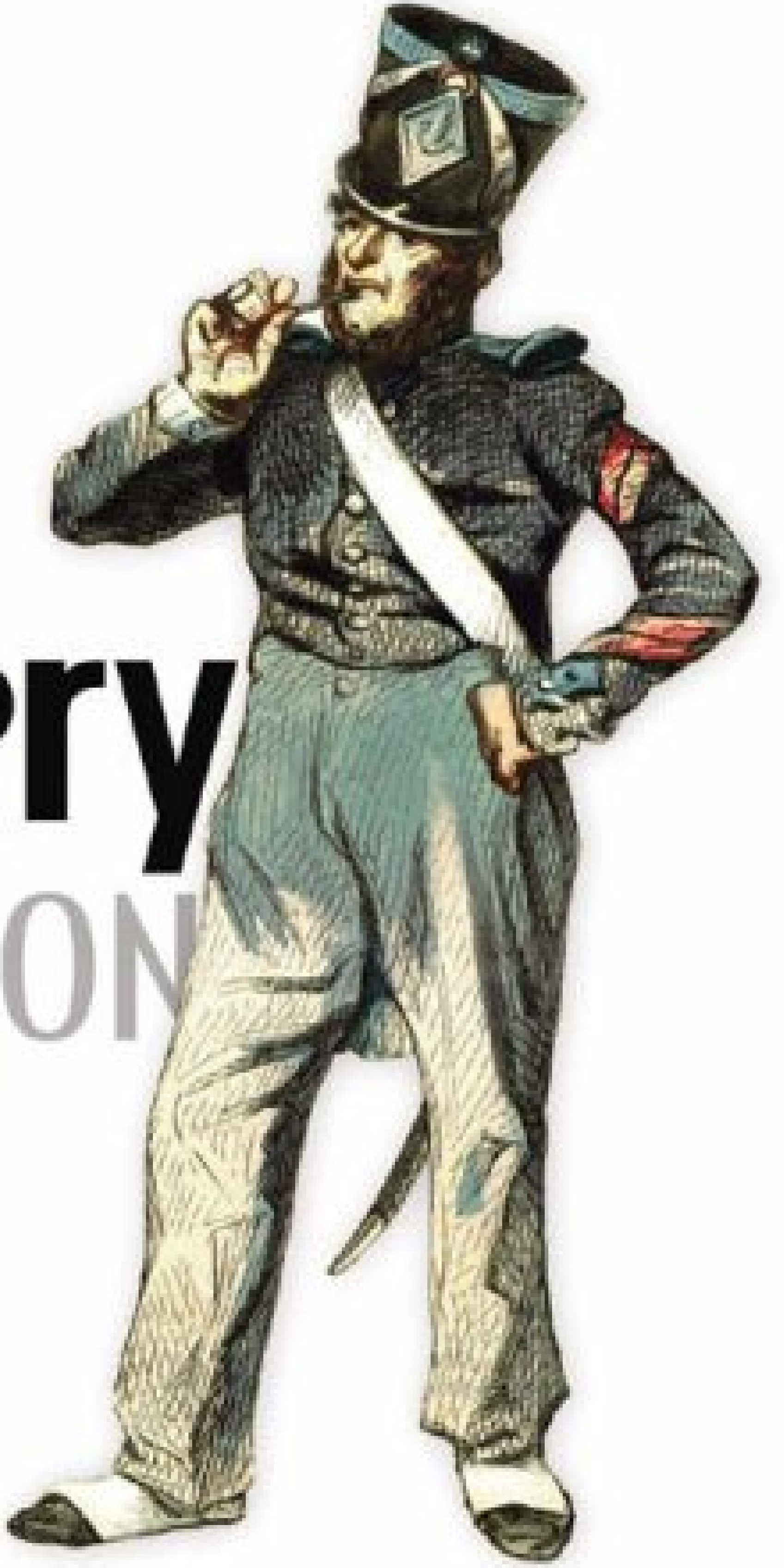
Bear Bibeault

Yehuda Katz

jQuery IN ACTION



MANNING

Table of Contents

appendix: JavaScript that you need to know but might not!

This appendix covers

- Which JavaScript concepts are important for effectively using jQuery
- JavaScript `Object` basics
- How functions are *first-class objects*
- Determining (and controlling) what `this` means
- What's a closure?

One of the great benefits that jQuery brings to our web applications is the ability to implement a great deal of scripting-enabled behavior without having to write a whole lot of script ourselves. jQuery handles the nuts-and-bolts details so that we can concentrate on the job of making our applications do what they need to do!

For the first few chapters in this book, we only needed rudimentary JavaScript skills to code and understand those examples. With the chapters on advanced topics such as event handling, animations, and Ajax, we must understand a handful of fundamental JavaScript concepts to make effective use of the jQuery library. You may find that a lot of things that you, perhaps, took for granted in JavaScript (or took on blind faith) will start to make more sense.

We're not going to go into an exhaustive study of all JavaScript concepts—that's not the purpose of this book. The purpose of this book is to get us up and running with effective jQuery in the shortest time possible. To that end, we'll concentrate on the fundamental concepts that we need to make the most effective use of jQuery in our web applications.

The most important of these concepts centers around the manner in which JavaScript defines and deals with functions, specifically the way in which functions are *first-class objects* in JavaScript. What do we mean by that? Well, in order to understand what it means for a function to be an object, let alone a *first-class* one, we must first make sure that we understand what a JavaScript object itself is all about. So let's dive right in.

A.1 JavaScript Object fundamentals

The majority of object-oriented (OO) languages define a fundamental `Object` type of some kind from which all other objects are derived. Likewise, in JavaScript, the fundamental `Object` serves as the basis for all other objects, but that's where the comparison stops. At its basic level, the JavaScript `Object` has little in common with the fundamental object defined by its OO brethren languages.

At first glance, a JavaScript `Object` may seem like a boring and mundane item. Once created, it holds no data and exposes little in the way of semantics. But those limited semantics *do* give it a great deal of potential.

Let's see how.

A.1.1 How objects come to be

A new object comes into existence via the `new` operator paired with the `Object` constructor. Creating an object is as easy as

```
var shinyAndNew = new Object();
```

It could be even easier (as we'll see shortly), but this will do for now.

But what can we *do* with this new object? It seemingly contains nothing: no information, no complex semantics, nothing. Our brand-new, shiny object doesn't get interesting until we start adding things to it—things known as *properties*.

A.1.2 Properties of objects

Like their server-side counterparts, JavaScript objects can contain data and possess methods (well…sort of, but that's getting ahead of ourselves). Unlike those server-side brethren, these elements aren't pre-declared for an object; we create them dynamically as needed.

Take a look at the following code fragment:

```
var ride = new Object();
ride.make = 'Yamaha';
ride.model = 'V-Star Silverado 1100';
ride.year = 2005;
ride.purchased = new Date(2005,3,12);
```

We create a new `Object` instance and assign it to a variable named `ride`. We then populate this variable with a number of *properties* of different types: two strings, a number, and an instance of the `Date` type.

We don't need to declare these properties prior to assigning them; they come into being merely by the act of our assigning a value to them. That's mighty powerful juju that gives us a great deal of flexibility. But before we get too giddy, let's remember that flexibility always comes with a price!

For example, let's say that in a subsequent part of the page, we want to change the value of the purchase date:

```
ride.purchased = new Date(2005,2,1);
```

No problem…unless we make an inadvertent typo such as

```
ride.purcahsed = new Date(2005,2,1);
```

There's no compiler to warn us that we've made a mistake; a new property named `purcahsed` is cheerfully created on our behalf, leaving us to wonder later on why the new date didn't *take* when we reference the correctly spelled property.

With great power comes great responsibility (where have we heard that before?), so type carefully!

NOTE JavaScript debuggers such as Firebug for Firefox can be lifesavers when dealing with such issues. Because typos such as these frequently result in no JavaScript errors, relying on JavaScript consoles or error dialog boxes is usually less than effective.

From this example, we've learned that an instance of the JavaScript `Object`, which we'll simply refer to as an *object* from here forward, is a collection of *properties*, each of which consists of a *name* and a *value*. The name of a property is a string, and the value can be any JavaScript object, be it a `Number`, `String`, `Date`, `Array`, basic `Object`, or any other JavaScript object type (including, as we shall see, functions).

This makes the primary purpose of an `Object` instance to serve as a container for a named collection of other objects. This may remind you of concepts in other languages: a Java map for example, or dictionaries or hashes in other languages.

When referencing properties, we can chain references to properties of objects serving as the properties of a parent object. Let's say that we add a new property to our `ride` instance that captures the owner of the vehicle. This property is another JavaScript object that contains properties such as the name and occupation of the owner:

```
var owner = new Object();
owner.name = 'Spike Spiegel';
owner.occupation = 'bounty hunter';
ride.owner = owner;
```

To access the nested property, we write

```
var ownerName = ride.owner.name;
```

There are no limits to the nesting levels we can employ (except the limits of good sense). When finished—up to this point—our object hierarchy looks as shown in figure A.1.

Note how each value is a distinct instance of a JavaScript type.

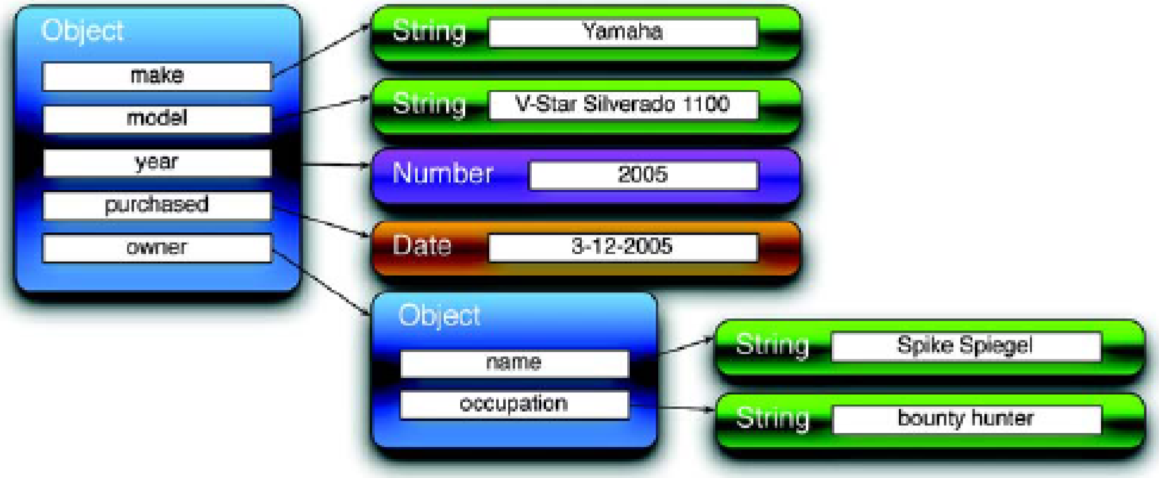


Figure A.1 Our object hierarchy shows that Objects are containers for named references to other Objects or JavaScript built-in objects.

By the way, there's no need for all the intermediary variables (such as `owner`) we've created for illustrative purposes in these code fragments. In a short while, we'll see more efficient and compact ways to declare objects and their properties.

Up to this point, we've referenced properties of an object by using the dot (period character) operator; but, as it turns out, that's a synonym for a more general operator for performing property referencing.

What if, for example, we have a property named `color.scheme`? Do you notice the period in the middle of the name? It throws a monkey wrench into the works because the JavaScript interpreter will try to look up `scheme` as a nested property of `color`.

"Well, just don't do that!" you say. But what about space characters? What about other characters that could be mistaken for delimiters rather than as part of a name? And most importantly, what if we don't even know what the property name is but have it as a value in another variable or as the result of an expression evaluation?

For all these cases the dot operator is inadequate, and we must use the more general notation for accessing properties. The format of the general property reference operator is

```
object[propertyNameExpression]
```

where *propertyNameExpression* is a JavaScript expression whose evaluation as a string forms the name of the property to be referenced. For example, all three of the following references are equivalent:

```
ride.make
ride['make']
ride['m'+'a'+'k'+'e']
```

as well as

```
var p = 'make';
ride[p];
```

Using the general reference operator is the only way to reference properties whose names don't form valid JavaScript identifiers, such as

```
ride["a property name that's rather odd!"]
```

which contains characters not legal for JavaScript identifiers, or whose names are the values of other variables.

Building up objects by creating new instances with the `new` operator and assigning each property using separate assignment statements can be a tedious affair. In the next section, we'll look at a more compact and easy-to-read notation for declaring our objects and their properties.

A.1.3 Object literals

In the previous section, we created an object that modeled some of the properties of a motorcycle, assigning it to a variable named `ride`. To do so, we used two `new` operations, an intermediary variable named `owner`, and a bunch of assignment statements. This is tedious—as well as wordy and error-prone—and makes it difficult for us to visually grasp the structure of the object during a quick inspection of the code.

Luckily, we can use a notation that's more compact and easier to visually scan. Consider the following statement:

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {
    name: 'Spike Spiegel',
    occupation: 'bounty hunter'
  }
};
```

Using an *object literal*, this fragment creates the same `ride` object that we built up with assignment statements in the previous section.

This notation, which has come to be termed *JSON* (JavaScript Object Notation[1]), is much preferred by most page authors over the multiple-assignment means of object building. Its structure is simple; an object is denoted by a matching pair of braces, within which properties are listed delimited by commas. Each property is denoted by listing its name and value separated by a colon character.

NOTE Technically, JSON has no means to express date values, primarily because JavaScript itself lacks any kind of date literal. When used in script, the `Date` constructor is usually employed as shown in the previous example. When used as an interchange format, dates are frequently expressed either as a string containing the ISO 8601 format or a number expressing the date as the millisecond value returned by `Date.getTime()`.

As we can see by the declaration of the `owner` property, object declarations can be nested.

[1] For more information, you can visit http://www.json.org/.

By the way, we can also express arrays in JSON by placing the comma-delimited list of elements within square brackets as in the following:

```
var someValues = [2,3,5,7,11,13,17,19,23,29,31,37];
```

As we've seen in the examples presented in this section, object reference are frequently stored in variables or in properties of other objects. Let's take a look at a special case of the latter scenario.

A.1.4 Objects as window properties

Up to this point, we've seen two ways to store a reference to a JavaScript object: variables and properties. These two means of storing references use differing notation, as shown in the following snippet:

```
var aVariable =
  'Before I teamed up with you, I led quite a normal life.';

someObject.aProperty =
  'You move that line as you see fit for yourself.';
```

These two statements each assign a `String` instance (created via literals) to a variable and an object property respectively using assignment operations. (Kudos to you if you can identify the source of the obscure quotes; no cheating with Google! There was a clue earlier in the chapter.)

But are these statements *really* performing different operations? As it turns out, they're not!

When the `var` keyword is used at the top level, outside the body of any containing function, it's only a programmer-friendly notation for referencing a property of the pre-defined JavaScript `window` object. Any reference made in top-level scope is implicitly made on the `window` instance.

This means that all of the following statements are equivalent:

```
var foo = bar;
```

and

```
window.foo = bar;
```

and

```
foo = bar;
```

Regardless of which notation is used, a `window` property named `foo` is created (if it's not already in existence) and assigned the value of `bar`. Also, note that because `bar` is unqualified, it's assumed to be a property on `window`.

It probably won't get us into conceptual trouble to think of top-level scope as *window* scope because any unqualified references *at the top level* are assumed to be `window` properties. The scoping rules get more complex when we delve deeper into the bodies of functions—much more complex, in fact—but we'll be addressing that soon enough.

That pretty much covers things for our overview of the JavaScript `Object`. The important concepts to take away from this discussion are

- A JavaScript object is an unordered collection of properties.
- Properties consist of a name and a value.
- Objects can be declared using object literals.
- Top-level *variables* are properties of `window`.

Now, let's discuss what we meant when we referred to JavaScript functions as *first-class objects*.

A.2 Functions as first-class citizens

In many traditional OO languages, objects can contain data, and they can possess methods. In these languages the data and the methods are usually distinct concepts; JavaScript walks a different path.

Functions in JavaScript are considered objects like any of the other object types that are defined in JavaScript, such as `Strings`, `Numbers`, or `Dates`. Like other objects, functions are defined by a JavaScript constructor—in this case `Function`—and can be

- Assigned to variables
- Assigned as a property of an object
- Passed as a parameter
- Returned as a function result
- Created using literals

Because functions are treated in the same way as other objects in the language, we say that functions are *first-class objects*.

But you might be thinking to yourself that functions are fundamentally different from other object types like `String` or `Number` because they possess not only a value (in the case of a `Function` instance, its body) but also a *name*.

Well, not so fast!

A.2.1 What's in a name?

A large percentage of JavaScript programmers operate under a false assumption that functions are named entities. Not so. If you're one of these programmers, you've been fooled by a Jedi mind trick. As with other instances of objects—be they `Strings`, `Dates`, or `Numbers`—functions are referenced *only* when they are assigned to variables, properties, or parameters.

Let's consider objects of type `Number`. We frequently express instances of `Number` by their literal notation such as `213`. The statement

```
213;
```

is perfectly valid, but it is also perfectly useless. The `Number` instance isn't all that useful unless it has been assigned to a property or a variable, or bound to a parameter name. Otherwise, we have no way to reference the disembodied instance.

The same applies to instances of `Function` objects.

"But, but, but..." you might be saying, "what about the following code?"

```
function doSomethingWonderful() {
  alert('does something wonderful');
}
```

"Doesn't that create a function *named* `doSomethingWonderful`?"

No, it doesn't. Although that notation may seem familiar and is ubiquitously used to create *top-level* functions, it's the same syntactic sugar used by `var` to create `window` properties. The `function` keyword automatically creates a `Function` instance and assigns it to a `window` property created using the function "name" (what we referred to earlier as a *Jedi mind trick*) as in the following:

```
doSomethingWonderful = function() {
  alert('does something wonderful');
}
```

If that looks weird to you, consider another statement using the exact same format, except this time using a `Number` literal:

```
aWonderfulNumber = 213;
```

There's nothing strange about that, and the statement assigning a function to a top-level variable (a.k.a. `window` property) is no different; a function literal is used to create an instance of `Function` and then is assigned to the variable `doSomethingWonderful` in the same way that our `Number` literal `213` was used to assign a `Number` instance to the variable `aWonderfulNumber`.

If you've never seen the syntax for a *function literal*, it might seem odd. It's composed of the keyword `function`, followed by its parameter list enclosed in parentheses, then followed by the function body.

When we declare a top-level named function, a `Function` instance is created and *assigned* to a property of `window` that's automatically created using the so-called function name. The `Function` instance itself no more has a name than a `Number` literal or a `String` literal. Figure A.2 illustrates this concept.

Gecko browsers and function names

Browsers based on the Gecko layout engine, such as Firefox and Camino, store the name of functions defined using the top-level syntax in a nonstandard property of the function instance named `name`. Although this may not be of much use to the general development public, particularly considering its confinement to Gecko-based browsers, it's of great value to writers of browser plugins and debuggers.

Remember that, when a top-level variable is created in an HTML page, the variable is created as a property of the `window` instance. Therefore, the following statements are all equivalent:

```
function hello(){ alert('Hi there!'); }

hello = function(){ alert('Hi there!'); }

window.hello = function(){ alert('Hi there!'); }
```

Although this may seem like syntactic juggling, it's important to understanding that `Function` instances are *values* that can be assigned to variables, properties, or parameters just like instances of other object types. And like those other object types, nameless disembodied instances aren't of any use unless they're assigned to a variable, property, or parameter through which they can be referenced.

We've seen examples of assigning functions to variables and properties, but what about passing functions as parameters? Let's take a look at why and how we do that.

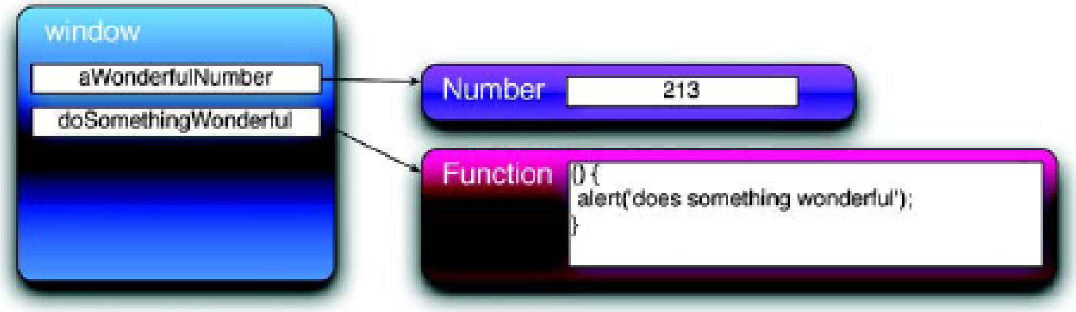


Figure A.2 A Function instance is a nameless object like the Number 213 or any other JavaScript value. It's named only by references that are made to it.

A.2.2 Functions as callbacks

Top-level functions are all well and good when our code follows a nice and orderly synchronous flow, but the nature of HTML pages—once loaded—is far from synchronous. Whether we are handling events, instituting timers, or making Ajax requests, the nature of the code in a web page is asynchronous. And one of the most prevalent concepts in asynchronous programming is the notion of a *callback* function.

Let's take the example of a timer. We can cause a timer to fire—let's say in five seconds—by passing the appropriate duration value to the `window.setTimeout()` method. But how does that method let us know when the timer has expired so that we can do whatever it is that we're waiting around for? It does so by invoking a function that *we* supply.

Let's consider the following code:

```
function hello() { alert('Hi there!'); }

setTimeout(hello,5000);
```

We declare a function named `hello` and set a timer to fire in 5 seconds, expressed as 5000 milliseconds by the second parameter. In the first parameter to the `setTimeout()` method, we pass a function reference. Passing a function as a parameter is no different than passing any other value—just as we passed a `Number` in the second parameter.

When the timer expires, the `hello` function is called. Because the `setTimeout()` method makes a call *back* to a function in our own code, that function is termed a *callback* function.

This code example would be considered naïve by most advanced JavaScript coders because the creation of the `hello` name is unnecessary. Unless the function is to be called elsewhere in the page, there's no need to create the `window` property `hello` to momentarily store the `Function` instance to pass it as the callback parameter.

The more elegant way to code this fragment is

```
setTimeout(function() { alert('Hi there!'); },5000);
```

in which we express the function literal directly in the parameter list, and no needless name is generated. This is an idiom that we'll often see used in jQuery code when there is no need for a function instance to be assigned to a top-level property.

The functions we've created in the examples so far are either top-level functions (which we know are top-level `window` properties) or assigned to parameters in a function call. We can also assign `Function` instances to properties of objects, and that's where things get really interesting. Read on…

A.2.3 What's this all about?

OO languages automatically provide a means to reference the current instance of an object from within a method. In languages like Java and C++, a variable named `this` points to that current instance. In JavaScript, a similar concept exists and even uses the same `this` keyword, which also provides access to an object associated with a function. But OO programmers beware! The JavaScript implementation of `this` differs from its OO counterparts in subtle but significant ways.

In class-based OO languages, the `this` pointer generally references the instance of the class within which the method has been declared. In JavaScript, where functions are first-class objects that aren't declared as part of anything, the object referenced by `this`—termed the *function context*—is determined not by how the function is declared but by how it's *invoked*.

This means that the *same* function can have *different* contexts depending on how it's called. That may seem freaky at first, but it can be quite useful.

In the default case, the context (`this`) of an invocation of the function is the object whose property contains the reference used to invoke the function. Let's look back to our motorcycle example for a demonstration, amending the object creation as follows (additions highlighted in bold):

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {name: 'Spike Spiegel',occupation: 'bounty hunter'},
  whatAmI: function() {
    return this.year+' '+this.make+' '+this.model;
  }
};
```

To our original example code, we add a property named `whatAmI` that references a `Function` instance. Our new object hierarchy, with the `Function` instance assigned to the property named `whatAmI`, is shown in figure A.3.

When the function is invoked through the property reference as in

```
var bike = ride.whatAmI();
```

the function context (the `this` reference) is set to the object instance pointed to by `ride`. As a result, the variable `bike` gets set to the string *2005 Yamaha V-Star*

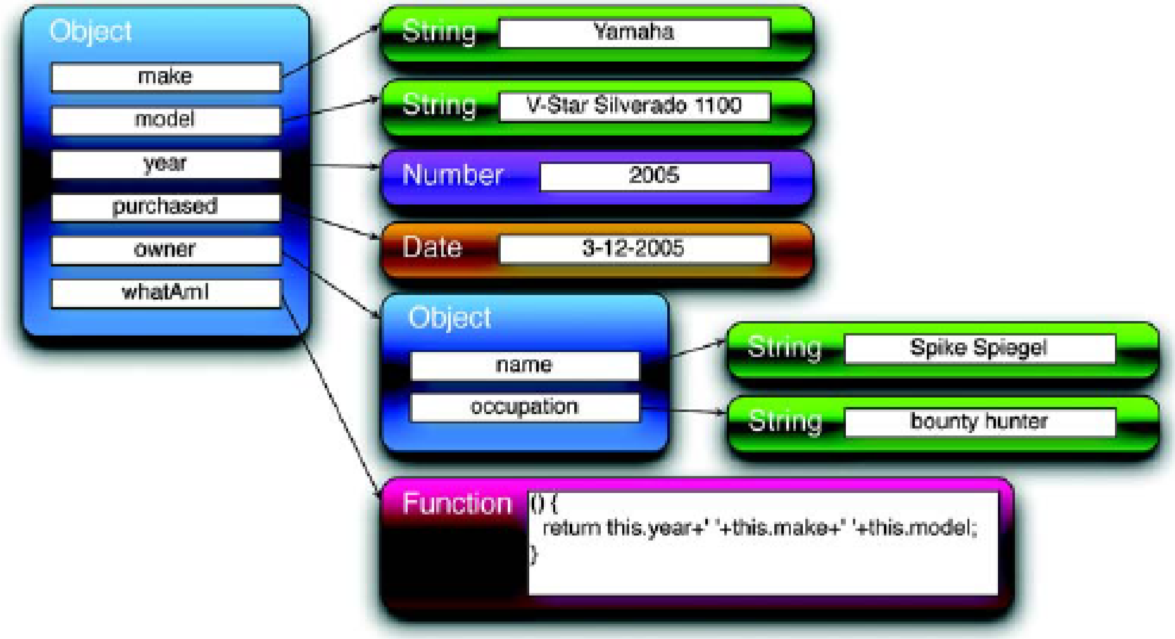


Figure A.3 This model clearly shows that the function isn't part of the Object but is only referenced from the Object property named `whatAmI`.

Silverado 1100 because the function picks up the properties of the object through which it was invoked via `this`.

The same is true of top-level functions. Remember that top-level functions are properties of `window`, so their function contexts, when called *as* top-level functions, are the window objects.

Although that may be the usual and implicit behavior, JavaScript gives us the means to explicitly control what's used as the function context. We can set the function context to whatever we want by invoking a function via the `Function` methods `call()` or `apply()`.

Yes, as first-class objects, even functions have methods as defined by the `Function` constructor.

The `call()` method invokes the function specifying, as its first parameter, the object to serve as the function context, while the remainder of the parameters become the parameters of the called function—the second parameter to `call()` becomes the first argument of the called function and so on. The `apply()` method works in a similar fashion except that its second parameter is expected to be an array of objects that become the arguments to the called function.

Confused? It's time for a more comprehensive example. Consider the code of listing A.1 (found in the downloadable code as appendixA/function.context.html).

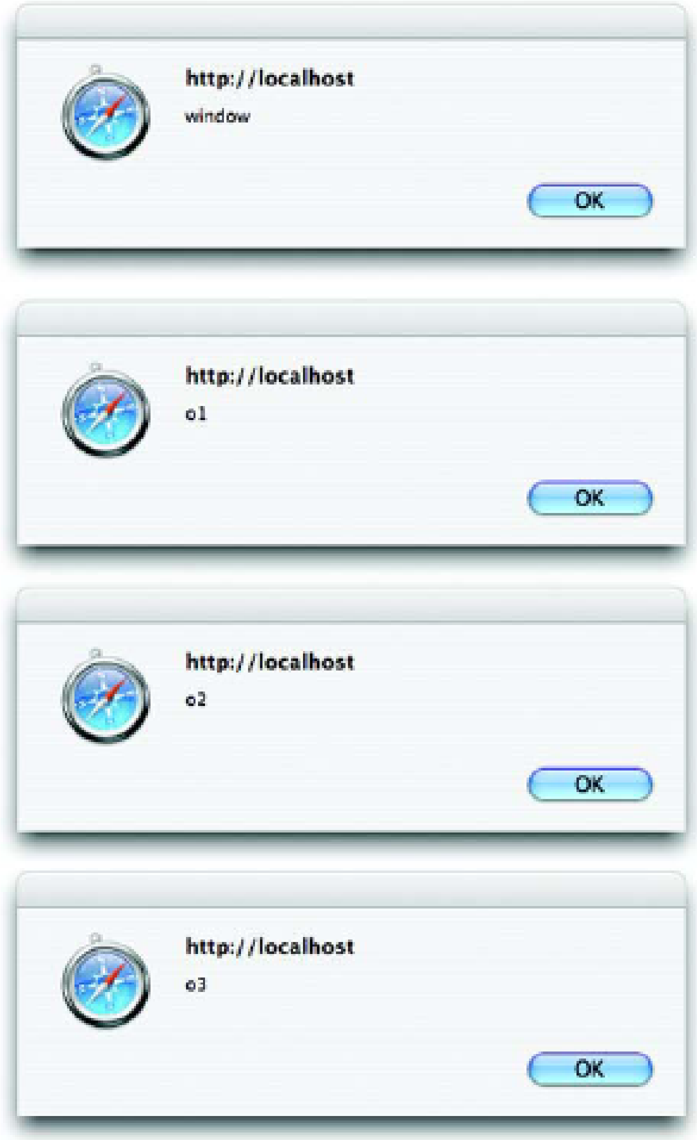
Listing A.1 Demonstrating that the value of the function context is dependent on how the function is invoked

```
<html>
  <head>
    <title>Function Context Example</title>
    <script>
      var o1 = {handle:'o1'};
      var o2 = {handle:'o2'};                ❶
      var o3 = {handle:'o3'};
      window.handle = 'window';

      function whoAmI() {
        return this.handle;              ❷
      }
                                          ❸
      o1.identifyMe = whoAmI;
                                   ❹
      alert(whoAmI());
      alert(o1.identifyMe());        ❺
      alert(whoAmI.call(o2));          ❻
      alert(whoAmI.apply(o3));
                                        ❼
    </script>
  </head>

  <body>
  </body>
</html>
```

In this example, we define three simple objects, each with a `handle` property that makes it easy to identify the object given a reference ❶. We also add a `handle` property to the `window` instance so that it's also readily identifiable.

We then define a top-level function that returns the value of the `handle` property for whatever object that serves as its function context ❷ and assign the *same* function instance to a property of object `o1` named `identifyMe` ❸. We can say that this creates a method on `o1` named `identifyMe`, although it's important to note that the function is declared independently of the object.

Finally, we issue four alerts, each of which uses a different mechanism to invoke the same function instance. When loaded into a browser, the sequence of four alerts is as shown in figure A.4.



http://localhost
window

http://localhost
o1

http://localhost
o2

http://localhost
o3

Figure A.4
The object serving as the function context changes with the manner in which the function is called.

This sequence of alerts illustrates the following:

- When the function is called directly as a top-level function, the function context is the `window` instance ❹.
- When called as a property of an object (`o1` in this case), the object becomes the function context of the function invocation ❺. We could say that the function acts as a *method* for that object—as in OO languages. But take care not to get too blasé about this analogy. You can be led astray if you're not careful, as the remainder of this example's results will show.

- Employing the `call()` method of `Function` causes the function context to be set to whatever object is passed as the first parameter to `call()`—in this case, `o2` ❻. In this example, the function acts like a method to `o2`, even though it has no association whatsoever—even as a property—with `o2`.
- As with `call()`, using the `apply()` method of `Function` sets the function context to whatever object is passed as the first parameter ❼. The difference between these two methods only becomes significant when parameters are passed to the function (which we didn't do in this example for simplicity).

This example page clearly demonstrates that the function context is determined on a per invocation basis and that a single function can be called with any object acting as its context. It's, therefore, probably never correct to say that a function *is* a method of an object. It's much more correct to state the following:

> *A function* `f` acts *as a method of object* `o` *when* `o` *serves as the function context of the invocation of* `f`*.*

As a further illustration of this concept, consider the effect of adding the following statement to our example:

```
alert(o1.identifyMe.call(o3));
```

Even though we reference the function as a property of `o1`, the function context for this invocation is `o3`, further emphasizing that it's not how a function is declared but how it's invoked that determines its function context.

When using jQuery commands and functions that employ callbacks, this proves to be an important concept. We saw this concept in action early on (even if you didn't realize it at the time) in section 2.3.3 where we supplied a callback function to the `filter()` method of `$` and that function was sequentially invoked with each element of the wrapped set serving as its function context in turn.

Now that we understand how functions can act as methods of objects, let's turn our attention to another advanced function topic that will play an important role in effective usage of jQuery—closures.

A.2.4 Closures

To page authors coming from a traditional OO or procedural programming background, *closures* are often an odd concept to grasp; whereas, to those with a functional programming background, they're a familiar and cozy concept. For the uninitiated, let's answer the question: What are closures?

Stated as simply as possible, a *closure* is a `Function` instance coupled with the local variables from its environment that are necessary for its execution.

When a function is declared, it has the ability to reference any variables that are in its scope at the point of declaration. These variables are carried along with the function *even after* the point of declaration has gone out of scope, *closing* the declaration.

The ability for callback functions to reference the local variables in effect when they were declared is an essential tool for writing effective JavaScript. Using a timer once again, let's look at the illustrative example in listing A.2 (the file appendixA/closure.html).

Listing A.2 Gaining access to the environment of a function declaration through closures

```
<html>
  <head>
    <title>Closure Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.js"></script>
    <script>
      $(function(){
        var local = 1;                                  ❶
        window.setInterval(function(){              ❷
          $('#display')
            .append('<div>At '+new Date()+' local='+local+'</div>');
          local++;                                    ❸
        },3000);
      });
    </script>
  </head>

  <body>
    <div id="display"></div>                ❹
  </body>
</html>
```

In this example, we define a ready handler that fires after the DOM loads. In this handler, we declare a local variable named `local` ❶ and assign it a numeric value of 1. We then use the `window.setInterval()` method to establish a timer that will fire every 3 seconds ❷. As the callback for the timer, we specify an inline function that references the `local` variable and shows the current time and the value of `local`, by writing a `<div>` element into an element named `display` that's defined in the page body ❹. As part of the callback, the `local` variable's value is also incremented ❸.

Prior to running this example, if we were unfamiliar with closures, we might look at this code and see some problems. We might surmise that, because the callback will fire off three seconds after the page is loaded (long after the ready handler has finished executing), the value of `local` is undefined during the execution of the callback function. After all, the block in which `local` is declared goes out of scope when the ready handler finishes, right?

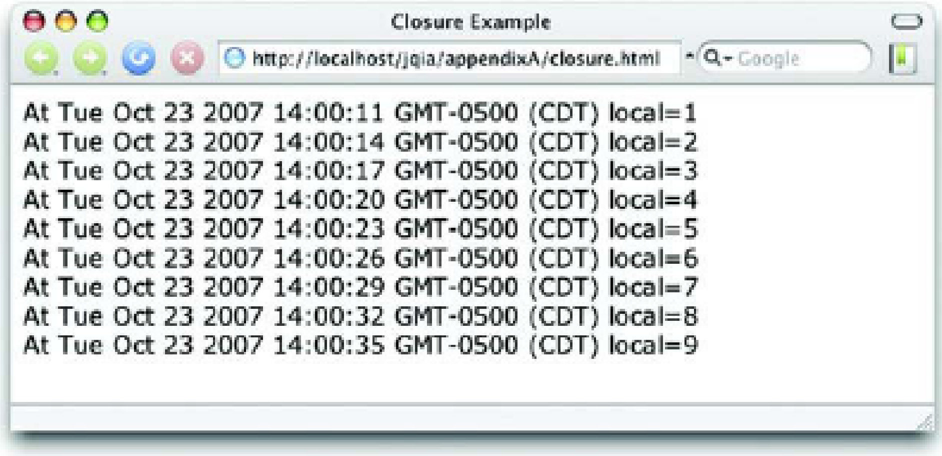
But on loading the page and letting it run for a short time, we see the display as shown in figure A.5.

It works! But how?

Although it *is* true that the block in which `local` is declared goes out of scope when the ready handler exits, the closure created by the declaration of the function, which includes `local`, stays in scope for the lifetime of the function.

NOTE You might have noted that the closure, as with all closures in JavaScript, was created implicitly without the need for explicit syntax as is required in some other languages that support closures. This is a double-edged sword that makes it easy to create closures (whether you intend to or not!) but can make them difficult to spot in the code.

Unintended closures can have unintended consequences. For example, circular references can lead to memory leaks. A classic example of this is the creation of DOM elements that refer back to closure variables, preventing those variables from being reclaimed.



```
At Tue Oct 23 2007 14:00:11 GMT-0500 (CDT) local=1
At Tue Oct 23 2007 14:00:14 GMT-0500 (CDT) local=2
At Tue Oct 23 2007 14:00:17 GMT-0500 (CDT) local=3
At Tue Oct 23 2007 14:00:20 GMT-0500 (CDT) local=4
At Tue Oct 23 2007 14:00:23 GMT-0500 (CDT) local=5
At Tue Oct 23 2007 14:00:26 GMT-0500 (CDT) local=6
At Tue Oct 23 2007 14:00:29 GMT-0500 (CDT) local=7
At Tue Oct 23 2007 14:00:32 GMT-0500 (CDT) local=8
At Tue Oct 23 2007 14:00:35 GMT-0500 (CDT) local=9
```

Figure A.5 Closures allow callbacks to access their environment even if that environment has gone out of scope.

Another important feature of closures is that a function context is never included as part of the closure. For example, the following code won't execute as we might expect:

```
...
this.id = 'someID';
$('*').each(function(){
   alert(this.id);
});
```

Remember that each function invocation has its own function context so that, in the code above, the function context within the callback function passed to `each()` is an element from the jQuery wrapped set, not the property of the outer function set to `'someID'`. Each invocation of the callback function displays an alert box showing the `id` of each element in the wrapped set in turn.

When access to the object serving as the function context in the outer function is needed, we can employ a common idiom to create a copy of the `this` reference in a local variable that *will* be included in the closure. Consider the following change to our example:

```
this.id = 'someID';
var outer = this;
$('*').each(function(){
   alert(outer.id);
});
```

The local variable `outer`, which is assigned a reference to the outer function's function context, becomes part of the closure and can be accessed in the callback function. The changed code now displays an alert showing the string `'someID'` as many times as there are elements in the wrapped set.

We'll find closures indispensable when creating elegant code using jQuery commands that utilize asynchronous callbacks, which is particularly true in the areas of Ajax requests and event handling.

A.3 Summary

JavaScript is a language that's widely used across the web, but it's often not *deeply* used by many of the page authors writing it. In this chapter, we introduced some of the deeper aspects of the language that we must understand to use jQuery effectively on our pages.

We saw that a JavaScript `Object` primarily exists to be a *container* for other objects. If you have an OO background, thinking of an `Object` instance as an unordered collection of name/value pairs may be a far cry from what you think of as an *object*, but it's an important concept to grasp when writing JavaScript of even moderate complexity.

Functions in JavaScript are *first-class citizens* that can be declared and referenced in a manner similar to the other object types. We can declare them using literal notation, store them in variables and object properties, and even pass them to other functions as parameters to serve as callback functions.

The term *function context* describes the object that's referenced by the `this` pointer during the invocation of a function. Although a function can be made to act like a method of an object by setting the object as the function context, functions aren't declared as methods of any single object. The manner of invocation (possibly explicitly controlled by the caller) determines the function context of the invocation.

Finally, we saw how a function declaration and its environment form a *closure* allowing the function, when later invoked, to access those local variables that become part of the closure.

With these concepts firmly under our belts, we're ready to face the challenges that confront us when writing effective JavaScript using jQuery on our pages.