The Array Object

CHAPTER 29

In This Chapter

Working with ordered collections of data

Simulating multidimensional arrays

Manipulating information stored in an array

An array is the sole JavaScript data structure provided for storing and manipulating ordered collections of data. But unlike some other programming languages, JavaScript's arrays are very forgiving as to the kind of data you store in each cell or entry of the array. This allows, for example, an array of arrays, providing the equivalent of multidimensional arrays customized to the kind of data your application needs.

If you have not done a lot of programming in the past, the notion of arrays may seem like an advanced topic. But if you ignore their capabilities, you set yourself up for a harder job when implementing many kinds of tasks. Whenever I approach a script, one of my first thoughts is about the data being controlled by the application and whether handling it as an array will offer some shortcuts for creating the document and handling interactivity with the user.

I hope that by the end of this chapter, you will not only be familiar with the properties and methods of JavaScript arrays, but you will begin to look for ways to make arrays work for you.

Structured Data

In programming, an array is defined as *an ordered collection of data*. You can best visualize an array as a table, not much different from a spreadsheet. In JavaScript, arrays are limited to a table holding one column of data, with as many rows as needed to hold your data. As you have seen in many chapters in Part III, a JavaScript-enabled browser creates a number of internal arrays for the objects in your HTML documents and browser properties. For example, if your document contains five links, the browser maintains a table of those links. You access them by number (with 0 being the first link) in the array syntax: the array name followed by the index number in square brackets, as in `document.links[0]`, which represents the first link in the document.

For many JavaScript applications, you will want to use an array as an organized warehouse for data that users of your page access, depending on their interaction with form elements. In one of the bonus applications (Chapter 48) on the CD-ROM, I show you an extended version of this usage in a page that lets users search a small table of data for a match

between the first three digits of their U.S. Social Security numbers and the state in which they registered. Arrays are the way JavaScript-enhanced pages can re-create the behavior of more sophisticated CGI programs on servers. When the collection of data you embed in the script is no larger than a typical .gif image file, the user won't experience significant delays in loading your page; yet he or she will have the full power of your small database collection for instant searching without any calls back to the server. Such database-oriented arrays are important applications of JavaScript for what I call *serverless CGIs*.

As you design an application, look for clues as to potential application of arrays. If you have a number of objects or data points that interact with scripts the same way, you have a good candidate for array structures. For example, with the exception of Internet Explorer 3, you can assign like names to every text field in a column of an order form. In that sequence like-named objects are treated as elements of an array. To perform repetitive row calculations down an order form, your scripts can use array syntax to perform all the extensions within a handful of JavaScript statements, rather than perhaps dozens of statements hard-coded to each field name. Chapter 49 on the CD-ROM shows an example of this application.

You can also create arrays that behave like the Java hash table: a lookup table that gets you to the desired data point instantaneously if you know the name associated with the entry. If you can conceive your data in a table format, an array is in your future.

Creating an Empty Array

Arrays are treated in JavaScript like objects, but the extent to which your scripts can treat them as objects depends on whether you're using the first version of JavaScript (in Navigator 2 and Internet Explorer 3 with the Version 1 JScript DLL) or more recent versions (in Navigator 3 or later and Internet Explorer with JScript DLL Version 2 or later). For the sake of compatibility, I'll begin by showing you how to create arrays that work in all scriptable browsers.

You begin by defining an object *constructor* that assigns a passed parameter integer value to the `length` property of the object:

```
function makeArray(n) {
        this.length = n
        return this
}
```

Then, to actually initialize an array for your script, use the `new` keyword to construct the object for you while assigning the array object to a variable of your choice:

```
var myArray = new makeArray(n)
```

where ***n*** is the number of entries you anticipate for the array. This initialization does not make any array entries or create any placeholders. Such preconditioning of arrays is not necessary in JavaScript.

In one important aspect, an array created in this "old" manner does not exhibit an important characteristic of standard arrays. The `length` property here is artificial in that it does not change with the size of the array (JavaScript arrays are

completely dynamic, letting you add items at any time). The length value here is hardwired by assignment. You can always change the value manually, but it takes a great deal of scripted bookkeeping to manage that task.

Another point to remember about this property scheme is that the value assigned to `this.length` in the constructor actually occupies the first entry (index 0) of the array. Any data you want to add to an array should not overwrite that position in the array if you expect to use the length to help a repeat loop look through an array's contents.

What the full-fledged newer array object gains you is behavior more like that of the arrays you work with elsewhere in JavaScript. You don't need to define a constructor function, because it's built into the JavaScript object mechanism. Instead, you create a new array object like this:

```
var myArray = new Array()
```

An array object automatically has a `length` property (0 for an empty array). Most importantly, this length value does not occupy one of the array entries; the array is entirely for data.

Should you want to presize the array (for example, preload entries with null values), you can specify an initial size as a parameter to the constructor. Here I create a new array to hold information about a 500-item compact disc collection:

```
var myCDCollection = new Array(500)
```

Presizing an array does not give you any particular advantage, because you can assign a value to any slot in an array at any time: The `length` property adjusts itself accordingly. For instance, if you assign a value to `myCDCollection[700]`, the array object adjusts its length upward to meet that slot (with the count starting at 0):

```
myCDCollection [700] = "Gloria Estefan/Destiny"
collectionSize = myCDCollection.length  // result = 701
```

A true array object also features a number of methods and the capability to add prototype properties, described later in this chapter.

Populating an Array

Entering data into an array is as simple as creating a series of assignment statements, one for each element of the array. Listing 29-1 (not on the CD-ROM) assumes that you're using the newer style array object and that your goal is to generate an array containing a list of the nine planets of the solar system.

Listing 29-1: Generating and Populating a New Array

```
solarSys = new Array(9)
solarSys[0] = "Mercury"
solarSys[1] = "Venus"
solarSys[2] = "Earth"
solarSys[3] = "Mars"
solarSys[4] = "Jupiter"
solarSys[5] = "Saturn"
```

```
solarSys[6] = "Uranus"
solarSys[7] = "Neptune"
solarSys[8] = "Pluto"
```

This way of populating a single array is a bit tedious when you're writing the code, but after the array is set, it makes accessing information collections as easy as any array reference:

```
onePlanet = solarSys[4] // result = "Jupiter"
```

A more compact way to create an array is available when you know that the data will be in the desired order (as the `solarSys[]` preceding array). Instead of writing a series of assignment statements (as in Listing 29-1), you can create what is called a ***dense array*** by supplying the data as parameters to the `Array()` constructor:

```
solarSys = new
Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
"Uranus","Neptune","Pluto")
```

The term "dense array" means that data is packed into the array without gaps starting at index position 0.

The example in Listing 29-1 shows what you might call a vertical collection of data. Each data point contains the same type of data as the other data points — the name of a planet — and the data points appear in the relative order of the planets from the Sun.

But not all data collections are vertical. You may, for instance, just want to create an array that holds various pieces of information about one planet. Earth is handy, so let's use some of its astronomical data to build a completely separate array of earthly info in Listing 29-2 (not on the CD-ROM).

Listing 29-2: **Creating a "Horizontal" Array**

```
earth = new Array()
earth.diameter = "7920 miles"
earth.distance = "93 million miles"
earth.year = "365.25 days"
earth.day = "24 hours"
earth.length     // result = 4
```

What you see in Listing 29-2 is an alternative way to populate an array. In a sense, you saw a preview of this method when I created an array in the old style earlier and assigned the `length` property name to its first entry. If you assign a value to a property name that has not yet been assigned for the array, JavaScript is smart enough to append a new property entry for that value.

In an important change from the old style of array construction, the way you define an array entry impacts how you access that information later. For example, when you populate an array based on index values (Listing 29-1), you can retrieve those array entries only via references that include the index values. Conversely, if

you define array entries by property name (Listing 29-2), you cannot access those values via the index way. In Navigator 2, for instance, the array assignments of Listing 29-2 can be retrieved by their corresponding index values:

```
earth.diameter          // result = "7920 miles"
earth["diameter"]       // result = "7920 miles"
earth[0]                // result = "7920 miles"
```

In Navigator 3 or 4, however, because these entries are defined as properties, they must be retrieved as properties, not index values:

```
earth.diameter          // result = "7920 miles"
earth["diameter"]       // result = "7920 miles"
earth[0]                // result = null
```

The impact here on your scripts is that you need to anticipate how you expect to retrieve data from your array. If an indexed repeat loop is in the forecast, populate the array with index values (as in Listing 29-1); if the property names are more important to you, then populate the array that way (as in Listing 29-2). Your choice of index value type for a single-column array is driven by the application, but you will want to focus on the named array entry style for creating what appear to be two-dimensional arrays.

JavaScript 1.2 Array Creation Enhancements

Navigator 4 added one new way to create a dense array and also cleared up a bug in the old way. These features are part of the JavaScript 1.2 specification, and so are also available in Internet Explorer 4.

A new, simpler way to create a dense array does not require the array object constructor. Instead, JavaScript 1.2 accepts what is called *literal notation* to generate an array. To demonstrate the difference, the following statement is the regular dense array constructor that works with Navigator 3:

```
solarSys = new
Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn",
"Uranus","Neptune","Pluto")
```

While JavaScript 1.2 fully accepts the preceding syntax, it also accepts the new literal notation:

```
solarSys = ["Mercury","Venus","Earth","Mars","Jupiter","Saturn",
"Uranus","Neptune","Pluto"]
```

The square brackets stand in for the call to the array constructor. You have to judge which browser types your audience will be using before deploying the JavaScript 1.2 version.

The bug fix has to do with how to treat the earlier dense array constructor if the scripter enters only the numeric value 1 as the parameter — `new Array(1)`. In Navigator 3 and Internet Explorer 4, JavaScript erroneously creates an array of length 1, but that element is undefined. For Navigator 4 (and inside a `<SCRIPT LANGUAGE="JavaScript1.2">` tag), the same statement creates that one-element array and places the value in that element.

Deleting Arrays and Array Entries

You can always set the value of an array entry to null or an empty string to wipe out any data that used to occupy that space. But until the `delete` operator in Navigator 4, you could not completely remove the element or the array.

Deleting an array element eliminates the index from the list of accessible index values, but does not reduce the array's length, as in the following sequence of statements:

```
myArray.length// result: 5
delete myArray[2]
myArray.length// result: 5
myArray[2] // result: undefined
```

See the `delete` operator in Chapter 32 for further details.

Simulating Two-Dimensional Arrays

As you may have deduced from my examples in Listings 29-1 and 29-2, what I'm really aiming for in this application is a two-dimensional array. If the data were in a spreadsheet, there would be columns for Name, Diameter, Distance, Year, and Day; also, each row would contain the data for each planet, filling a total of 45 cells or data points (9 planets times 5 data points each). Although JavaScript does not have a mechanism for explicit two-dimensional arrays, you can create an array of objects, which accomplishes the same thing.

The mechanism for the array of objects consists of a primary array object creation (whether created by the old or new way), a separate constructor function that builds objects, and the main, data-stuffing assignment statements you saw for the vertical array style. Listing 29-3 (not on the CD-ROM) shows the constructor and stuffer parts of the solar system application.

Listing 29-3: **Building a Two-Dimensional Array**

```
function planet(name,diameter, distance, year, day){
        this.name = name
        this.diameter = diameter
        this.distance = distance
        this.year = year
        this.day = day
}
solarSys = new Array(9)  // Navigator 3.0 array object constructor
solarSys[0] = new planet("Mercury","3100 miles", "36 million miles",
"88 days", "59 days")
solarSys[1] = new planet("Venus", "7700 miles", "67 million miles",
"225 days", "244 days")
solarSys[2] = new planet("Earth", "7920 miles", "93 million miles",
"365.25 days","24 hours")
solarSys[3] = new planet("Mars", "4200 miles", "141 million miles",
"687 days", "24 hours, 24 minutes")
```

```
solarSys[4] = new planet("Jupiter","88,640 miles","483 million miles",
"11.9 years", "9 hours, 50 minutes")
solarSys[5] = new planet("Saturn", "74,500 miles","886 million miles",
"29.5 years", "10 hours, 39 minutes")
solarSys[6] = new planet("Uranus", "32,000 miles","1.782 billion
miles","84 years", "23 hours")
solarSys[7] = new planet("Neptune","31,000 miles","2.793 billion
miles","165 years", "15 hours, 48 minutes")
solarSys[8] = new planet("Pluto", "1500 miles", "3.67 billion miles",
   "248 years", "6 days, 7 hours")
```

After creating the main, nine-data-element array, `solarSys`, the script uses that `new` keyword again to populate each entry of the `solarSys` array with an object fashioned in the `planet()` constructor function. Each call to that function passes five data points, which, in turn, are assigned to named property entries in the planet object. Thus, each entry of the `solarSys` array contains a five-element object of its own.

The fact that all of these subobjects have the same data structure now makes it easy for your scripts to extract the data from anywhere within this 45-entry, two-dimensional array. For example, to retrieve the name value of the fifth entry of the `solarSys` array, the syntax is this:

```
planetName = solarSys[4].name
```

This statement has the same appearance and behavior as properties of JavaScript's built-in arrays. It is, indeed, the very same model. To understand why you want to create this table, study Listing 29-4. Extracting data from the two-dimensional array is quite simple in the `showData()` function. The array structure even makes it possible to create a pop-up button listing from the same array data.

Listing 29-4: Two-Dimensional Array Results

```
<HTML>
<HEAD>
<TITLE>Our Solar System</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
<!-- start script
// stuff "rows" of data for our pseudo-two-dimensional array
function planet(name,diameter, distance, year, day){
        this.name = name
        this.diameter = diameter
        this.distance = distance
        this.year = year
        this.day = day
}
// create our pseudo-two-dimensional array
solarSys = new Array(9)
solarSys[0] = new planet("Mercury","3100 miles", "36 million miles",
"88 days", "59 days")
```

(continued)

Listing 29-4 *(continued)*

```
solarSys[1] = new planet("Venus", "7700 miles", "67 million miles",
"225 days", "244 days")
solarSys[2] = new planet("Earth", "7920 miles", "93 million miles",
"365.25 days","24 hours")
solarSys[3] = new planet("Mars", "4200 miles", "141 million miles",
"687 days", "24 hours, 24 minutes")
solarSys[4] = new planet("Jupiter","88,640 miles","483 million miles",
"11.9 years", "9 hours, 50 minutes")
solarSys[5] = new planet("Saturn", "74,500 miles","886 million miles",
"29.5 years", "10 hours, 39 minutes")
solarSys[6] = new planet("Uranus", "32,000 miles","1.782 billion
miles","84 years", "23 hours")
solarSys[7] = new planet("Neptune","31,000 miles","2.793 billion
miles","165 years", "15 hours, 48 minutes")
solarSys[8] = new planet("Pluto", "1500 miles", "3.67 billion miles",
"248 years", "6 days, 7 hours")

// fill text area object with data from selected planet
function showData(form) {
       i = form.planets.selectedIndex
       var result = "The planet " + solarSys[i].name
       result += " has a diameter of " + solarSys[i].diameter + ".\n"
       result += "At a distance of " + solarSys[i].distance + ", "
       result += "it takes " + solarSys[i].year + " to circle the
Sun.\n"
       result += "One day lasts " + solarSys[i].day + " of Earth
time."
       form.output.value = result
}
// end script -->
</SCRIPT>
<BODY onLoad="document.forms[0].planets.onchange()">
<H1>The Daily Planet</H1>
<HR>
<FORM>
<SCRIPT LANGUAGE = "JavaScript">
<!-- start script again
var page = "" // start assembling next part of page and form
page += "Select a planet to view its planetary data: "
page += "<SELECT NAME='planets' onChange='showData(this.form)'> "
// build popup list from array planet names
for (var i = 0; i < solarSys.length; i++) {
       page += "<OPTION" // OPTION tags
       if (i == 0) { // pre-select first item in list
          page += " SELECTED"
       }
       page += ">" + solarSys[i].name
}
page += "</SELECT><P>" // close selection item tag
document.write(page) // lay out this part of the page
```

```
// really end script -->
</SCRIPT>
<TEXTAREA NAME="output" ROWS=5 COLS=75>
</TEXTAREA>
</FORM>
</BODY>
</HTML>
```

The Web page code shown in Listing 29-4 uses two blocks of JavaScript scripts. In the upper block, the scripts create the arrays described earlier and define a function that the page uses to accumulate and display data in response to user action (see Figure 29-1).

The body of the page is constructed partially out of straight HTML, with some JavaScript coding in between. I hard-code the `<H1>` heading, divider, and start of the form definition. From there, I hand-off page layout to JavaScript. It begins assembling the next chunk of the page in a string variable, `page`. The start of a select object definition follows a line of instructions. To assign values to the `<OPTION>` tags of the select object, I use a repeat loop that cycles through each entry of the `solarSys` array, extracting only the `name` property for each and plugging it into the accumulated HTML page for the select object. Notice how I applied the `SELECTED` attribute to the first option. I then close out the select object definition in the `page` variable and write the entire variable's contents out to the browser. The browser sees this rush of HTML as just more HTML to obey as it fills in the page. After the variable's HTML is loaded, the rest of the hard-wired page is loaded, including an output textarea object and the close of all opened tag pairs.

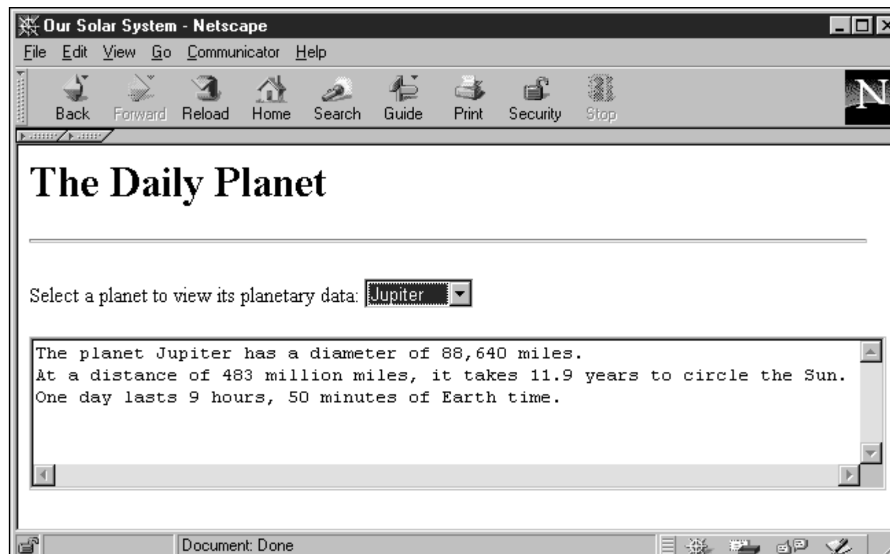


Figure 29-1: The page constructed from Listing 29-4

After the document is loaded into the browser, all activity takes place on the client machine. If the network connection were to drop, the planet data would still be intact. In fact, the user could save the source code on the client computer's hard disk and open it as a file at any time, without reconnecting to the server. Without JavaScript, a CGI program on the server would have to reply to a query from the document, fetch the data, and send it back to the PC — involving two extra network transfers. Another serverless CGI has been born.

Array Object Properties

`length`

Value: Integer **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

A true array object's `length` property reflects the number of entries in the array. An entry can be any kind of JavaScript value, including null. If there is an entry in the 10th cell and the rest are null, the length of that array is 10. Note that because array index values are zero-based, the index of the last cell of an array is one less than the length.

`prototype`

Value: Variable or Function **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

Inside JavaScript, an array object has its dictionary definition of methods and `length` property — items that all array objects have in common. The `prototype` property enables your scripts to ascribe additional properties or methods that apply to all the arrays you create in the currently loaded documents. You can override this prototype, however, for any individual objects as you want.

To demonstrate how the `prototype` property works, Listing 29-5 creates a `prototype` property for all array objects. As the script generates new arrays, the property automatically becomes a part of those arrays. In one array, `c`, you override the value of the prototype `sponsor` property. By changing the value for that one object, you don't alter the value of the prototype for the array object. Therefore, another array created afterward, `d`, still gets the original `prototype` property value.

Listing 29-5: **Adding a prototype Property**

```
<HTML>
<HEAD>
<TITLE>Array prototypes</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
// add prototype to all Array objects
Array.prototype.sponsor = "DG"
a = new Array(5)
b = new Array(5)
c = new Array(5)
// override prototype property for one 'instance'
c.sponsor = "JS"
// this one picks up the original prototype
d = new Array(5)
</SCRIPT>
<BODY><H2>
<SCRIPT LANGUAGE="JavaScript">
document.write("Array a is brought to you by: " + a.sponsor + "<P>")
document.write("Array b is brought to you by: " + b.sponsor + "<P>")
document.write("Array c is brought to you by: " + c.sponsor + "<P>")
document.write("Array d is brought to you by: " + d.sponsor + "<P>")
</SCRIPT>
</H2>
</BODY>
</HTML>
```

You can assign properties and functions to a prototype. To assign a function, define the function as you normally would in JavaScript. Then assign the function by name:

```
function newFunc(param1) {
        // statements
}
Array.prototype.newMethod = newFunc  // omit parentheses in this
reference
```

When you need to call upon that function (which has essentially become a new temporary method for the array object), invoke it as you would any object method. Therefore, if an array named `CDCollection` has been created and a prototype method `showCoverImage()` has been attached to the array, the call to invoke the method for a tenth listing in the array is

```
CDCollection[9].showCoverImage(this)
```

where `this` passes a reference to this particular array entry and all properties and methods associated with it.

Array Object Methods

After you have information stored in an array, JavaScript provides several methods to help you manage that data. These methods have evolved over time, so observe carefully which browser versions a desired method works with.

arrayObject.concat(*array2*)

Returns: Nothing.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

The `array.concat()` method allows you to join together two array objects into a new, third array object. The action of concatenating the arrays does not alter the contents or behavior of the two original arrays. To join the arrays together, you refer to the first array object to the left of the period before the method; a reference to the second array is the parameter to the method. For example

```
var array1 = new Array(1,2,3)
var array2 = new Array("a","b","c")
var array3 = array1.concat(array2)
       // result: array with values 1,2,3,"a","b","c"
```

If an array element is a string or number value (not a string or number object), the values are copied from the original arrays into the new one. All connection with the original arrays ceases for those items. But if an original array element is a reference to an object of any kind, JavaScript copies a reference from the original array's entry into the new array. This means that if you make a change to either array's entry, the change occurs to the object, and both array entries reflect the change to the object.

Example

Listing 29-6 is a bit complex, but it demonstrates both how arrays can be joined with the `array.concat()` method and how values and objects in the source arrays do or do not propagate based on their data type. The page is shown in Figure 29-2.

When you load the page, you see readouts of three arrays. The first array consists of all string values; the second array has two string values and a reference to a form object on the page (a textbox named "original" in the HTML). In the initialization routine of this page, not only are the two source arrays created, but they are joined together with the `array.concat()` method, and the result is shown in the third box. To show the contents of these arrays in columns, I use the `array.join()` method, which brings the elements of an array together as a string delimited in this case by a return character — giving us an instant column of data.

Two series of fields and buttons let you experiment with the way values and object references are linked across concatenated arrays. In the first group, if you enter a new value to be assigned to `arrayThree[0]`, the new value replaces the string value in the combined array. Because regular values do not maintain a link back to the original array, only the entry in the combined array is changed. A call to `showArrays()` proves that only the third array is affected by the change.

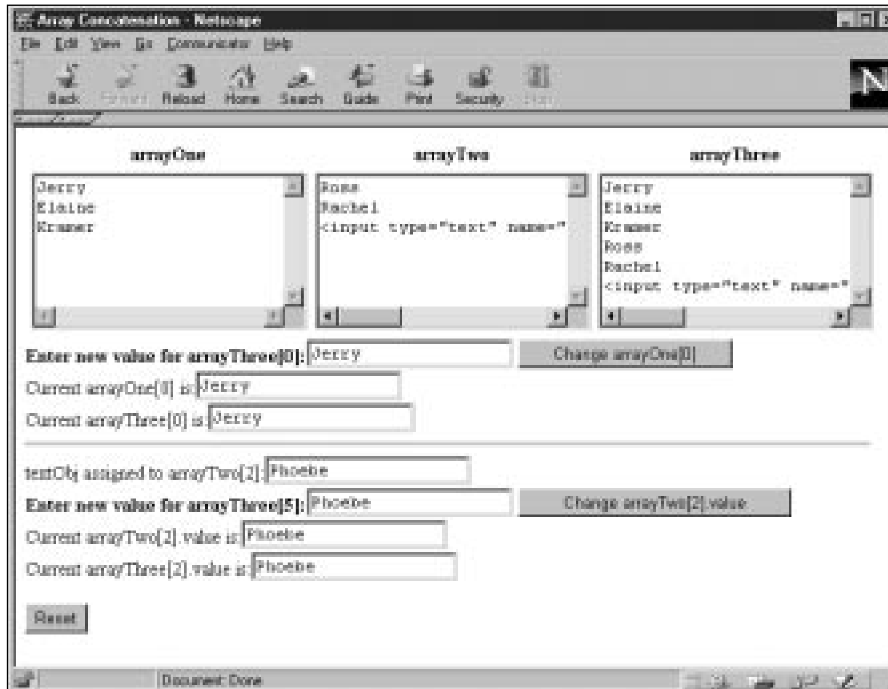


Figure 29-2: Object references remain "alive" in a concatenated array.

More complex is the object relationship for this demonstration. A reference to the first text box of the second grouping has been assigned to the third entry of `arrayTwo`. After concatenation, the same reference is now in the last entry of the combined array. If you enter a new value for a property of the object in the last slot of `arrayThree`, the change goes all the way back to the original object — the first text box in the lower grouping. Thus, the text of the original field changes in response to the change of `arrayThree[5]`. And because all references to that object yield the same result, the reference in `arrayTwo[2]` points to the same text object, yielding the same new answer. The display of the array contents doesn't change, because both arrays still contain a reference to the same object (and the `VALUE` attribute showing in the `<INPUT>` tag of the column listings refers to the default value of the tag, not its current algorithmically retrievable value shown in the last two fields of the page).

Listing 29-6: **Array Concatenation**

```
<HTML>
<HEAD>
<TITLE>Array Concatenation</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
// global variables
var arrayOne, arrayTwo, arrayThree, textObj
// initialize after load to access text object in form
function initialize() {
        var form = document.forms[0]
        textObj = form.original
        arrayOne = new Array("Jerry", "Elaine","Kramer")
        arrayTwo = new Array("Ross", "Rachel",textObj)
        arrayThree = arrayOne.concat(arrayTwo)
        update1(form)
        update2(form)
        showArrays()
}
// display current values of all three arrays
function showArrays() {
        var form = document.forms[0]
        form.array1.value = arrayOne.join("\n")
        form.array2.value = arrayTwo.join("\n")
        form.array3.value = arrayThree.join("\n")
}
// change the value of first item in Array Three
function update1(form) {
        arrayThree[0] = form.source1.value
        form.result1.value = arrayOne[0]
        form.result2.value = arrayThree[0]
        showArrays()
}
// change value of object property pointed to in Array Three
function update2(form) {
        arrayThree[5].value = form.source2.value
        form.result3.value = arrayTwo[2].value
        form.result4.value = arrayThree[5].value
        showArrays()
}
</SCRIPT>
</HEAD>
<BODY onLoad="initialize()">
<FORM>
<TABLE>
<TR><TH>arrayOne</TH><TH>arrayTwo</TH><TH>arrayThree</TH></TR>
<TR>
<TD><TEXTAREA NAME="array1" COLS=25 ROWS=6></TEXTAREA></TD>
<TD><TEXTAREA NAME="array2" COLS=25 ROWS=6></TEXTAREA></TD>
<TD><TEXTAREA NAME="array3" COLS=25 ROWS=6></TEXTAREA></TD>
</TR>
</TABLE>
```

```
<B>Enter new value for arrayThree[0]:</B><INPUT TYPE="text"
NAME="source1" VALUE="Jerry">
<INPUT TYPE="button" VALUE="Change arrayOne[0]"
onClick="update1(this.form)"><BR>
Current arrayOne[0] is:<INPUT TYPE="text" NAME="result1"><BR>
Current arrayThree[0] is:<INPUT TYPE="text" NAME="result2"><BR>
<HR>

textObj assigned to arrayTwo[2]:<INPUT TYPE="text" NAME="original"
onFocus="this.blur()"></BR>
<B>Enter new value for arrayThree[5]:</B><INPUT TYPE="text"
NAME="source2" VALUE="Phoebe">
<INPUT TYPE="button" VALUE="Change arrayTwo[2].value"
onClick="update2(this.form)"><BR>
Current arrayTwo[2].value is:<INPUT TYPE="text" NAME="result3"><BR>
Current arrayThree[2].value is:<INPUT TYPE="text" NAME="result4"><P>

<INPUT TYPE="button" VALUE="Reset" onClick="location.reload()">
</FORM>
</BODY>
</HTML>
```

Related Items: `array.join()` method.

arrayObject.join(*separatorString*)

Returns: String of entries from the array delimited by the *separatorString* value.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

You cannot view data in an array when it's in that form. Nor can you put an array into a form element for transmittal to a server CGI program. To make the transition from discrete array elements to string, the `array.join()` method handles what would otherwise be a nasty string manipulation exercise.

The sole parameter for this method is a string of one or more characters that you want to act as a delimiter between entries. For example, if you want commas between array items in their text version, the statement is

```
var arrayText = myArray.join(",")
```

Invoking this method does not change the original array in any way. Therefore, you need to assign the results of this method to another variable or a `value` property of a form element.

Example

The script in Listing 29-7 converts the now familiar array of planet names into a text string. The page provides you with a field to enter the delimiter string of your choice and shows the results in a textarea.

Listing 29-7: Using the Array.join() Method

```
<HTML>
<HEAD>
<TITLE>Array.join()</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
solarSys = new Array(9)
solarSys[0] = "Mercury"
solarSys[1] = "Venus"
solarSys[2] = "Earth"
solarSys[3] = "Mars"
solarSys[4] = "Jupiter"
solarSys[5] = "Saturn"
solarSys[6] = "Uranus"
solarSys[7] = "Neptune"
solarSys[8] = "Pluto"

// join array elements into a string
function convert(form) {
       var delimiter = form.delim.value
       form.output.value = unescape(solarSys.join(delimiter))
}
</SCRIPT>
<BODY>
<H2>Converting arrays to strings</H2>
This document contains an array of planets in our solar system.<HR>
<FORM>
Enter a string to act as a delimiter between entries:
<INPUT TYPE="text" NAME="delim" VALUE="," SIZE=5><P>
<INPUT TYPE="button" VALUE="Display as String"
onClick="convert(this.form)">
<INPUT TYPE="reset">
<TEXTAREA NAME="output" ROWS=4 COLS=40 WRAP="virtual">
</TEXTAREA>
</FORM>
</BODY>
</HTML>
```

Notice that this method takes the parameter very literally. If you want to include nonalphanumeric characters, such as a newline or tab, do so with escaped characters (`"%0D"` for a carriage return; `"%09"` for a tab) instead of inline string literals. In Listing 29-7, the results of the `Array.join()` method are subjected to the `unescape()` function in order to display them in the textarea.

Related Items: `String.split()` method.

arrayObject.pop()
arrayObject.push(valueOrObject)
arrayObject.shift()
arrayObject.unshift(valueOrObject)

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			

The notion of a *stack* is well known to experienced programmers, especially those who know about the inner workings of assembly language at the CPU level. Even if you've never programmed a stack before, you have encountered the concept in real life many times. The classic analogy is the spring-loaded pile of cafeteria trays. If the pile were created one at a time, each tray would be pushed into the stack of trays. When a customer comes along, the topmost tray (the last one to be pushed onto the stack) gets popped off. The last one to be put on the stack is the first one to be taken off.

JavaScript 1.2 in Navigator lets you turn an array into one of these spring-loaded stacks. But instead of placing trays on the pile, you can place any kind of data at either end of the stack, depending on which method you use to do the stacking. Similarly, you can extract an item from either end.

Perhaps the most familiar terminology for this is *push* and *pop*. When you `push()` a value onto an array, the value is appended as the last entry in the array. When you issue the `array.pop()` method, the last item in the array is removed from the stack and is returned, plus the array shrinks in length by one. In the following sequence of statements, watch what happens to the value of the array used as a stack:

```
var source = new Array("Homer","Marge","Bart","Lisa","Maggie")
var stack = new Array()
        // stack = <empty>
stack.push(source[0])
        // stack = "Homer"
stack.push(source[2])
        // stack = "Homer","Bart"
var Simpson1 = stack.pop()
        // stack = "Homer" ; Simpson1 = "Bart"
var Simpson2 = stack.pop()
        // stack = <empty> ; Simpson2 = "Homer"
```

While `push()` and `pop()` work at the end of an array, another pair of methods work at the front. Their names are not as picturesque as `push()` and `pop()`. To insert a value at the front of an array, use the `array.unshift()` method; to grab the first element and remove it from the array, use `array.shift()`. Of course you are not required to use these methods in matching pairs. If you `push()` a series of values onto the back end of an array, you can `shift()` them off from the front end without complaint. It all depends on how you need the data.

You may readily see existing parallels to some of these operations (especially the `push()` method), but if you need a temporary staging ground for data, the array stacking methods should help out quite a bit.

Related Items: `array.concat()` method; `array.slice()` method.

arrayObject.reverse()

Returns: Array of entries in the opposite order of the original.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

Occasionally, you may find it more convenient to work with an array of data in reverse order. Although you can concoct repeat loops to count backward through index values, a CGI program on the server may prefer the data in a sequence opposite to the way it was most convenient for you to script it.

You can have JavaScript switch the contents of an array for you: Whatever element was last in the array becomes the 0 index item in the array. Bear in mind that when you do this, you're restructuring the original array, not copying it. A reload of the document restores the order as written in the HTML document.

Example

In Listing 29-8, I enhanced Listing 29-7 by including another button and function that reverses the array and displays it as a string in a text area.

Listing 29-8: Array.reverse() Method

```
<HTML>
<HEAD>
<TITLE>Array.reverse()</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
solarSys = new Array(9)
solarSys[0] = "Mercury"
solarSys[1] = "Venus"
solarSys[2] = "Earth"
solarSys[3] = "Mars"
solarSys[4] = "Jupiter"
solarSys[5] = "Saturn"
solarSys[6] = "Uranus"
solarSys[7] = "Neptune"
solarSys[8] = "Pluto"

// show array as currently in memory
function showAsIs(form) {
        var delimiter = form.delim.value
        form.output.value = unescape(solarSys.join(delimiter))
}
```

```
// reverse array order, then display as string
function reverseIt(form) {
        var delimiter = form.delim.value
        solarSys.reverse()   // reverses original array
        form.output.value = unescape(solarSys.join(delimiter))
}
</SCRIPT>
<BODY>
<H2>Reversing array element order</H2>
This document contains an array of planets in our solar system.<HR>
<FORM>
Enter a string to act as a delimiter between entries:
<INPUT TYPE="text" NAME="delim" VALUE="," SIZE=5><P>
<INPUT TYPE="button" VALUE="Array as-is" onClick="showAsIs(this.form)">
<INPUT TYPE="button" VALUE="Reverse the array"
onClick="reverseIt(this.form)">
<INPUT TYPE="reset">
<INPUT TYPE="button" VALUE="Reload" onClick="self.location.reload()">
<TEXTAREA NAME="output" ROWS=4 COLS=60>
</TEXTAREA>
</FORM>
</BODY>
</HTML>
```

Notice that the `solarSys.reverse()` method stood by itself while the method modified the `solarSys` array. You then run the now inverted `solarSys` array through the `array.join()` method for your text display.

Related Items: `array.sort()` method.

arrayObject.slice(*startIndex* [, *endIndex*])

Returns: Array.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

Behaving like its like-named string method, `array.slice()` lets you extract a contiguous series of items from an array. The extracted segment becomes an entirely new array object. Values and objects from the original array have the same kind of behavior as arrays created with the `array.concat()` method.

One parameter is required — the starting index point for the extraction. If you don't specify a second parameter, the extraction goes all the way to the end of the array; otherwise the extraction goes to *but does not include* the index value supplied as the second parameter. For example, extracting Earth's neighbors from an array of planet names would look like this:

```
var solarSys = new
Array("Mercury","Venus","Earth","Mars","Jupiter","Saturn","Uranus","Nep
tune","Pluto")
var nearby = solarSys.slice(1,4)
        // result: new array of "Venus", "Earth", "Mars"
```

Related Items: `string.slice()` method.

arrayObject.sort([*compareFunction*])

Returns: Array of entries in the order as determined by the *compareFunction* algorithm.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

JavaScript array sorting is both powerful and a bit complex to script if you haven't had experience with this kind of sorting methodology. The purpose, obviously, is to let your scripts sort entries of an array by almost any kind of criterion that you can associate with an entry. For entries consisting of strings, the criterion may be their alphabetical order or their length; for numeric entries, the criterion may be their numerical order.

Let's look first at the kind of sorting you can do with the `array.sort()` method by itself (for example, without calling a comparison function). When no parameter is specified, JavaScript takes a snapshot of the contents of the array and converts items to strings. From there, it performs a string sort of the values. ASCII values of characters govern the sort, which means that numbers are sorted by their string values, not their numeric values. This fact has strong implications if your array consists of numeric data: The value 201 sorts before 88, because the sorting mechanism compares the first characters of the strings (`"2"` versus `"8"`) to determine the sort order. For simple alphabetical sorting of string values in arrays, the plain `Array.sort()` method should do the trick.

Fortunately, additional intelligence is available that you can add to array sorting. The key tactic is to define a function that helps the `sort()` method compare items in the array. A comparison function is passed two values from the array (what you don't see is that the `Array.sort()` method rapidly sends numerous pairs of values from the array to help it sort through all entries). The comparison function lets the `sort()` method know which of the two items comes before the other based on the value the function returns. Assuming that the function compares two values, `a` and `b`, the returned value reveals information to the `sort()` method, as shown in Table 29-1.

Table 29-1
Comparison Function Return Values

Return Value Range	*Meaning*
< 0	Value b should sort above a
0	The order of a and b should not change
> 0	Value a should sort above b

Consider the following example:

```
myArray = new Array(12, 5, 200, 80)
function compare(a,b) {
        return a - b
}
myArray.sort(compare)
```

The array has four numeric values in it. To sort the items in numerical order, you define a comparison function (arbitrarily named `compare()`), which is called from the `sort()` method. Note that unlike invoking other functions, the parameter of the `sort()` method uses a reference to the function, which lacks parentheses.

When the `compare()` function is called, JavaScript automatically sends two parameters to the function in rapid succession until each element has been compared against the others. Every time `compare()` is called, JavaScript assigns two of the array's values to the parameter variables (`a` and `b`). In the preceding example, the returned value is the difference between `a` and `b`. If `a` is larger than `b`, then a positive value goes back to the `sort()` method, telling it to sort `a` above `b` (that is, position `a` at a lower value index position than `b`). Therefore, `a` may end up at `myArray[0]`, whereas `b` ends up at a higher index-valued location. On the other hand, if `b` is larger than `a`, then the returned negative value tells `sort()` to put `b` in a lower index value spot than `a`.

Evaluations within the comparison function can go to great lengths, as long as some data connected with array values can be compared. For example, instead of numerical comparisons, as just shown, you can perform string comparisons. The following function sorts alphabetically by the last character of each array string entry:

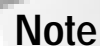
```
function compare(a,b) {
        // last character of array strings
        var aComp = a.charAt(a.length - 1)
        var bComp = b.charAt(b.length - 1)
        if (aComp < bComp) {return -1}
        if (aComp > bComp) {return 1}
        return 0
}
```

First, this function extracts the final character from each of the two values passed to it. Then, because strings cannot be added or subtracted like numbers, you compare the ASCII values of the two characters, returning the corresponding

values to the `sort()` method to let it know how to treat the two values being checked at that instant.

Array sorting, unlike sorting routines you might find in other scripting languages, is not a stable sort. This means that succeeding sort routines on the same array are not cumulative. Also remember that sorting changes the sort order of the original array. If you don't want the original array harmed, make a copy of it before sorting; or reload the document to restore an array to its original order.

Should an array element be null, the method sorts such elements at the end of the sorted array starting with Navigator 4 (instead of leaving them in their original places as in Navigator 3).


Note

Unfortunately, this powerful method does not work in the Macintosh version of Navigator 3. All platforms have the feature starting with Navigator 4.

Example

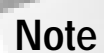
You can look to Listing 29-9 for a few examples of sorting an array of string values. Four buttons summon different sorting routines, three of which invoke comparison functions. This listing sorts the planet array alphabetically (forward and backward) by the last character of the planet name and also by the length of the planet name. Each comparison function demonstrates different ways of comparing data sent during a sort.

Listing 29-9: **Array.sort() Possibilities**

```
<HTML>
<HEAD>
<TITLE>Array.sort()</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
solarSys = new Array(9)
solarSys[0] = "Mercury"
solarSys[1] = "Venus"
solarSys[2] = "Earth"
solarSys[3] = "Mars"
solarSys[4] = "Jupiter"
solarSys[5] = "Saturn"
solarSys[6] = "Uranus"
solarSys[7] = "Neptune"
solarSys[8] = "Pluto"
// comparison functions
function compare1(a,b) {
        // reverse alphabetical order
        if (a > b) {return -1}
        if (b > a) {return 1}
        return 0
}
function compare2(a,b) {
        // last character of planet names
        var aComp = a.charAt(a.length - 1)
        var bComp = b.charAt(b.length - 1)
        if (aComp < bComp) {return -1}
        if (aComp > bComp) {return 1}
        return 0
```

```
}
function compare3(a,b) {
        // length of planet names
        return a.length - b.length
}
// sort and display array
function sortIt(form, compFunc) {
        var delimiter = ";"
        if (compFunc == null) {
           solarSys.sort()
        } else {
           solarSys.sort(compFunc)
        }
        // display results in field
        form.output.value = unescape(solarSys.join(delimiter))
}
</SCRIPT>
<BODY onLoad="document.forms[0].output.value =
unescape(solarSys.join(';'))">
<H2>Sorting array elements</H2>
This document contains an array of planets in our solar system.<HR>
<FORM>
Click on a button to sort the array:<P>
<INPUT TYPE="button" VALUE="Alphabetical A-Z"
onClick="sortIt(this.form)">
<INPUT TYPE="button" VALUE="Alphabetical Z-A"
onClick="sortIt(this.form,compare1)">
<INPUT TYPE="button" VALUE="Last Character"
onClick="sortIt(this.form,compare2)">
<INPUT TYPE="button" VALUE="Name Length"
onClick="sortIt(this.form,compare3)">
<INPUT TYPE="button" VALUE="Reload Original"
onClick="self.location.reload()">
<INPUT TYPE="text" NAME="output" SIZE=62>
</TEXTAREA>
</FORM>
</BODY>
</HTML>
```

Related Items: `array.reverse()` method.



Note

As I show you in the next chapter, many regular expression object methods generate arrays as their result (for example, an array of matching values in a string). These special arrays have a custom set of named properties that assist your script in analyzing the findings of the method. Beyond that, these regular expression result arrays behave like all others.