Introducing JavaScript OSA

Contents

CHAPTER 1

What is JavaScript OSA?

JavaScript OSA is an OSA component that makes the JavaScript language available to you for scripting at system level. To understand what this means, you need to know what an OSA component is, so that's what the first section of this chapter is about. Then the JavaScript language is briefly described.

The OSA

The OSA stands for the Open Scripting Architecture. This section explains what it is.

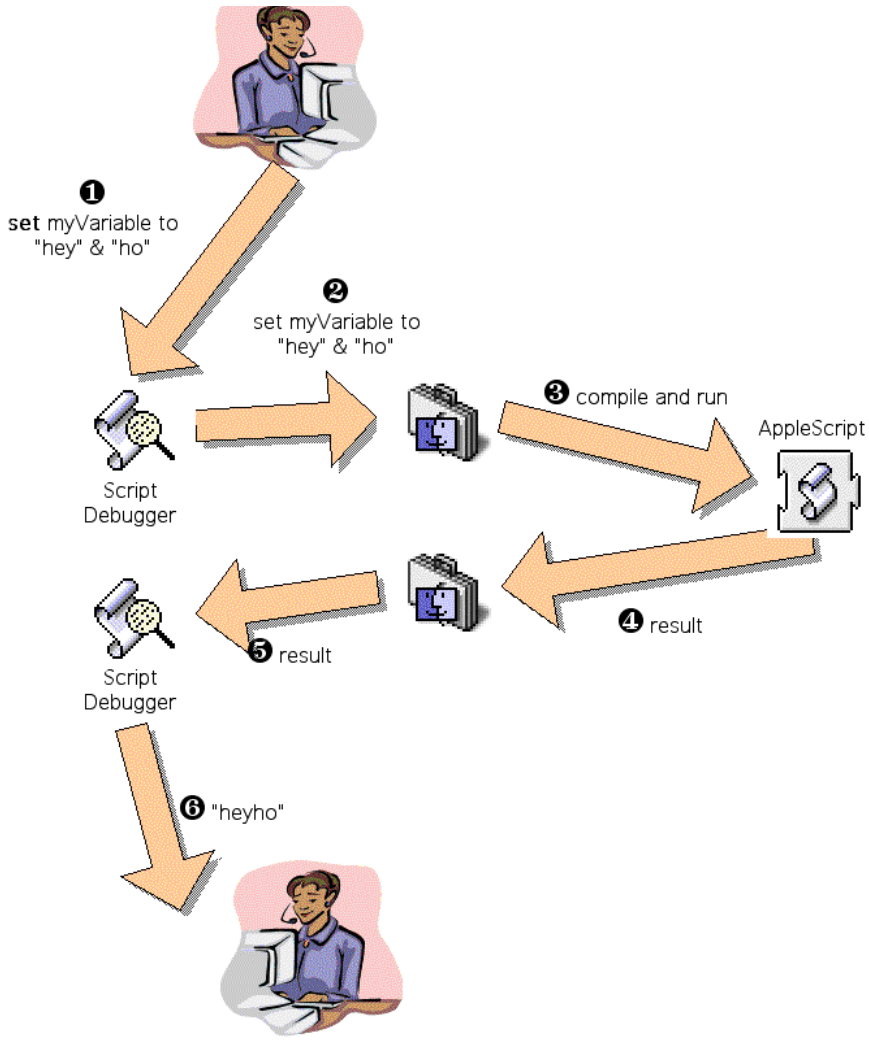
You may have written and executed AppleScript scripts without giving too much thought to what's going on behind the scenes. If you've been using Script Debugger to edit, compile and run your scripts, you may think of Script Debugger as doing all the work; but this isn't actually true. Script Debugger is an editor, and it does some very cool stuff to help you develop and debug your scripts, but when the time comes to compile your script and run it, Script Debugger doesn't actually do much of anything. It's the system which is actually doing all the work.

For example, let's say you create and run the following script in Script Debugger:

```
set myVariable to "hey" & "ho"
```

and Script Debugger replies with the result, which is `"heyho"`. Now, clearly somebody took the strings `"hey"` and `"ho"`, obeyed the command to concatenate them, created a variable `myVariable`, put `"heyho"` into it, and returned this as the result. But that somebody was not actually Script Debugger. In fact, Script Debugger merely acted as a kind of front end, a window for the user on what was really an operation behind the scenes at system level. We can envision this operation in simplified form by way of the diagram shown in Figure 1-1.

Figure 1-1
How Script Debugger Talks To the OSA



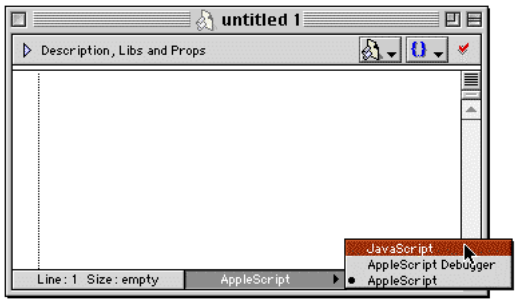
Let's talk about what's happening here. The user types the script into Script Debugger, but when the time comes to compile and run the script, Script Debugger hands it off the system. The system passes it on to AppleScript, which returns a result; Script Debugger presents this result to the user in readable form, but doesn't actually generate it.

Now, for our purposes, the interesting part of the diagram is steps 3 and 4, and in particular the presence of AppleScript as an independent entity. You may recognize the icon here as that of the AppleScript extension. It is this extension which is responsible for making the AppleScript language available at system level. What's important about this architecture is that it is open (it's an Open Scripting Architecture, remember?). This means that it is perfectly possible for a different entity, something other than the AppleScript extension, to make a different language available at system level. When this happens, you can compile and run scripts in that other language from within any OSA-savvy application, just as you can with AppleScript. (Such applications, giving you access to any OSA languages that may be present, include HyperCard, REALbasic, UserLand Frontier, Apple's Script Editor, and of course Script Debugger.)

The openness of the OSA is not very heavily used, by which I mean that there are not very many alternatives to AppleScript of the sort we're talking about here. One is UserLand's UserTalk, which is present as an OSA language whenever UserLand Frontier (or Radio UserLand) is running. Another is CE Software's QuicKeys Script, which is available if QuicKeys is installed.

Late Night Software's JavaScript OSA is another such OSA language. If the JavaScript extension is present when your Macintosh starts up, then the JavaScript language is available to OSA-savvy applications. This means that you can use Script Debugger (or Frontier, HyperCard, and so on) to compile and run scripts in the JavaScript language. To do so, simply flip the popup menu at the bottom of the script editing window to JavaScript, as shown in Figure 1-2.

Figure 1-2
Selecting JavaScript as OSA Language



Why JavaScript?

You now understand what JavaScript OSA is: it's an extension parallel to Apple's AppleScript extension. Just as the AppleScript extension makes AppleScript available as an OSA language, so the JavaScript extension makes JavaScript available as an OSA language.

But why would you want to use JavaScript as a scripting language? Without making any invidious comparisons with AppleScript, here are some possible considerations:

- JavaScript syntax is easy to read and write. It's a compact, neatly structured language. Someone who knows C, Java, UserTalk, or even Basic (including REALbasic), can probably learn JavaScript in a day.

- JavaScript has powerful native types. It has dynamic arrays, along with methods for working with them. It has a Date class, with a full complement of getter and setter methods. It has strings, along with a number of valuable methods for working with them, plus support for search and replace by way of regular expressions.
- JavaScript is object-based. Objects can have properties and methods; an object can have a constructor; there are static variables and methods; one object can inherit from another.
- JavaScript is a cross-platform universal standard. This means, among other things, that there are gazillions of JavaScript examples out there, many of them embedded in Web pages, just waiting for you to take advantage of them as working models. Information about JavaScript is extremely plentiful on the Internet.
- JavaScript is cool. Most of its best features are parallel to the finest aspects of Java, UserTalk, and Lisp. Of particularly noteworthy coolness are the following: variables are not strongly typed, are automatically converted as necessary, and can be queried at runtime as to their datatype; names are lexically scoped; blocks can be labeled for use with `break` and `continue`; properties can be added dynamically to an object, and an object can be treated as an associative array; strings can be secondarily evaluated as commands; object methods can be called with a supplied `this`-substitute; functions can be passed any number of parameters, can learn who called them, are first-class datatypes, and can be defined anonymously.

In sum, the real reason for using JavaScript is not merely to script with it but to program with it. JavaScript is a great language, but hitherto it has been available chiefly from inside a browser, which is rather a restrictive environment. The JavaScript OSA extension makes JavaScript available to you for programming tasks in general.

CHAPTER 2

JavaScript Examples

This chapter presents a few brief examples of JavaScript in use from within Script Debugger. If you're already familiar with the JavaScript language, you can skip this chapter.

Learning JavaScript

Although the examples in this chapter were developed specifically to illustrate the nature of the JavaScript language, no actual attempt will be made here to teach you any JavaScript. The best way to learn the language is to use David Flanagan's splendid book, *JavaScript: The Definitive Guide*, from O'Reilly & Associates. At present (3rd edition), this is brought up only through JavaScript 1.2 (though it does mention certain JavaScript 1.3 features that were incipient at the time of publication), so you will want to supplement it with further readings. A very helpful document is Netscape's reference on the core features of the current version of JavaScript, which is presently here:

http://developer.netscape.com/docs/manuals/js/core/jsref/CoreReferenceJS14.pdf

Unfortunately, even this is not up to date; the codebase documented by the Netscape reference is JavaScript 1.4, whereas JavaScript OSA is based on JavaScript 1.5. You can obtain the ECMA specification for JavaScript 1.5 here:

http://www.mozilla.org/js/language/E262-3.pdf

This document, however, is technical rather than instructive; and it still lacks information about important features supported by JavaScript OSA's implementation of the JavaScript language. For example, it doesn't mention the extremely useful `__proto__` property (which fortunately is explained by David Flanagan's book). And JavaScript 1.5 contains some bleeding-edge new features, such as getters and setters, on which documentation can be extremely hard to find; the best I've been able to come up with is this Web page:

http://www.mozilla.org/js/js15.html

But even this only gives clues and hints, not clear explanations, about what's new in JavaScript 1.5. The wisest course is probably to confine oneself for now to what is documented in the Netscape JavaScript 1.4 reference, plus the `__proto__` property documented by David Flanagan's book.

It's important to be clear about what elements of JavaScript are present in the JavaScript OSA implementation. Many people are introduced to JavaScript through its use as a client language inside a Web browser, and to them, JavaScript is likely to mean the DOM, the document object model which makes available various aspects of a browser document such as its images, elements of its forms, details of its HTML, the position of the window, and so forth. But in JavaScript OSA, there is no browser, so there is no DOM. Furthermore, there is no connection with Java. In JavaScript as implemented in a browser, it is assumed that Java is present in the same environment, and the two languages are able to communicate through LiveConnect and JavaScript's `java` object. But in JavaScript OSA, there is no `java` object, and packages in general are absent.

On the other hand, JavaScript OSA has some power that normal JavaScript does not, because it extends JavaScript by way of the `MacOS` object, which makes available a number of supplementary functionalities, the most significant being the ability to run AppleScript commands and send Apple events. This makes JavaScript OSA suitable for the main purpose for which you were probably doing any scripting in the first place, namely, to drive other applications. The `MacOS` object is dealt with starting in the next chapter.

On to the examples! These have been constructed primarily to illustrate the compactness and clarity of JavaScript, and to show off its elegance and power. They are pure JavaScript examples; there is nothing special about them with regard to JavaScript OSA or Script Debugger. Do note, however, that some attention is given to the question of how to return a useful reply to the host environment (assumed to be Script Debugger). Users accustomed to working with JavaScript in a browser context should observe that in JavaScript OSA you are probably going to want to make sure you return a value of some sort. The reason this deserves emphasis is that even though JavaScript is a functional language, in the browser context it is more commonly used for its side effects, such as calling `document.write()` or setting document properties; thus, the

script typically displays its effects in the course of being run. With JavaScript OSA, however, where you initiate an action by running the script from some OSA-compliant environment such as Script Debugger, the value returned at the end is the only way you have of knowing just what has occurred, or of deriving useful information from the running of the script. Furthermore, since your main JavaScript routine cannot have a `return` statement, the value returned is going to be the value of the last statement executed. This works fine, because every JavaScript statement does have a value; but you may not be certain what it is. A reliable strategy, therefore, is to have the last executed statement consist of just the expression whose value you wish to return. That's the strategy adopted here.

Sieve of Eratosthenes

We begin by implementing the Sieve of Eratosthenes, a crude but effective device for finding all prime numbers from 2 to any integer *n*. The idea is to cycle forward through the integers remaining in the sieve, eliminating at every step those divisible by the integer we are presently looking at, except for the integer itself. Thus, we shall first eliminate all integers divisible by 2 (except for 2), then all integers divisible by 3 (except for 3), then all integers divisible by 5 (this is the next integer, because 4 was eliminated by the first step), and so forth. When we are done we will be left with just the prime numbers.

Our strategy will be in three parts. First, we will generate all the integers between 2 and our maximum *n*, taking advantage of JavaScript's dynamic arrays. Here, *n* is taken to be 500:

```
var ints = new Array();
for (var i = 2; i < 500; i++) ints.push(i);
```

Next, we write a generalized array filter function. Given any array and any function that returns a Boolean (or something coercible to a Boolean), we eliminate every element of the array that doesn't pass the test. We take advantage of the fact that in JavaScript a function such as our Boolean test is a first-class citizen and can be passed as a parameter, as well as of the fact that arrays are passed by reference.

```
function filter (arr, test) {
   for (var i = arr.length - 1; i >= 0; i--)
      if (!test(arr[i])) arr.splice(i,1);
}
```

Finally, we loop through the array, calling our filter function repeatedly. This is the really cool part. On each pass, we use the fact that JavaScript functions are first-class citizens to form a new anonymous function to pass as a Boolean test to the filter function. In other words, the first time through, we make a function that tests whether its parameter is divisible by 2; the second time through, we make a function that tests whether its parameter is divisible by 3; and so forth. The reason this is possible is because of lexical scoping; in particular, the meaning of the variable `p` inside our test function depends upon the value of `p` at the time the test function is formed, which means that it differs each time through the loop.

```
var ix = 0;
do {
   var p = ints[ix++];
   var f = function (i) {return i % p != 0 || i == p;}
   filter (ints, f);
} while (ix < ints.length);
```

The last step is to return a value, which in this case is the `ints` array. In Script Debugger, this is simply represented to us directly in the Script Results window as a list of integers.

```
ints;
```

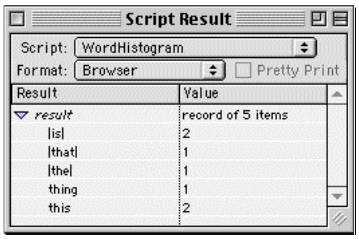
Word Histogram Example

Next we'll write a word-histogram program — that is, we will count the occurrences of every distinct word in a string, and report the counts. This is almost trivial in JavaScript because an object is a dynamic associative array; the task of mapping names to values, along with all the details of maintaining a dynamic internal data structure, are thus completely handled for us. In our first attempt, we start with a literal array consisting of the individual words. If a word doesn't already appear in the map, we add it, with a count of 1; if it does, we increment its count.

```
var wordArray = ["this", "is", "the", "thing", "that", "this", "is"]
var map = new Object;
var wordCount = 0;
var u = wordArray.length;
for (var i = 0; i < u; i++) {
   var aWord = wordArray[i];
   if (aWord.length)
      map[aWord] == undefined ? map[aWord]=1 : map[aWord]++;
}
map;
```

As with the previous example, the final statement ensures that the value of the `map` object is the result returned to Script Debugger. Objects are nicely represented by Script Debugger as records. In Browser view, this is what you'll see in the Script Result window (see Figure 2-1; the pipes around the names of the first three entries are just because those are AppleScript reserved words).

Figure 2-1 *Results From JavaScript in Script Debugger*



Now let's rewrite the example so far, expanding it to derive the word array from a string. Our definition of a word will be overly simplistic, but quite sufficient to hint at the power of JavaScript's built-in regular expressions capabilities; we will take a word to be just any continuous run of the letters `a` through `z`:

```
var s = "This is the thing that this is."
s = s.toLowerCase();
var wordArray = s.split(/[^a-z]+/);
var map = new Object;
var wordCount = 0;
var u = wordArray.length;
for (var i = 0; i < u; i++) {
   var aWord = wordArray[i];
   if (aWord.length)
```

```
      map[aWord] == undefined ? map[aWord]=1 : map[aWord]++;
}
map;
```

Let's now say that our real purpose is to learn which words are most frequently used; therefore we want to see the properties of `map` in order of their value, from highest to lowest. The properties of an object, however, cannot be sorted. But the elements of an array can be! Therefore, having constructed `map`, we will feed its name–value pairs into an array called `sorter`, taking advantage of JavaScript's extremely cool ability to cycle through the properties of an object without our having to know their names. Then we sort the `sorter` array on the second element of each of its items (the count for that item), again taking advantage of JavaScript's ability to construct and pass an anonymous function:

```
var sorter = [];
for (prop in map)
   sorter.push([prop, map[prop]]);
sorter.sort(function(a,b) {return b[1]-a[1];});
sorter;
```

If you feel that this doesn't present the results in a very pretty way, you can add the following code to cycle through the `sorter` array and build a string which will look good in Script Debugger's Script Result window, especially when the format is set to Best:

```
s = "";
for (i = 0; i < u; i++)
   s += (sorter[i][0] + ": " +  sorter[i][1] + "\r");
s;
```

Later, we'll make the example practical by permitting the user to run this operation on an arbitrary text file.

Area of a Polygon

Finally, let's have an example that illustrates JavaScript's ability to package properties and methods into objects. We're going to calculate the area of a polygon, given its vertices. So we're going to need a Polygon object and a Point object. For the sake of simplicity, we will assume that the polygon is convex; therefore it will suffice to break it up into triangles and sum their areas. So we will need a Triangle object, and the area of a triangle depends upon the length of its sides, so we will need a Line object.

We begin at the bottom, with points and lines. As usual in JavaScript, we'll initialize properties in the constructor while defining functions in the prototype. A Point will consist of an `x` and `y` value; a Line will consist of two points and a `length()` function that uses Pythagoras's Theorem.

```
function Point (x, y) {
   this.x = x;
   this.y = y;
}

function Line (pt1, pt2) {
   this.pt1 = pt1;
   this.pt2 = pt2;
}

Line.prototype.length = function () {
```

```
   var side1 = this.pt1.x-this.pt2.x;
   var side2 = this.pt1.y-this.pt2.y;
   return Math.sqrt(side1*side1 + side2*side2);
}
```

To test our code so far, we can append the following couple of lines:

```
var aLine = new Line (new Point (1,1), new Point (2,2));
aLine.length();
```

We expect to get the square root of two, and we do. So far, so good.

Now we'll add the triangle. A triangle will be initialized by three points, but since what we really need is the three sides, we'll convert the points to lines in the constructor. We can then write the triangle's `area()` function, which is simply a matter of asking each side for its length and applying Hero's formula:

```
function Triangle (pt1, pt2, pt3) {
   this.side1 = new Line (pt1, pt2);
   this.side2 = new Line (pt2, pt3);
   this.side3 = new Line (pt3, pt1);
}

Triangle.prototype.area = function () {
   var a = this.side1.length();
   var b = this.side2.length();
   var c = this.side3.length();
   var s = (a + b + c) / 2;
   return Math.sqrt(s*(s-a)*(s-b)*(s-c))
}
```

To test again, we can append the following:

```
var aTriangle = new Triangle
   (new Point(0,0), new Point(2,0), new Point (0,2));
aTriangle.area();
```

We expect to get 2 (half the area of a two-by-two square), and that's what we do get.

Finally we are ready for a polygon. The constructor is rather tricky; we don't know how many points the polygon will have, so we don't even try to define constructor parameters; instead, we look in JavaScript's local `arguments` variable. There are two chief cases. If there are three parameters, the polygon is a triangle, so we make a triangle and return that. Otherwise, we just store the arguments in a property called `pts`. However, we first check to see if there is only *one* parameter, which is an array. This is to cover a special case: we would like to be able to call the constructor using two different possible syntaxes, either listing all the points or else passing an array which lists them. The reason why we need to do this will be clear in a moment; the point for now is that if we are called with just one parameter when that parameter is an array, we extract that array before proceeding. After that, since `arguments` itself is an array, we've got an array of points either way.

```
function Polygon () {
   var args = arguments;
   if (args.length == 1 && args[0].constructor == Array)
      args = args[0];
   if (args.length == 3)
```

```
      return new Triangle (args[0], args[1], args[2])
   else
      this.pts = args;
}
```

Now we are ready to write the polygon's `area()` function. Our approach, remember, will be to break the polygon into triangles. However, I am much too lazy and ignorant to figure out how to do this; the only thing I know how to do is to form *one* triangle, namely the one consisting of the first three vertices of the polygon, and to break it off from the polygon by throwing away its middle vertex, leaving another polygon, one vertex smaller. At this point the word "recursion" leaps inevitably to mind, and the algorithm becomes obvious: to find the area of a convex polygon, break the polygon into a triangle and the remaining polygon, and sum their areas.

```
Polygon.prototype.area = function () {
   var aTriangle = new Triangle
      (this.pts[0], this.pts[1], this.pts[2]);
   var otherPts = ([]).concat(this.pts); otherPts.splice(1,1);
   var theRest = new Polygon (otherPts);
   return aTriangle.area() + theRest.area();
}
```

The second line here is a bit tricky (and rather clever, if I do say so myself): we use JavaScript's powerful `concat()` function to copy our array into a temporary variable, throw away the copy's second element, and pass the remainder as the vertices of our new smaller polygon. This is why it was desirable that the polygon constructor should be able to accept an array.

To test, we will construct the regular hexagon inscribed within the unit circle centered at the origin and ask for its area:

```
var pt1 = new Point (1,0);
var a60 = Math.PI/3;
var pt2 = new Point (Math.cos(a60), Math.sin(a60));
var pt3 = new Point (Math.cos(a60*2), Math.sin(a60*2));
var pt4 = new Point (Math.cos(a60*3), Math.sin(a60*3));
var pt5 = new Point (Math.cos(a60*4), Math.sin(a60*4));
var pt6 = new Point (Math.cos(a60*5), Math.sin(a60*5));
new Polygon (pt1, pt2, pt3, pt4, pt5, pt6).area();
```

We expect to get about 2.5981, and we do.

CHAPTER 3

The Core Object

JavaScript OSA enhances JavaScript through the addition of a `Core` object. This object is used for obtaining certain platform-independent information about the environment (in the expectation that perhaps a cross-platform implementation will some day be created) , and for a little basic interface communication with the user. (The `Core.load()` method, used to implement libraries, is discussed in a subsequent chapter.)

Talking to the User

`Core.message()` puts up a simple dialog with just an OK button. The text of the dialog is the parameter of the function. So, for example: `Core.message("Hello world!")`

`Core.warning()` is just like `Core.message()`, except that an alert icon appears in the dialog. For example: `Core.warning("Danger, Will Robinson!")`

`Core.error()` is just like `Core.warning()`, except that the icon is the stop sign. For example: `Core.error("I'm sorry, Dave, I really can't do that.")`

`Core.pick()` puts up a dialog where the user can choose from a scrolling list of alternatives. The first parameter is the prompt; the second parameter is the list, as an array of strings. The result is the text of the item chosen, or `undefined` if the user cancels. For example: `Core.pick("Pick a Pep Boy:", ["Manny", "Moe", "Jack"])`

Places

`Core.home` is the pathname of the desktop folder.

`Core.me` is the pathname of the application running the current script.

`Core.temp` is the pathname to the folder where temporary files are kept. On the Macintosh, this is the system's invisible Temporary Items folder.

Where Am I?

`Core.platform` returns the name of the platform on which we are running; at present this will be `MacOS` (for Mac OS 8 and 9) or `MacOSX` (for Mac OS X).

`Core.platformVersion` returns the system version. On my machine this is `9.0.4`.

Who Am I?

`Core.version` is the version of JavaScript OSA; at present this will probably be `1.0` or some variant thereof.

`Core.copyright` is Late Night Software's copyright notice for JavaScript OSA.

`Core.URL` is the URL of Late Night Software's main support Web page for JavaScript OSA.

CHAPTER 4

The MacOS Object

JavaScript OSA enhances JavaScript through the addition of a `MacOS` object. It is through this object that communication takes place with other applications. It is also used to get references to folders, files, and applications, as well as for a few types of dialog to be presented to the user. In all probability, scripting other applications will be most important to you, since that's why you're probably doing any scripting in the first place. However, since the other uses are simpler, that's what we'll deal with first; use of the `MacOS` object to drive other applications is discussed in subsequent chapters.

System Information

`MacOS.gestalt()` permits you to call the system's `gestalt()` function. The parameter is the selector specifying what information you want. For the selector codes and their meanings, see http://developer.apple.com/techpubs/mac/OSUtilities/OSUtilities-27.html and http://developer.apple.com/techpubs/mac/OSUtilities/OSUtilities-13.html. For example, on my machine, `MacOS.gestalt("lram")` returns `268435456`, revealing that I have 256MB of RAM.

Talking to the User

`MacOS.ask()` puts up a dialog containing a prompt (which is the parameter handed to it), a Yes button, and a No button. It returns a Boolean corresponding the user's response. For example: `MacOS.ask("Shall I erase your hard disk?")`.

`MacOS.beep()` beeps.

Places

`MacOS.desktopFolder`, `MacOS.appleMenuFolder`, `MacOS.controlPanelsFolder`, `MacOS.extensionsFolder`, `MacOS.fontsFolder`, `MacOS.preferencesFolder`, `MacOS.temporaryItemsFolder`, `MacOS.startupItemsFolder`, `MacOS.systemFolder`, `MacOS.trashFolder`. These all return references to the special folders which they denote. They are all actually just special cases of `MacOS.findFolder()`, which is treated next.

`MacOS.findFolder()` returns a reference to a special folder; its parameter is the four-letter code specifying the folder. A list of the chief such four-letter codes may be found at http://developer.apple.com/techpubs/macos8/Files/FolderManager/FolderMgrRef/FolderMgrRef.d.html.

`MacOS.pathToMe()` returns a `MacOS.FileSpec` object referring to the application from which the current script is being run.

What's returned by the methods and properties listed in this section is not a pathname. It will be represented as a pathname externally if you ask for its value from within Script Debugger, but that's because it has been coerced to a string through its `toString()` method. It is actually a `MacOS.FileSpec` object, suitable for querying through its various properties (these properties are discussed in a later section). For example, to learn the name of the volume on which the Desktop Folder resides, you can say `MacOS.desktopFolder.volume`.

The distinction being made here is an important one, because the `Core` properties that look equivalent to some of what's listed in this section, are not in fact equivalent; the `Core` properties really *are* pathnames. So, for example, asking Script Debugger for `Core.home` and `MacOS.desktopFolder`, you seem to be getting the same answer. But you aren't, as is proven by the fact that you won't learn anything by asking for `Core.home.volume`. Similarly, `MacOS.pathToMe()` is not identical to `Core.me`, which is just a pathname.

Applications

A reference to an application, as treated in this section, is a `MacOS.AEApp` object, suitable for sending Apple events to. Such a reference can be obtained in the following ways. (**Warning**: creating a reference to an application will launch the application if it is not running!)

`MacOS.finder()`, `MacOS.desktop()`, `MacOS.urlAccess()`, `MacOS.bbEdit()`. These all refer to common applications. The first two denote the Finder (called equivalently the Desktop because that's its Mac OS X

name); next is the URL Access Scripting Addition; finally we have Bare Bones' text editor BBEdit. These are all just special cases of `MacOS.appBySignature()`, which is treated next.

`MacOS.appBySignature()` takes as parameter the four-letter creator code of an application.

`MacOS.appSelf()` returns a reference to the application from which the current script is being run.

The question of how to send Apple events to an application is discussed starting in the next chapter.

The two verbs `MacOS.appBySignature()` and `MacOS.appSelf()` take an optional extra parameter, a Boolean specifying whether the application's dictionary should be loaded. The default is `true`, because if you're about to send an AppleScript-type command to the application, you're going to need the dictionary. This point will be explained further later on.

Targeting of remote applications is not implemented at the present time.

FileSpecs

In the previous sections we met several verbs and properties of the `MacOS` object which return a `MacOS.FileSpec` object. It is now time to discuss what you can do with such an object. But first, let's summarize how you obtain a `MacOS.FileSpec` object.

- As we've already seen, you can call `MacOS.pathToMe()`; or you can call `MacOS.FindFolder()` or any of the special cases which actually call it.
- Having obtained a `MacOS.AEApp`, as described in the previous section, you can ask for its `_path` property. Recall that you are actually driving the application in question when you do this, so it will be launched if it is not already running.
- You can construct it from a pathname. For example, on my machine, `new MacOS.FileSpec("main:Installer Logs")` returns a `MacOS.FileSpec` object which I am then free to manipulate. In such a construction, all the items along the pathname must exist, except the last one (because otherwise you could never refer to a file in order to create it); otherwise, you'll get an exception.
- You can return it from an Apple event. Any Apple event which returns an alias or filespec is returning a value suitable for immediate use as a `MacOS.FileSpec` object. For example, here is how to call the Choose File scripting extension to let the user choose a textfile in a standard file dialog; afterwards, if the user doesn't cancel the dialog, `f` is a `MacOS.FileSpec` (the details of this example will be explained in the next chapter):

```
var f = MacOS.appSelf().choose_file(null, ["TEXT", "ttro"]);
```

We come now to how you can manipulate a `MacOS.FileSpec`. It has three chief properties: `name`, `container`, and `volume`, which are respectively the file's name, its containing folder, and its top-level containing volume. These obviously permit you to work your way up the file hierarchy; you can also work your way down the file hierarchy by using string addition and pathname construction. For example, on my machine, the following yields a valid `MacOS.FileSpec` referring to an existing folder:

```
var f = new MacOS.FileSpec("main:");
f = new MacOS.FileSpec (f + "Installer Logs");
```

A `MacOS.FileSpec` also has a `folder` property which is a Boolean indicating whether or not the item referred to is a folder.

There is no property indicating whether the item referred to by a `MacOS.FileSpec` actually exists. In general, you are expected to use the Finder to ask questions of any complexity about items on disk.

CHAPTER 5

Driving Other Applications

We come at last to the moment you've been waiting for: how to send AppleScript-like scripting commands with JavaScript OSA.

The Basics

Driving other applications with JavaScript OSA turns out to be extremely easy, because of a piece of clever sleight-of-hand that goes on behind the scenes. You start by obtaining a `MacOS.AEApp` referring to the application to which you intend to send a command. It was explained in an earlier section how to obtain this reference; and at that time, I noted that there is an optional parameter to the verbs which yield such a reference, specifying whether or not the application's dictionary should be loaded. The default value of this parameter is `true`. Here's the really cool part: when the application's dictionary is loaded, the entries in the dictionary are automatically transformed into methods and properties of the `MacOS.AEApp` object, and of one another, in conformance with the object hierarchy described in the dictionary. You can therefore use almost exactly the same commands you would use in writing an AppleScript!

We'll get into details as we go along, but just to get started, here are the basics.

- There are no `get()` or `set()` verbs. Simply capture or assign a value directly using the assignment operator (`=`).
- The object hierarchy (containment) is expressed by dot-notation.
- An object's index, whether by name or by number, is expressed by array notation.
- No spaces are allowed in the middle of a JavaScript word, so if one occurs in an AppleScript name, you have to substitute an underscore.

Knowing just these rules, you can immediately start sending some AppleScript-type commands. Here's an example. For instructional purposes, we supply first the AppleScript version, then the JavaScript OSA equivalent command.

```
// tell application "Finder" to get
//    creation date of item "main:installer logs:"
MacOS.finder().item["main:installer logs"].creation_date;
```

Here's another:

```
// tell application "Microsoft Word" to get sentence 1 of window 1
MacOS.appBySignature("MSWD").window[1].sentence[1];
```

If you try that second example, you'll discover something that is not always so clear when you're using AppleScript, but which is immediately obvious when you look at the result from JavaScript OSA: what you've just fetched is a reference to the sentence, not the text of the sentence. That's good, because the resulting object can now be used as a reference in whatever way you like; but it's bad if what you wanted was the text!

The way to make sure you're getting a value and not a reference is usually to apply the object's `valueOf()` method:

```
MacOS.appBySignature("MSWD").window[1].sentence[1].valueOf();
```

But watch out for applications that define their own property for getting a value instead of a reference. For example:

```
// tell application "Microsoft Excel" to get
//    Value of Column 2 of Row 1 of Worksheet 1
MacOS.appBySignature("XCEL").Worksheet[1].Row[1].Cell[2].Value;
```

That example reveals some other rather tricky points. The dictionary looks as if asking for a column of a row would work, and in AppleScript it does; but JavaScript OSA chokes when you try this, so you have to ask for a cell of a row instead. Also, notice the

capitalization. AppleScript takes care of this for you, but JavaScript is case-sensitive and doesn't compensate. If you fail to capitalize any of the class names properly, you get an error.

By the way, those accustomed to AppleScript may be surprised at the precise moment in the process at which this sort of error occurs. In AppleScript, when you make a mistake with respect to the target application's dictionary, the script doesn't even compile; this is because AppleScript compilation involves a translation into "byte code" which can take place only in the presence of the dictionary, so when you compile, the dictionary is loaded and the script is interpreted in the light of the dictionary. In JavaScript, on the other hand, compilation means compiling the JavaScript code itself, and no more: anything that is syntactically legal JavaScript will compile, and problems with respect to the dictionary won't reveal themselves until runtime.

Technically, a collection reference such as `MacOS.finder().item` is a `MacOS.AEColl` object, which is a subclass of JavaScript's `Array`; a reference to a particular element such as `MacOS.finder().item["main:installer logs"]` is a `MacOS.AEClass` object; and `MacOS.AEApp` is a subclass of `MacOS.AEClass`. A conscious awareness of these facts, however, is neither necessary nor particularly useful in order to create object references and drive applications; I mention them only for the sake of rigorous completeness.

Reference Forms

Here is a summary of the various ways in which an element can be specified.

- **Name or index number**. This, as we have already seen, is expressed in JavaScript OSA through the use of array notation.

```
MacOS.finder().disk["main"];
MacOS.finder().disk[1];
```

Actually, to be quite honest, you can refer to an element by name using ordinary dot-notation if the name is a legal JavaScript property name and not a reserved word, as in my use of `main` in the following expression:

```
MacOS.finder().disk.main.file[1].name;
```

However, this sort of thing is confusing and not recommended.

- **Named position**. JavaScript OSA supplies properties `every`, `first`, `middle`, `any`, `last`.

```
MacOS.finder().disk.every.name;
```

- **Range**. JavaScript OSA supplies a method `byRange`, taking two parameters, the lower bound index and the upper bound index.

```
MacOS.appBySignature("MSWD").word.byRange(1,4).valueOf();
```

As all veteran AppleScripters know, the previous example gets you a list of words; to get the actual text across a range, you have to use syntax like this:

```
//tell application "Microsoft Word" to get
//    text from word 1 to word 4 of document 1
with (MacOS.appBySignature("MSWD"))
      text.byRange(word[1], word[4]).valueOf();
```

- **ID**. JavaScript OSA supplies a method `byID`, whose parameter is the ID number.

```
MacOS.finder().disk.byID(-1).name
```

- **Relative**. JavaScript OSA supplies properties `next`, `previous`.

```
var MSWD = MacOS.appBySignature ("MSWD");
var aWord = MSWD.document[1].word.first;
aWord.next.valueOf();
```

Boolean filter tests (`whose`-clauses) are not implemented. On the whole, however, they are not needed, because you can cycle through the objects of a collection within JavaScript itself. This is a behavior you have probably trained yourself not to use in AppleScript, where it is much more efficient to ask the target application to do the looping and selection first, and then to hand back the results. However, in JavaScript there is no good reason not to do the looping yourself.

```
// tell application "Finder" to get
//    every disk whose name does not contain "e"
var allDiskNames = MacOS.finder().disk.every.name;
var allDisks = MacOS.finder().disk.every.valueOf();
var result = [];
for (var i = 0; i < allDiskNames.length; i++)
   if (allDiskNames[i].indexOf("e") < 0)
      result.push(allDisks[i]);
result;
```

Be aware of the difference between array indexing in AppleScript and JavaScript. AppleScript arrays are 1-based; JavaScript arrays are 0-based. This will not pose a problem provided you remember whom you're talking to. For instance, in the preceding example, we deliberately start with two arrays, one containing the disk names, the other containing references to the disks. These are both JavaScript arrays, so there's no problem. Things would have been different if we had not gathered the array of disk references at the outset; we can ask for each disk reference as the need arises, during the loop, but in that case we must compensate for the difference in indexing:

```
var allDiskNames = MacOS.finder().disk.every.name;
var result = [];
for (var i = 0; i < allDiskNames.length; i++)
   if (allDiskNames[i].indexOf("e") < 0)
      result.push(MacOS.finder().disk[i+1].valueOf());
result;
```

Coercion

You may be wondering at this point about how to ask for a coercion. In AppleScript there is a `get as` command; but since there is no `get` in JavaScript OSA, obviously there is no `get as` either. There are two techniques. If you're asking for a property, you can ask instead using a method whose name is the name of the property followed by `As`, where the parameter is the four-letter data type code. So, for example:

```
// tell app "Microsoft Excel" to get
//    Value of Column 2 of Row 1 of Worksheet 1 as integer
MacOS.appBySignature("XCEL").Worksheet[1].Row[1].Cell[2].
   ValueAs("long");
```

On the other hand, if you're asking for an element's value with `valueOf()`, you can supply a data type code to request the coercion:

```
// tell application "Microsoft Word" to get
//    word 1 of window 2 as integer
MacOS.appBySignature("MSWD").window[2].word[1].valueOf("long");
```

Counting

The `count` verb is not supported. Instead, specify the desired collection and, since this is a JavaScript array, ask for its `length`. The collection can be referred to in its singular or plural forms.

```
//tell application "Finder" to count files of control panels folder
MacOS.finder().folder[MacOS.controlPanelsFolder].file.length;
```

Bad Dictionary

Sometimes (rather more often than not, in fact) you come up against a bad dictionary, where the containment hierarchy is not properly structured. In such cases, you need to throw in an intermediate step where you set the `MacOS.AEApp`'s `_strict` property to `false`. This allows a looser interpretation of the hierarchy. A good example is Eudora:

```
// tell application "Eudora" to get subject of message 5 of mailbox "In"
var eudora = MacOS.appBySignature("CSOm");
eudora._strict = false;
eudora.mailbox["In"].message[5].subject;
```

Without that intermediate step, JavaScript OSA will complain that there is no `mailbox` property — because, strictly speaking, there is no such class defined as being top-level in its dictionary. So the solution is to speak less strictly!

Similarly for BBEdit; here, we show an alternate syntax using `with`:

```
with (MacOS.bbEdit()) {
   _strict = false;
   window[1].word[3] = "cool";}
```

You need make only one `_strict` setting per session; once you've said in a script that a certain application's dictionary interpretation is to be non-strict, it will remain non-strict throughout, even if you summon a new instance of that application as a `MacOS.AEApp`.

Insertion Locations

Absolute insertion locations are often expressed as simple references to an element of a class provided by the application. For example:

```
// tell application "Microsoft Word" to select insertion point 4
with (MacOS.appBySignature("MSWD"))
   select(insertion_point[4]);
```

Relative insertion locations are expressed using the properties `atNext`, `toNext`, `atPrevious`, `toPrevious`, `atBefore`, `toBefore`, `atAfter`, `toAfter`, `atBeginning`, `toBeginning`, `atEnd`, `toEnd`, `atReplace`, `toReplace`. The pairs are synonyms; that is, `atNext` and `toNext` are identical, and so forth. These properties modify the object to which they apply, in a perfectly natural way.

```
// tell document 1 of application "Microsoft Word" to move word 1 to end
with (MacOS.appBySignature("MSWD"))
   move (document[1].word[1], document[1].atEnd)
```

When creating an object, you also need to be aware of the fact that what you're being asked to specify as the first parameter of the `make()` verb is a class, not an object. In AppleScript this distinction is obscured, but in JavaScript OSA you have to maintain it strictly. To specify a class, use the `_types` property followed by the name of the class, using dot-notation. For example, the document class is denoted by `_types.document`.

Thus, here's how to make a new message in Eudora:

```
// tell application "Eudora" to make new message at end of mailbox "Out"
var eudora = MacOS.appBySignature("CSOm");
eudora._strict = false;
with (eudora)
   make(_types.message, mailbox["Out"].atEnd);
```

In this example, the `with` syntax is essential. Since `_types` and `mailbox` are used independently inside the parentheses, there would otherwise be no way for JavaScript to know that they are supposed to be properties of `eudora`. There is an alternative, of course; you can be explicit, at the cost of repetition:

```
eudora.make(eudora._types.message, eudora.mailbox["Out"].atEnd);
```

Here's another example, where `make()` takes a third parameter:

```
// tell app "Tex-Edit Plus" to make
//    word at after word 2 of window 1 with data "not"
with (MacOS.appBySignature("TBB6")) {
   _strict = false;
   make (_types.word, window[1].word[2].atAfter, "not"); }
```

Missing Parameter and Named Parameter

The last example from the previous section raises the question of how to specify parameters to a function. Our use of `make()` with three parameters worked because Tex-Edit's `make()` command is defined such that its first three parameters are the class, the location, and the data, which are exactly the parameters we wanted. But what about BBEdit, where the `make()` parameters are class, location, properties, and data? How can we supply the data without the properties? In AppleScript, this is not a problem because we name every parameter explicitly except for the direct object; so here, `with data "not"`. How do we do the same thing in JavaScript OSA?

It turns out that we have two choices. One is to supply `null` as the value of any missing parameters between the parameters we do supply. So:

```
// tell app "BBEdit 6.0" to make
//    new word at end of word 2 of window 1 with data "not"
with (MacOS.bbEdit()) {
   _strict = false;
   make (_types.word, window[1].word[2].atEnd, null, "not"); }
```

The other alternative is to use a named parameter. This is quite tricky. First, in giving the name of the verb, you have to put an underscore in front of it, to signify to JavaScript OSA that you're going to be using named parameters. Second, you construct the parameter names and values as an object, and pass that object as the single parameter of the verb. (You cannot mix named and unnamed parameter syntax in a single call.) Third, you have to be careful of reserved JavaScript words that you may be compelled to use as names of object properties; you can do it, but you have to refer to them indirectly, in quotes. Look at how we use quotes to avoid direct use of the reserved word `new` in this example:

```
// tell app "BBEdit 6.0" to make
//    new word at end of word 2 of window 1 with data "not"
with (MacOS.bbEdit()) {
   _strict = false;
   _make ({ "new":_types.word,
            at:window[1].word[2].atEnd, with_data:"not"});}
```

Here's another syntax for exactly the same example. We construct the parameter object first, then pass it. Here, array-style syntax avoids direct use of the reserved word `new`:

```
// tell app "BBEdit 6.0" to make
//    new word at end of word 2 of window 1 with data "not"
var bb = MacOS.bbEdit();
bb._strict = false;
var params = new Object();
params["new"] = bb._types.word;
params.at = bb.window[1].word[2].atEnd;
params.with_data = "not";
bb._make (params);
```

It may sometimes happen, in using named parameters, that you need to specify the default parameter, which has no name. Since you can't combine unnamed and named parameter syntax, you need a name for the default parameter; therefore, JavaScript OSA supplies one: it is `ae_----`.

Constant Enumerations

It will often happen that an object's legal values must be one of a set of constant enumerations. These are the values which appear in Script Debugger's Explorer window as popup menus listing the various possibilities, and in the dictionary as a series of names to which the value is limited. For example, the orientation of the Application Switcher's palette can be `horizontal` or `vertical` — that's a constant enumeration.

When you ask for such a value in JavaScript OSA, what comes back is the value's four-letter constant designator, not an English-like word. For example:

```
with (MacOS.appBySignature("apsw"))
   palette.orientation; // --> vert
```

To convert to an English-like word, ask for explicit conversion to a string:

```
with (MacOS.appBySignature("apsw"))
   palette.orientation.toString(); // --> vertical
```

When you wish to set such a value, you can use the four-letter designator:

```
with (MacOS.appBySignature("apsw"))
   palette.orientation = "vert";
```

But alternatively, you can use the English-like term. You access this through the application's `_constants` object. So:

```
with (MacOS.appBySignature("apsw"))
   palette.orientation = _constants.vertical;
```

Scripting Additions

Scripting additions are injected, as it were, directly into the language, and therefore whatever they make available appears at the top level of every application's dictionary. It then becomes merely a matter of picking an application — any application. This will usually be the frontmost application, or the one to which commands are currently being directed; in the absence of any other clear choice, an obvious candidate is `appSelf()`, the current application.

For example, recall our earlier example of a word-histogram. It wasn't a very powerful example, because we hard-coded the text to be analyzed. It would be much more useful if the user could choose a textfile. Here's a way to do that:

```
with (MacOS.appSelf()) {
   var f = choose_file (null, ["TEXT", "ttro"])
   var fRef = open_for_access (f);
   var s = read(fRef);
   close_access (fRef);
}
s = s.toLowerCase(); // and the rest is as before...
```

Exceptions

A problem with the example in the preceding section is that if the user cancels out of the standard file Open dialog in the first line, an exception is thrown, and a rather ugly "uncaught exception" error message appears on the screen. The way to prevent this sort of thing is to take advantage of JavaScript's excellent facilities for handling exceptions.

Let's think how to construct our exception-handling architecture in this instance. If the user cancels out of the Open dialog, we'd like to abort the script. But JavaScript doesn't provide any facility for aborting at the top level; we cannot, for example, simply say `return`. One obvious solution is to embed the entire script in a `try`:

```
try {
   with (MacOS.appSelf()) {
      var f = choose_file (null, ["TEXT", "ttro"])
      var fRef = open_for_access (f);
      var s = read(fRef);
      close_access (fRef);
   }
   s = s.toLowerCase();
   // and so on...
}
catch (whatError) {};
```

This intercepts any runtime error, no matter the cause, and jettisons it. This, however, may be a bit too broad; all we really wanted to do was intercept the runtime error caused by the user's canceling the dialog. Other errors deserve to be shown. We can implement this by suppressing only `userCanceled` errors and passing all others to top level:

```
catch (whatError) {
   if (whatError.message.indexOf("userCanceled") < 0)
      throw whatError};
```

The `message` property here is one of several supplied for any exception generated from an Apple event (such exceptions are actually objects of type `MacOS.MacOSError`). Besides the `message`, which is the complete text of the error, there is also the `error`, which is the error number, the `offendingObject`, the `partialResult`, and the `expectedType`. In most cases this will be much more than you need; as here, the message tells you enough to go on with.

A completely different and more elegant architecture is to write the entire routine as a function, and call it. Now we *can* simply return if the user cancels, and our exception handling is neatly localized.

```
function doYourThing () {
   with (MacOS.appSelf()) {
      try {var f = choose_file (null, ["TEXT", "ttro"])}
```

```
      catch (whatError) {return;};
// ... and so on...
   return s;
}
doYourThing();
```

Waiting For Replies (Or Not)

If something goes wrong when you send an Apple event, your script can get stuck forever waiting for the answer. To avoid this, Apple events are accompanied by a timeout value; if no reply arrives in the requisite time, an error is thrown, and your script, even though it didn't get the response it was looking for, is at least allowed to get on with its life.

In JavaScript OSA, this timeout value is accessed as the application's `_timeout` property. It is measured in ticks, that is to say, in sixtieths of a second. The default, `-1`, actually refers to AppleScript's default, which is two seconds.

A completely different approach is to proceed without waiting for a reply at all. Obviously you would do this only in the case of a command to which no response is needed. Asynchronous Apple events of this sort used to be called Finder events, because the Finder was the chief application that needed to be treated this way. To send an asynchronous Apple event, you must send a raw Apple event, as described in the next section, except that wherever the `sendAE()` function is referred to, you'd use `sendAENoReply()` instead.

CHAPTER 6

Raw Apple Events

In the event that JavaScript's syntax for AppleScript-like commands doesn't permit you to send the Apple event you need, or if you wish to drive an application that can receive Apple events but has no dictionary to make it scriptable with AppleScript, or just for the sheer downright speed of the thing, you can construct and send a raw Apple event with JavaScript OSA.

Getting Started

The first step in sending a raw Apple event with JavaScript OSA is to know the structure of the Apple event you wish to send. If you're speaking to an application which isn't AppleScriptable, there is presumably some sort of documentation which tells you the structure of the Apple events the application is prepared to receive. If you're trying to copy something you're able to do through AppleScript, you can learn the structure of the Apple event by sending the AppleScript command and intercepting it in order to get a look at it; Script Debugger's Apple Event Log window permits you to do this.

A very full explanation of how to understand the structure of an Apple event is available online as part of *Frontier: The Definitive Guide*: http://www.ojai.net/matt/frontierdef/ch32.html#pgfId-184. Alternatively, see Chapter 31 of *REALbasic: The Definitive Guide*. The details won't be repeated here; we'll just start with an Apple event and discuss how to construct it in JavaScript OSA.

From an AEBuild String

We'll use as our first example this AppleScript command:

```
tell application "Finder" to get the name of every file of control panels folder
```

This shows in Script Debugger's Apple Event Log window, in what's known as AEBuild format, as:

```
core\getd{'----':obj {form:prop, want:type(prop), seld:type(pnam), from:obj {form:indx, want:type(file), seld:abso(«616C6C20»), from:obj {form:prop, want:type(prop), seld:type(ctrl), from:'null'()}}}}
```

The way to send a raw Apple event is with the `sendAE()` method of the application object. It takes three parameters: the class and event IDs of the command being sent, and the Apple event parameters as a single JavaScript object. Thus, the structure of our command will be:

```
MacOS.finder().sendAE ("core", "getd", {...});
```

Our problem is now to decide what goes into the curly braces. There is only one parameter, the default parameter, `'----'`; recall that this is expressed in JavaScript OSA as `ae____`. So our command must look like this:

```
MacOS.finder().sendAE ("core", "getd", {"ae____": ...});
```

We now come to the problem of how to express an Apple event object descriptor. This is clearly not a JavaScript native datatype; and for expressing Apple event datatypes not supported by JavaScript, JavaScript OSA supplies the `MacOS.AEDesc` object type. The simplest way to proceed, since we already have the AEBuild string version of the object, is to let the `MacOS.AEDesc`'s `build()` command construct it for us; this command converts an AEBuild string to a single piece of actual Apple event object data, which is just what we want done. The way to use this command is always in two steps: construct the `AEDesc`, then set it using a string, like this:

```
var obj = new MacOS.AEDesc();
obj.build ("...");
```

In our example, there are three object descriptors, and each must be formed separately. The objects are embedded in one another — each is the `"from"` of the one that follows — and this can be expressed by simple string concatenation (not forgetting to close the expression with a final curly brace). Apart from this, the strings themselves can simply be copied from the original Apple Event Log window string and pasted directly into our script. Here is the final result; study it carefully to see how its pieces relate to the original AEBuild string from which we started:

```
var obj1 = new MacOS.AEDesc();
obj1.build ("obj {   form:prop, want:type(prop),
                     seld:type(ctrl), from:'null'()}");

var obj2 = new MacOS.AEDesc();
obj2.build ("obj {   form:indx, want:type(file),
                     seld:abso(«616C6C20»), from:" + obj1 +"}");

var obj3 = new MacOS.AEDesc();
obj3.build ("obj {   form:prop, want:type(prop),
                     seld:type(pnam), from:" + obj2 + "}");

MacOS.finder().sendAE ("core", "getd", {"ae_----":obj3});
```

As a second example, we will tell the Finder to copy one folder into another. The original AppleScript command is:

```
tell application "Finder" to copy item "main:Mac OS Read Me Files" to
folder "main:Installer Logs"
```

Capturing this in Script Debugger's Apple Event Log window, we get this AEBuild string:

```
core\clon{'----':obj {form:name, want:type(cobj), seld:“main:Mac OS Read
Me Files”, from:'null'()}, insh:obj {form:name, want:type(cfol),
seld:“main:Installer Logs”, from:'null'()}}
```

Things here are a little different. There are two object descriptors, but they aren't embedded in each other; instead, they constitute the two different parameters of the command. The thing to be aware of here is that in constructing our parameter list, it is necessary to prefix each parameter name with `"ae_"`. This is what tells JavaScript OSA that it is forming a keyed property (as opposed to a mere user property). Thus, our command will have the form:

```
MacOS.finder().sendAE ("core", "clon", {"ae_----":..., ae_insh:...});
```

In fact, at this point the entire shape of the script becomes clear:

```
var obj1 = new MacOS.AEDesc();
obj1.build ("...");

var obj2 = new MacOS.AEDesc();
obj2.build ("...");

MacOS.finder().sendAE ("core", "clon", {"ae_----":obj1, ae_insh:obj2});
```

But now it is simply a matter of cutting and pasting, and we are done:

```
var obj1 = new MacOS.AEDesc();
obj1.build ("obj {   form:name, want:type(cobj),
                     seld:“main:Mac OS Read Me Files”, from:'null'()}");

var obj2 = new MacOS.AEDesc();
obj2.build ("obj {   form:name, want:type(cfol),
                     seld:“main:Installer Logs”, from:'null'()}");

MacOS.finder().sendAE ("core", "clon", {"ae_----":obj1, ae_insh:obj2});
```

From Individual Properties

We may think of the AEBuild string technique discussed in the previous section as constructing a raw Apple event from the top down: we have the entire event expressed as a string already, and we just translate that into objects and assemble them. Now we're going to build a raw Apple event from the bottom up, creating each property of each object ourselves.

The command we're going to build corresponds to this simple AppleScript command:

```
tell application "Finder" to get disk 1
```

This corresponds to the following AEBuild string:

```
core\getd{'----':obj {form:indx, want:type(cdis), seld:1, from:'null'()}}
```

We already know that this means our command will have the following structure:

```
MacOS.finder().sendAE ("core", "getd", {"ae_----":...});
```

Our problem is to construct the direct object parameter. This is an `"obj "` record (technically known as an AERecord) made up of four properties. The third is an integer, and the fourth is null, but the first two are rather mysterious: `indx` is evidently not a string, or it would be surrounded by quotes, and `cdis` is specifically marked as being a type called `type`. How are we going to build these properties, which involve datatypes seemingly out of our reach?

The answer is that the `MacOS.AEDesc` supplies an `as()` method which permits us to coerce data to any type we care to name; we name it using its four-letter designator. So, we will use the `MacOS.AEDesc` constructor to get a string version of our data, and coerce it. For `"cdis"`, this is easy, because the AEBuild string tells us to coerce it to a `"type"`. We'll store the result temporarily in a variable:

```
var disk = new MacOS.AEDesc("cdis").as("type");
```

To deal with `indx`, you simply have to know that a bit of four-letter unmarked data like this is an `"enum"`:

```
var index = new MacOS.AEDesc("indx").as("enum");
```

We are now ready to form the object. As before, it is keyed, so we must prefix `"ae_"` to the names of the properties. Having constructed the object, we must coerce it to an AERecord — that is, we must make sure that it has type `"obj "`.

```
var obj = new MacOS.AEDesc({  ae_form:index, ae_want:disk,
                              ae_seld:1, ae_from:null}).as("obj ");
```

That's all. The variable `obj` is now ready to be used as the value of the direct object parameter. To recap, here is the entire script:

```
var disk = new MacOS.AEDesc("cdis").as("type");
var index = new MacOS.AEDesc("indx").as("enum");
var obj = new MacOS.AEDesc({  ae_form:index, ae_want:disk,
                              ae_seld:1, ae_from:null}).as("obj ");
MacOS.finder().sendAE ("core", "getd", {"ae_----":obj});
```

This was a very simple example, but a more elaborate one would be pointless. No matter how large or complex the desired Apple event, the principles for constructing it from the ground up in this way are simply more of the same.

CHAPTER 7

Executing Other Scripts

JavaScript OSA has the ability to execute code in any OSA language. If the code is a compiled script file, you can load and execute it; if the code is a string, you can compile and execute it. You can even call a particular subroutine within a script, passing it parameters if need be.

The OSA Object

The magic of executing OSA code with JavaScript OSA is performed through a `MacOS.OSA` object. If you're going to be handing this object a string to compile, it needs to be keyed in advance to some particular OSA language; you do this by handing its constructor a language specifier, which will typically be `MacOS.OSA.JavaScript`, `MacOS.OSA.AppleScript` or `MacOS.OSA.Frontier`. If you're going to be loading a script that is already compiled, you can use `MacOS.OSA.Generic`, because the language information is already built into the compiled script.

So, as a rather silly but nonetheless revealing example, we'll call the `display dialog` command through AppleScript:

```
var ascr = new MacOS.OSA(MacOS.OSA.AppleScript);
var s = 'display dialog "It is working!"'
ascr.compile(s); ascr.execute();
```

The really cool part is that the result (the button pressed by the user) comes back to us in good order, just if we had called the `display dialog` command directly.

Next, we'll target a compiled script. We compile and save the following AppleScript code:

```
on hello(who)
   display dialog "Hello " & who
end hello

on goodbye(who)
   display dialog "Goodbye " & who
end goodbye

hello("amigo")
goodbye("amigo")
```

I save this at `main:testing`. Now I run this JavaScript code to load and execute my AppleScript:

```
var ascr = new MacOS.OSA(MacOS.OSA.Generic);
ascr.load ("main:testing");
ascr.execute();
```

But the really cool part here is that instead of running the entire script, I can call one piece of it myself.

```
var ascr = new MacOS.OSA(MacOS.OSA.Generic);
ascr.load ("main:testing");
ascr.hello("friend");
```

That last line means that we call just the handler `hello()`, handing it the parameter `"friend"`. This works because `MacOS.OSA.load()` creates a kind of pseudo-dictionary, parallel to the behavior a `MacOS.AEApp` with a loaded dictionary: handlers in the OSA script are promoted to methods, and globals and properties of the OSA script are promoted to properties, of the `MacOS.OSA` instance.

External Editors

Since we can compile and execute any string, any text editing environment can serve as a base for running scripts, provided we can run the one script that will compile and execute the text. A good example of such an environment is BBEdit. BBEdit is a nice

editing environment for JavaScript; it provides syntax coloring, powerful search-and-replace, and so forth. It also has a Script menu from which any compiled script in its Scripts folder can be run.

So, for example, you might use Script Debugger to create and compile this script, saving it in BBEdit's Scripts folder under the name Run JavaScript:

```
var bb = MacOS.bbEdit();
bb._strict = false;
var s = bb.window[1].text.contents;
var js = new MacOS.OSA(MacOS.OSA.JavaScript);
js.compile(s)
var result = js.execute();
try {
   result = "\r// " + result.toString();
   with (bb) make (_types.word, window[1].atEnd, {contents:result});
} catch (what) {};
```

Now you can write a JavaScript program in BBEdit, and execute it immediately, without leaving BBEdit, by choosing Run JavaScript from the Script menu. If the script produces a result, it is appended to the program as a comment. (You may need to increase BBEdit's memory allocation before trying this.)

Libraries

The ability to load and execute scripts from disk means that you can break your script down into libraries which are stored on disk. JavaScript OSA also provides a shortcut for doing this, `Core.load()`, which loads, compiles, and executes a JavaScript textfile in a single step. If this textfile is prepared properly, its functions can be called directly. The trick is that the result of the textfile must be an object whose methods are the textfile's functions — effectively, we must perform ourselves the same magic of promoting the functions to become the methods of an object that JavaScript OSA does when it loads a dictionary.

For example, let's suppose we frequently have reason to reverse strings, and we wish to break this functionality out into a library. Our `reverse()` function might look like this:

```
function reverse(s) {
   var result = "";
   for (var i = s.length-1; i >= 0; i--)
      result = result + s.charAt(i);
   return result;
}
```

If we wish to make this function available through a library, we must add code to this script which captures the `reverse()` function as a property of an object, and returns that object:

```
var obj = new Object; obj.reverse = reverse;
obj;
```

The library is now ready, and we save it as text. Let's say we save it at `main:myLib.js`. Now, in a different script, we can load the library and compile and execute it, capturing the object it returns, and then call the `reverse()` function through that object:

```
var lib = Core.load ("main:myLib.js");
lib.reverse ("howdy");
```

Since the library is just a textfile, it can be created and saved with any text editor. For example, it could be created and saved with BBEdit, and then BBEdit could be used to create and run the script which calls it, as shown in the previous section.

Technically speaking, `Core.load()` is a shortcut in more ways than one. Because it loads, compiles, and executes its script within the current OSA context, it avoids the overhead of creating and maintaining an independent `MacOS.OSA` instance.

CHAPTER 8

Receiving Apple Events

It is possible for a script written in the JavaScript OSA language to receive Apple events. The situation is roughly parallel to that of sending Apple events; just as we can send Apple events using AppleScript terminology or as raw Apple events, so too we can receive them using AppleScript terminology or through a raw Apple event terminology.

Raw Apple Event Terminology

Let's take raw Apple event terminology first. A typical example of where one would write a script designed to receive Apple events is a script application (an "applet"), so let's write one as our first example. Suppose we wish to count, and list the names of, the files dropped onto our application. As you know from AppleScript, this functionality lives in the script application's Open handler, which will receive a single parameter that is a list of aliases. In raw Apple event terms, the script application is sent an `aevt/odoc` Apple event, and the list of aliases is its direct-object parameter. This, then, is what we'd like to receive.

To receive an Apple event using raw Apple event terminology, we use a function whose name is `ae_` followed by the eight characters that identify the Apple event. If there are to be any parameters, they will arrive as a single parameter which is a JavaScript object; the properties of this object are the actual Apple event parameters, and they can be accessed under the name `ae_` followed by the four letters that identify the parameter.

So, in this case, our function will be called `ae_aevtodoc()`, and when the parameter arrives, we will obtain its `ae_----` property (the direct-object parameter of the Apple event), and expect to find there an array of `MacOS.FileSpec` objects. Here is the script application:

```
function ae_aevtodoc (theParameters) {
   var theFiles = theParameters["ae_----"];
   var count = 0, total = "";
   for (var i = 0; i < theFiles.length; i++, count++)
      total += MacOS.finder().item[theFiles[i].toString()].name+"\r";
   MacOS.message(count + " files:\r" + total);
}
```

There is actually another way to name our function. Instead of calling it `ae_aevtodoc`, we are permitted to assign it any name we like, provided we "register" the function as an alias for `ae_aevtodoc` using a special JavaScriptOSA constructor called `OSAInit()`. The syntax for this is easier to grasp by example than by description:

```
function countAndName (theParameters) {
   var theFiles = theParameters.of;
   var count = 0, total = "";
   for (var i = 0; i < theFiles.length; i++, count++)
      total += MacOS.finder().item[theFiles[i].toString()].name+"\r";
   MacOS.message(count + " files:\r" + total);
}

function OSAInit() {
   this["ae_aevtodoc"] = countAndName;
}
```

(Notice that you may need to run this script once inside Script Debugger before saving it as a script application; otherwise, since there is no "real" Open handler here, it won't flip the bit that notifies the Finder to allow drops onto the applet icon.) The purpose of this device isn't merely to allow you to give your function a more convenient name; it's to allow you to give your function an *illegal* name! The eight letters that identify an Apple event could include characters that JavaScript would consider illegal in a function name; using `OSAInit()`, you give your function a legal name and put the eight-letter identifier inside a string, where any characters are permitted.

AppleScript Terminology

Now let's proceed to receiving an Apple event using AppleScript terminology. For this to work, your script must obviously be inside an application context where there is an AppleScript dictionary. In this situation, you don't have to declare the function using its Apple event name; you can use its AppleScript name instead. The parameters of the Apple event still arrive as the properties of a single function parameter, but the names of these properties are their AppleScript names (with underlines instead of spaces, just as when you send an Apple event).

You are now probably asking: What *is* an application context where there is an AppleScript dictionary? Well, one such context is when you write an applet and embed an `aete` resource into it. For example, suppose we want to write an (admittedly trivial) applet that reports the name of the author. Let's say we'd like it to accept AppleScript commands of the following (admittedly unusual) forms:

```
reveal author
reveal author particularly firstname
reveal author particularly lastname
```

If the author's name is John Doe, our applet script can look like this:

```
function reveal_author(params) {
   if (params.particularly == undefined)
      return "John Doe";
   switch (String(params.particularly)) {
      case "lastname" :
         return "Doe";
      case "firstname" :
         return "John";
      default:
         return "";
   }
}
```

The amazing thing is that this is entirely English-like — that is, the names `reveal_author` and `particularly`, and the values `lastname` and `firstname`, come right out of our AppleScript command definitions. The trick is merely to supply the glue which makes this possible, and that glue is the `aete` resource. After saving our script as a stay-open application named, let's say, Revealer, we then use our favorite `aete` resource editor to define an Apple event `reve/auth` as `reveal author`, with an optional parameter `part` as `particularly`, whose type is `whic`, which is an enumeration consisting of enumerators `firs` and `last` defined as `firstname` and `lastname`. (All the four-letter codes here are completely arbitrary, of course, since both the script which speaks to our application and the script inside it speak in terms of the English-like names; but it's nice to have our four-letter codes be easy to remember, in case someone wishes to send our applet raw Apple events instead.)

If the `aete` resource specifies that the `particularly` parameter can be of any datatype, we can permit a further syntax where the user asks for the number of the word rather than using an enumeration:

```
reveal author particularly 1 -- same as firstname
reveal author particularly 2 -- same as lastname
```

The script takes advantage of the `MacOS.AEDesc` class's `_type` method, which returns the four-letter code of the AppleScript datatype of the incoming data:

```
function reveal_author(params) {
```

```
   if (params.particularly == undefined)
      return "John Doe";
   if (params.particularly._type == "enum")
      params.particularly = String(params.particularly);
   switch (params.particularly) {
      case "lastname" :
      case 2:
         return "Doe";
      case "firstname" :
      case 1:
         return "John";
      default:
         return "";
   }
}
```

Another context in which there is an AppleScript dictionary is where the system itself has "published" the dictionary. This is the case with scripting additions, which means we can receive scripting addition commands using AppleScript terminology. For an example of this, look at the JavaScript CGI script included with JavaScript OSA. It starts like this:

```
function handle_CGI_request(ae_params) {
   try {
      var paramPath = ae_params.of;
      var searchingFor = ae_params.searching_for;
      var withPostedData = ae_params.with_posted_data;
      // ... and so on ...
```

Except for the name `of`, which designates the direct object and is defined by JavaScript OSA, all these names — `handle_CGI_request`, `searching_for`, `with_posted_data` — come from the Standard Additions scripting addition.

Finally, the system defines the AppleScript terminology for some standard Apple events, such as `quit`. Here's an extremely badly behaved script application that refuses to quit; it puts up a dialog instead. Ignore for the moment its bad behavior; the important thing is that we don't have to name the function `ae_aevtquit()`:

```
function quit (params) {
   MacOS.message ("Heck no, I won't go!");
}
```

Save that as a stay-open script application called Refuser, and run the application. Now, in AppleScript, you can say:

```
tell application "Refuser"
   activate
   quit
end tell
```

Resuming an Apple Event

To "resume" an Apple event means to handle it but to pass it along to higher context for further processing. The Refuser script application from the previous section makes a good example. It is reasonable that we might wish to intercept the `quit` command and perform some processing of our own, but it is also incumbent upon us to quit in

response to it. But this is not something we can do ourselves; we must pass the Apple event on up the contextual hierarchy so that the system can perform its normal behavior.

This is done with the `MacOS.aeResume()` function; its first two parameters are the two four-letter identifiers of the Apple event. So:

```
function quit (params) {
   MacOS.message ("Okay, I'm leaving, but I don't like it!");
   MacOS.aeResume ("aevt", "quit");
}
```

The `MacOS.aeResume()` function can take a third parameter, which is to permit you to pass parameters along with your Apple event. You can supply a simple value, which then becomes the direct object parameter, or you can supply a `MacOS.AEDesc` object such as we constructed in sending a raw Apple event earlier, or you can supply a JavaScript object with `ae_` property names. In the latter two cases, the name–value pairs supplied by the `AEDesc` object or JavaScript object become the parameters of the Apple event.