:

```

getElement('myBoxDiv').
  move(350, 150).
  width(100).
  height(100).
  color('red').
  border('10px outset').
  padding('4px').
  appendText("Please stand by").

```

```

on('mousedown', function (m) {
    this.startDrag(m, this.getNinth(m));
}).
on('mousemove', 'drag').
on('mouseup', 'stopDrag').
later(2000, function () {
    this.
        color('yellow').
        setHTML("What hath God wrought?").
        slide(400, 40, 200, 200);
}).
tip('This box is resizable');

```

In this example, the `getElement` function produces an object that gives functionality to the DOM element with `id="myBoxDiv"`. The methods allow us to move the element, change its dimensions and styling, and add behavior. Each of those methods returns the object, so the result of the invocation can be used for the next invocation.

Cascading can produce interfaces that are very expressive. It can help control the tendency to make interfaces that try to do too much at once.

Curry

Functions are values, and we can manipulate function values in interesting ways. *Currying* allows us to produce a new function by combining a function and an argument:

```

var add1 = add.curry(1);
document.writeln(add1(6));    // 7

```

`add1` is a function that was created by passing `1` to `add`'s `curry` method. The `add1` function adds `1` to its argument. JavaScript does not have a `curry` method, but we can fix that by augmenting `Function.prototype`:

```

Function.method('curry', function () {
    var args = arguments, that = this;
    return function () {
        return that.apply(null, args.concat(arguments));
    };
}); // Something isn't right...

```

The `curry` method works by creating a closure that holds that original function and the arguments to curry. It returns a function that, when invoked, returns the result of calling that original function, passing it all of the arguments from the invocation of `curry` and the current invocation. It uses the `Array concat` method to concatenate the two arrays of arguments together.

Unfortunately, as we saw earlier, the `arguments` array is not an array, so it does not have the `concat` method. To work around that, we will apply the `array slice` method on both of the arguments arrays. This produces arrays that behave correctly with the `concat` method:

```

Function.method('curry', function () {
  var slice = Array.prototype.slice,
      args = slice.apply(arguments),
      that = this;
  return function () {
    return that.apply(null, args.concat(slice.apply(arguments)));
  };
});

```

Memoization

Functions can use objects to remember the results of previous operations, making it possible to avoid unnecessary work. This optimization is called *memoization*. JavaScript's objects and arrays are very convenient for this.

Let's say we want a recursive function to compute Fibonacci numbers. A Fibonacci number is the sum of the two previous Fibonacci numbers. The first two are 0 and 1:

```

var fibonacci = function (n) {
  return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};

for (var i = 0; i <= 10; i += 1) {
  document.writeln('// ' + i + ': ' + fibonacci(i));
}

// 0: 0
// 1: 1
// 2: 1
// 3: 2
// 4: 3
// 5: 5
// 6: 8
// 7: 13
// 8: 21
// 9: 34
// 10: 55

```

This works, but it is doing a lot of unnecessary work. The `fibonacci` function is called 453 times. We call it 11 times, and it calls itself 442 times in computing values that were probably already recently computed. If we *memoize* the function, we can significantly reduce its workload.

We will keep our memoized results in a `memo` array that we can hide in a closure. When our function is called, it first looks to see if it already knows the result. If it does, it can immediately return it:

```

var fibonacci = function () {
  var memo = [0, 1];
  var fib = function (n) {
    var result = memo[n];
    if (typeof result !== 'number') {

```

```

        result = fib(n - 1) + fib(n - 2);
        memo[n] = result;
    }
    return result;
};
return fib;
})();

```

This function returns the same results, but it is called only 29 times. We called it 11 times. It called itself 18 times to obtain the previously memoized results.

We can generalize this by making a function that helps us make memoized functions. The `memoizer` function will take an initial `memo` array and the fundamental function. It returns a shell function that manages the memo store and that calls the fundamental function as needed. We pass the shell function and the function's parameters to the fundamental function:

```

var memoizer = function (memo, fundamental) {
    var shell = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = fundamental(shell, n);
            memo[n] = result;
        }
        return result;
    };
    return shell;
};

```

We can now define `fibonacci` with the `memoizer`, providing the initial `memo` array and fundamental function:

```

var fibonacci = memoizer([0, 1], function (shell, n) {
    return shell(n - 1) + shell(n - 2);
});

```

By devising functions that produce other functions, we can significantly reduce the amount of work we have to do. For example, to produce a memoizing factorial function, we only need to supply the basic factorial formula:

```

var factorial = memoizer([1, 1], function (shell, n) {
    return n * shell(n - 1);
});

```