


- 14.4 In Fig. 14.2, we subdivided the author element into more detailed pieces. How might you subdivide the date element? Use the date May 5, 2005, as an example.
- 14.5 Write a processing instruction that includes style sheet wap.xsl.
- 14.6 Write an XPath expression that locates contact nodes in letter.xml (Fig. 14.4).

Exercises

- 14.7 (*Nutrition Information XML Document*) Create an XML document that marks up the nutrition facts for a package of Grandma White's cookies. A package of cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 4 grams of saturated fat, 5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugars and 5 grams of protein. Name this document `nutrition.xml`. Load the XML document into Internet Explorer. [*Hint:* Your markup should contain elements describing the product name, serving size/amount, calories, sodium, cholesterol, proteins, etc. Mark up each nutrition fact/ingredient listed above.]
- 14.8 (*Nutrition Information XSL Style Sheet*) Write an XSL style sheet for your solution to Exercise 14.7 that displays the nutritional facts in an XHTML table. Modify Fig. 14.38 to output the results.
- 14.9 (*Sorting XSLT Modification*) Modify Fig. 14.23 (`sorting.xml`) to sort by the number of pages rather than by chapter number. Save the modified document as `sorting_byPage.xml`.
- 14.10 Modify Fig. 14.38 to use `sorting.xml` (Fig. 14.22), `sorting.xsl` (Fig. 14.23) and `sorting_byPage.xml` (from Exercise 14.9). Display the result of transforming `sorting.xml` using each style sheet. [*Hint:* Remove the `xml:stylesheet` processing instruction from line 2 of `sorting.xml` before attempting to transform the file programmatically.]




Ajax-Enabled Rich Internet Applications

OBJECTIVES

In this chapter you will learn:

- What Ajax is and why it is important for building Rich Internet Applications.
- What asynchronous requests are and how they help give web applications the feel of desktop applications.
- What the XMLHttpRequest object is and how it's used to create and manage asynchronous requests to servers and to receive asynchronous responses from servers.
- Methods and properties of the XMLHttpRequest object.
- How to use XHTML, JavaScript, CSS, XML, JSON and the DOM in Ajax applications.
- How to use Ajax frameworks and toolkits, specifically Dojo, to conveniently create robust Ajax-enabled Rich Internet Applications.
- About resources for studying Ajax-related issues such as security, performance, debugging, the "back-button problem" and more.



... the challenges are for the designers of these applications: to forget what we think we know about the limitations of the Web, and begin to imagine a wider, richer range of possibilities. It's going to be fun.

—Jesse James Garrett

Dojo is the standard library JavaScript never had.

—Alex Russell

To know how to suggest is the great art of teaching. To attain it we must be able to guess what will interest ...

—Henri-Fredric Amiel

It is characteristic of the epistemological tradition to present us with partial scenarios and then to demand whole or categorical answers as if were.

—Avrum Stroll

O! call back yesterday, bid time return.

—William Shakespeare

- 15.1 Introduction
- 15.2 Traditional Web Applications vs. Ajax Applications
- 15.3 Rich Internet Applications (RIAs) with Ajax
- 15.4 History of Ajax
- 15.5 “Raw” Ajax Example Using the XMLHttpRequest Object
- 15.6 Using XML and the DOM
- 15.7 Creating a Full-Scale Ajax-Enabled Application
- 15.8 Ajax Toolkits
- 15.9 Web Resources

Summary | Terminology | Self-Review Exercises | Exercises

15.1 Introduction

Despite the tremendous technological growth of the Internet over the past decade, the usability of web applications has lagged behind compared to that of desktop applications. Every significant interaction in a web application results in a waiting period while the application communicates over the Internet with a server. **Rich Internet Applications (RIAs)** are web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and a rich GUI.

RIA performance comes from **Ajax (Asynchronous JavaScript and XML)**, which uses client-side scripting to make web applications more responsive. Ajax applications separate client-side user interaction and server communication, and run them in parallel, reducing the delays of server-side processing normally experienced by the user.

There are many ways to implement Ajax functionality. “**Raw**” Ajax uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM (see Section 15.5). “Raw” Ajax is best suited for creating small Ajax components that asynchronously update a section of the page. However, when writing “raw” Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications. These portability issues are hidden by **Ajax toolkits**, such as **Dojo** (Section 15.8), **Prototype**, **Script.aculo.us** and **ASP.NET Ajax**, which provide powerful ready-to-use controls and functions that enrich web applications, and simplify JavaScript coding by making it cross-browser compatible.

Traditional web applications use XHTML forms (Chapter 4) to build simple and thin GUIs compared to the rich GUIs of Windows, Macintosh and desktop systems in general. We achieve rich GUI in RIAs with Ajax toolkits and with RIA environments such as Adobe’s **Flex** (Chapter 18), Microsoft’s **Silverlight** (Chapter 19) and **JavaServer Faces** (Chapters 26–27). Such toolkits and environments provide powerful ready-to-use controls and functions that enrich web applications.

Previous chapters discussed XHTML, CSS, JavaScript, dynamic HTML, the DOM and XML. This chapter uses these technologies to build Ajax-enabled web applications. The client-side of Ajax applications is written in XHTML and CSS, and uses JavaScript to add functionality to the user interface. XML is used to structure the data passed between the server and the client. We’ll also use **JSON (JavaScript Object Notation)** for this purpose. The Ajax component that manages interaction with the server is usually implemented

with JavaScript’s `XMLHttpRequest` object—commonly abbreviated as XHR. The server processing can be implemented using any server-side technology, such as PHP, ASP.NET, JavaServer Faces and Ruby on Rails—each of which we cover in later chapters.

This chapter begins with several examples that build basic Ajax applications using JavaScript and the `XMLHttpRequest` object. We then build an Ajax application with a rich calendar GUI using the Dojo Ajax toolkit. In subsequent chapters, we use tools such as Adobe Flex, Microsoft Silverlight and JavaServer Faces to build RIAs using Ajax. In Chapter 24, we’ll demonstrate features of the Prototype and Script.aculo.us Ajax libraries, which come with the Ruby on Rails framework (and can be downloaded separately). Prototype provides capabilities similar to Dojo. Script.aculo.us provides many “eye candy” effects that enable you to beautify your Ajax applications and create rich interfaces. In Chapter 27, we present Ajax-enabled JavaServer Faces (JSF) components. JSF uses Dojo to implement many of its client-side Ajax capabilities.

15.2 Traditional Web Applications vs. Ajax Applications

In this section, we consider the key differences between traditional web applications and Ajax-based web applications.

Traditional Web Applications

Figure 15.1 presents the typical interactions between the client and the server in a traditional web application, such as one that uses a user registration form. First, the user fills in the form’s fields, then submits the form (Fig. 15.1, *Step 1*). The browser generates a request to the server, which receives the request and processes it (*Step 2*). The server generates and sends a response containing the exact page that the browser will render (*Step 3*), which causes the browser to load the new page (*Step 4*) and temporarily makes the browser window blank. Note that the client *waits* for the server to respond and *reloads the entire page* with the data from the response (*Step 4*). While such a **synchronous request** is being

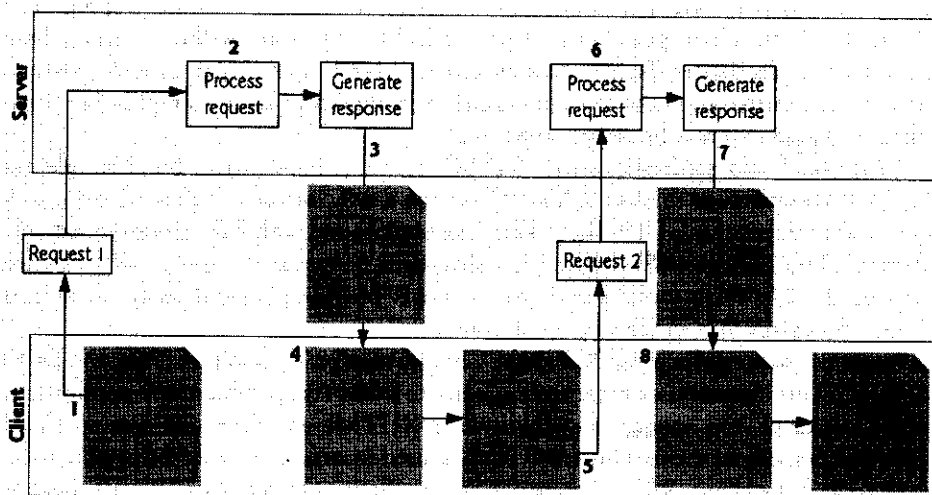


Fig. 15.1 | Classic web application reloading the page for every user interaction.

processed on the server, the user cannot interact with the client web page. Frequent long periods of waiting, due perhaps to Internet congestion, have led some users to refer to the World Wide Web as the “World Wide Wait.” If the user interacts with and submits another form, the process begins again (Steps 5–8).

This model was originally designed for a web of hypertext documents—what some people call the “brochure web.” As the web evolved into a full-scale applications platform, the model shown in Fig. 15.1 yielded “choppy” application performance. Every full-page refresh required users to re-establish their understanding of the full-page contents. Users began to demand a model that would yield the responsive feel of desktop applications.

Ajax Web Applications

Ajax applications add a layer between the client and the server to manage communication between the two (Fig. 15.2). When the user interacts with the page, the client creates an XMLHttpRequest object to manage a request (Step 1). The XMLHttpRequest object sends the request to the server (Step 2) and awaits the response. The requests are asynchronous, so the user can continue interacting with the application on the client-side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server (Steps 3 and 4). Once the server responds to the original request (Step 5), the XMLHttpRequest object that issued the request calls a client-side function to process the data returned by the server. This function—known as a **callback function**—uses **partial page updates** (Step 6) to display the data in the existing web page *without reloading the entire page*. At the same time, the server may be responding to the second request (Step 7) and the client-side may be starting to do another partial page update (Step 8). The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications. The web application does not load a new page while the user interacts with it.

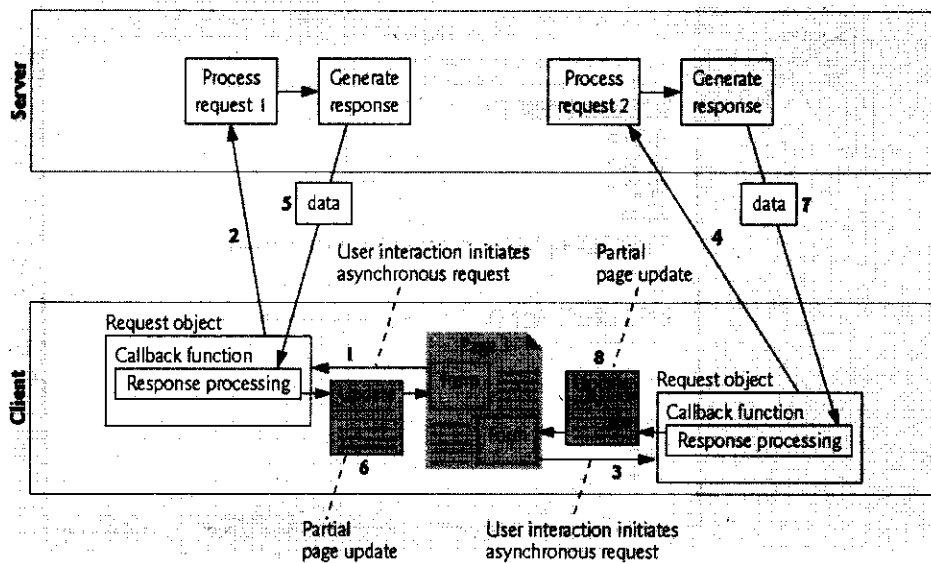


Fig. 15.2 | Ajax-enabled web application interacting with the server asynchronously.

15.3 Rich Internet Applications (RIAs) with Ajax

Ajax improves the user experience by making interactive web applications more responsive. Consider a registration form with a number of fields (e.g., first name, last name, e-mail address, telephone number, etc.) and a **Register** (or **Submit**) button that sends the entered data to the server. Usually each field has rules that the user's entries have to follow (e.g., valid e-mail address, valid telephone number, etc.).

When the user clicks **Register**, a classic XHTML form sends the server all of the data to be validated (Fig. 15.3). While the server is validating the data, the user cannot interact with the page. The server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser. Once the user fixes the errors and clicks the **Register** button, the cycle repeats until no errors are found, then the data is stored on the server. The entire page reloads every time the user submits invalid data.

Ajax-enabled forms are more interactive. Rather than sending the entire form to be validated, entries are validated dynamically as the user enters data into the fields. For example, consider a website registration form that requires a unique e-mail address. When the user enters an e-mail address into the appropriate field, then moves to the next form field to continue entering data, an asynchronous request is sent to the server to validate the e-mail address. If the e-mail address is not unique, the server sends an error message that is displayed on the page informing the user of the problem (Fig. 15.4). By sending each entry asynchronously, the user can address each invalid entry quickly, versus making edits and resubmitting the entire form repeatedly until all entries are valid. Asynchronous

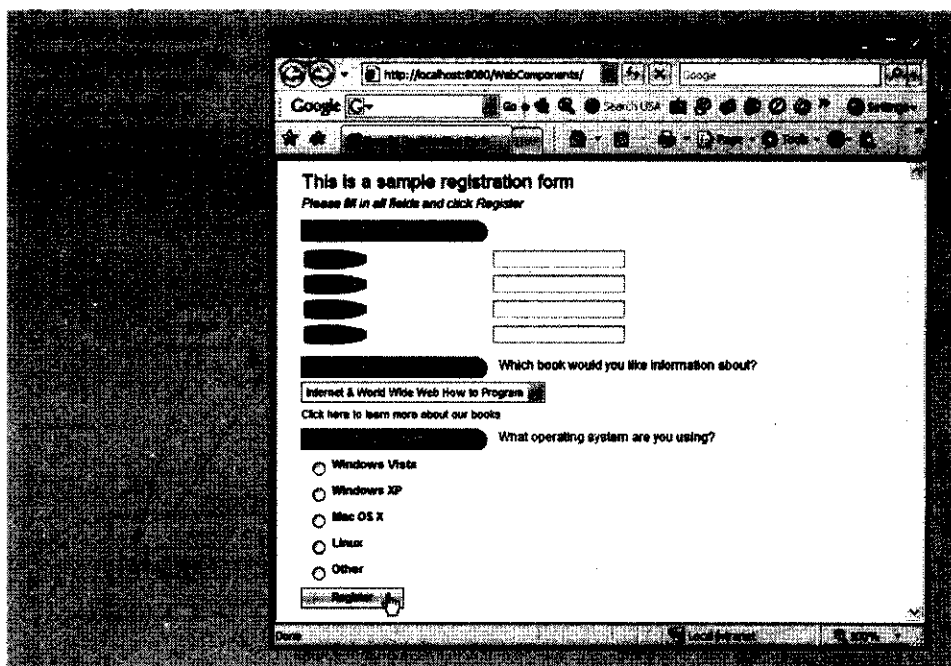


Fig. 15.3 | Classic XHTML form: User submits entire form to server, which validates the data entered (if any). Server responds indicating fields with invalid or missing data. (Part 1 of 2.)

b) The server responds by indicating all the form fields with missing or invalid data. The user must correct the problems and resubmit the entire form repeatedly until all errors are corrected.

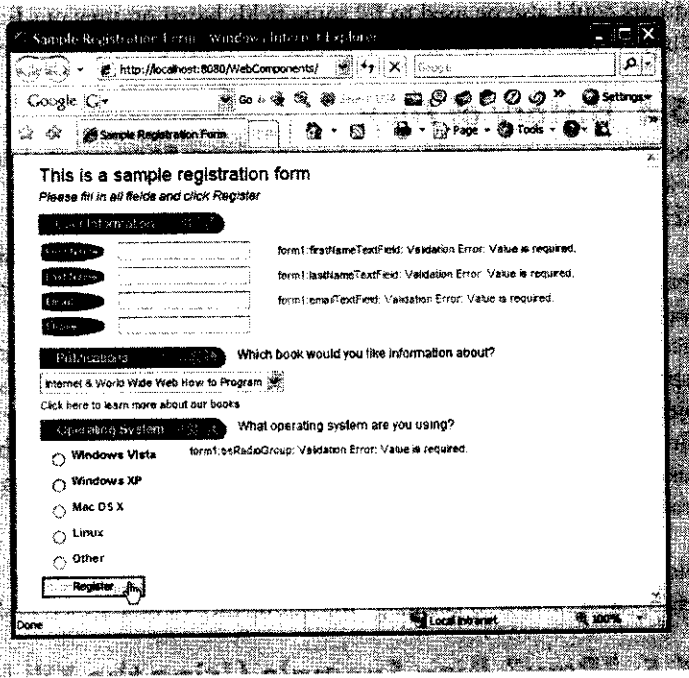


Fig. 15.3 | Classic XHTML form: User submits entire form to server, which validates the data entered (if any). Server responds indicating fields with invalid or missing data. (Part 2 of 2.)

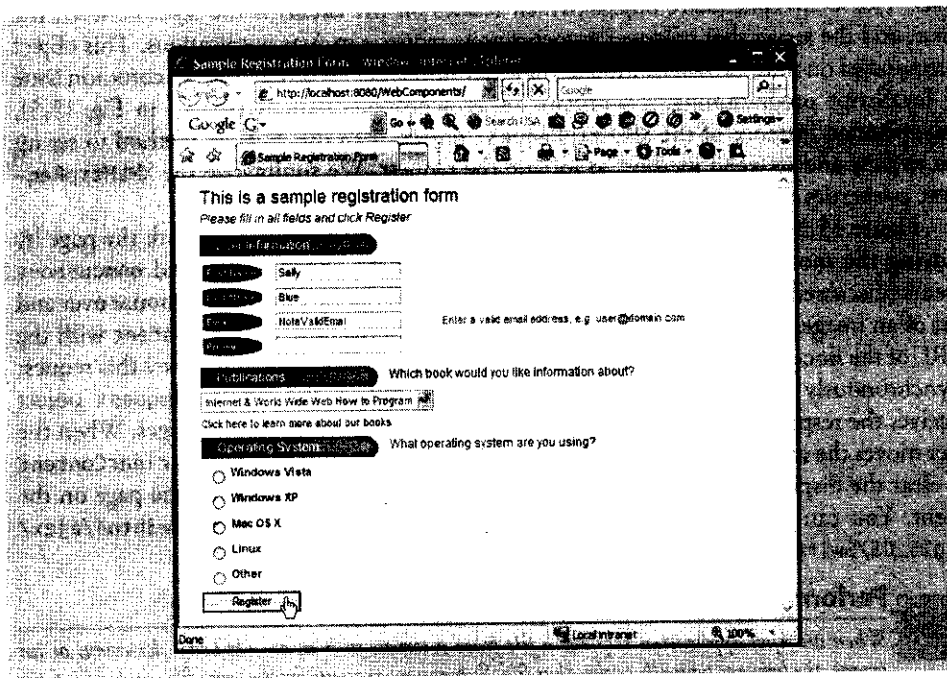


Fig. 15.4 | Ajax-enabled form shows errors asynchronously when user moves to another field.

requests could also be used to fill some fields based on previous fields (e.g., automatically filling in the “city” and “state” fields based on the zip code entered by the user).

15.4 History of Ajax

The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client. The technologies of Ajax (XHTML, JavaScript, CSS, the DOM and XML) have all existed for many years.

Asynchronous page updates can be traced back to earlier browsers. In the 1990s, Netscape’s LiveScript made it possible to include scripts in web pages (e.g., web forms) that could run on the client. LiveScript evolved into JavaScript. In 1998, Microsoft introduced the XMLHttpRequest object to create and manage asynchronous requests and responses. Popular applications like Flickr and Google’s Gmail use the XMLHttpRequest object to update pages dynamically. For example, Flickr uses the technology for its text editing, tagging and organizational features; Gmail continuously checks the server for new e-mail; and Google Maps allows you to drag a map in any direction, downloading the new areas on the map without reloading the entire page.

The name Ajax immediately caught on and brought attention to its component technologies. Ajax has become one of the hottest web-development technologies, enabling webtop applications to challenge the dominance of established desktop applications.

15.5 “Raw” Ajax Example Using the XMLHttpRequest Object

In this section, we use the XMLHttpRequest object to create and manage asynchronous requests. The XMLHttpRequest object (which resides on the client) is the layer between the client and the server that manages asynchronous requests in Ajax applications. This object is supported on most browsers, though they may implement it differently—a common issue in JavaScript programming. To initiate an asynchronous request (shown in Fig. 15.5), you create an instance of the XMLHttpRequest object, then use its open method to set up the request and its send method to initiate the request. We summarize the XMLHttpRequest properties and methods in Figs. 15.6–15.7.

Figure 15.5 presents an Ajax application in which the user interacts with the page by moving the mouse over book-cover images. We use the onmouseover and onmouseout events (discussed in Chapter 13) to trigger events when the user moves the mouse over and out of an image, respectively. The onmouseover event calls function getContent with the URL of the document containing the book’s description. The function makes this request asynchronously using an XMLHttpRequest object. When the XMLHttpRequest object receives the response, the book description is displayed below the book images. When the user moves the mouse out of the image, the onmouseout event calls function clearContent to clear the display box. These tasks are accomplished without reloading the page on the client. You can test-drive this example at test.deitel.com/examples/iw3http4/ajax/fig15_05/SwitchContent.html.



Performance Tip 15.1

When an Ajax application requests a file from a server, such as an XHTML document or an image, the browser typically caches that file. Subsequent requests for the same file can load it from the browser’s cache rather than making the round trip to the server again.


```
31     catch ( exception )
32     {
33         alert( 'Request failed.' );
34     } // end catch
35 } // end function getContent
36
37 // displays the response area on the page
38 function stateChange()
39 {
40     if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
41     {
42         document.getElementById( 'contentArea' ).innerHTML =
43             asyncRequest.responseText; // places text in contentArea
44     } // end if
45 } // end function stateChange
46
47 // clear the content of the box
48 function clearContent()
49 {
50     document.getElementById( 'contentArea' ).innerHTML = ""
51 } // end function clearContent
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

<div class = "box" id = "contentArea"> </div>

Fig. 15.5 | Asynchronously display content without reloading the page. (Part 2 of 3.)

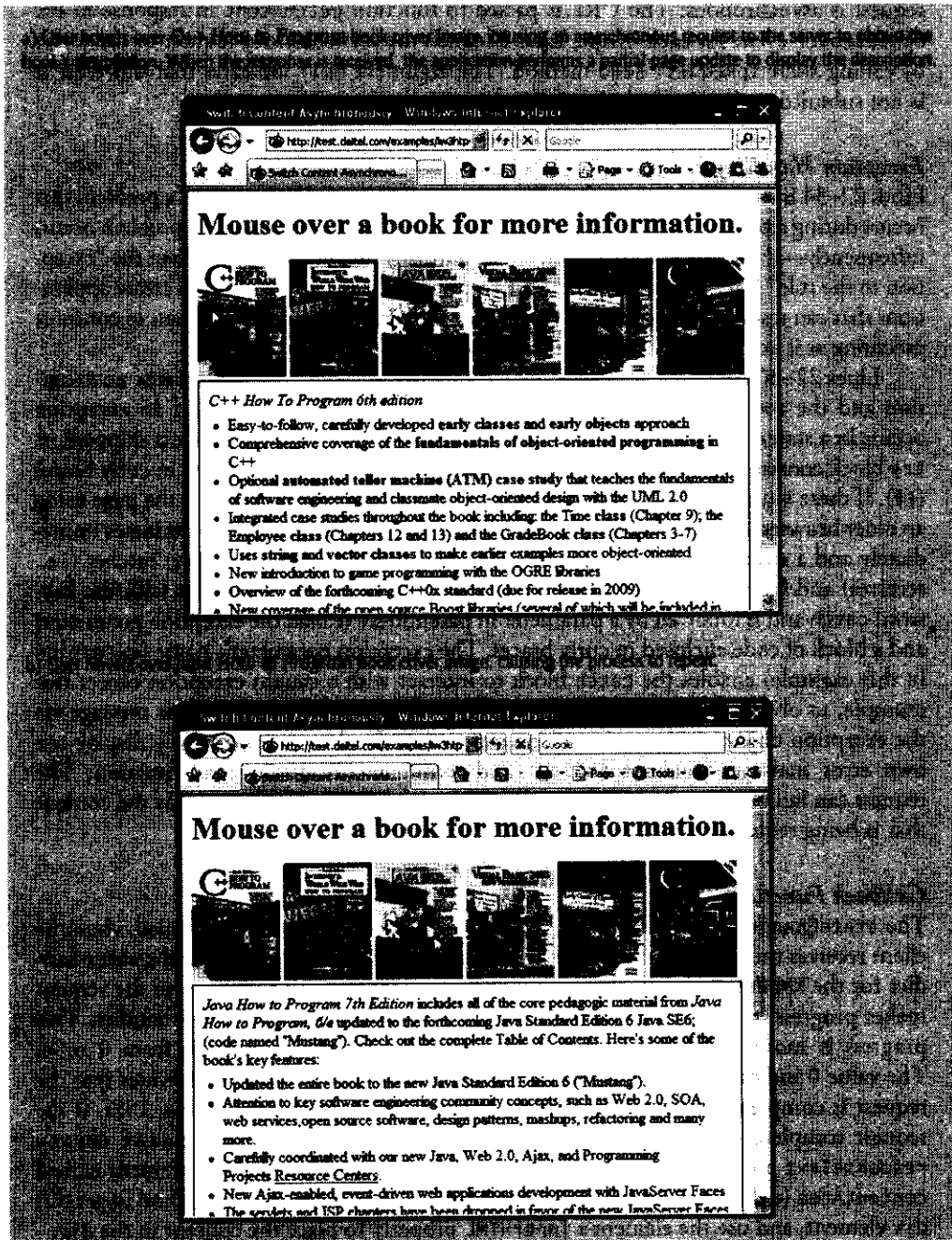


Fig. 15.5 | Asynchronously display content without reloading the page. (Part 3 of 3.)

Line 28 calls the XMLHttpRequest open method to prepare an asynchronous GET request. In this example, the url parameter specifies the address of an HTML document containing the description of a particular book. When the third argument is true, the

request is asynchronous. The URL is passed to function `getContent` in response to the `onmouseover` event for each image. Line 29 sends the asynchronous request to the server by calling `XMLHttpRequest` `send` method. The argument `null` indicates that this request is not submitting data in the body of the request.

Exception Handling

Lines 22–34 introduce **exception handling**. An **exception** is an indication of a problem that occurs during a program's execution. The name "exception" implies that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the "exception to the rule" is that a problem occurs. Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered.

Lines 22–30 contain a **try block**, which encloses the code that might cause an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs in a statement of the try block, the remaining code in the try block is skipped). A try block consists of the keyword `try` followed by a block of code enclosed in curly braces (`{}`). If there is a problem sending the request—e.g., if a user tries to access the page using an older browser that does not support `XMLHttpRequest`—the try block terminates immediately and a **catch block** (also called a **catch clause** or **exception handler**) catches (i.e., receives) and handles an exception. The catch block (lines 31–34) begins with the keyword `catch` and is followed by a parameter in parentheses (called the exception parameter) and a block of code enclosed in curly braces. The exception parameter's name (exception in this example) enables the catch block to interact with a caught exception object (for example, to obtain the name of the exception or an exception-specific error message via the exception object's `name` and `message` properties). In this case, we simply display our own error message 'Request Failed' and terminate the `getContent` function. The request can fail because a user accesses the web page with an older browser or the content that is being requested is located on a different domain.

Callback Functions

The `stateChange` function (lines 38–45) is the callback function that is called when the client receives the response data. Line 27 registers function `stateChange` as the event handler for the `XMLHttpRequest` object's `onreadystatechange` event. Whenever the request makes progress, the `XMLHttpRequest` calls the `onreadystatechange` event handler. This progress is monitored by the `readyState` property, which has a value from 0 to 4. The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete—all the values for this property are summarized in Fig. 15.6. If the request completes successfully (line 40), lines 42–43 use the `XMLHttpRequest` object's `responseText` property to obtain the response data and place it in the `div` element named `contentArea` (defined at line 81). We use the DOM's `getElementById` method to get this `div` element, and use the element's `innerHTML` property to place the content in the `div`.

XMLHttpRequest Object Properties and Methods

Figures 15.6 and 15.7 summarize some of the `XMLHttpRequest` object's properties and methods, respectively. The properties are crucial to interacting with asynchronous quests. The methods initialize, configure and send asynchronous requests.

<code>onreadystatechange</code>	Stores the callback function—the event handler that gets called when the server responds.
<code>readyState</code>	Keeps track of the request's progress. It is usually used in the callback function to determine when the code that processes the response should be launched. The <code>readyState</code> value 0 signifies that the request is uninitialized; 1 signifies that the request is loading; 2 signifies that the request has been loaded; 3 signifies that data is actively being sent from the server; and 4 signifies that the request has been completed.
<code>responseText</code>	Text that is returned to the client by the server.
<code>responseXML</code>	If the server's response is in XML format, this property contains the XML document; otherwise, it is empty. It can be used like a document object in JavaScript, which makes it useful for receiving complex data (e.g. populating a table).
<code>status</code>	HTTP status code of the request. A status of 200 means that request was successful. A status of 404 means that the requested resource was not found. A status of 500 denotes that there was an error while the server was processing the request.
<code>statusText</code>	Additional information on the request's status. It is often used to display the error to the user when the request fails.

Fig. 15.6 | XMLHttpRequest object properties.

<code>open</code>	Initializes the request and has two mandatory parameters—method and URL. The method parameter specifies the purpose of the request—typically GET if the request is to take data from the server or POST if the request will contain a body in addition to the headers. The URL parameter specifies the address of the file on the server that will generate the response. A third optional boolean parameter specifies whether the request is asynchronous—it's set to <code>true</code> by default.
<code>send</code>	Sends the request to the sever. It has one optional parameter, <code>data</code> , which specifies the data to be POSTed to the server—it's set to <code>null</code> by default.
<code>setRequestHeader</code>	Alters the header of the request. The two parameters specify the header and its new value. It is often used to set the <code>content-type</code> field.

Fig. 15.7 | XMLHttpRequest object methods. (Part 1 of 2.)

<code>getResponseHeader</code>	Returns the header data that precedes the response body. It takes one parameter, the name of the header to retrieve. This call is often used to determine the response's type, to parse the response correctly.
<code>getAllResponseHeaders</code>	Returns an array that contains all the headers that precede the response body.
<code>abort</code>	Cancels the current request.

Fig. 15.7 | XMLHttpRequest object methods. (Part 2 of 2.)

15.6 Using XML and the DOM

When passing structured data between the server and the client, Ajax applications often use XML because it is easy to generate and parse. When the XMLHttpRequest object receives XML data, it parses and stores the data as an XML DOM object in the responseXML property. The example in Fig. 15.8 asynchronously requests from a server XML documents containing URLs of book-cover images, then displays the images in an HTML table. The code that configures the asynchronous request is the same as in Fig. 15.5. You can test-drive this application at test.deitel.com/examples/iw3http4/ajax/fig15_08/PullImagesOntoPage.html (the book-cover images will be easier to see on the screen).

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5 <html>
6 <head>
7 <title>XML Example</title>
8 </head>
9 <body>
10 <table border="1">
11 <thead>
12 <tr>
13 <th>Book Title</th>
14 <th>Book Cover</th>
15 </tr>
16 </thead>
17 <tbody>
18 <tr>
19 <td>The Hobbit</td>
20 <td></td>
21 </tr>
22 <tr>
23 <td>The Lord of the Rings</td>
24 <td></td>
25 </tr>
26 </tbody>
27 </table>
28 </body>
29 </html>

```

Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 1 of 4.)

```

25 // register event handler
26 XMLHttpRequest.onreadystatechange = function() {
27     if (XMLHttpRequest.readyState == 4) { // request complete
28         // send the request
29         // and so
30         break;
31     }
32     // start "Request Failed"
33     // and so
34     // and function path
35 }
36 // parse the XML response, dynamically creates a table using tbody
37 // populates it with the response data, discards the tbody of the page
38 // function processResponse()
39 {
40     // if request completed successfully and response is not null
41     if ( XMLHttpRequest.readyState == 4 && XMLHttpRequest.status == 200 &&
42         XMLHttpRequest.responseXML )
43     {
44         clearTable(); // prepare to display a new set of images
45
46         // get the covers from the responseXML
47         var covers = XMLHttpRequest.responseXML.getElementsByTagName(
48             "cover" );
49
50         // get base URL for the images
51         var baseUrl = XMLHttpRequest.responseXML.getElementsByTagName(
52             "baseUrl" ).item( 0 ).firstChild.nodeValue;
53
54         // get the placeholder div element named covers
55         var tbody = document.getElementsByTagName( "tbody" );
56
57         // create a table to display the images
58         var table = document.createElement( "table" );
59
60         // create the table's body
61         var tbody2 = document.createElement( "tbody" );
62
63         var rowCount = 0; // track number of images to display
64         var tbody2.innerHTML = document.createElement( "tr" ); // create row
65
66         // place images in row
67         for ( cover = 0; i < covers.length; i++)
68         {
69             var cover = covers.item( i ); // get a cover from covers
70
71             // create the image element
72             var img = document.createElement( "img" );
73             img.src = baseUrl + cover.firstChild.nodeValue;
74
75             // create table cell and append to tbody
76             var imgCell = document.createElement( "td" );

```

Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 2 of 4.)

```

77     var imageTag = document.createElement( "img" );
78
79     // set img element's src attribute
80     imageTag.setAttribute( "src", baseUrl + escapedImage );
81     imageCell.appendChild( imageTag ); // place img in cell
82     imageRow.appendChild( imageCell ); // place cell in row
83     rowCount++; // increment number of images in row
84
85     // if there are 6 images in the row, append the row to
86     // table and start a new row
87     if ( rowCount == 6 || 1 + 1 < covers.length )
88     {
89         tableBody.appendChild( imageRow );
90         imageRow = document.createElement( "tr" );
91         rowCount = 0;
92     } // end if statement
93 } // end for statement
94
95 tableBody.appendChild( imageRow ); // append row to table body
96 imageTable.appendChild( tableBody ); // append body to table
97 output.appendChild( imageTable ); // append table to covers div
98 } // end if
99 } // end function processResponse
100
101 // deletes the data in the table.
102 function clearTable()
103 {
104     document.getElementById( "covers" ).innerHTML = "";
105 } // end function clearTable
106
107 </script>
108
109 <div type = "radio" checked = "unchecked" name = "books" value = "all">
110     <input type = "radio" checked = "unchecked" name = "books" value = "all" /> All books
111 </div>
112 <div type = "radio" checked = "unchecked"
113     name = "books" value = "simple">
114     <input type = "radio" checked = "unchecked" name = "books" value = "simple" /> Simply Books
115 </div>
116 <div type = "radio" checked = "unchecked"
117     name = "books" value = "howto">
118     <input type = "radio" checked = "unchecked" name = "books" value = "howto" /> How to Program Books
119 </div>
120 <div type = "radio" checked = "unchecked"
121     name = "books" value = "dotnet">
122     <input type = "radio" checked = "unchecked" name = "books" value = "dotnet" /> .NET Books
123 </div>
124 <div type = "radio" checked = "unchecked"
125     name = "books" value = "javacpp">
126     <input type = "radio" checked = "unchecked" name = "books" value = "javacpp" /> Java, C, C++ Books
127 </div>
128 <div type = "radio" checked = "checked" name = "books" value = "none">
129     <input checked = "checked" type = "radio" name = "books" value = "none" /> None
130 </div>
131 </div id = "covers"></div>
132
133 </script>
134 </div>

```

Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 3 of 4.)

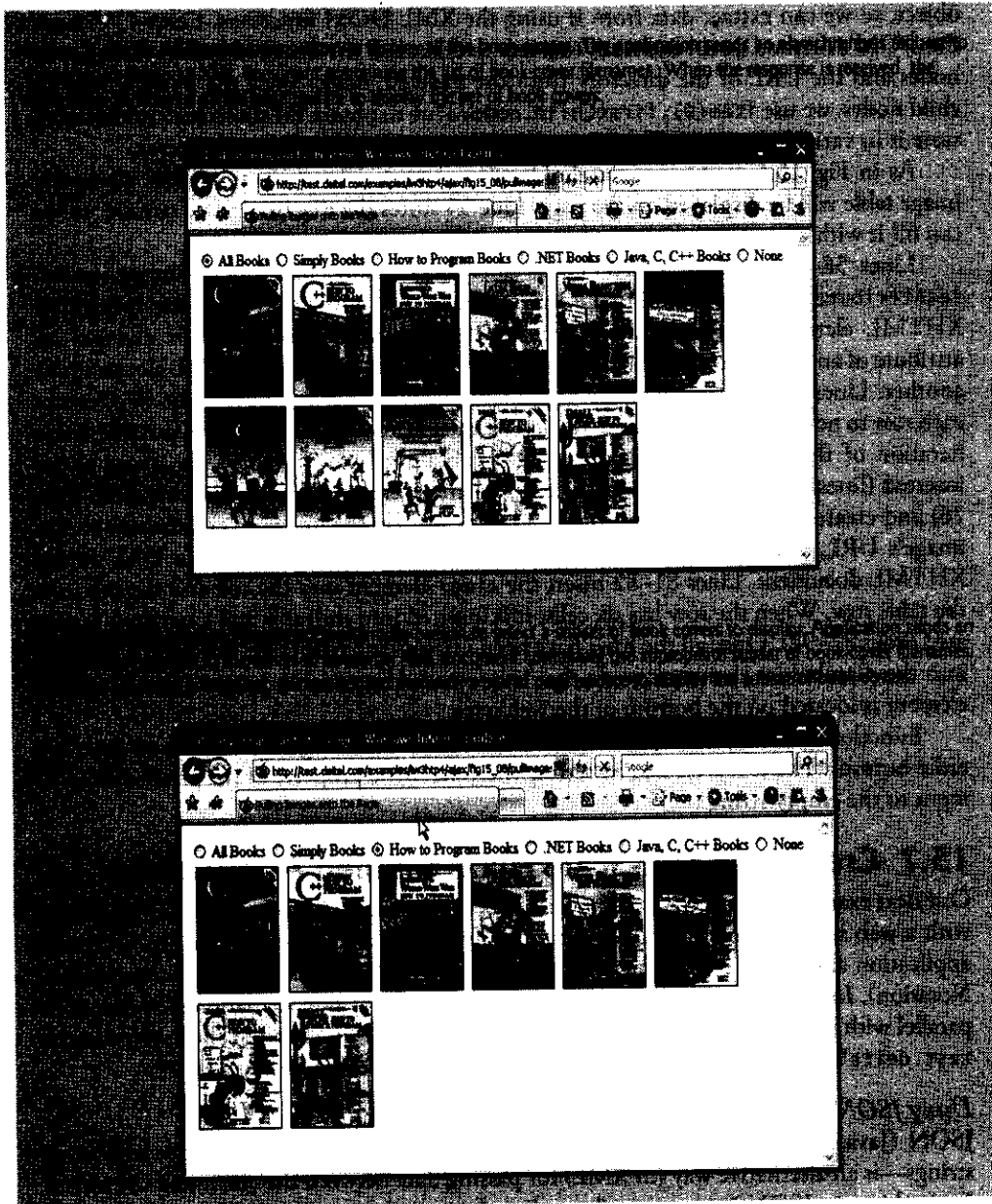


Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 4 of 4.)

When the `XMLHttpRequest` object receives the response, it invokes the callback function `processResponse` (lines 38–99). We use `XMLHttpRequest` object's `responseXML` property to access the XML returned by the server. Lines 41–42 check that the request was successful, and that the `responseXML` property is not empty. The XML file that we requested includes a `baseUrl` node that contains the address of the image directory and a collection of cover nodes that contain image filenames. `responseXML` is a document

object, so we can extract data from it using the XML DOM functions. Lines 47–52 use the DOM's method `getElementsByTagName` to extract all the image filenames from cover nodes and the URL of the directory from the `baseURL` node. Since the `baseURL` has no child nodes, we use `item(0).firstChild.nodeValue` to obtain the directory's address and store it in variable `baseURL`. The image filenames are stored in the `covers` array.

As in Fig. 15.5 we have a placeholder `div` element (line 126) to specify where the image table will be displayed on the page. Line 55 stores the `div` in variable `output`, so we can fill it with content later in the program.

Lines 58–93 generate an XHTML table dynamically, using the `createElement`, `setAttribute` and `appendChild` DOM methods. Method `createElement` creates an XHTML element of the specified type. Method `setAttribute` adds or changes an attribute of an XHTML element. Method `appendChild` inserts one XHTML element into another. Lines 58 and 61 create the `table` and `tbody` elements, respectively. We restrict each row to no more than six images, which we track with variable `rowCount` variable. Each iteration of the `for` statement (lines 67–93) obtains the filename of the image to be inserted (lines 69–73), creates a table cell element where the image will be inserted (line 76) and creates an `` element (line 77). Line 80 sets the image's `src` attribute to the image's URL, which we build by concatenating the filename to the base URL of the XHTML document. Lines 81–82 insert the `` element into the cell and the cell into the table row. When the row has six cells, it is inserted into the table and a new row is created (lines 87–92). Once all the rows have been inserted into the table, the table is inserted into the placeholder element `covers` that is referenced by variable `output` (line 97). This element is located on the bottom of the web page.

Function `clearTable` (lines 102–105) is called to clear images when the user switches radio buttons. The text is cleared by setting the `innerHTML` property of the placeholder element to the empty string.

15.7 Creating a Full-Scale Ajax-Enabled Application

Our next example demonstrates additional Ajax capabilities. The web application interacts with a web service to obtain data and to modify data in a server-side database. The web application and server communicate with a data format called JSON (JavaScript Object Notation). In addition, the application demonstrates server-side validation that occurs in parallel with the user interacting with the web application. You can test the application at test.deitel.com/examples/iw3http4/ajax/fig15_09_10/AddressBook.html.

Using JSON

JSON (JavaScript Object Notation)—a simple way to represent JavaScript objects as strings—is an alternative way (to XML) for passing data between the client and the server. Each object in JSON is represented as a list of property names and values contained in curly braces, in the following format:

```
{ "propertyName1" : value1, "propertyName2": value2 }
```

Arrays are represented in JSON with square brackets in the following format:

```
[ value1, value2, value3 ]
```

Each value can be a string, a number, a JSON representation of an object, `true`, `false` or `null`. You can convert JSON strings into JavaScript objects with JavaScript's `eval`

function. To evaluate a JSON string properly, a left parenthesis should be placed at the beginning of the string and a right parenthesis at the end of the string before the string is passed to the `eval` function.

The `eval` function creates a potential security risk—it executes any embedded JavaScript code in its string argument, possibly allowing a harmful script to be injected into JSON. A more secure way to process JSON is to use a JSON parser. In our examples, we use the open source parser from www.json.org/js.html. When you download its JavaScript file, place it in the same folder as your application. Then, link the `json.js` file into your XHTML file with the following statement in the head section:

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

You can now call function `parseJSON` on a JSON string to convert it to a JavaScript object.

JSON strings are easier to create and parse than XML, and require fewer bytes. For these reasons, JSON is commonly used to communicate in client/server interaction. For more information on JSON, visit our JSON Resource Center at www.deitel.com/json.

Rich Functionality

The previous examples in this chapter requested data from static files on the server. The example in Fig. 15.9 is an address-book application that communicates with a server-side application. The application uses server-side processing to give the page the functionality and usability of a desktop application. We use JSON to encode server-side responses and to create objects on the fly.

Initially the address book loads a list of entries, each containing a first and last name (Fig. 15.9(a)). Each time the user clicks a name, the address book uses Ajax functionality to load the person's address from the server and expand the entry *without reloading the page* (Fig. 15.9(b))—and it does this *in parallel* with allowing the user to click other names. The application allows the user to search the address book by typing a last name. As the user enters each keystroke, the application asynchronously displays the list of names in which the last name starts with the characters the user has entered so far (Fig. 15.9(c), Fig. 15.9(d) and Fig. 15.9(e))—a popular feature called *type ahead*.

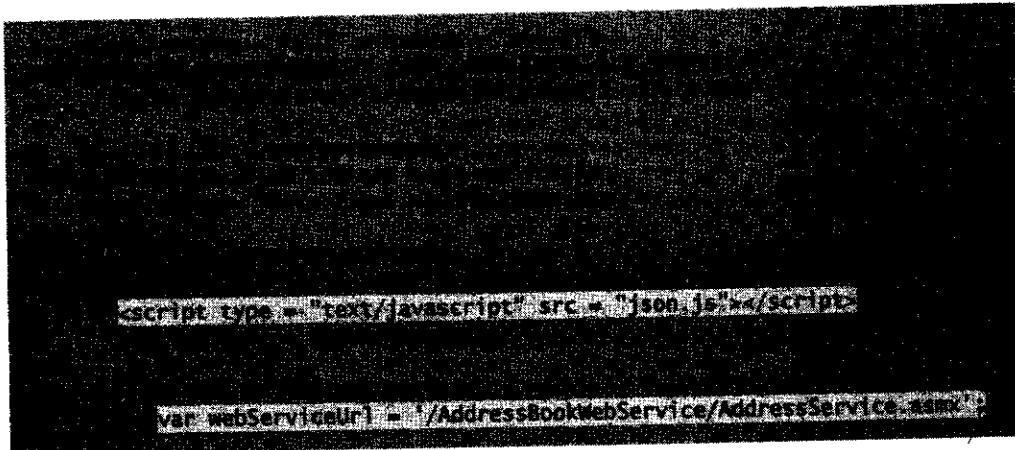


Fig. 15.9 | Ajax-enabled address-book application. (Part 1 of 10.)

```

var isLoading = false; // indicates if the response is loading
var isLoading = false; // indicates if the response is loading
// get a list of names from the service and display them
function showNames() {
    // hide the "address" form and show the address list
    document.getElementById( "address" ).style.display = "none";
    document.getElementById( "names" ).style.display = "block";

    var params = "[ ]"; // create an empty object
    callWebService( 'getAllNames', params, parseData );
}

// send the asynchronous request to the web service
function callWebService( method, params ) {
    // build request URI string
    var requestUrl = webServiceUrl + "/" + method;
    var params = paramString.parseJSON();
    // build the parameter string to add to the url
    for ( var i = 0; i < params.length; i++ )
    {
        // checks whether it is the first parameter and builds
        // the parameter string accordingly
        if ( i == 0 )
            requestUrl = requestUrl + "?" + params[ i ].param +
                "=" + params[ i ].value; // add first parameter to url
        else
            requestUrl = requestUrl + "&" + params[ i ].param +
                "=" + params[ i ].value; // add other parameters to url
    } // end for

    // attempt to send the asynchronous request

    asyncRequest.onreadystatechange = function()
    {
        callback( asyncRequest );
    }; // end anonymous function

    asyncRequest.setRequestHeader( "Accept",
        "application/json; charset=utf-8" );
}

```

Fig. 15.9 | Ajax-enabled address-book application. (Part 2 of 10.)

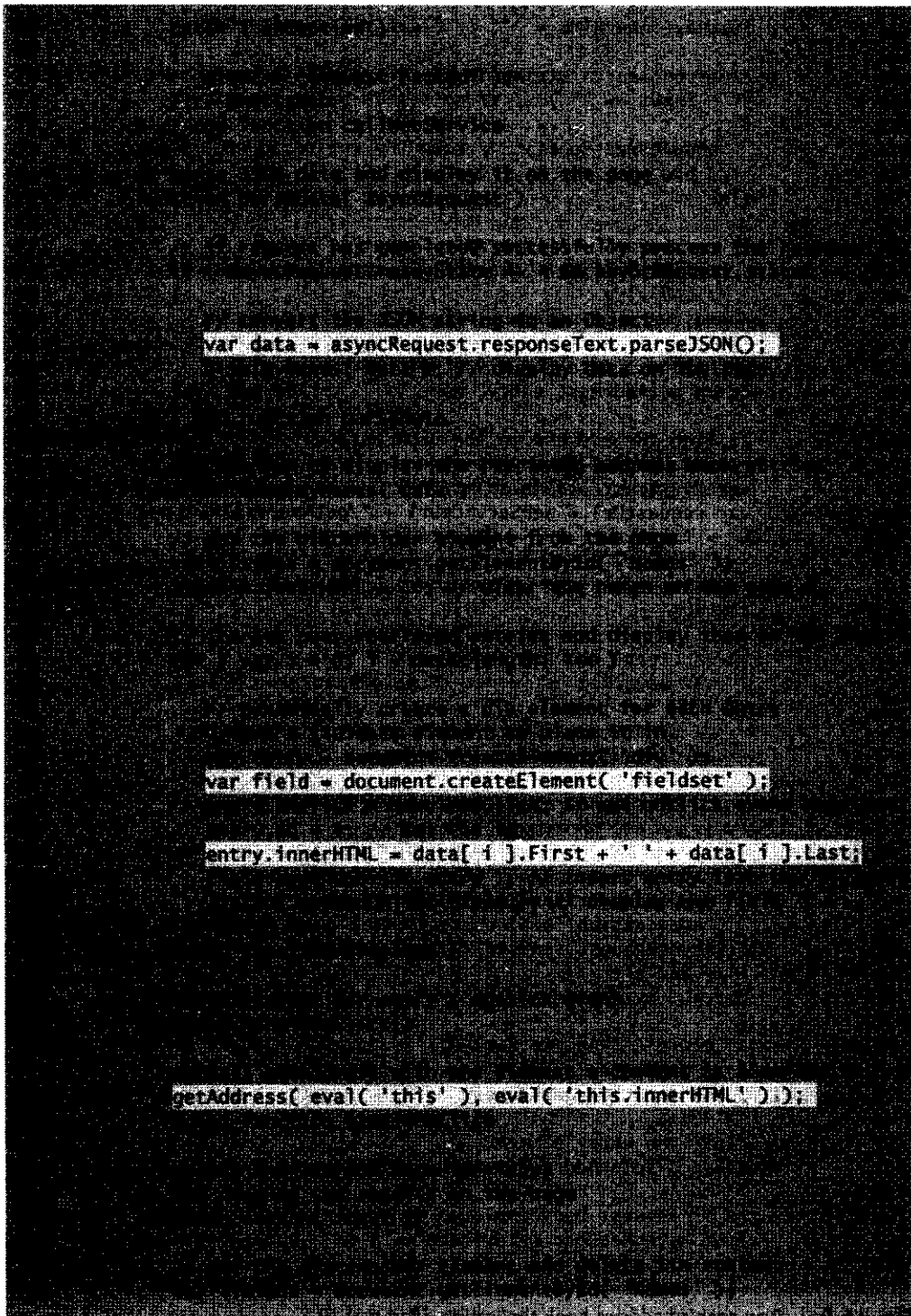


Fig. 15.9 | Ajax-enabled address-book application. (Part 3 of 10.)

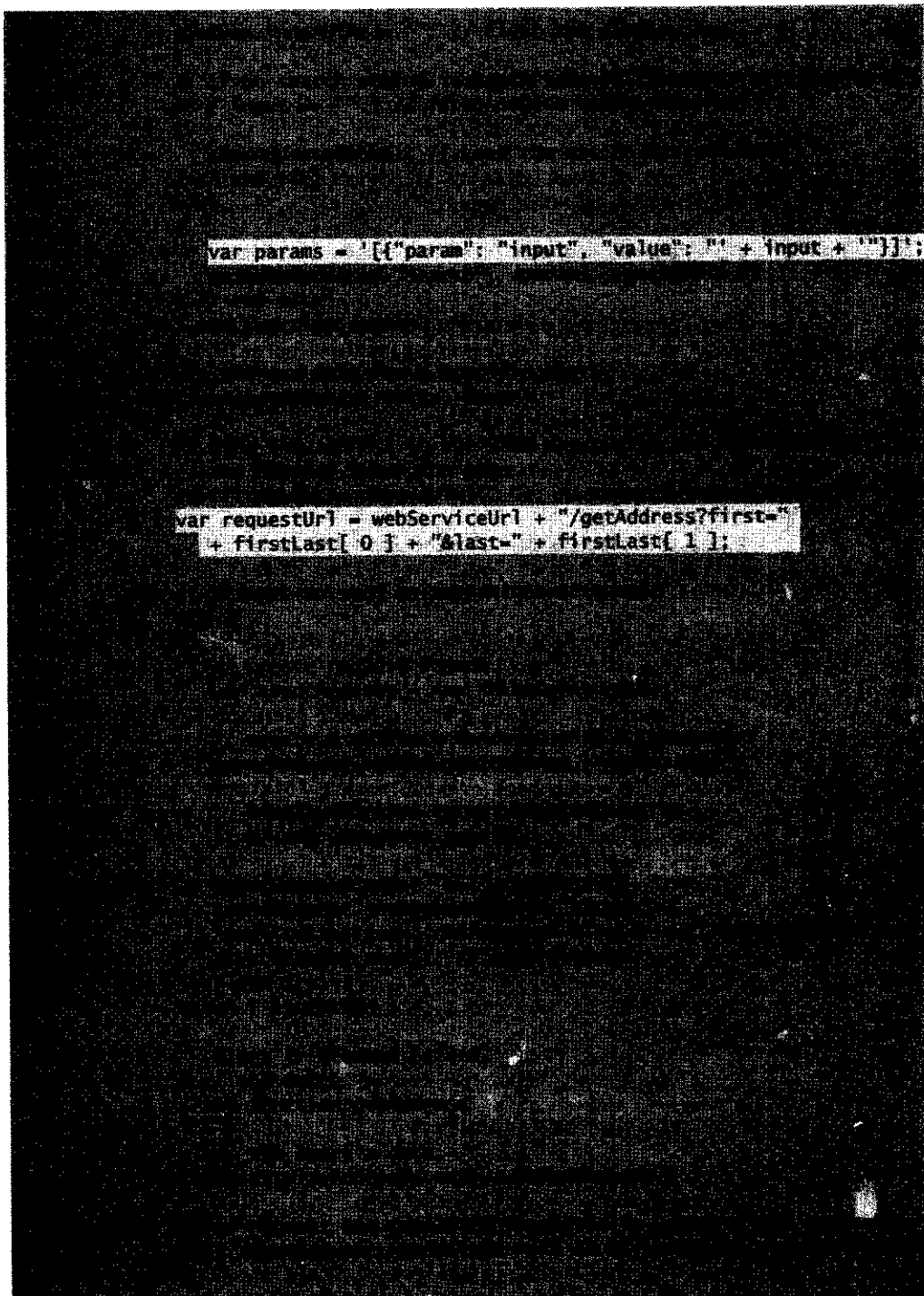


Fig. 15.9 | Ajax-enabled address-book application. (Part 4 of 10.)

```

197 // convert the JSON response to an object
198 var data = asyncRequest.responseText.parseJSON();
199 var name = entry.innerHTML // save the name string
200 entry.innerHTML = name + '<br/>' + data.Street +
201 '<br/>' + data.City + ', ' + data.State
202 + ', ' + data.Zip + '<br/>' + data.Telephone;
203
204 // clicking on the entry reveals the address
205 entry.onclick = function() {
206     // retrieve address and display it
207     getAddress( entry, name );
208 }
209
210 // add function to reveal address
211 function getAddress( entry, name ) {
212     // set the entry to display only the name
213     entry.innerHTML = name; // set the entry to display only the name
214     entry.onclick = function() // set onclick event
215     {
216         getAddress( entry, name ); // retrieve address and display it
217     }; // end function
218 }
219
220 // add function to reveal address
221 function addEntry() {
222     document.getElementById( 'addressBook' ).style.display = 'block';
223     document.getElementById( 'addEntry' ).style.display = 'block';
224 } // end function addEntry
225
226 // add the zip code to be validated and to generate city and state
227 function validateZip() {
228     // build the parameters
229     var params = [{"param": "zip", "value": "" + zip + ""}];
230     callWebService( "validateZip", params, showCityState );
231 } // end function validateZip
232
233 // get city and state that were generated using the zip code
234 // and display them on the page
235 function showCityState( asyncRequest ) {
236     // display message while request is being processed
237     document.getElementById( 'validateZip' ).
238         innerHTML = "Checking zip...";
239
240     // if request was processed successfully, process the response
241     if ( asyncRequest.readyState == 4 ) {
242

```

Fig. 15.9 | Ajax-enabled address-book application. (Part 5 of 10.)

```
var data = asyncRequest.responseText.parseJSON();

else if ( asyncRequest.status == 500 )
{
    document.getElementById( 'validateZip' ).
        innerHTML = 'Zip validation service not available';

var params = '[{"param": "tel", "value": "' + phone + '"}]';
callWebService( "validateTel", params, showPhoneError );

var data = asyncRequest.responseText.parseJSON();
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 6 of 10.)


```

311
312 // call the web service to insert data into the database
313 callWebService( "addEntry", params, parseData );
314 } // end else
315 } // end function - saveForm
316 // ->
317 </script>
318 </head>
319 <body onload = "showAddressBook()" >
320 <div>
321 <input type = "button" value = "Address Book"
322 onclick = "showAddressBook()" />
323 <input type = "button" value = "Add an Entry"
324 onclick = "addEntry()" />
325 </div>
326 <div id = "addressbook" style = "display : block;" >
327 Search By Last Name:
328 <input onkeyup = "search( this.value )" />
329 <br/>
330 <div id = "Names" >
331 </div>
332 </div>
333 <div id = "addEntry" style = "display : none" >
334 First Name: <input id = "first" />
335 <br/>
336 Last Name: <input id = "last" />
337 <br/>
338 <strong> Address: </strong>
339 <br/>
340 Street: <input id = "street" />
341 <br/>
342 City: <span id = "city" class = "validator" /> </span>
343 <br/>
344 State: <span id = "state" class = "validator" /> </span>
345 <br/>
346 Zip: <input id = "zip" onblur = "validateZip( this.value )" />
347 <span id = "validateZip" class = "validator" />
348 </span>
349 <br/>
350 Telephone: <input id = "phone"
351 onblur = "validatePhone( this.value )" />
352 <span id = "validatePhone" class = "validator" />
353 </span>
354 <br/>
355 <input type = "button" value = "Submit"
356 onclick = "saveForm()" />
357 <br/>
358 <div id = "success" class = "validator" >
359 </div>
360 </div>
361 </body>
362 </html>

```

Fig. 15.9 | Ajax-enabled address-book application. (Part 8 of 10.)

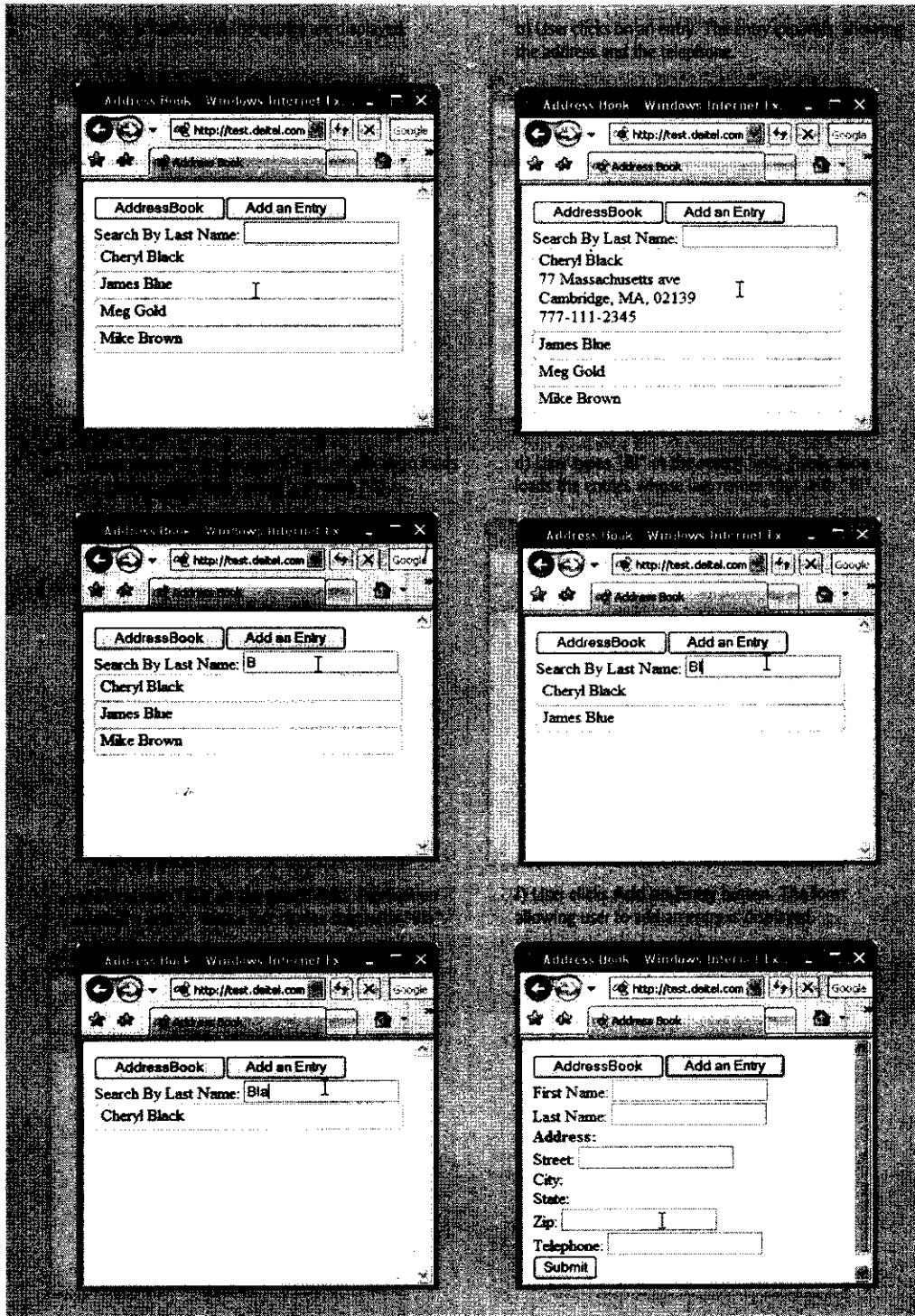


Fig. 15.9 | Ajax-enabled address-book application. (Part 9 of 10.)

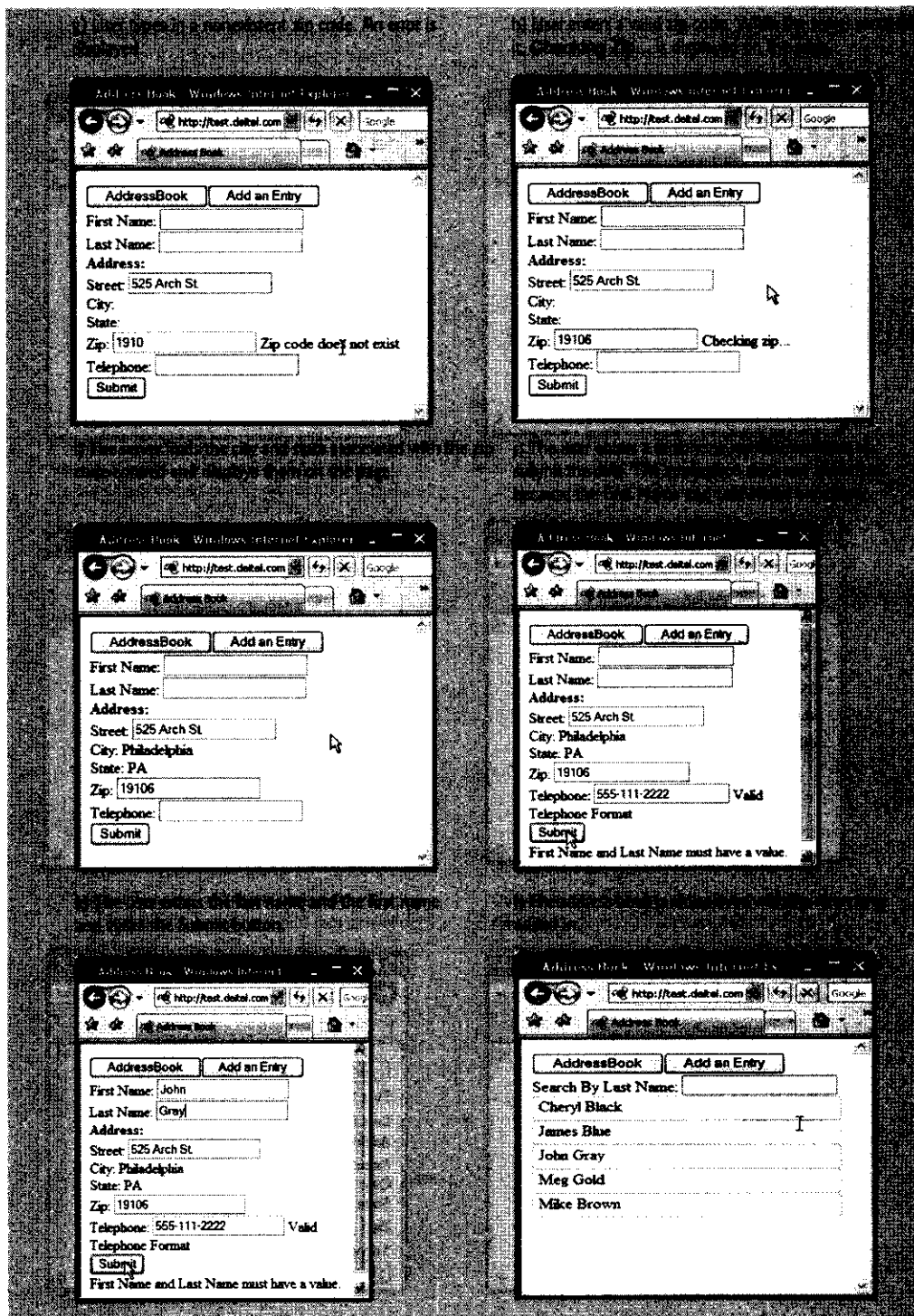


Fig. 15.9 | Ajax-enabled address-book application. (Part 10 of 10.)

The application also enables the user to add another entry to the address book by clicking the `addEntry` button (Fig. 15.9(f)). The application displays a form that enables live field validation. As the user fills out the form, the zip-code value is validated and used to generate the city and state (Fig. 15.9(g), Fig. 15.9(h) and Fig. 15.9(i)). The telephone number is validated for correct format (Fig. 15.9(j)). When the `Submit` button is clicked, the application checks for invalid data and stores the values in a database on the server (Fig. 15.9(k) and Fig. 15.9(l)). You can test-drive this application at test.deitel.com/examples/iw3http4/ajax/fig15_09_10/AddressBook.html.

Interacting with a Web Service on the Server

When the page loads, the `onload` event (line 339) calls the `showAddressBook` function to load the address book onto the page. Function `showAddressBook` (lines 21–29) shows the `addressBook` element and hides the `addEntry` element using the HTML DOM (lines 24–25). Then it calls function `callWebService` to make an asynchronous request to the server (line 28). Function `callWebService` requires an array of parameter objects to be sent to the server. In this case, the function we are invoking on the server requires no arguments, so line 27 creates an empty array to be passed to `callWebService`. Our program uses an ASP.NET web service that we created for this example to do the server-side processing. The web service contains a collection of methods that can be called from a web application.

Function `callWebService` (lines 32–72) contains the code to call our web service, given a method name, an array of parameter bindings (i.e., the method's parameter names and argument values) and the name of a callback function. The web-service application and the method that is being called are specified in the request URL (line 35). When sending the request using the GET method, the parameters are concatenated URL starting with a `?` symbol and followed by a list of `parameter=value` bindings, each separated by an `&`. Lines 39–49 iterate over the array of parameter bindings that was passed as an argument, and add them to the request URL. In this first call, we do not pass any parameters because the web method that returns all the entries requires none. However, future web method calls will send multiple parameter bindings to the web service. Lines 52–71 prepare and send the request, using similar functionality to the previous two examples. There are many types of user interaction in this application, each requiring a separate asynchronous request. For this reason, we pass the appropriate `asyncRequest` object as an argument to the function specified by the `callback` parameter. However, event handlers cannot receive arguments, so lines 57–60 assign an anonymous function to `asyncRequest`'s `onreadystatechange` property. When this anonymous function gets called, it calls function `callback` and passes the `asyncRequest` object as an argument. Lines 64–65 set an `Accept` request header to receive JSON formatted data.

Parsing JSON Data

Each of our web service's methods in this example returns a JSON representation of an object or array of objects. For example, when the web application requests the list of names in the address book, the list is returned as a JSON array, as shown in Fig. 15.10. Each object in Fig. 15.10 has the attributes `first` and `last`.

Line 11 links the `json.js` script to the XHTML file so we can parse JSON data. When the `XMLHttpRequest` object receives the response, it calls function `parseData` (lines 75–84). Line 81 calls the string's `parseJSON` function, which converts the JSON string into a JavaScript object. Then line 82 calls function `displayNames` (lines 87–106), which

```

1 [{"first": "Cheryl", "last": "Black"},
2  {"first": "James", "last": "Blue"},
3  {"first": "Mike", "last": "Brown"},
4  {"first": "Ray", "last": "Gold"}]

```

Fig. 15.10 | Address-book data formatted in JSON.

displays the first and last name of each address-book entry passed to it. Lines 90–91 use the DOM to store the placeholder div element `Names` in the variable `listbox`, and clear its content. Once parsed, the JSON string of address-book entries becomes an array, which this function traverses (lines 94–105).

Creating XHTML Elements and Setting Event Handlers on the Fly

Line 99 uses an XHTML `fieldset` element to create a box in which the entry will be placed. Line 100 registers function `handleOnClick` as the `onclick` event handler for the div created in line 98. This enables the user to expand each address-book entry by clicking it. Function `handleOnClick` (lines 109–113) calls the `getAddress` function whenever the user clicks an entry. The parameters are generated dynamically and not evaluated until the `getAddress` function is called. This enables each function to receive arguments that are specific to the entry the user clicked. Line 102 displays the names on the page by accessing the `first` (first name) and `last` (last name) fields of each element of the data array.

Function `getAddress` (lines 136–166) is called when the user clicks an entry. This request must keep track of the entry where the address is to be displayed on the page. Lines 151–154 set the `displayAddress` function (lines 168–187) as the callback function, and pass it the entry element as a parameter. Once the request completes successfully, lines 174–178 parse the response and display the addresses. Lines 181–184 update the div's `onclick` event handler to hide the address data when that div is clicked again by the user. When the user clicks an expanded entry, function `clearField` (lines 190–197) is called. Lines 192–196 reset the entry's content and its `onclick` event handler to the values they had before the entry was expanded.

Implementing Type-Ahead

The `input` element declared in line 348 enables the user to search the address book by last name. As soon as the user starts typing in the input box, the `onkeyup` event handler calls the `search` function (lines 117–133), passing the `input` element's value as an argument. The `search` function performs an asynchronous request to locate entries with last names that start with its argument value. When the response is received, the application displays the matching list of names. Each time the user changes the text in the input box, function `search` is called again to make another asynchronous request.

The `search` function (lines 117–133) first clears the address-book entries from the page (lines 120–121). If the `input` argument is the empty string, line 126 displays the entire address book by calling function `showAddressBook`. Otherwise lines 130–131 send a request to the server to search the data. Line 130 creates a JSON string to represent the parameter object to be sent as an argument to the `callWebServices` function. Line 131 converts the string to an object and calls the `callWebServices` function. When the server responds, callback function `parseData` is invoked, which calls function `displayNames` to display the results on the page.

Implementing a Form with Asynchronous Validation

When the **Add an Entry** button (lines 343–344) is clicked, the `addEntry` function (lines 200–204) is called, which hides the `addressBook` element and shows the `addEntry` element that allows the user to add a person to the address book. The `addEntry` element (lines 353–380) contains a set of entry fields, some of which have event handlers that enable validation that occurs asynchronously as the user continues to interact with the page. When a user enters a zip code, the `validateZip` function (lines 207–212) is called. This function calls an external web service to validate the zip code. If it is valid, that external web service returns the corresponding city and state. Line 210 builds a parameter object containing `validateZip`'s parameter name and argument value in JSON format. Line 211 calls the `callWebService` function with the appropriate method, the parameter object created in line 210 and `showCityState` (lines 216–258) as the callback function.

Zip-code validation can take a long time due to network delays. The `showCityState` function is called every time the request object's `readyState` property changes. Until the request completes, lines 219–220 display "Checking zip code..." on the page. After the request completes, line 228 converts the JSON response text to an object. The response object has four properties—`validity`, `errorMessage`, `city` and `state`. If the request is valid, line 233 updates the `zipValid` variable that keeps track of zip-code validity (declared at line 18), and lines 237–239 show the city and state that the server generated using the zip code. Otherwise lines 243–245 update the `zipValid` variable and show the error code. Lines 248–249 clear the city and state elements. If our web service fails to connect to the zip-code validator web service, lines 252–256 display an appropriate error message.

Similarly, when the user enters the telephone number, the function `validatePhone` (lines 261–265) sends the phone number to the server. Once the server responds, the `showPhoneError` function (lines 268–288) updates the `validatePhone` variable (declared at line 17) and shows the message that the web service returned.

When the **Submit** button is clicked, the `saveForm` function is called (lines 291–335). Lines 294–300 retrieve the data from the form. Lines 303–308 check if the zip code and telephone number are valid, and display the appropriate error message in the `Success` element on the bottom of the page. Before the data can be entered into a database on the server, both the first-name and last-name fields must have a value. Lines 309–314 check that these fields are not empty and, if they are empty, display the appropriate error message. Once all the data entered is valid, lines 318–321 hide the entry form and show the address book. Lines 324–333 build the parameter object using JSON and send the data to the server using the `callWebService` function. Once the server saves the data, it queries the database for an updated list of entries and returns them; then function `parseData` displays the entries on the page.

15.8 Dojo Toolkit

Developing web applications in general, and Ajax applications in particular, involves a certain amount of painstaking and tedious work. Cross-browser compatibility, DOM manipulation and event handling can get cumbersome, particularly as an application's size increases. Dojo is a free, open source JavaScript library that takes care of these issues. Dojo reduces asynchronous request handling to a single function call. Dojo also provides cross-browser DOM functions that simplify partial page updates. It covers many more areas of web development, from simple event handling to fully functional rich GUI controls.

To install Dojo, download the Dojo version 0.4.3 from www.dojotoolkit.org/downloads to your hard drive. Extract the files from the archive file you downloaded to your web development directory or web server. Including the `dojo.js` script file in your web application will give you access to all the Dojo functions. To do this, place the following script in the head element of your XHTML document:

```
<script type = "text/javascript" src = "path/Dojo.js">
```

where *path* is the relative or complete path to the Dojo toolkit's files. Quick installation instructions for Dojo are provided at Dojotoolkit.org/book/Dojo-book-0-9/part-1-life-Dojo/quick-installation.

Figure 15.11 is a calendar application that uses Dojo to create the user interface, communicate with the server asynchronously, handle events and manipulate the DOM. The application contains a calendar control that shows the user six weeks of dates (see the screen captures in Fig. 15.11). Various arrow buttons allow the user to traverse the calendar. When the user selects a date, an asynchronous request obtains from the server a list of the scheduled events for that date. There is an **Edit** button next to each scheduled event. When the **Edit** button is clicked, the item is replaced by a text box with the item's content, a **Save** button and a **Cancel** button. When the user presses **Save**, an asynchronous request saves the new value to the server and displays it on the page. This feature, often referred to as **edit-in-place**, is common in Ajax applications. You can test-drive this application at test.deitel.com/examples/iw3http4/ajax/fig15_11/calendar.html.

```

1 <?xml version = "1.0" encoding = "utf-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN
3 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
4
5 <!-- Fig. 15.11 Calendar.html -->
6 <!-- Calendar application built with Dojo -->
7 <xhtml:xmlns = "http://www.w3.org/2000/xhtml"
8
9 <head>
10 <script type = "text/javascript" src = "/dojo043/dojo.js"></script>
11 <script type = "text/javascript" src = "calendar.js"></script>
12 </head>
13
14 // specify all the required dojo scripts
15 dojo.require( "dojo.event.*" ); // use scripts from event package
16 dojo.require( "dojo.widget.*" ); // use scripts from widget package
17 dojo.require( "dojo.dom.*" ); // use scripts from dom package
18 dojo.require( "dojo.io.*" ); // use scripts from the io package
19
20 // configure calendar event handler
21 function connectEventHandler()
22 {
23     var calendar = dojo.widget.byId( "calendar" ); // get calendar
24     calendar.setDate( "2007-07-04" );
25     dojo.event.connect(
26         calendar, "onValueChanged", "retrieveItems" );
27 } // end function connectEventHandler

```

Fig. 15.11 | Calendar application built with Dojo. (Part 1 of 7.)

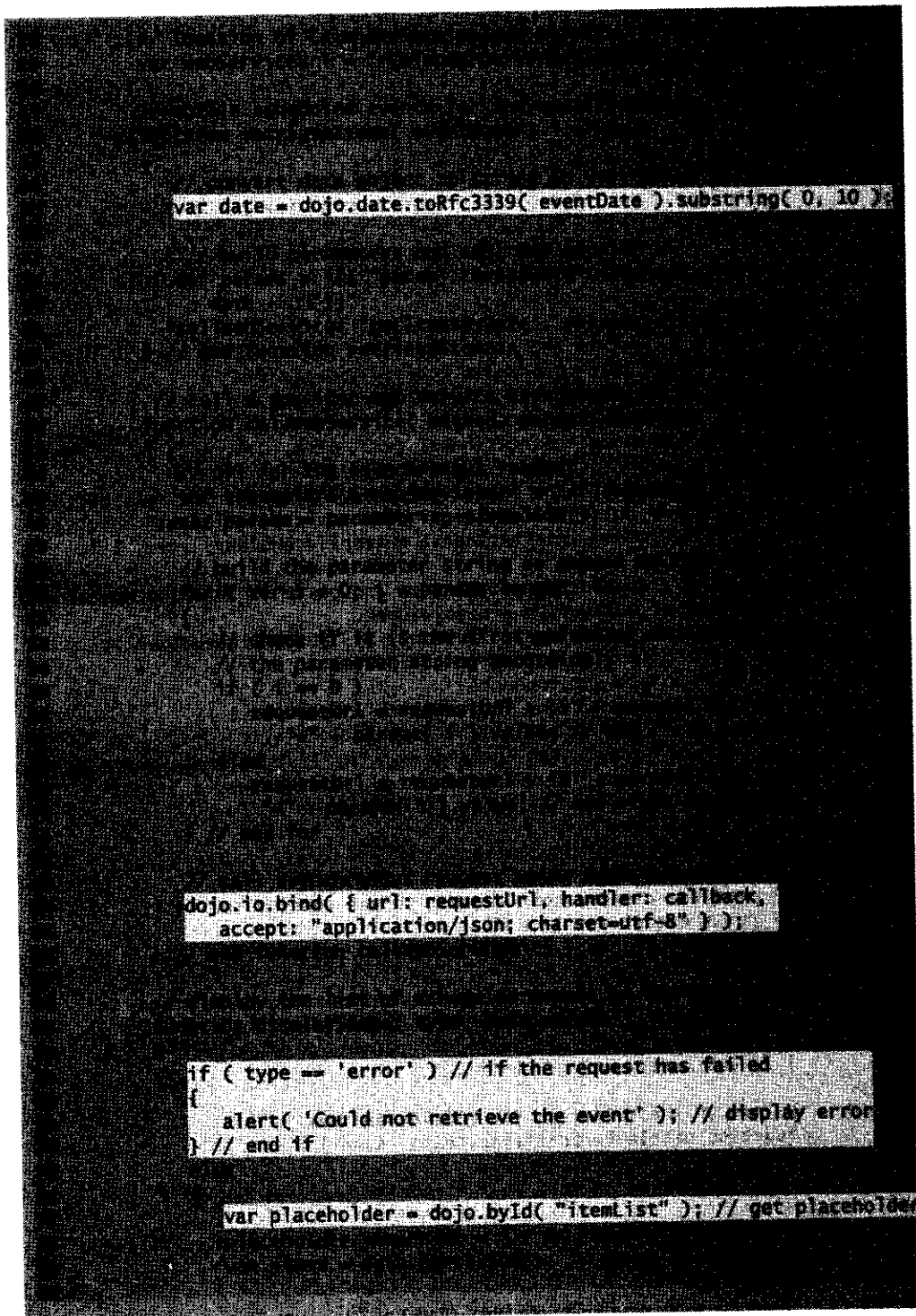


Fig. 15.11 | Calendar application built with Dojo. (Part 2 of 7.)

```
dojo.dom.insertAtIndex( text, item, 0 );

var buttonPlaceholder = document.createElement( "div" );
dojo.dom.insertAtIndex( buttonPlaceholder, item, 1 );

// create the edit button and paste it into the container
var editButton = dojo.widget.createWidget( "Button", {}, buttonPlaceholder );
editButton.setCaption( "Edit" );
dojo.event.connect( editButton, 'buttonClick', handleEdit );

dojo.dom.insertAtIndex( item, placeholder, 1 );

var id = event.currentTarget.parentNode.id; // retrieve id
```

Fig. 15.11 | Calendar application built with Dojo. (Part 3 of 7.)

```

133     else
134     {
135         var item = data.parseJSON($j("#calendar").data("calendar"));
136         var id = item.id; // calendar id
137         // create div element for calendar content
138         var calendar = document.createElement("div");
139         var buttonElement = document.createElement("div");
140
141         // hide the unfiltered content
142         var calendar = dojo.byId(id);
143         calendar.id = id + "calendar";
144         calendar.style.display = "none";
145         calendar.id = id; // calendar id
146
147         // create a section and header
148         var calendar = document.createElement("div");
149         calendar.id = id + "calendar";
150         dojo.dom.insertAt(calendar, "before", "calendar");
151
152         // create button element for calendar
153         // this will be the calendar id
154         var calendar = document.createElement("div");
155         var cancelButton = document.createElement("div");
156         dojo.dom.insertAt(calendar, "before", "calendar");
157         dojo.dom.insertAt(calendar, "before", "calendar");
158         dojo.dom.insertAt(calendar, "before", "calendar");
159
160         // create save and cancel buttons
161         var calendar = document.createElement("div");
162         dojo.dom.insertAt(calendar, "before", "calendar");
163         var cancelButton = document.createElement("div");
164         dojo.dom.insertAt(calendar, "before", "calendar");
165         dojo.dom.insertAt(calendar, "before", "calendar");
166         dojo.dom.insertAt(calendar, "before", "calendar");
167
168         // add an event listener to the calendar
169         dojo.on(calendar, "click", function(e) {
170             // do something with the calendar
171             // do something with the calendar
172             // do something with the calendar
173         });
174
175         // make the calendar visible on the page
176         dojo.dom.insertAt(calendar, "before", "calendar");
177     } // end else
178 } // end function parseJSON
179
180 // make the calendar visible on the page
181 function hideCalendar(id) {
182     // get the calendar id
183     var id = event.currentTarget.id;
184     var div = dojo.byId(id);
185     div.style.display = "none";
186 }

```

Fig. 15.11 | Calendar application built with Dojo. (Part 4 of 7.)

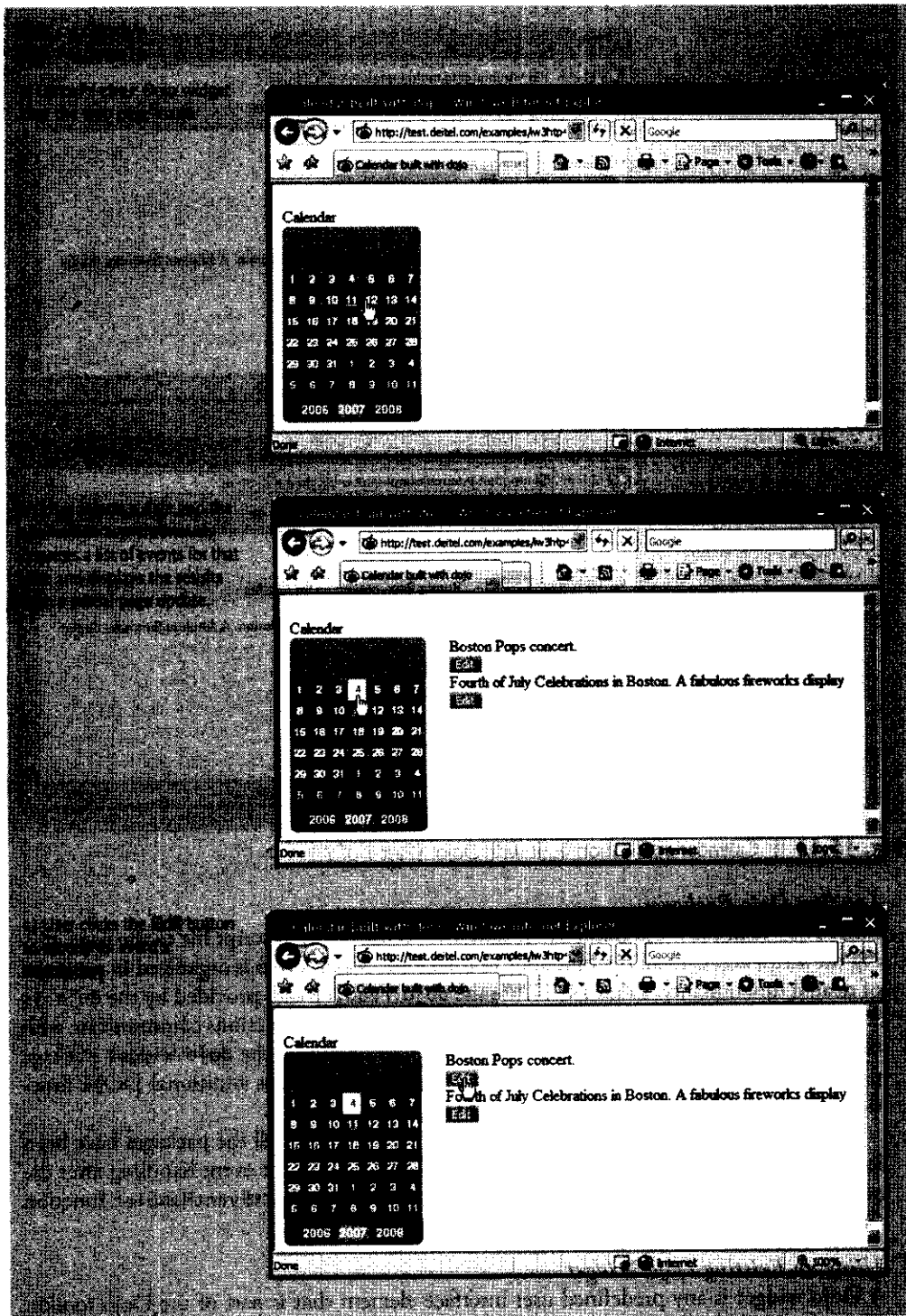


Fig. 15.11 | Calendar application built with Dojo. (Part 6 of 7.)

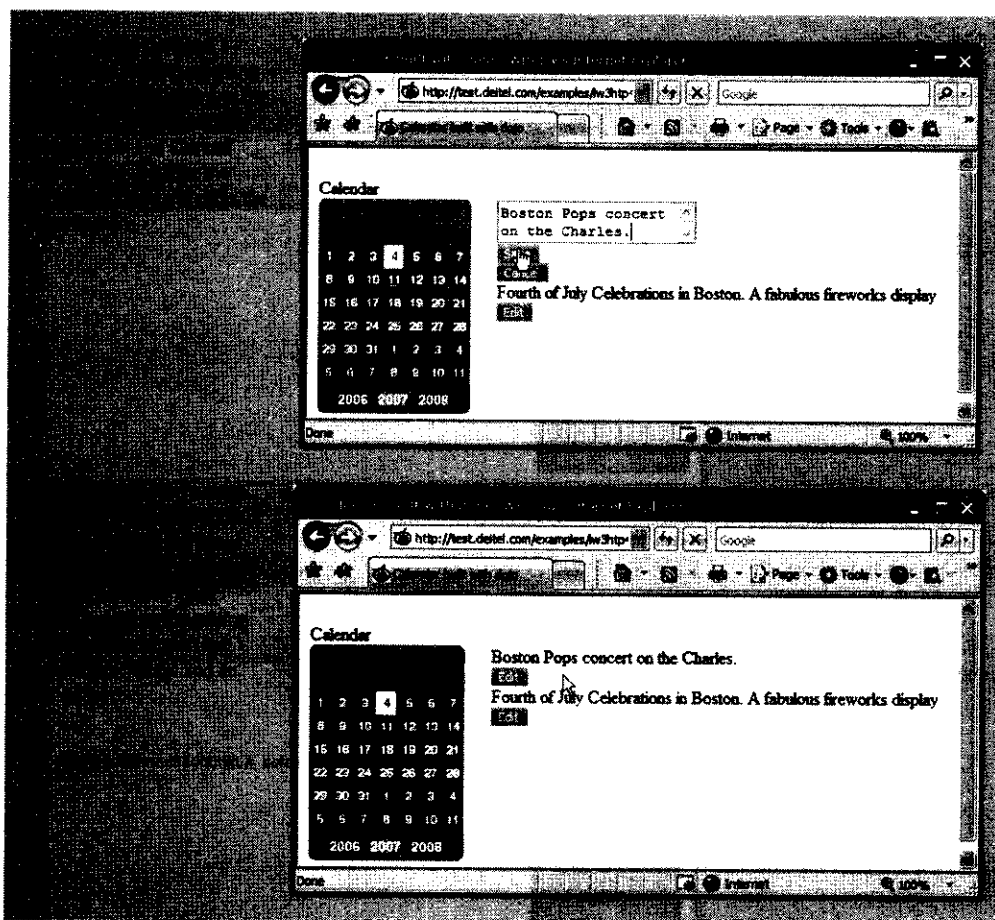


Fig. 15.11 | Calendar application built with Dojo. (Part 7 of 7.)

Loading Dojo Packages

Lines 9–17 load the Dojo framework. Line 9 links the `dojo.js` script file to the page, giving the script access to all the functions in the Dojo toolkit. Dojo is organized in packages of related functionality. Lines 14–17 use the `dojo.require` call, provided by the `dojo.js` script to include the packages we need. The `dojo.io` package functions communicate with the server, the `dojo.event` package simplifies event handling, the `dojo.widget` package provides rich GUI controls, and the `dojo.dom` package contains additional DOM functions that are portable across many different browsers.

The application cannot use any of this functionality until all the packages have been loaded. Line 229 uses the `dojo.addOnLoad` method to set up the event handling after the page loads. Once all the packages have been loaded, the `connectEventHandler` function (lines 20–26) is called.

Using an Existing Dojo Widget

A **Dojo widget** is any predefined user interface element that is part of the Dojo toolkit. The calendar control on the page is the `DatePicker` widget. To incorporate an existing

Dojo widget onto a page, you must set the `DojoType` attribute of any HTML element to the type of widget that you want it to be (line 236). Dojo widgets also have their own `widgetID` property (line 237). Line 22 uses the `dojo.widget.byId` method, rather than the DOM's `document.getElementById` method, to obtain the calendar widget element. The `dojo.events.connect` method links functions together. Lines 24–25 use it to connect the calendar's `onValueChanged` event handler to the `retrieveItems` function. When the user picks a date, a special `onValueChanged` event that is part of the `DatePicker` widget calls `retrieveItems`, passing the selected date as an argument. The `retrieveItems` function (lines 32–41) builds the parameters for the request to the server, and calls the `callWebService` function. Line 35 uses the `dojo.date.toRfc3339` method to convert the date passed by the calendar control to `yyyy-mm-dd` format.

Asynchronous Requests in Dojo

The `callWebService` function (lines 44–66) sends the asynchronous request to the specified web-service method. Lines 47–61 build the request URL using the same code as Fig. 15.9. Dojo reduces the asynchronous request to a single call to the `dojo.io.bind` method (lines 64–65), which works on all the popular browsers such as Firefox, Internet Explorer, Opera, Mozilla and Safari. The method takes an array of parameters, formatted as a JavaScript object. The `url` parameter specifies the destination of the request, the `handler` parameter specifies the callback function, and the `mimetype` parameter specifies the format of the response. The `handler` parameter can be replaced by the `load` and `error` parameters. The function passed as `load` handles successful requests and the function passed as `error` handles unsuccessful requests.

Response handling is done differently in Dojo. Rather than calling the callback function every time the request's `readyState` property changes, Dojo calls the function passed as the “handler” parameter when the request completes. In addition, in Dojo the script does not have access to the request object. All the response data is sent directly to the callback function. The function sent as the `handler` argument must have three parameters—`type`, `data` and `event`.

In the first request, the function `displayItems` (lines 69–115) is set as the callback function. Lines 71–74 check if the request is successful, and display an error message if it isn't. Lines 77–78 obtain the place-holder element (`itemList`), where the items will be displayed, and clear its content. Line 79 converts the JSON response text to a JavaScript object, using the same code as the example in Fig. 15.9.

Partial Page Updates Using Dojo's Cross-Browser DOM Manipulation Capabilities

The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser portable manner. Lines 83–86 check if the server-side returned any items, and display an appropriate message if it didn't. For each item object returned from the server, lines 91–92 create a `div` element and set its `id` to the item's `id` in the database. Lines 95–97 create a container element for the item's description. Line 98 uses Dojo's `dojo.dom.insertAtIndex` method to insert the description element as the first element in the item's element.

For each entry, the application creates an **Edit** button that enables the user to edit the event's content on the page. Lines 101–109 create a Dojo Button widget programmatically. Lines 101–102 create a `buttonPlaceHolder` `div` element for the button and paste it on the page. Lines 105–106 convert the `buttonPlaceHolder` element to a Dojo Button

widget by calling the `dojo.widget.createWidget` function. This function takes three parameters—the type of widget to be created, a list of additional widget parameters and the element which is to be converted to a Dojo widget. Line 107 uses the button's `setCaption` method to set the text that appears on the button. Line 112 uses the `insertAtIndex` method to insert the items into the `itemList` placeholder, in the order in which they were returned from the server.

Adding Edit-In-Place Functionality

Dojo Button widgets use their own `buttonClick` event instead of the DOM `onclick` event to store the event handler. Lines 108–109 use the `dojo.event.connect` method to connect the `buttonClick` event of the Dojo Button widget and the `handleEdit` event handler (lines 119–124). When the user clicks the **Edit** button, the Event object gets passed to the event handler as an argument. The Event object's `currentTarget` property contains the element that initiated the event. Line 121 uses the `currentTarget` property to obtain the `id` of the item. This `id` is the same as the item's `id` in the server database. Line 123 calls the web service's `getItemById` method, using the `callWebService` function to obtain the item that needs to be edited.

Once the server responds, the function `displayForEdit` (lines 127–178) replaces the item on the screen with the user interface used for editing the item's content. The code for this is similar to the code in the `displayItems` function. Lines 129–132 make sure the request was successful and parse the data from the server. Lines 139–140 create the container elements into which we insert the new user-interface elements. Lines 143–146 hide the element that displays the item and change its `id`. Now the `id` of the user-interface element is the same as the `id` of the item that it's editing stored in the database. Lines 149–152 create the text-box element that will be used to edit the item's description, paste it into the text box, and paste the resulting text box on the page. Lines 156–173 use the same syntax that was used to create the **Edit** button widget to create **Save** and **Cancel** button widgets. Line 176 pastes the resulting element, containing the text box and two buttons, on the page.

When the user edits the content and clicks the **Cancel** button, the `handleCancel` function (lines 194–202) restores the item element to what it looked like before the button was clicked. Line 198 deletes the edit UI that was created earlier, using Dojo's `removeNode` function. Lines 200–201 show the item with the original element that was used to display the item, and change its `id` back to the item's `id` on the server database.

When the user clicks the **Save** button, the `handleSave` function (lines 181–191) sends the text entered by the user to the server. Line 185 obtains the text that the user entered in the text box. Lines 188–190 send to the server the `id` of the item that needs to be updated and the new description.

Once the server responds, `displayEdited` (lines 205–226) displays the new item on the page. Lines 214–217 contain the same code that was used in `handleCancel` to remove the user interface used to edit the item and redisplay the element that contains the item. Line 221 changes the item's description to its new value.

15.9 Web Resources

www.deitel.com/ajax

Our *Ajax Resource Center* contains links to some of the best Ajax resources on the web from which you can learn more about Ajax and its component technologies. Find categorized

links to Ajax tools, code, forums, books, libraries, frameworks, conferences, podcasts and more. Check out the tutorials for all skill levels, from introductory to advanced. See our comprehensive list of developer toolkits and libraries. Visit the most popular Ajax community websites and blogs. Explore many popular commercial and free open-source Ajax applications. Download code snippets and complete scripts that you can use on your own website. Also, be sure to visit our Resource Centers with information on Ajax's component technologies, including XHTML (www.deitel.com/xhtml/), CSS 2.1 (www.deitel.com/css21/), XML (www.deitel.com/XML/), and JavaScript (www.deitel.com/javascript/). For a complete list of Resource Centers, visit www.deitel.com/ResourceCenters.html.

Summary

Section 15.1 Introduction

- Despite the tremendous technological growth of the Internet over the past decade, the usability of web applications has lagged behind compared to desktop applications.
- Rich Internet Applications (RIAs) are web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and rich GUI.
- RIA performance comes from Ajax (Asynchronous JavaScript and XML), which uses client-side scripting to make web applications more responsive.
- Ajax applications separate client-side user interaction and server communication, and run them in parallel, making the delays of server-side processing more transparent to the user.
- “Raw” Ajax uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM.
- When writing “raw” Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications.
- Portability issues are hidden by Ajax toolkits, such as Dojo, Prototype and Script.aculo.us, which provide powerful ready-to-use controls and functions that enrich web applications and simplify JavaScript coding by making it cross-browser compatible.
- We achieve rich GUI in RIAs with Ajax toolkits and with RIA environments such as Adobe's Flex, Microsoft's Silverlight and JavaServer Faces. Such toolkits and environments provide powerful ready-to-use controls and functions that enrich web applications.
- The client-side of Ajax applications is written in XHTML and CSS, and uses JavaScript to add functionality to the user interface.
- XML and JSON are used to structure the data passed between the server and the client.
- The Ajax component that manages interaction with the server is usually implemented with JavaScript's XMLHttpRequest object—commonly abbreviated as XHR.

Section 15.2 Traditional Web Applications vs. Ajax Applications

- In traditional web applications, the user fills in the form's fields, then submits the form. The browser generates a request to the server, which receives the request and processes it. The server generates and sends a response containing the exact page that the browser will render, which causes the browser to load the new page and temporarily makes the browser window blank. The client waits for the server to respond and reloads the entire page with the data from the response.

- While a synchronous request is being processed on the server, the user cannot interact with the client web browser.
- The synchronous model was originally designed for a web of hypertext documents—what some people call the “brochure web.” This model yielded “choppy” application performance.
- In an Ajax application, when the user interacts with a page, the client creates an XMLHttpRequest object to manage a request. The XMLHttpRequest object sends the request to and awaits the response from the server. The requests are asynchronous, allowing the user to continue interacting with the application while the server processes the request concurrently. When the server responds, the XMLHttpRequest object that issued the request invokes a callback function, which typically uses partial page updates to display the returned data in the existing web page *without reloading the entire page*.
- The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications.

Section 15.3 Rich Internet Applications (RIAs) with Ajax

- A classic XHTML registration form sends all of the data to be validated to the server when the user clicks the Register button. While the server is validating the data, the user cannot interact with the page. The server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser. Once the user fixes the errors and clicks the Register button, the cycle repeats until no errors are found, then the data is stored on the server. The entire page reloads every time the user submits invalid data.
- Ajax-enabled forms are more interactive. Entries are validated dynamically as the user enters data into the fields. If a problem is found, the server sends an error message that is asynchronously displayed to inform the user of the problem. Sending each entry asynchronously allows the user to address invalid entries quickly, rather than making edits and resubmitting the entire form repeatedly until all entries are valid. Asynchronous requests could also be used to fill some fields based on previous fields' values.

Section 15.4 History of Ajax

- The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client.
- All of the technologies involved in Ajax (XHTML, JavaScript, CSS, dynamic HTML, the DOM and XML) have existed for many years.
- In 1998, Microsoft introduced the XMLHttpRequest object to create and manage asynchronous requests and responses.
- Popular applications like Flickr, Google's Gmail and Google Maps use the XMLHttpRequest object to update pages dynamically.
- The name Ajax immediately caught on and brought attention to its component technologies. Ajax has quickly become one of the hottest technologies in web development, as it enables web-top applications to challenge the dominance of established desktop applications.

Section 15.5 “Raw” Ajax Example using the XMLHttpRequest Object

- The XMLHttpRequest object (which resides on the client) is the layer between the client and the server that manages asynchronous requests in Ajax applications. This object is supported on most browsers, though they may implement it differently.
- To initiate an asynchronous request, you create an instance of the XMLHttpRequest object, then use its open method to set up the request, and its send method to initiate the request.

- When an Ajax application requests a file from a server, the browser typically caches that file. Subsequent requests for the same file can load it from the browser's cache.
- For security purposes, the XMLHttpRequest object does not allow a web application to request resources from servers other than the one that served the web application.
- Making a request to a different server is known as cross-site scripting (also known as XSS). You can implement a server-side proxy—an application on the web application's web server—that can make requests to other servers on the web application's behalf.
- When the third argument to XMLHttpRequest method open is true, the request is asynchronous.
- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered.
- A try block encloses code that might cause an exception and code that should not execute if an exception occurs. A try block consists of the keyword try followed by a block of code enclosed in curly braces ({}).
- When an exception occurs, a try block terminates immediately and a catch block (also called a catch clause or exception handler) catches (i.e., receives) and handles an exception.
- The catch block begins with the keyword catch and is followed by an exception parameter in parentheses and a block of code enclosed in curly braces.
- The exception parameter's name enables the catch block to interact with a caught exception object, which contains name and message properties.
- A callback function is registered as the event handler for the XMLHttpRequest object's onreadystatechange event. Whenever the request makes progress, the XMLHttpRequest calls the onreadystatechange event handler.
- Progress is monitored by the readyState property, which has a value from 0 to 4. The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete.

Section 15.6 Using XML and the DOM

- When passing structured data between the server and the client, Ajax applications often use XML because it consumes little bandwidth and is easy to parse.
- When the XMLHttpRequest object receives XML data, the XMLHttpRequest object parses and stores the data as a DOM object in the responseXML property.
- The XMLHttpRequest object's responseXML property contains the XML returned by the server.
- DOM method createElement creates an XHTML element of the specified type.
- DOM method setAttribute adds or changes an attribute of an XHTML element.
- DOM method appendChild inserts one XHTML element into another.
- The innerHTML property of a DOM element can be used to obtain or change the XHTML that is displayed in a particular element.

Section 15.7 Creating a Full-Scale Ajax-Enabled Application

- JSON (JavaScript Object Notation)—a simple way to represent JavaScript objects as strings—is an alternative way (to XML) for passing data between the client and the server.
- Each JSON object is represented as a list of property names and values contained in curly braces.
- An array is represented in JSON with square brackets containing a comma-separated list of values.

- Each value in a JSON array can be a string, a number, a JSON representation of an object, true, false or null.
- JavaScript's `eval` function can convert JSON strings into JavaScript objects. To evaluate a JSON string properly, a left parenthesis should be placed at the beginning of the string and a right parenthesis at the end of the string before the string is passed to the `eval` function.
- The `eval` function creates a potential security risk—it executes any embedded JavaScript code in its string argument, possibly allowing a harmful script to be injected into JSON. A more secure way to process JSON is to use a JSON parser.
- JSON strings are easier to create and parse than XML and require fewer bytes. For these reasons, JSON is commonly used to communicate in client/server interaction.
- When a request is sent using the GET method, the parameters are concatenated on the URL. URL parameter strings start with a ? symbol and have a list of *parameter-value* bindings, each separated by an &.
- To implement type-ahead, you can use an element's `onkeyup` event handler to make synchronous requests.

Section 15.8 Dojo Toolkit

- Developing web applications in general, and Ajax applications in particular, involves a certain amount of painstaking and tedious work. Cross-browser compatibility, DOM manipulation and event handling can get cumbersome, particularly as an application's size increases. Dojo is a free, open source JavaScript library that takes care of these issues.
- Dojo reduces asynchronous request handling to a single function call.
- Dojo provides cross-browser DOM functions that simplify partial page updates. It also provides event handling and rich GUI controls.
- To install Dojo, download the latest release from www.dojotoolkit.org/downloads to your hard drive. Extract the files from the archive file you downloaded to your web development directory or web server. To include the `Dojo.js` script file in your web application, place the following script in the head element of your XHTML document:


```
<script type = "text/javascript" src = "path/Dojo.js">
```

 where *path* is the relative or complete path to the Dojo toolkit's files.
- Edit-in-place enables a user to modify data directly in the web page, a common feature in Ajax applications.
- Dojo is organized in packages of related functionality.
- The `dojo.require` method is used to include specific Dojo packages.
- The `dojo.io` package functions communicate with the server, the `dojo.event` package simplifies event handling, the `dojo.widget` package provides rich GUI controls, and the `dojo.dom` package contains additional DOM functions that are portable across many different browsers.
- A Dojo widget is any predefined user interface element that is part of the Dojo toolkit.
- To incorporate an existing Dojo widget onto a page, you must set the `dojoType` attribute of any HTML element to the type of widget that you want it to be.
- The `dojo.widget.byId` method can be used to obtain a Dojo widget.
- The `dojo.events.connect` method links functions together.
- The `dojo.date.toRFC3339` method converts a date to *yyyy-mm-dd* format.
- The `dojo.io.btrnd` method configures and sends asynchronous requests. The method takes an array of parameters, formatted as a JavaScript object. The `url` parameter specifies the destination.

of the request, the handler parameter specifies the callback function, and the datatype parameter specifies the format of the response. The handler parameter can be replaced by the load and error parameters. The function passed as the load handler processes successful requests and the function passed as the error handler processes unsuccessful requests.

- Dojo calls the function passed as the handler parameter only when the request completes.
- In Dojo, the script does not have access to the request object. All the response data is sent directly to the callback function.
- The function sent as the handler argument must have three parameters—type, data and event.
- The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser manner.
- Dojo's `dojo.dom.insertAtIndex` method inserts an element at the specified index in the DOM.
- Dojo's `removeNode` function removes an element from the DOM.
- Dojo button widgets use their own `buttonClick` event instead of the DOM `onClick` event to open the event handler.
- The Event object's `currentTarget` property contains the element that initiated the event.

Terminology

Ajax	partial page update
Ajax toolkit	Prototype Ajax library
asynchronous request	"raw" Ajax
callback function	<code>readyState</code> property of <code>XMLHttpRequest</code> object
catch block	<code>responseText</code> property of <code>XMLHttpRequest</code> object
catch clause	<code>responseXML</code> property of <code>XMLHttpRequest</code> object
catch keyword	same origin policy (SOP)
cross-browser compatibility	Script.aculo.us Ajax library
cross-site scripting (XSS)	<code>send</code> method of <code>XMLHttpRequest</code>
Dojo Ajax library	<code>setRequestHeader</code> method of <code>XMLHttpRequest</code> object
edit-in-place	<code>status</code> property of <code>XMLHttpRequest</code> object
exception	<code>statusText</code> property of <code>XMLHttpRequest</code> object
exception handler	synchronous request
exception handling	try block
GET method of <code>XMLHttpRequest</code> object	try keyword
<code>getResponseHeader</code> method of <code>XMLHttpRequest</code> object	type ahead
JavaScript Object Notation (JSON)	XHR (abbreviation for <code>XMLHttpRequest</code>)
<code>readyStateChange</code> property of the <code>XMLHttpRequest</code> object	<code>XMLHttpRequest</code> object
open method of <code>XMLHttpRequest</code>	

Self-Review Exercises

15.1 Fill in the blanks in each of the following statements:

- a) Ajax applications use _____ requests to create Rich Internet Applications.
- b) In Ajax applications, the _____ object manages asynchronous interaction with the server.
- c) The event handler called when the server responds is known as a(n) _____ function.

- d) The _____ attribute can be accessed through the DOM to update an XHTML element's content without reloading the page.
- e) JavaScript's XMLHttpRequest object is commonly abbreviated as _____.
- f) _____ is a simple way to represent JavaScript objects as strings.
- g) Making a request to a different server is known as _____.
- h) JavaScript's _____ function can convert JSON strings into JavaScript objects.
- i) A(n) _____ encloses code that might cause an exception and code that should not execute if an exception occurs.
- j) The XMLHttpRequest object's _____ contains the XML returned by the server.

15.2 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) Ajax applications must use XML for server responses.
- b) The technologies that are used to develop Ajax applications have existed since the 1990s.
- c) The event handler that processes the response is stored in the `readyState` property of XMLHttpRequest.
- d) An Ajax application can be implemented so that it never needs to reload the page on which it runs.
- e) The `responseXML` property of the XMLHttpRequest object stores the server's response as a raw XML string.
- f) The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser manner.
- g) An exception indicates successful completion of a program's execution.
- h) When the third argument to XMLHttpRequest method `open` is `false`, the request is asynchronous.
- i) For security purposes, the XMLHttpRequest object does not allow a web application to request resources from servers other than the one that served the web application.
- j) The `innerHTML` property of a DOM element can be used to obtain or change the XHTML that is displayed in a particular element.

Exercises

15.3 Consider the AddressBook application in Fig. 15.9. Describe how you could reimplement the type-ahead capability so that it could perform the search using data previously downloaded rather than making an asynchronous request to the server after every keystroke.

15.4 Describe each of the following terms in the context of Ajax:

- a) type-ahead
- b) edit-in-place
- c) partial page update
- d) asynchronous request
- e) XMLHttpRequest
- f) "raw" Ajax
- g) callback function
- h) same origin policy
- i) Ajax libraries
- j) RIA

[Note to Instructors and Students: Due to security restrictions on using XMLHttpRequest, Ajax applications must be placed on a web server (even one on your local computer) to enable the applications to work correctly, and when they need to access other resources, those must reside on the same web server. Students: You'll need to work closely with your instructors to understand your lab

setup so you can run your solutions to the exercises (the examples are already posted on our web server) and to run many of the other server-side applications that you'll learn later in the book.]

15.5 The XML files used in the book-cover catalog example (Fig. 15.8) also store the titles of the books in a `title` attribute of each cover node. Modify the example so that every time the mouse hovers over an image, the book's title is displayed below the image.

15.6 Create an Ajax-based product catalog that obtains its data from JSON files located on the server. The data should be separated into four JSON files. The first file should be a summary file, containing a list of products. Each product should have a title, an image filename for a thumbnail image and a price. The second file should contain a list of descriptions for each product. The third file should contain a list of filenames for the full-size product images. The last file should contain a list of the thumbnail image file names. Each item in a catalogue should have a unique ID that should be included with the entries for that product in every file. Next, create an Ajax-enabled web page that displays the product information in a table. The catalog should initially display a list of product names with their associated thumbnail images and prices. When the mouse hovers over a thumbnail image, the larger product image should be displayed. When the user moves the mouse away from that image, the original thumbnail should be redisplayed. You should provide a button that the user can click to display the product description.

15.7 Create a version of Exercise 15.6 that uses Dojo's capabilities and widgets to display the product catalog. Modify the asynchronous requests to use `dojo.io.bind` functions rather than raw Ajax. Use Dojo's DOM functionality to place elements on the page. Improve the look of the page by using Dojo's button widgets rather than XHTML button elements.

3

Rich Internet Application Client Technologies

The user should feel in control of the computer; not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.

—Inside Macintosh, Volume 1,
Apple Computer, Inc., 1985


16

Adobe Flash CS3

OBJECTIVES

In this chapter you will learn:

- Flash CS3 multimedia development.
- To develop Flash movies.
- Flash animation techniques.
- ActionScript 3.0, Flash's object-oriented programming language.
- To create a preloading animation for a Flash movie.
- To add sound to Flash movies.
- To publish a Flash movie.
- To create special effects with Flash.
- To create a Splash Screen.



*Science and technology and
the various forms of art, all
unite humanity in a single
and interconnected system.*

—Zhores Aleksandrovich
Medvede

*All the world's a stage, and
all the men and women
merely players; they have
their exits and their
entrances; and one man in
his time plays many parts. . .*

—William Shakespeare

*Music has charms to soothe a
savage breast,
To soften rocks, or bend a
knotted oak.*

—William Congreve

*A flash and where previously
the brain held a dead fact,
the soul grasps a living truth!
At moments we are all artists.*

—Arnold Bennett

- 16.1 Introduction
- 16.2 Flash Movie Development
- 16.3 Learning Flash with Hands-On Examples
 - 16.3.1 Creating a Shape with the Oval Tool
 - 16.3.2 Adding Text to a Button
 - 16.3.3 Converting a Shape into a Symbol
 - 16.3.4 Editing Button Symbols
 - 16.3.5 Adding Keyframes
 - 16.3.6 Adding Sound to a Button
 - 16.3.7 Verifying Changes with Test Movie
 - 16.3.8 Adding Layers to a Movie
 - 16.3.9 Animating Text with Tweening
 - 16.3.10 Adding a Text Field
 - 16.3.11 Adding ActionScript
- 16.4 Publishing Your Flash Movie
- 16.5 Creating Special Effects with Flash
 - 16.5.1 Importing and Manipulating Bitmaps
 - 16.5.2 Creating an Advertisement Banner with Masking
 - 16.5.3 Adding Online Help to Forms
- 16.6 Creating a Website Splash Screen
- 16.7 ActionScript
- 16.8 Web Resources

Summary | Terminology | Self-Review Exercises | Exercises

16.1 Introduction

Adobe Flash CS3 (Creative Suite 3) is a commercial application that you can use to produce interactive, animated movies. Flash can be used to create web-based banner advertisements, interactive websites, games and web-based applications with stunning graphics and multimedia effects. It provides tools for drawing graphics, generating animations, and adding sound and video. Flash movies can be embedded in web pages, distributed on CDs and DVDs as independent applications, or converted into stand-alone, executable programs. Flash includes tools for coding in its scripting language—ActionScript 3.0—which is similar to JavaScript and enables interactive applications. A fully functional, 30-day trial version of Flash CS3 is available for download from:

www.adobe.com/products/flash/

To follow along with the examples in this chapter, please install this software before continuing. Follow the on-screen instructions to install the trial version of the Flash software.

To play Flash movies, the Flash Player plug-in must be installed in your web browser. The most recent version of the plug-in (at the time of this writing) is version 9. You can download the latest version from:

www.adobe.com/go/getflashplayer

According to Adobe's statistics, approximately 98.7 percent of web users have Flash Player version 6 or greater installed, and 83.4 percent of web users have Flash Player version 9 installed.¹ There are ways to detect whether a user has the appropriate plug-in to view Flash content. Adobe provides a tool called the Flash Player Detection Kit which contains files that work together to detect whether a suitable version of Adobe Flash Player is installed in a user's web browser. This kit can be downloaded from:

www.adobe.com/products/flashplayer/download/detection_kit/

This chapter introduces building Flash movies. You'll create interactive buttons, add sound to movies, create special graphic effects and integrate ActionScript in movies.

16.2 Flash Movie Development

Once Flash CS3 is installed, open the program. Flash's **Welcome Screen** appears by default. The **Welcome Screen** contains options such as **Open a Recent Item**, **Create New** and **Create from Template**. The bottom of the page contains links to useful help topics and tutorials. [Note: For additional help, refer to Flash's **Help** menu.]

To create a blank Flash document, click **Flash File (ActionScript 3.0)** under the **Create New** heading. Flash opens a new file called **Untitled-1** in the Flash development environment (Fig. 16.1).

At the center of the development environment is the movie **stage**—the white area in which you place graphic elements during movie development. Above the stage is the **timeline**, which represents the time period over which a movie runs. The timeline is divided into increments called **frames**, represented by gray and white rectangles. Each frame depicts a moment in time during the movie, into which you can insert movie elements. The **playhead** indicates the current frame.



Common Programming Error 16.1

Elements placed off stage can still appear if the user changes the aspect ratio of the movie. If an element should not be visible, use an alpha of 0% to hide the element.

The development environment contains several windows that provide options and tools for creating Flash movies. Many of these tools are located in the **Tools bar**, the vertical window located at the left side of the development environment. The **Tools bar** (Fig. 16.2) is divided into multiple sections, each containing tools and functions that help you create Flash movies. The tools near the top of the **Tools bar** select, add and remove graphics from Flash movies. The **Hand** and **Zoom** tools allow you to pan and zoom in the stage. Another section of tools provides colors for shapes, lines and filled areas. The last section contains settings for the **active tool** (i.e., the tool that is highlighted and in use). You can make a tool behave differently by selecting a new mode from the options section of the **Tools bar**.

Application windows called **panels** organize frequently used movie options. Panel options modify the size, shape, color, alignment and effects associated with a movie's graphic elements. By default, panels line the right and bottom edges of the window. Panels

1. Flash Player statistics from Adobe's Flash Player Penetration Survey website at www.adobe.com/products/player_census/flashplayer/version_penetration.html.

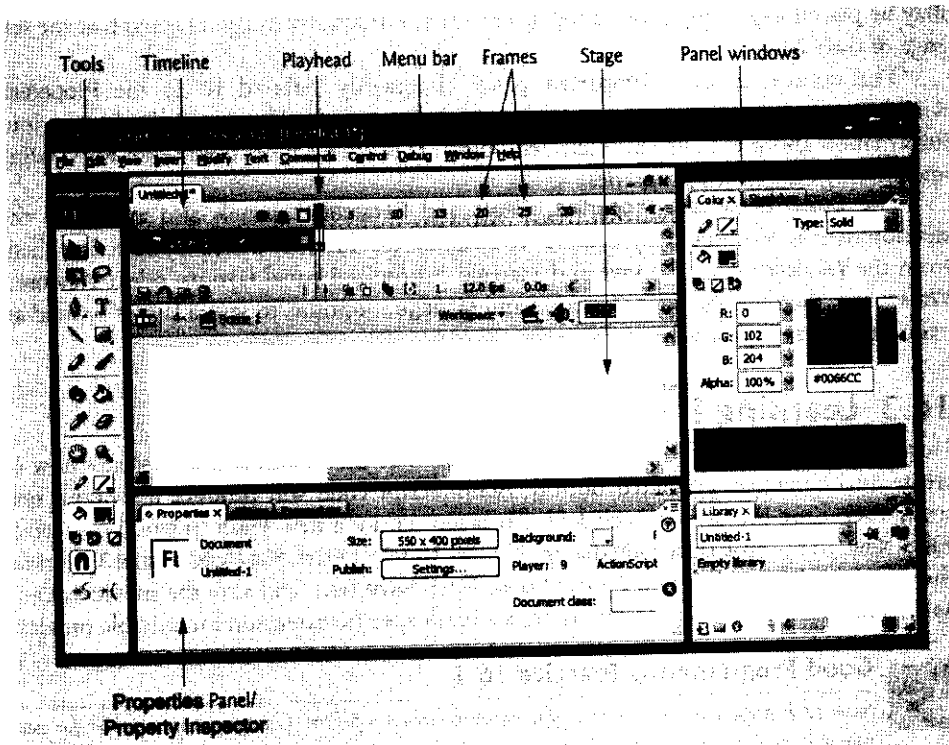


Fig. 16.1 | Flash CS3 development environment.

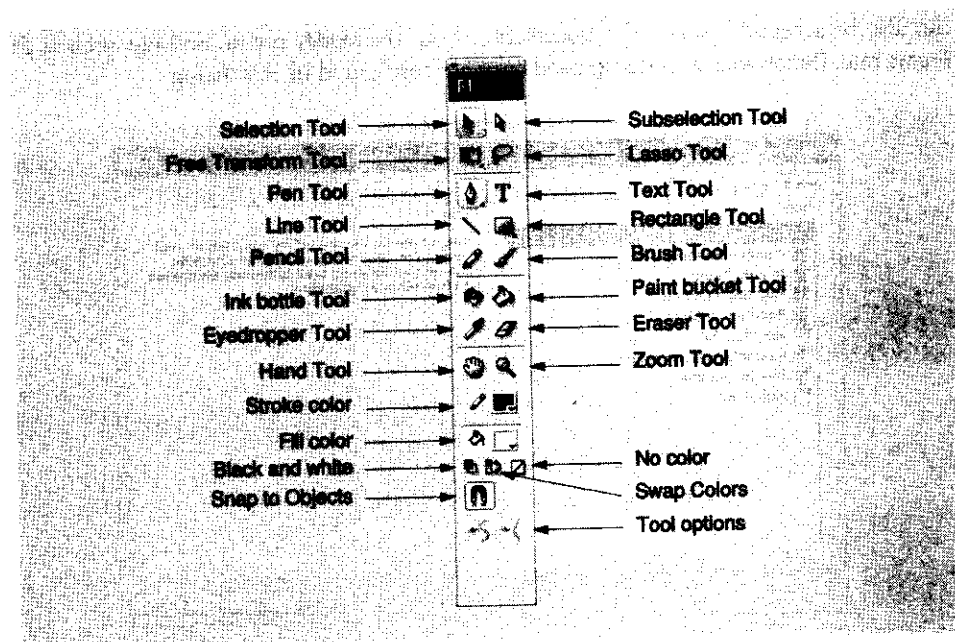


Fig. 16.2 | CS3 Tools bar.

may be placed anywhere in the development environment by dragging the tab at the left edge of their bars.

The context-sensitive **Properties** panel (frequently referred to as the **Properties** window) is located at the bottom of the screen by default. This panel displays various information about the currently selected object. It is Flash's most useful tool for viewing and altering an object's properties.

The **Color**, **Swatches**, **Properties**, **Filters** and **Parameters** panels also appear in the development environment by default. You can access different panels by selecting them from the **Window** menu. To save and manage customized panel layouts, select **Window > Workspace**, then use the **Save Current...** and **Manage...** options to save a layout or load an existing layout, respectively.

16.3 Learning Flash with Hands-On Examples

Now you'll create several complete Flash movies. The first example demonstrates how to create an interactive, animated button. ActionScript code will produce a random text string each time the button is clicked. To begin, create a new Flash movie. First, select **File > New**. In the **New Document** dialog (Fig. 16.3), select **Flash File (ActionScript 3.0)** under the **General** tab and click **OK**. Next, choose **File > Save As...** and save the movie as **Ceo-Assistant.fla**. The **.fla** file extension is a Flash-specific extension for editable movies.



Good Programming Practice 16.1

Save each project with a meaningful name in its own folder. Creating a new folder for each movie helps keep projects organized.

Right click the stage to open a menu containing different movie options. Select **Document Properties...** to display the **Document Properties** dialog (Fig. 16.4). This dialog can also be accessed by selecting **Document...** from the **Modify** menu. Settings such as the **Frame rate**, **Dimensions** and **Background color** are configured in this dialog.

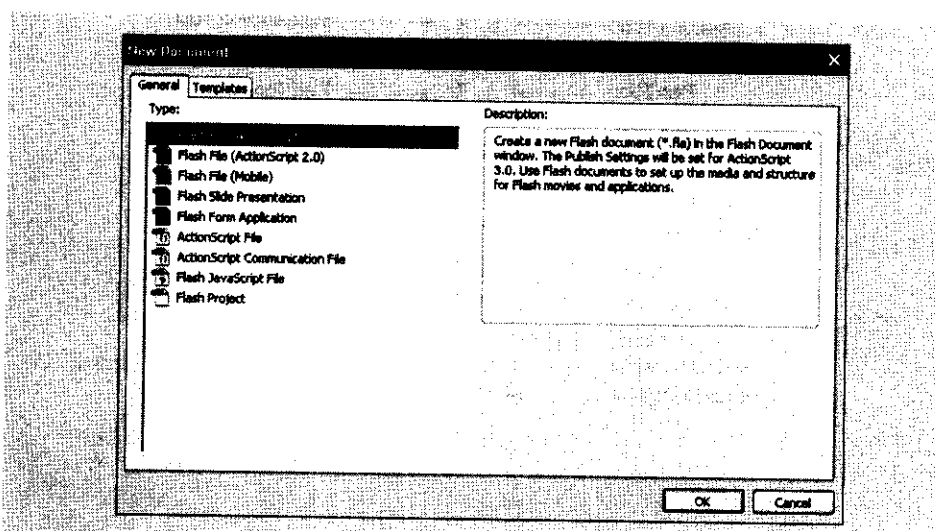


Fig. 16.3 | New Document dialog.

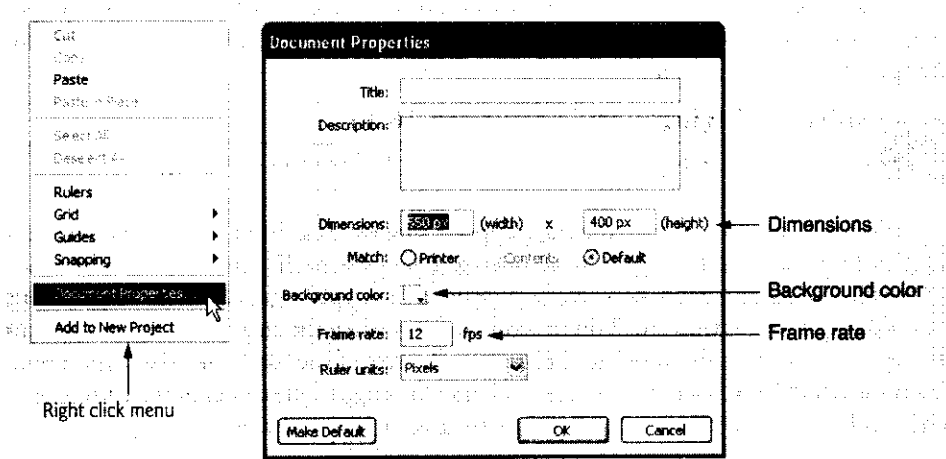


Fig. 16.4 | Document Properties dialog.

The **Frame rate** sets the speed at which movie frames display. A higher frame rate causes more frames to be displayed in a given unit of time (the standard measurement is seconds), thus creating a faster movie. The frame rate for Flash movies on the web is generally between 12 and 60 frames per second (**fps**). Flash’s default frame rate is 12 fps. For this example, set the **Frame Rate** to 10 frames per second.



Performance Tip 16.1

Higher frame rates increase the amount of information to process, and thus increase the movie’s processor usage and file size. Be especially aware of file sizes when catering to low bandwidth web users.

The background color determines the color of the stage. Click the background-color box (called a **swatch**) to select the background color. A new panel opens, presenting a web-safe palette. Web-safe palettes and color selection are discussed in detail in Chapter 3. Note that the mouse pointer changes into an eyedropper, which indicates that you may select a color. Choose a light blue color (Fig. 16.5).

The box in the upper-left corner of the dialog displays the new background color. The **hexadecimal notation** for the selected color appears to the right of this box. The hexadecimal notation is the color code that a web browser uses to render color.

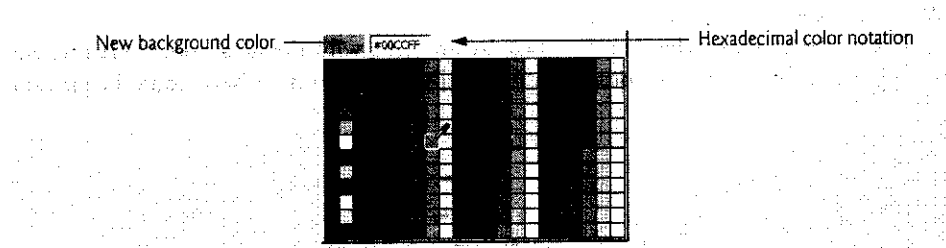


Fig. 16.5 | Selecting a background color.

Dimensions define the size of the movie as it displays on the screen. For this example, set the movie **width** to 200 pixels and the movie **height** to 180 pixels. Click OK to apply the changes in the movie settings.



Software Engineering Observation 16.1

A movie's contents are not resized when you change the size of the movie stage.

With the new dimensions, the stage appears smaller. Select the **Zoom Tool** from the toolbox (Fig. 16.2) and click the stage once to enlarge it to 200 percent of its size (i.e., zoom in). The current zoom percentage appears in the upper-right above the stage editing area. Editing a movie with small dimensions is easier when the stage is enlarged. Press the *Alt* key while clicking the zoom tool to reduce the size of the work area (i.e., zoom out). Select the **Hand Tool** from the toolbox, and drag the stage to the center of the editing area. The hand tool may be accessed at any time by holding down the *spacebar* key.

16.3.1 Creating a Shape with the Oval Tool

Flash provides several editing tools and options for creating graphics. Flash creates shapes using **vectors**—mathematical equations that Flash uses to define size, shape and color. Some other graphics applications create raster graphics or bitmapped graphics. When vector graphics are saved, they are stored using equations. **Raster graphics** are defined by areas of colored **pixels**—the unit of measurement for most computer monitors. Raster graphics typically have larger file sizes because the computer saves the information for every pixel. Vector and raster graphics also differ in their ability to be resized. Vector graphics can be resized without losing clarity, whereas raster graphics lose clarity as they are enlarged or reduced.

We will now create an interactive button out of a circular shape. You can create shapes by dragging with the shape tools. Select the Oval tool from the toolbox. If the Oval tool is not already displayed, click and hold the Rectangle/Oval tool to display the list of rectangle and oval tools. We use this tool to specify the button area. Every shape has a **Stroke color** and a **Fill color**. The stroke color is the color of a shape's outline, and the fill color is the color that fills the shape. Click the swatches in the **Colors** section of the toolbox (Fig. 16.6) to set the fill color to red and the stroke color to black. Select the colors from the web-safe palette or enter their hexadecimal values.

Clicking the **Black and white** button resets the stroke color to black and the fill color to white. Selecting the **Swap colors** option switches the stroke and fill colors. A shape can be created without a fill or stroke color by selecting the **No color** option () when you select either the stroke or fill swatch.

Create the oval anywhere on the stage by dragging with the Oval tool while pressing the *Shift* key. The *Shift* key constrains the oval's proportions to have equal height and

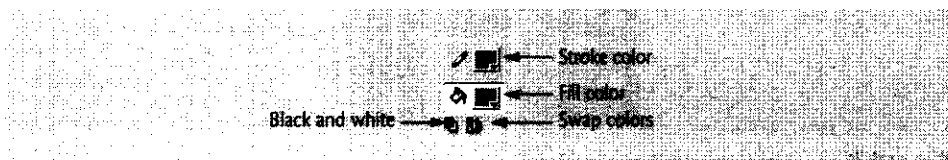


Fig. 16.6 | Setting the fill and stroke colors.

width (i.e., a circle). The same technique creates a square with the Rectangle tool or draws a straight line with the Pencil tool. Drag the mouse until the circle is approximately the size of a dime, then release the mouse button.

After you draw the oval, a dot appears in frame 1, the first frame of the timeline for **Layer 1**. This dot signifies a **keyframe** (Fig. 16.7), which indicates a point of change in a timeline. Whenever you draw a shape in an empty frame, Flash creates a keyframe.

The shape's fill and stroke may be edited individually. Click the red area with the **Selection** tool (black arrow) to select the circle fill. A grid of white dots appears over an object when it is selected (Fig. 16.8). Click the black stroke around the circle while pressing the *Shift* key to add to this selection. You can also make multiple selections by dragging with the selection tool to draw a selection box around specific items.

A shape's size can be modified with the **Properties** panel when the shape is selected (Fig. 16.9). If the panel is not open, open it by selecting **Properties** from the **Window** menu or pressing *<Ctrl>-F3*.

Set the width and height of the circle by typing **30** into the **W:** text field and **30** into the **H:** text field. Entering an equal width and height maintains a **constrained aspect ratio** while changing the circle's size. A constrained aspect ratio maintains an object's proportions as it is resized. Press *Enter* to apply these values.

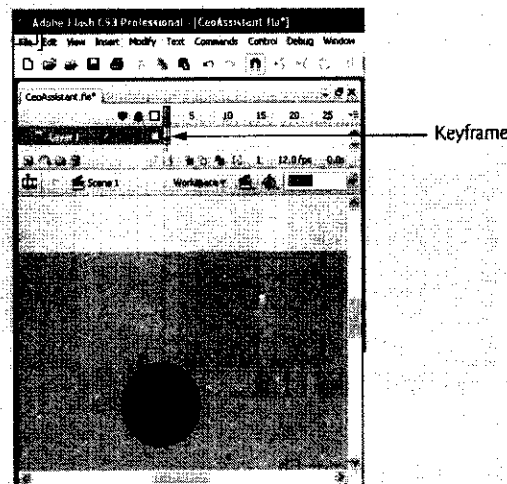


Fig. 16.7 | Keyframe added to the timeline.

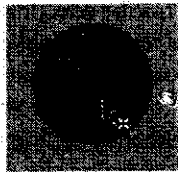


Fig. 16.8 | Making multiple selections with the Selection tool.

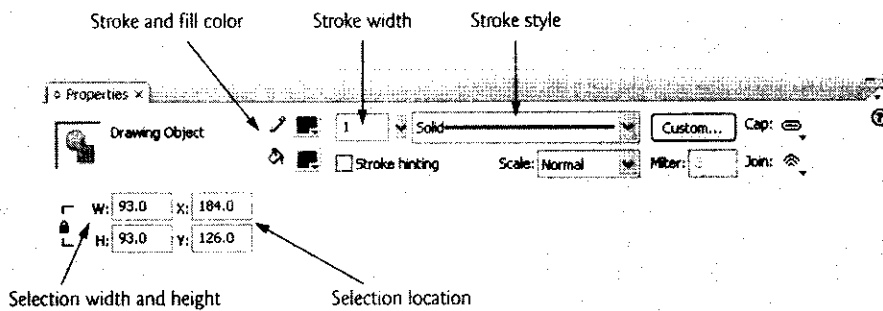


Fig. 16.9 | Modifying the size of a shape with the **Properties** window.

The next step is to modify the shape's color. We will apply a **gradient fill**—a gradual progression of color that fills the shape. Open the **Swatches** panel (Fig. 16.10), either by selecting **Swatches** from the **Window** menu or by pressing **<Ctrl>-F9**. The **Swatches** panel provides four **radial gradients** and three **linear gradients**, although you also can create and edit gradients with the **Color** panel.

Click outside the circle with the Selection tool to deselect the circle. Now, select only the red fill with the Selection tool. Change the fill color by clicking the red radial gradient fill in the **Swatches** panel. The gradient fills are located at the bottom of the **Swatches** panel (Fig. 16.10). The circle should now have a red radial gradient fill with a black stroke surrounding it.

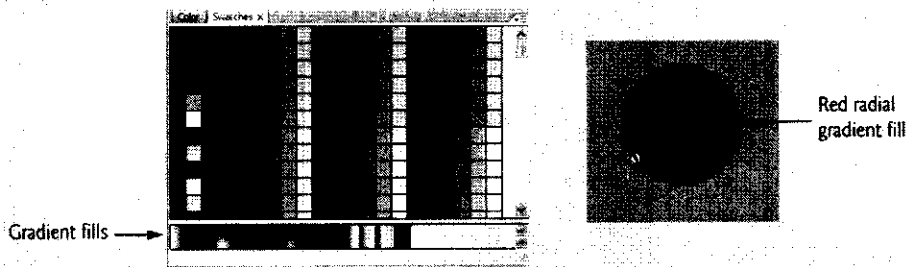


Fig. 16.10 | Choosing a gradient fill.

16.3.2 Adding Text to a Button

Button titles communicate a button's function to the user. The easiest way to create a title is with the **Text** tool. Create a button title by selecting the **Text** tool and clicking the center of the button. Next, type **GO** in capital letters. Highlight the text with the **Text** tool. Once text is selected, you can change the font, text size and font color with the **Properties** window (Fig. 16.11). Select a sans-serif font, such as **Arial** or **Verdana**, from the font drop-down list. Set the font size to **14 pt** either by typing the size into the font size field or by pressing the arrow button next to it, revealing the **size selection slider**—a vertical slider that, when moved, changes the font size. Set the font weight to **bold** by clicking the **bold**

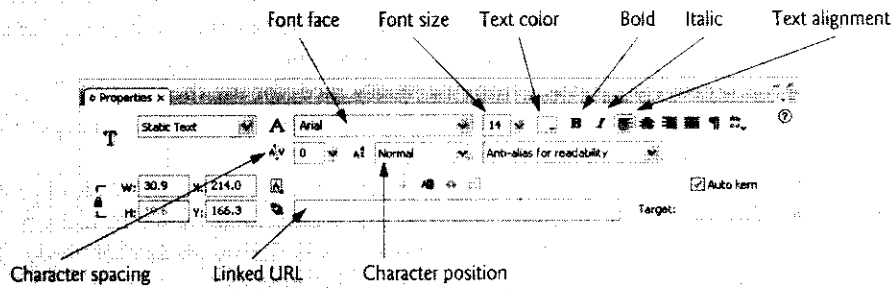


Fig. 16.11 | Setting the font face, size, weight and color with the **Properties window**.

button (B). Finally, change the font color by clicking the text color swatch and selecting white from the palette.

Look-and-Feel Observation 16.1



Sans-serif fonts, such as Arial, Helvetica and Verdana, are easier to read on a computer monitor, and therefore ensure better usability.

If the text does not appear in the correct location, drag it to the center of the button with the Selection tool. The button is almost complete and should look similar to Fig. 16.12.



Fig. 16.12 | Adding text to the button.

16.3.3 Converting a Shape into a Symbol

A Flash movie consists of **scenes** and **symbols**. Each scene contains all graphics and symbols. The parent movie may contain several symbols that are reusable movie elements, such as **graphics**, **buttons** and **movie clips**. A scene timeline can contain numerous symbols, each with its own timeline and properties. A scene may have several **instances** of any given symbol (i.e., the same symbol can appear multiple times in one scene). You can edit symbols independently of the scene by using the symbol's **editing stage**. The editing stage is separate from the scene stage and contains only one symbol.

Good Programming Practice 16.2



Reusing symbols can drastically reduce file size, thereby allowing faster downloads.

To make our button interactive, we must first convert the button into a button symbol. The button consists of distinct text, color fill and stroke elements on the parent

stage. These items are combined and treated as one object when the button is converted into a symbol. Use the Selection tool to drag a **selection box** around the button, selecting the button fill, the button stroke and the text all at one time (Fig. 16.13).

Now, select **Convert to Symbol...** from the **Modify** menu or use the shortcut *F8* on the keyboard. This opens the **Convert to Symbol** dialog, in which you can set the properties of a new symbol (Fig. 16.14).

Every symbol in a Flash movie must have a unique name. It is a good idea to name symbols by their contents or function, because this makes them easier to identify and reuse. Enter the name **go button** into the **Name** field of the **Convert to Symbol** dialog. The **Behavior** option determines the symbol's function in the movie.

You can create three different types of symbols—movie clips, buttons and graphics. A **movie clip symbol's** behavior is similar to that of a scene and thus it is ideal for recurring animations. **Graphic symbols** are ideal for static images and basic animations. **Button symbols** are objects that perform button actions, such as **rollovers** and hyperlinking. A rollover is an action that changes the appearance of a button when the mouse passes over it. For this example, select **Button** as the type of symbol and click **OK**. The button should now be surrounded by a blue box with crosshairs in the upper-left corner, indicating that the button is a symbol. Also, in the **Properties** window panel, name this instance of the **go button** symbol **goButton** in the field containing **<Instance Name>**. Use the selection tool to drag the button to the lower-right corner of the stage.

The **Library** panel (Fig. 16.15) stores every symbol present in a movie and is accessed through the **Window** menu or by the shortcuts **<Ctrl>-L** or *F11*. Multiple instances of a symbol can be placed in a movie by dragging and dropping the symbol from the **Library** panel onto the stage.

The **Movie Explorer** displays the movie structure and is accessed by selecting **Movie Explorer** from the **Window** menu or by pressing **<Alt>-F3** (Fig. 16.16). The **Movie Explorer** panel illustrates the relationship between the current scene (**Scene 1**) and its symbols.

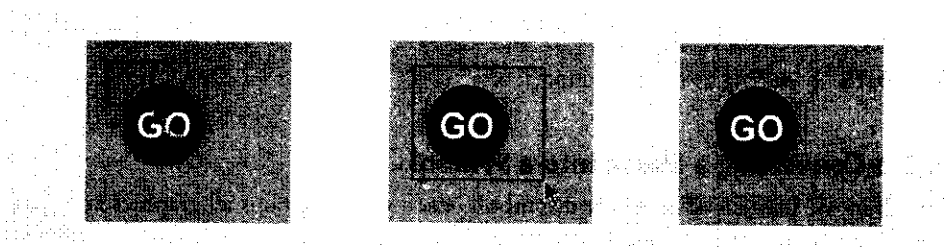


Fig. 16.13 | Selecting an object with the selection tool.

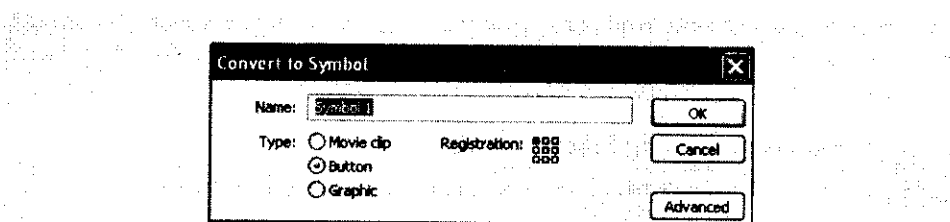


Fig. 16.14 | Creating a new symbol with the **Convert to Symbol** dialog.

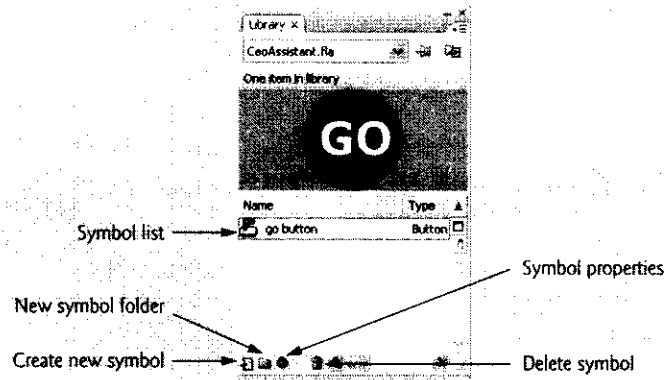


Fig. 16.15 | Library panel.

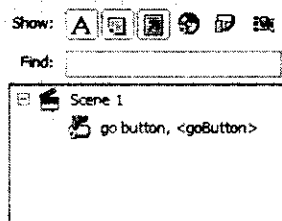


Fig. 16.16 | Movie Explorer for CeoAssistant.fla.

16.3.4 Editing Button Symbols

The next step in this example is to make the button symbol interactive. The different components of a button symbol, such as its text, color fill and stroke, may be edited in the symbol's editing stage, which you can access by double clicking the icon next to the symbol in the **Library**. A button symbol's timeline contains four frames, one for each of the button states (up, over and down) and one for the hit area.

The **up state** (indicated by the **Up** frame on screen) is the default state before the user presses the button or rolls over it with the mouse. Control shifts to the **over state** (i.e., the **Over** frame) when the user rolls over the button with the mouse cursor. The button's **down state** (i.e., the **Down** frame) plays when a user presses a button. You can create interactive, user-responsive buttons by customizing the appearance of a button in each of these states. Graphic elements in the **hit state** (i.e., the **Hit** frame) are not visible to a viewer of the movie; they exist simply to define the active area of the button (i.e., the area that can be clicked). The hit state will be discussed further in Section 16.6.

By default, buttons have only the up state activated when they are created. You may activate other states by adding keyframes to the other three frames. Keyframes for a button, discussed in the next section, determine how a button reacts when it is rolled over or clicked with the mouse.

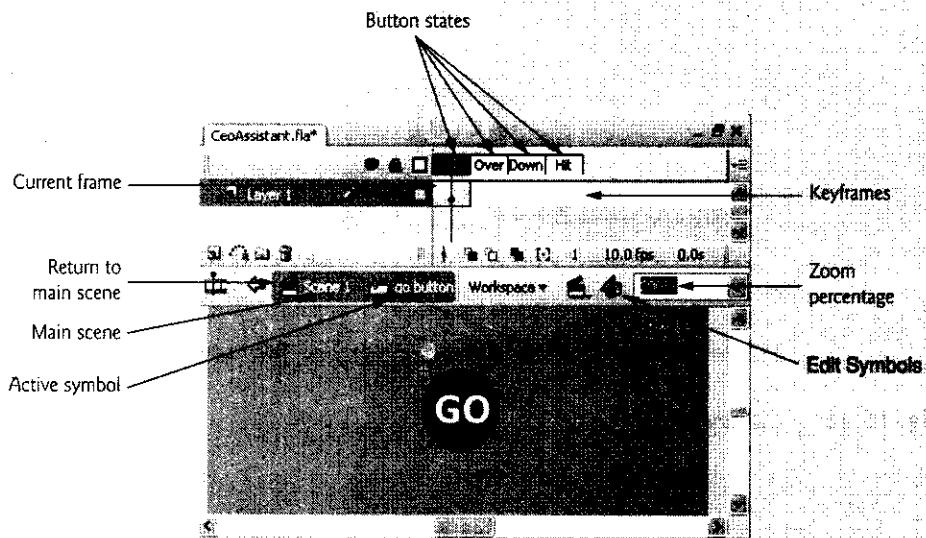


Fig. 16.17 | Modifying button states with a button's editing stage.

16.3.5 Adding Keyframes

Keyframes are points of change in a Flash movie and appear in the timeline with a dot. By adding keyframes to a button symbol's timeline, you can control how the button reacts to user interactions. The following step shows how to create a button rollover effect, which is accomplished by inserting a keyframe in the button's **Over** frame, then changing the button's appearance in that frame. Right click the **Over** frame and select **Insert Keyframe** from the resulting menu or press *F6* (Fig. 16.18).

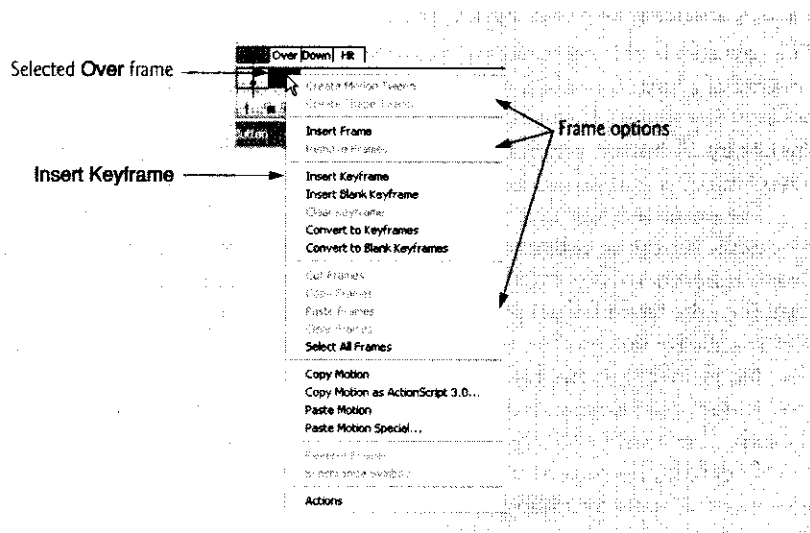


Fig. 16.18 | Inserting a keyframe.

Select the **Over** frame and click outside the button area with the selection tool to deselect the button's components. Change the color of the button in the **Over** state from red gradient fill to green gradient fill by selecting only the fill portion of the button with the Selection tool. Select the green gradient fill in the **Swatches** panel to change the color of the button in the **Over** state. Changing the color of the button in the over state does not affect the color of the button in the up state. Now, when the user moves the cursor over the button (in the up state) the button animation is replaced by the animation in the **Over** state. Here, we change only the button's color, but we could have created an entirely new animation in the **Over** state. The button will now change from red to green when the user rolls over the button with the mouse. The button will return to red when the mouse is no longer positioned over the button.

16.3.6 Adding Sound to a Button

The next step is to add a sound effect that plays when a user clicks the button. Flash imports sounds in the **WAV** (Windows), **AIFF** (Macintosh) or **MP3** formats. Several button sounds are available free for download from sites such as Flashkit (www.flashkit.com) and Muinar (www.sounds.muinar.com). For this example, download the **cash register** sound in **WAV** format from

www.flashkit.com/soundfx/Industrial_Commercial/Cash

Click the **Download** link to download the sound from this site. This link opens a new web page from which the user chooses the sound format. Choose **MP3** as the file format by clicking the **mp3** link. Save the file to the same folder as **CeoAssistant.fla**. Extract the sound file and save it in the same folder as **CeoAssistant.fla**.

Once the sound file is extracted, it can be imported into Flash. Import the sound into the **Library** by choosing **Import to Library...** from the **Import** submenu of the **File** menu. Select **All Formats** in the **Files of type** field of the **Import** dialog so that all available files are displayed. Select the sound file and press **Open**. This imports the sound file and places it in the movie's **Library**, making it available to use in the movie.

You can add sound to a movie by placing the sound clip in a keyframe or over a series of frames. For this example, we add the sound to the button's down state so that the sound plays when the user presses the button. Select the button's **Down** frame and press **F6** to add a keyframe.

Add the sound to the **Down** keyframe by dragging it from the **Library** to the stage. Open the **Properties** window (Fig. 16.19) and select the **Down** frame in the timeline to define the sound's properties in the movie. To ensure the desired sound has been added to the keyframe, choose the sound filename from the **Sound** drop-down list. This list contains all the sounds that have been added to the movie. Make sure the **Sync** field is set to **Event** so that the sound plays when the user clicks the button. If the **Down** frame has a blue wave or line through it, the sound effect has been added to the button.

Next, optimize the sound for the web. Double click the sound icon in the **Library** panel to open the **Sound Properties** dialog (Fig. 16.20). The settings in this dialog change the way that the sound is saved in the final movie. Different settings are optimal for different sounds and different audiences. For this example, set the **Compression** type to **MP3**, which reduces file size. Ensure that **Use imported MP3 quality** is selected. If the sound clip

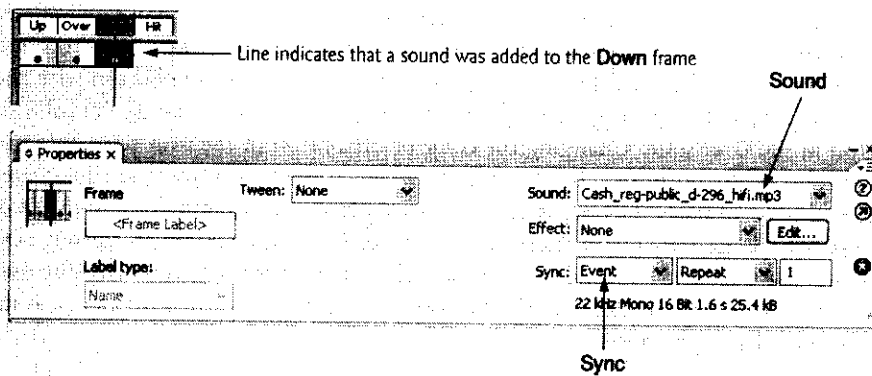


Fig. 16.19 | Adding sound to a button.

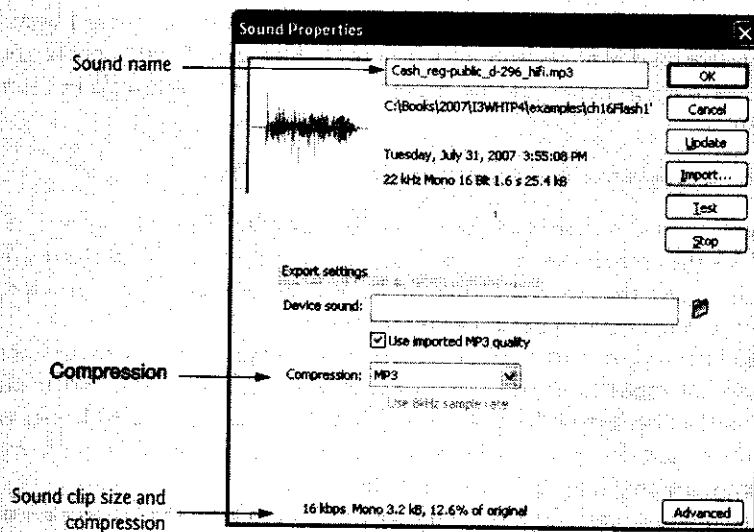


Fig. 16.20 | Optimizing sound with the Sound Properties dialog.

is long, or if the source MP3 was encoded with a high bitrate, you may want to deselect this and specify your own bitrate to save space.

The sound clip is now optimized for use on the web. Return to the scene by pressing the **Edit Scene** button (🖱️) and selecting **Scene 1** or by clicking **Scene 1** at the top of the movie window.

16.3.7 Verifying Changes with Test Movie

It is a good idea to ensure that movie components function correctly before proceeding further with development. Movies can be viewed in their **published** state with the Flash Player. The published state of a movie is how it would appear if viewed over the web or with the Flash Player. Published Flash movies have the Shockwave Flash extension **.swf** (pronounced “swiff”). SWF files can be viewed but not edited.

Select **Test Movie** from the **Control** menu (or press <Ctrl>-Enter) to **export** the movie into the Flash Player. A window opens with the movie in its published state. Move the cursor over the **GO** button to view the color change (Fig. 16.21), then click the button to play the sound. Close the test window to return to the stage. If the button's color does not change, return to the button's editing stage and check that you followed the steps correctly.

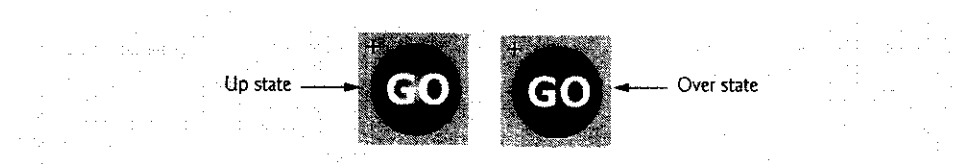


Fig. 16.21 | GO button in its up and over states.

16.3.8 Adding Layers to a Movie

The next step in this example is to create the movie's title animation. It's a good idea for you to create a new **layer** for new movie items. A movie can be composed of many layers, each having its own attributes and effects. Layers organize movie elements so that they can be animated and edited separately, making the composition of complex movies easier. Graphics in higher layers appear over the graphics in lower layers.

Before creating a new title layer, double click the text **Layer 1** in the timeline. Rename the layer by entering the text **Button** into the name field (Fig. 16.22).

Create a new layer for the title animation by clicking the **Insert a new layer** button or by selecting **Layer** from the **Timeline** submenu of the **Insert** menu. The **Insert a new layer** button places a layer named **Layer 2** above the selected layer. Change the name of **Layer 2** to **Title**. Activate the new layer by clicking its name.



Good Programming Practice 16.3

Always give movie layers descriptive names. Descriptive names are especially helpful when working with many layers.

Select the **Text** tool to create the title text. Click with the **Text** tool in the center of the stage toward the top. Use the **Property** window to set the font to **Arial**, the text color to navy blue (hexadecimal value #000099) and the font size to **20 pt** (Fig. 16.23). Set the text alignment to center by clicking the center justify button.

Type the title **CEO Assistant 1.0** (Fig. 16.24), then click the selection tool. A blue box appears around the text, indicating that it is a **grouped object**. This text is a grouped object because each letter is a part of a text string and cannot be edited independently. Text can be broken apart for color editing, shape modification or animation (shown in a later example). Once text has been broken apart, it may not be edited with the **Text** tool.

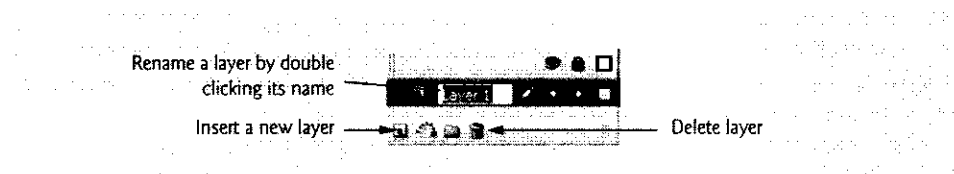


Fig. 16.22 | Renaming a layer.

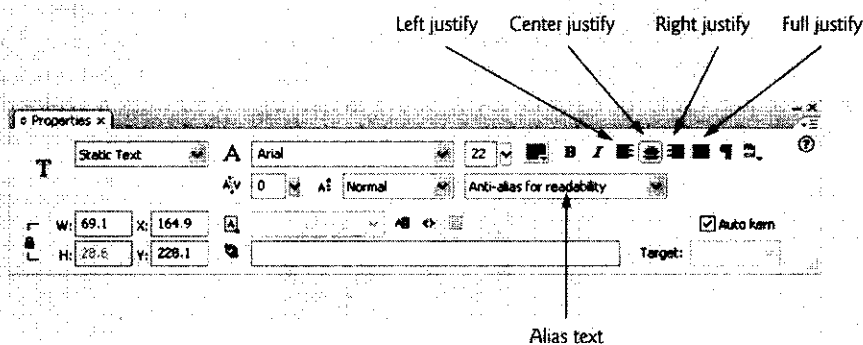


Fig. 16.23 | Setting text alignment with the Properties window.



Fig. 16.24 | Creating a title with the Text tool.

16.3.9 Animating Text with Tweening

Animations in Flash are created by inserting keyframes into the timeline. Each keyframe represents a significant change in the position or appearance of the animated object.

You may use several methods to animate objects in Flash. One is to create a series of successive keyframes in the timeline. Modifying the animated object in each keyframe creates an animation as the movie plays. Another method is to insert a keyframe later in the timeline representing the final appearance and position of the object, then create a tween between the two keyframes. **Tweening** is an automated process in which Flash creates the intermediate steps of the animation between two keyframes.

Flash provides two tweening methods. **Shape tweening** morphs an object from one shape to another. For instance, the word “star” could morph into the shape of a star. Shape tweening can be applied only to ungrouped objects, not symbols or grouped objects. Be sure to break apart text before attempting to create a shape tween. **Motion tweening** moves objects around the stage. Motion tweening can be applied to symbols or grouped objects.

You can only have one symbol per layer if you intend to tween the symbol. At this point in the development of the example movie, only frame 1 is occupied in each layer. Keyframes must be designated in the timeline before adding the motion tween. Click frame 15 in the **Title** layer and press **F6** to add a new keyframe. All the intermediate frames in the timeline should turn gray, indicating that they are active (Fig. 16.25). Until the motion tween is added, each active frame contains the same image as the first frame.

The button disappears from the movie after the first frame because only the first frame is active in the button layer. Before the movie is completed, we’ll move the button to frame 15 of its layer so that the button appears once the animation stops.

We now create a motion tween by modifying the position of the title text. Select frame 1 of the **Title** layer and select the title text with the Selection tool. Drag the title text directly above the stage. When the motion tween is added, the title will move onto the stage. Add

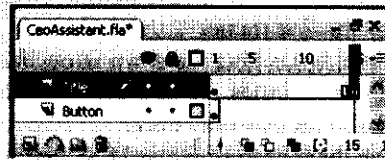


Fig. 16.25 | Adding a keyframe to create an animation.

the motion tween by right clicking frame 1 in the **Title** layer. Then select **Create Motion Tween** from the **Insert > Timeline** menu. Tweens also can be added using the **Tween type** drop down menu in the **Properties** window. Frames 2–14 should turn light blue, with an arrow pointing from the keyframe in frame 1 to the keyframe in frame 15 (Fig. 16.26).

Test the movie again with the Flash Player by pressing **<Ctrl>-Enter** to view the new animation. Note that the animation continues to loop—all Flash movies loop by default. Adding the ActionScript function **stop** to the last frame in the movie stops the movie from looping. For this example, click frame 15 of the **Title** layer, and open the **Actions** panel by selecting **Window > Actions** or by pressing **F9** (Fig. 16.27). The **Actions** panel is used to add

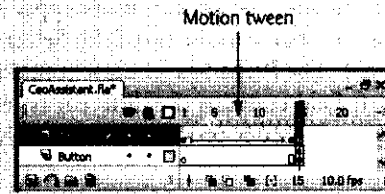


Fig. 16.26 | Creating a motion tween.

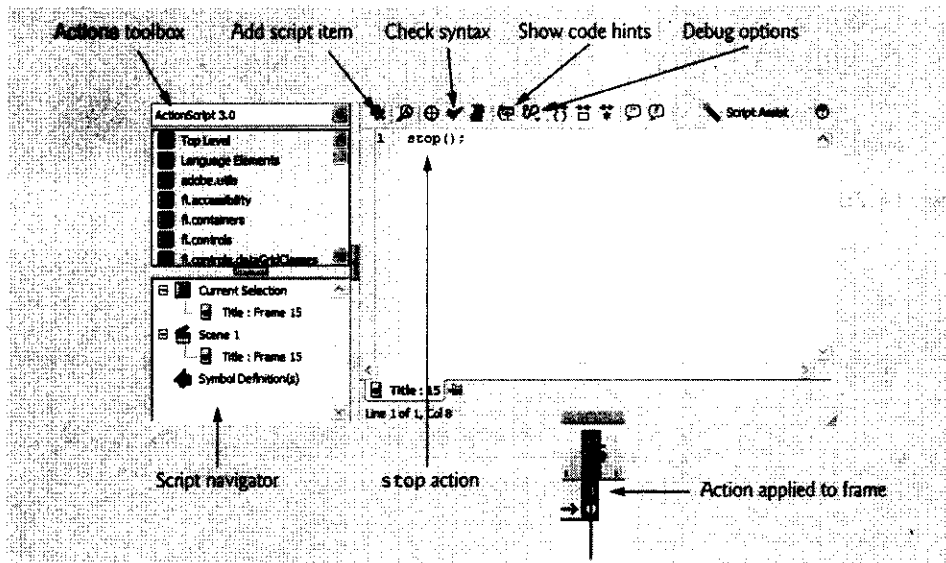


Fig. 16.27 | Adding ActionScript to a frame with the **Actions** panel.

actions (i.e., scripted behaviors) to symbols and frames. Here, add `stop()`; so that the movie does not loop back to the first frame.

Minimize the **Actions** panel by clicking the down arrow in its title bar. The small letter **a** in frame 15 of the **Title** layer indicates the new action. Test the movie again in Flash Player. Now, the animation should play only once.

The next step is to move the button to frame 15 so that it appears only at the end of the movie. Add a keyframe to frame 15 of the **Button** layer. A copy of the button should appear in the new keyframe. Select the button in the first frame and delete it by pressing the *Delete* key. The button will now appear only in the keyframe at the end of the movie.

16.3.10 Adding a Text Field

The final component of our movie is a **text field**, which contains a string of text that changes every time the user presses the button. An instance name is given to the text field so that ActionScript added to the button can control its contents.

Create a layer named **Advice** for the new text field, and add a keyframe to frame 15 of the **Advice** layer. Select the Text tool and create the text field by dragging with the mouse in the stage (Fig. 16.28). Place the text field directly below the title. Set the text font to **Courier New**, **12 pt** and the style to **bold** in the **Properties** window. You can alter the size of the text field by dragging the **anchor** that appears in its upper-right corner.

You'll now assign an instance name to the text field. Select the text field and open the **Properties** window (Fig. 16.29). The **Properties** window contains several options for modifying text fields. The top-left field contains the different types of text fields: **Static Text**, the default setting for this panel, creates text that does not change. The second option,

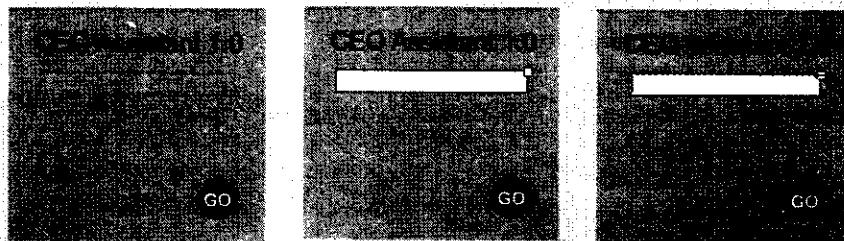


Fig. 16.28 | Creating a text field.

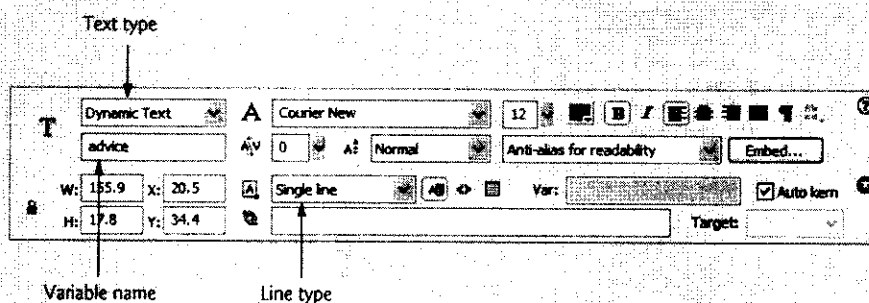


Fig. 16.29 | Creating a dynamic text field with the **Properties** window.

Dynamic Text, creates text that can be changed or determined by outside variables through ActionScript. When you select this text type, new options appear below this field. The **Line type** drop-down list specifies the text field size as either a single line or multiple lines of text. The **Instance Name** field allows you to give the text field an instance name by which it can be referenced in script. For example, if the text field instance name is `newText`, you could write a script setting `newText.text` equal to a string or a function output. The third text type, **Input Text**, creates a text field into which the viewers of the movie can input their own text. For this example, select **Dynamic Text** as the text type. Set the line type to **Single Line** and enter `advice` as the instance name. This instance name will be used in ActionScript later in this example.

16.3.1.1 Adding ActionScript

All the movie objects are now in place, so `CEO Assistant 1.0` is almost complete. The final step is to add ActionScript to the button, enabling the script to change the contents of the text field every time a user clicks the button. Our script calls a built-in Flash function to generate a random number. This random number corresponds to a message in a list of possible messages to display. [Note: The ActionScript in this chapter has been formatted to conform with the code-layout conventions of this book. The Flash application may produce code that is formatted differently.]

Select frame 15 of the **Button** layer and open the **Actions** panel. We want the action to occur when the user clicks the button. To achieve this, insert the statement:

```
goButton.addEventListener( MouseEvent.MOUSE_DOWN, goFunction );
```

This statement uses the button object's instance name (`goButton`) to call the **addEventListener** function, which registers an event handler (`goFunction` in this example) that will be called when the event takes place (i.e., when you click the button). The first argument, `MouseEvent.MOUSE_DOWN`, specifies that an action is performed when the user presses the button with the mouse.

The next step is to add the function that handles this event. Create a new function named `goFunction` by using the code

```
function goFunction( event : MouseEvent ) : void
{
} // end function goFunction
```

The function's one parameter is a `MouseEvent`, implying that the function has to be provided with a mouse action to be accessed. The function does not return anything, hence the `void` return value. Inside this function, add the following statement:

```
var randomNumber : int = Math.floor( ( Math.random() * 5 ) );
```

which creates an integer variable called `randomNumber` and assigns it a random value. For this example, we use the `Math.random` function to choose a random number from 0 to 1. `Math.random` returns a random floating-point number from 0.0 up to, but not including, 1.0. Then, it is scaled accordingly, depending on what the range should be. Since we want all the numbers between 0 and 4, inclusive, the value returned by the `Math.random` should be multiplied by 5 to produce a number in the range 0.0 up to, but not including, 5.0.

Finally, this new number should be rounded down to the largest integer smaller than itself, using the `Math.floor` function.



Error-Prevention Tip 16.1

ActionScript is case sensitive. Be aware of the case when entering arguments or variable names.

The value of `randomNumber` determines the text string that appears in the text field. A `switch` statement sets the text field's value based on the value of `randomNumber`. [*Note:* For more on `switch` statements, refer to Chapter 8.] On a new line in the `goFunction` function, insert the following `switch` statement:

```
switch ( randomNumber )
{
    case 0:
        advice.text = "Hire Someone!";
        break;
    case 1:
        advice.text = "Buy a Yacht!";
        break;
    case 2:
        advice.text = "Buy stock!";
        break;
    case 3:
        advice.text = "Go Golfing!";
        break;
    case 4:
        advice.text = "Hold a meeting!";
        break;
} // end switch
```

This statement displays different text in the `advice` text field based on the value of the variable `randomNumber`. The text field's `text` property specifies the text to display. If you feel ambitious, increase the number of `advice` statements by producing a larger range of random values and adding more cases to the `switch` statement. Minimize the **Actions** panel to continue.

Congratulations! You have now completed building *CEO Assistant 1.0*. Test the movie by pressing `<Ctrl>-Enter` and clicking the **GO** button. After testing the movie with the Flash Player, return to the main window and save the file.

16.4 Publishing Your Flash Movie

Flash movies must be **published** for users to view them outside the Flash CS3 environment and Flash Player. This section discusses the more common methods of publishing Flash movies. For this example, we want to publish in two formats, **Flash** and **Windows Projector**, which creates a standard Windows-executable file that works even if the user hasn't installed Flash. Select **Publish Settings...** from the **File** menu to open the **Publish Settings** dialog.

Select the **Flash**, **HTML** and **Windows Projector** checkboxes and uncheck all the others. Then click the **Flash** tab at the top of the dialog. This section of the dialog allows you to choose the Flash settings. Flash movies may be published in an older Flash version if you

wish to support older Flash Players. Note that ActionScript 3.0 is not supported by older players, so choose a version with care. Publish the movie by clicking **Publish** in the **Publish Settings** dialog or by selecting **Publish** from the **File** menu. After you've published the movie, the directory in which you saved the movie will have several new files (Fig. 16.30). If you wish to place your movie on a website, be sure to copy the HTML, JavaScript and SWF files to your server.



Good Programming Practice 16.4

It is not necessary to transfer the .fla version of your Flash movie to a web server unless you want other users to be able to download the editable version of the movie.

As we can see in the Ceo Assistant 1.0 example, Flash is a feature-rich program. We have only begun to use Flash to its full potential. ActionScript can create sophisticated programs and interactive movies. It also enables Flash to interact with ASP.NET (Chapter 25), PHP (Chapter 23), and JavaScript (Chapters 6–11), making it a program that integrates smoothly into a web environment.

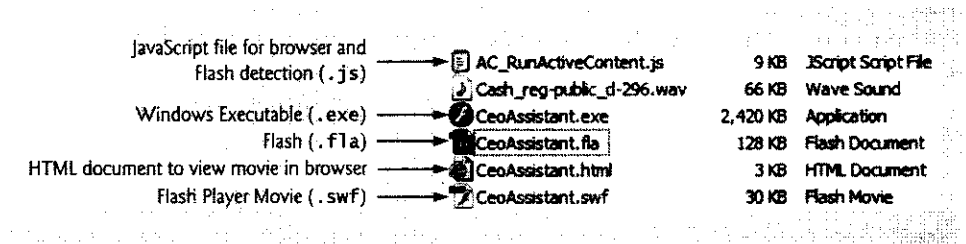


Fig. 16.30 | Published Flash files.

16.5 Creating Special Effects with Flash

The following sections introduce several Flash special effects. The preceding example familiarized you with basic movie development. The next sections cover many additional topics, from importing bitmaps to creating splash screens that display before a web page loads.

16.5.1 Importing and Manipulating Bitmaps

Some of the examples in this chapter require importing bitmapped images and other media into a Flash movie. The importing process is similar for all types of media, including images, sound and video. The following example shows how to import an image into a Flash movie.

Begin by creating a new Flash document. The image we are going to import is located in the Chapter 16 examples folder. Select **File > Import > Import to Stage...** (or press **<Ctrl>-R**) to display the **Import** dialog. Browse to the folder on your system containing this chapter's examples and open the folder labeled **images**. Select **bug.bmp** and click **OK** to continue. A bug image should appear on the stage. The **Library** panel stores imported images. You can convert imported images into editable shapes by selecting the image and pressing **<Ctrl>-B** or by choosing **Break Apart** from the **Modify** menu. Once an imported image is broken apart, it may be shape tweened or edited with editing tools, such as the

Lasso, Paint bucket, Eraser and Paintbrush. The editing tools are found in the toolbox and apply changes to a shape.

Dragging with the **Lasso tool** selects areas of shapes. The color of a selected area may be changed or the selected area may be moved. Click and drag with the Lasso tool to draw the boundaries of the selection. As with the button in the last example, when you select a shape area, a mesh of white dots covers the selection. Once an area is selected, you may change its color by selecting a new fill color with the fill swatch or by clicking the selection with the Paint bucket tool. The Lasso tool has different options (located in the **Options** section of the toolbox) including **Magic wand** and **Polygon mode**. The Magic wand option changes the Lasso tool into the Magic wand tool, which selects areas of similar colors. The polygonal lasso selects straight-edged areas.

The **Eraser tool** removes shape areas when you click and drag the tool across an area. You can change the eraser size using the tool options. Other options include settings that make the tool erase only fills or strokes.

The **Brush tool** applies color in the same way that the eraser removes color. The paintbrush color is selected with the fill swatch. The paintbrush tool options include a **Brush mode** option. These modes are **Paint behind**, which sets the tool to paint only in areas with no color information; **Paint selection**, which paints only areas that have been selected; and **Paint inside**, which paints inside a line boundary.

Each of these tools can create original graphics. Experiment with the different tools to change the shape and color of the imported bug graphic.



Portability Tip 16.1

When building Flash movies, use the smallest possible file size and web-safe colors to ensure that most people can view the movie regardless of bandwidth, processor speed or monitor resolution.

16.5.2 Creating an Advertisement Banner with Masking

Masking hides portions of layers. A **masking layer** hides objects in the layers beneath it, revealing only the areas that can be seen through the shape of the mask. Items drawn on a masking layer define the mask's shape and cannot be seen in the final movie. The next example, which builds a website banner, shows how to use masking frames to add animation and color effects to text.

Create a new Flash document and set the size of the stage to **470** pixels wide by **60** pixels high. Create three layers named **top**, **middle** and **bottom** according to their positions in the layer hierarchy. These names help track the masked layer and the visible layers. The **top** layer contains the mask, the **middle** layer becomes the masked animation and the **bottom** layer contains an imported bitmapped logo. Import the graphic `bug_apple.bmp` (from the `images` folder in this chapter's examples folder) into the first frame of the **top** layer, using the method described in the preceding section. This image will appear too large to fit in the stage area. Select the image with the selection tool and align it with the upper-left corner of the stage. Then select the **Free transform** tool in the toolbox (Fig. 16.31).

The Free transform tool allows us to resize an image. When an object is selected with this tool, anchors appear around its corners and sides. Click and drag an anchor to resize the image in any direction. Holding the *Shift* key while dragging a corner anchor ensures

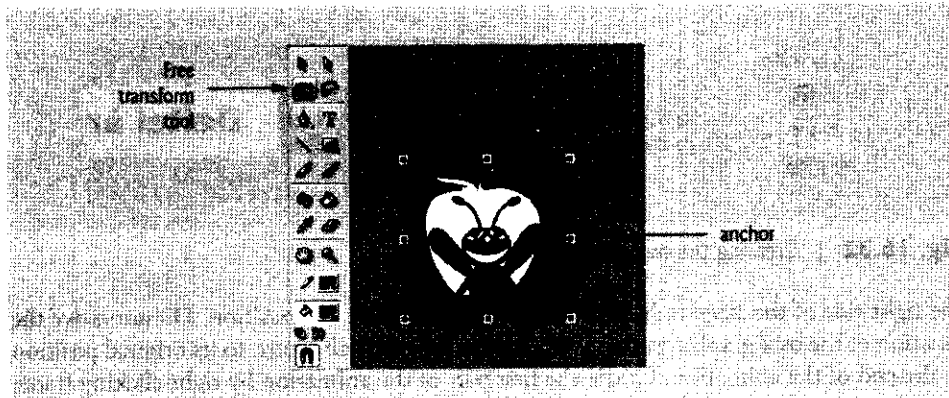


Fig. 16.31 | Resizing an image with the Free transform tool.

that the image maintains the original height and width ratio. Hold down the *Shift* key while dragging the lower-right anchor upward until the image fits on the stage.

Use the text tool to add text to frame 1 of the **top** layer. Use **Verdana, 28 pt bold**, as the font. Select a blue text color, and make sure that **Static Text** is selected in the **Properties** window. Type the banner text “Deitel and Associates”, making sure that the text fits inside the stage area, and use the Selection tool to position the text next to the image. This text becomes the object that masks an animation.

We must convert the text into a shape before using it as a mask. Click the text field with the Selection tool to ensure that it is active and select **Break Apart** twice from the **Modify** menu. Breaking the text apart once converts each letter into its own text field. Breaking it apart again converts the letters into shapes that cannot be edited with the text tool, but can be manipulated as regular graphics.

Copy the contents of the **top** layer to the **bottom** layer before creating the mask, so that the text remains visible when the mask is added. Right click frame 1 of the **top** layer, and select **Copy Frames** from the resulting menu. Paste the contents of the **top** layer into frame 1 of the **bottom** layer by right clicking frame 1 of the **bottom** layer and selecting **Paste Frames** from the menu. This shortcut pastes the frame’s contents in the same positions as the original frame. Delete the extra copy of the bug image by selecting the bug image in the **top** layer with the selection tool and pressing the *Delete* key.

Next, you’ll create the animated graphic that the banner text in the **top** layer masks. Click in the first frame of the **middle** layer and use the Oval tool to draw a circle to the left of the image that is taller than the text. The oval does not need to fit inside the banner area. Set the oval stroke to **no color** by clicking the stroke swatch and selecting the **No color** option. Set the fill color to the rainbow gradient (Fig. 16.32), found at the bottom of the **Swatches** panel.

Select the oval by clicking it with the Selection tool, and convert the oval to a symbol by pressing *F8*. Name the symbol **oval** and set the behavior to **Graphic**. When the banner is complete, the oval will move across the stage; however, it will be visible only through the text mask in the **top** layer. Move the oval just outside the left edge of the stage, indicating the point at which the oval begins its animation. Create a keyframe in frame 20 of the **middle** layer and another in frame 40. These keyframes indicate the different locations of the **oval** symbol during the animation. Click frame 20 and move the oval just outside

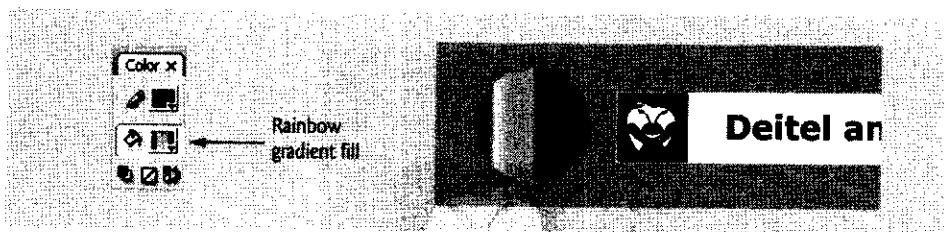


Fig. 16.32 | Creating the oval graphic.

the right side of the stage to indicate the animation's next key position. Do not move the position of the oval graphic in frame 40, so that the oval will return to its original position at the end of the animation. Create the first part of the animation by right clicking frame 1 of the **middle** layer and choosing **Create Motion Tween** from the menu. Repeat this step for frame 20 of the **middle** layer, making the oval symbol move from left to right and back. Add keyframes to frame 40 of both the **top** and **bottom** layers so that the other movie elements appear throughout the movie.

Now that all the supporting movie elements are in place, the next step is to apply the masking effect. To do so, right click the **top** layer and select **Mask** (Fig. 16.33). Adding a mask to the **top** layer masks only the items in the layer directly below it (the **middle** layer), so the bug logo in the **bottom** layer remains visible at all times. Adding a mask also locks the **top** and **middle** layers to prevent further editing.

Now that the movie is complete, save it as `banner.fla` and test it with the Flash Player. The rainbow oval is visible through the text as it animates from left to right. The text in the bottom layer is visible in the portions not containing the rainbow (Fig. 16.34).

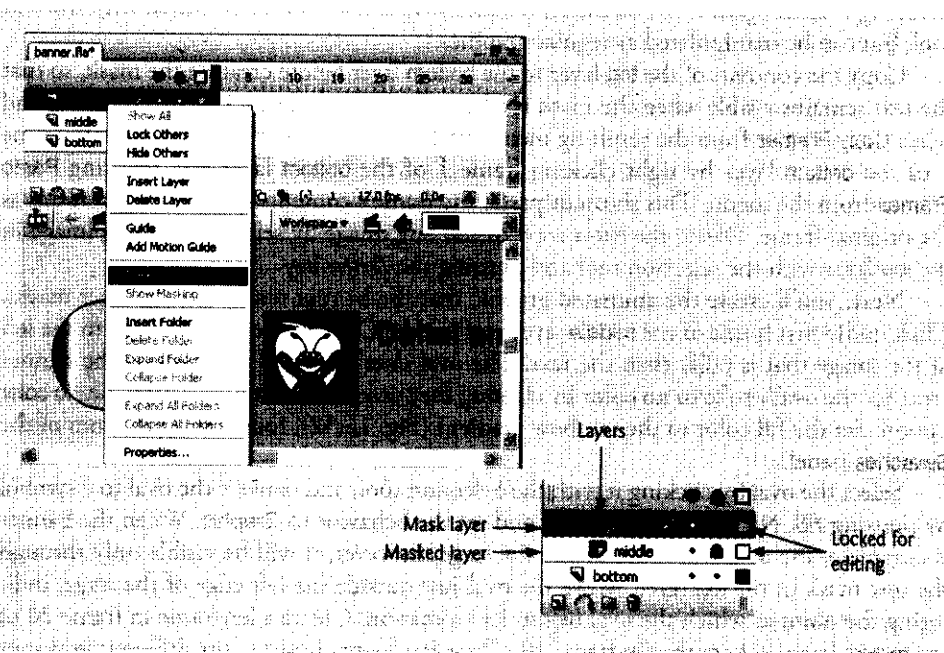


Fig. 16.33 | Creating a mask layer.

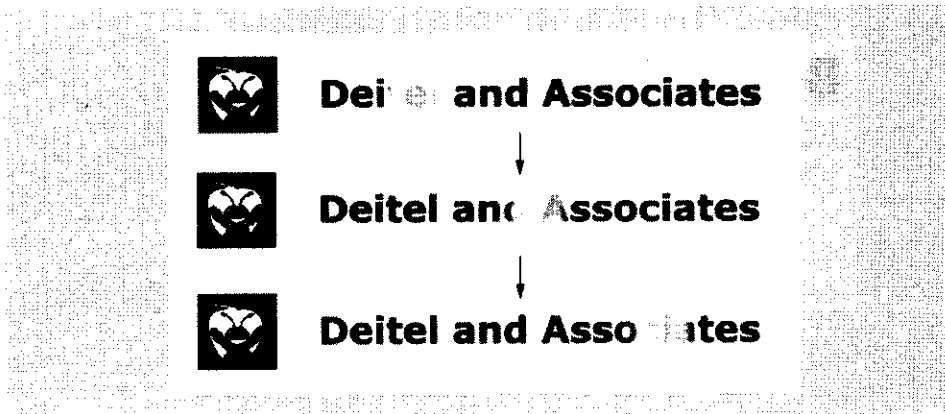


Fig. 16.34 | Completed banner.

16.5.3 Adding Online Help to Forms

In this section, we build on Flash techniques introduced earlier in this chapter, including tweening, masking, importing sound and bitmapped images, and writing ActionScript. In the following example, we apply these various techniques to create an online form that offers interactive help. The interactive help consists of animations that appear when a user presses buttons located next to the form fields. Each button contains a script that triggers an animation, and each animation provides the user with information regarding the form field that corresponds to the pressed button.

Each animation is a movie-clip symbol that is placed in a separate frame and layer of the scene. Adding a **stop** action to frame 1 pauses the movie until the user presses a button.

Begin by creating a new movie, using default movie size settings. Set the frame rate to 24 fps. The first layer will contain the site name, form title and form captions. Change the name of **Layer 1** to **text**. Add a **stop** action to frame 1 of the text layer. Create the site name **Bug2Bug.com** as static text in the **text** layer using a large, bold font, and place the title at the top of the page. Next, place the form name **Registration Form** as static text beneath the site name, using the same font, but in a smaller size and different color. The final text element added to this layer is the text box containing the form labels. Create a text box using the **Text Tool**, and enter the text: **Name:**, **Member #:** and **Password:**, pressing **Enter** after entering each label to put it on a different line. Next, adjust the value of the **Line Spacing** field (the amount of space between lines of text) found by clicking the **Edit Format Options** button (¶) in the **Properties** window. Change the form field caption line spacing to **22** in the **Format Options** dialog (Fig. 16.35) and set the text alignment (found in the **Properties** window) to right justify.

Now we'll create the form fields for our help form. The first step in the production of these form fields is to create a new layer named **form**. In the **form** layer, draw a rectangle that is roughly the same height as the caption text. This rectangle will serve as a background for the form text fields (Fig. 16.36). We set a **Rectangle corner radius** of 6 px in the **Properties** panel. Feel free to experiment with other shapes and colors.

The next step is to convert the rectangle into a symbol so that it may be reused in the movie. Select the rectangle fill and stroke with the selection tool and press **F8** to convert the selection to a symbol. Set the symbol behavior to **Graphic** and name the symbol **form**

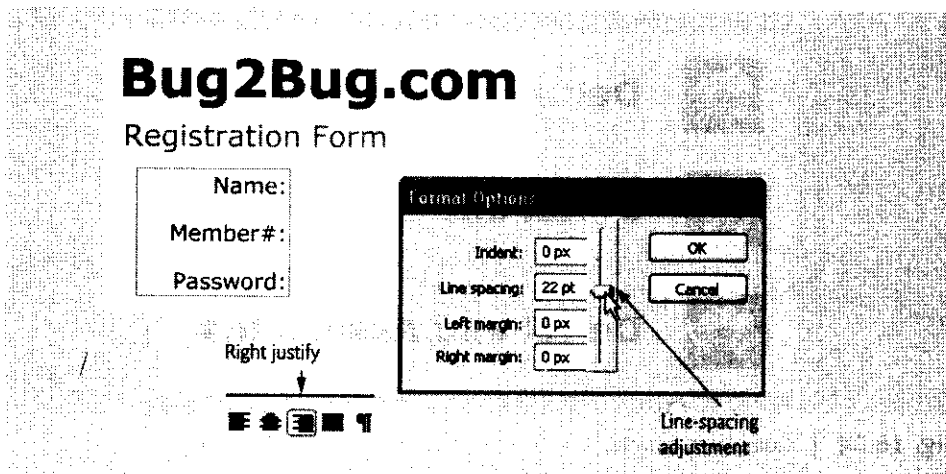


Fig. 16.35 | Adjusting the line spacing with the **Format Options** dialog.

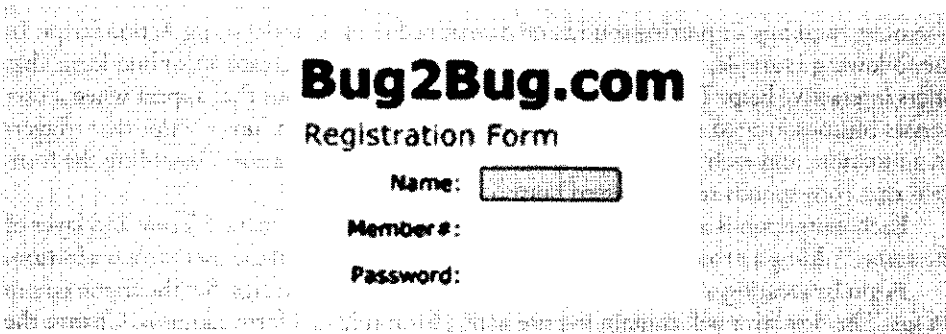


Fig. 16.36 | Creating a rectangle with rounded corners.

field. This symbol should be positioned next to the **Name:** caption. When the symbol is in place, open the **Library** panel by pressing **<Ctrl>-L**, select the **form** layer and drag two copies of the **form field** symbol from the **Library** onto the stage. This will create two new instances of this symbol. Use the Selection tool to align the fields with their corresponding captions. For more precise alignment, select the desired object with the Selection tool and press the arrow key on the keyboard in the direction you want to move the object. After alignment of the **form field** symbols, the movie should resemble Fig. 16.37.

We now add **input text fields** to our movie. An input text field is a text field into which the user can enter text. Select the Text tool and, using the **Properties** window, set the font to **Verdana**, **16 pt**, with dark blue as the color. In the **Text type** pull-down menu in the **Properties** window, select **Input Text** (Fig. 16.38). Then, click and drag in the stage to create a text field slightly smaller than the **form field** symbol we just created. With the Selection tool, position the text field over the instance of the **form field** symbol associated with the name. Create a similar text field for member number and password. Select the Password text field, and select **Password** in the Line type pull-down menu in the **Properties** window. Selecting **Password** causes any text entered into the field by the user to appear as an asterisk (*). We have now created all the input text fields for our help form. In this

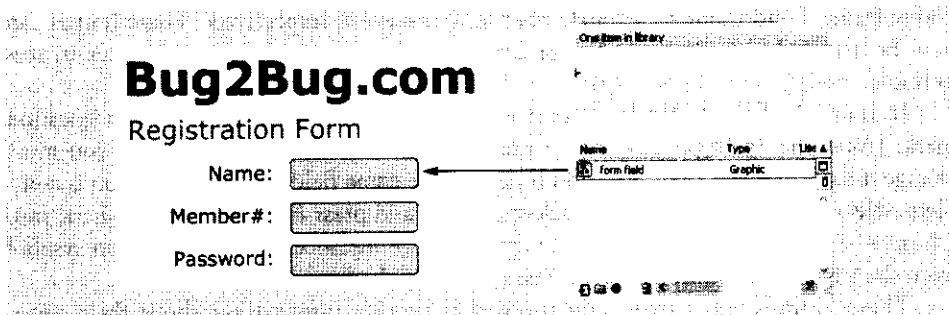


Fig. 16.37 | Creating multiple instances of a symbol with the Library panel.

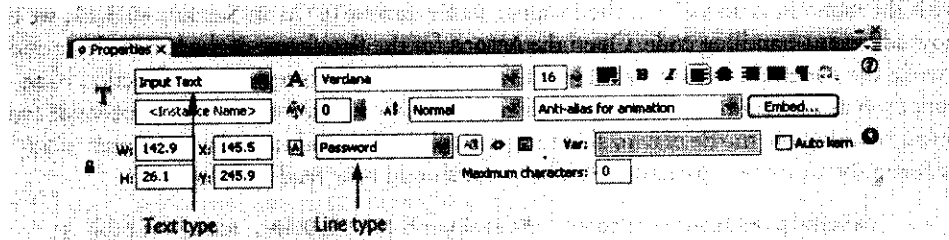


Fig. 16.38 | Input and password text-field creation.

example, we won't actually process the text entered into these fields. Using ActionScript, we could give each input text field a variable name, and send the values of these variables to a server-side script for processing.

Now that the form fields are in place, we can create the help associated with each field. Add two new layers. Name one layer **button** and the other **labels**. The **labels** layer will hold the **frame label** for each keyframe. A frame label is a text string that corresponds to a specific frame or series of frames. In the **labels** layer, create keyframes in frames 2, 3 and 4. Select frame 2 and enter **name** into the **Frame** field in the **Properties** window (Fig. 16.39).

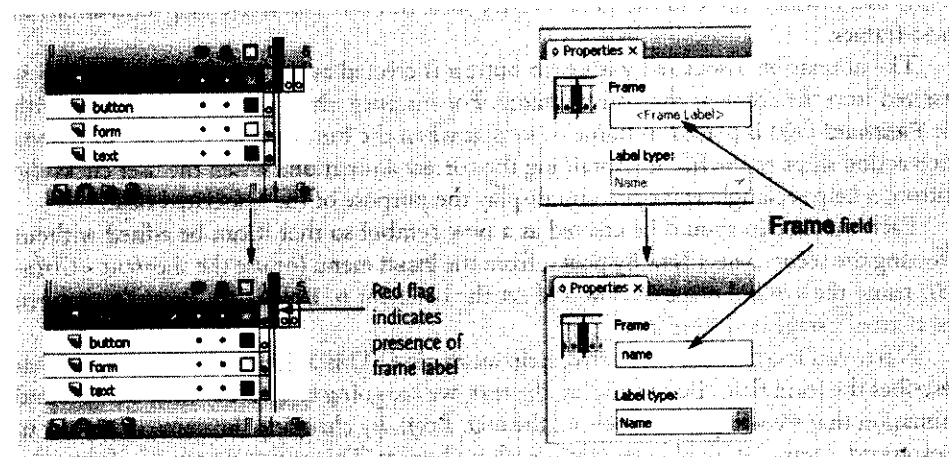


Fig. 16.39 | Adding Frame Labels using the Properties window.

Name frame 3 and frame 4 `memberNumber` and `password`, respectively. These frames can now be accessed either by number or by name. We use the labels again later in this example.

In frame 1 of the `button` layer, create a small circular button containing a question mark. [*Note:* the `Text type` property of the `Text Tool` will still be `Input Text`, so you must change it back to `Static Text`]. Position it next to the `name` field. When the button is complete, select all of its pieces with the selection tool, and press *F8* to convert the shape into a button symbol named `helpButton`. Drag two more instances of the `helpButton` symbol from the `Library` panel onto the stage next to each of the form fields.

These buttons will trigger animations that provide information about their corresponding form fields. A script will be added to each button so that the movie jumps to a particular frame when a user presses the button. Click the `helpButton` symbol associated with the `name` field and give it the instance name `nameHelp`. As in Section 16.3.11, we'll now add event-handling code. Open the `Actions` for the first frame of the `buttons` layer and invoke the `nameHelp` button's `addEventListener` function to register the function `nameFunction` as the handler of the mouse-click event. In `nameFunction`, add a `gotoAndStop` action, causing the movie play to skip to a particular frame and stop playing. Enter "name" between the function's parentheses. The script should now read as follows:

```
nameHelp.addEventListener( MouseEvent.MOUSE_DOWN, nameFunction );

function nameFunction( event : MouseEvent ) : void
{
    gotoAndStop( "name" );
}
```

When the user presses the `nameHelp` button, this script advances to the frame labeled `name` and stops. [*Note:* We could also have entered `gotoAndStop(2)`, referenced by frame number, in place of `gotoAndStop("name")`.] Add similar code for the `memberHelp` and `passwordHelp` buttons, changing the frame labels to `memberNumber` and `password`, the button instance names to `memberHelp` and `passwordHelp` and the function names to `memberFunction` and `passwordFunction`, respectively. Each button now has an action that points to a distinct frame in the timeline. We next add the interactive help animations to these frames.

The animation associated with each button is created as a movie-clip symbol that is inserted into the scene at the correct frame. For instance, the animation associated with the `Password` field is placed in frame 4, so that when the button is pressed, the `gotoAndStop` action skips to the frame containing the correct animation. When the user clicks the button, a help rectangle will open and display the purpose of the associated field.

Each movie clip should be created as a **new symbol** so that it can be edited without affecting the scene. Select **New Symbol...** from the **Insert** menu (or use the shortcut *<Ctrl>-F8*), name the symbol `nameWindow` and set the behavior to **Movie Clip**. Press **OK** to open the symbol's stage and timeline.

Next, you'll create the interactive help animation. This animation contains text that describes the form field. Before adding the text, we are going to create a small background animation that we will position behind the text. Begin by changing the name of **Layer 1** to **background**. Draw a dark blue rectangle with no border. This rectangle can be of any size, because we will customize its proportions with the **Properties** window. Select the rectangle

with the Selection tool, then open the **Properties** window. Set the **W:** field to **200** and the **H:** field to **120**, to define the rectangle's size. Next, center the rectangle by entering **-100** and **-60** into the **X:** and **Y:** fields, respectively (Fig. 16.40).

Now that the rectangle is correctly positioned, we can begin to create its animation. Add keyframes to frames 5 and 10 of the **background** layer. Use the **Properties** window to change the size of the rectangle in frame 5, setting its height to **5.0**. Next, right click frame 5 and select **Copy Frames**. Then right click frame 1 and select **Paste Frames**. While in frame 1, change the width of the rectangle to **5.0**.

The animation is created by applying shape tweening to frames 1 and 5. Recall that shape tweening morphs one shape into another. The shape tween causes the dot in frame 1 to grow into a line by frame 5, then into a rectangle in frame 10. Select frame 1 and apply the shape tween by right clicking frame 1 and selecting **Create Shape Tween**. Do the same for frame 5. Shape tweens appear green in the timeline (Fig. 16.41). Follow the same procedure for frame 5.

Now that this portion of the animation is complete, it may be tested on the stage by pressing **Enter**. The animation should appear as the dot from frame 1 growing into a line by frame 5 and into a rectangle by frame 10.

The next step is to add a mock form field to this animation which demonstrates what the user would type in the actual field. Add two new layers above the **background** layer, named **field** and **text**. The **field** layer contains a mock form field, and the **text** layer contains the help information.

First, create an animation similar to the growing rectangle we just created for the mock form field. Add a keyframe to frame 10 in both the **field** and **text** layers. Fortunately, we have a form field already created as a symbol. Select frame 10 of the **field** layer, and drag

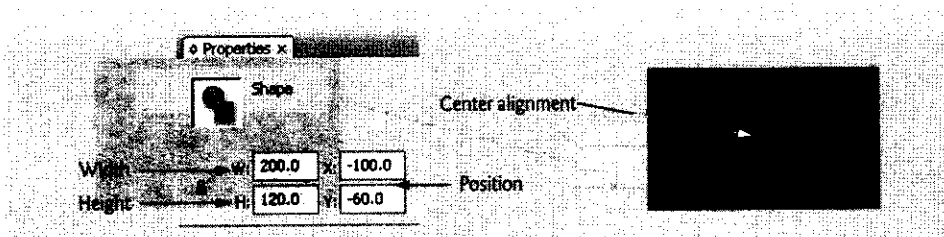


Fig. 16.40 | Centering an image on the stage with the **Properties** window.

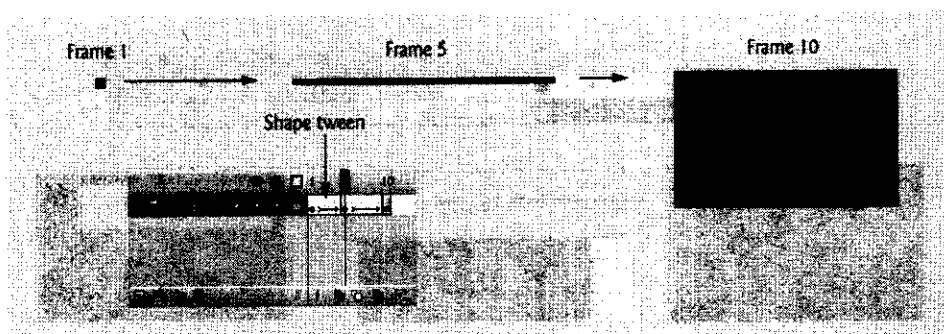


Fig. 16.41 | Creating a shape tween.

the **form field** symbol from the **Library** panel onto the stage, placing it within the current movie clip. Symbols may be embedded in one another; however, they cannot be placed within themselves (i.e., an instance of the **form field** symbol cannot be dragged onto the **form field** symbol editing stage). Align the **form field** symbol with the upper-left corner of the background rectangle, as shown in Fig. 16.42.

Next, set the end of this movie clip by adding keyframes to the **background** and **field** layers in frame 40. Also add keyframes to frames 20 and 25 of the **field** layer. These keyframes define intermediate points in the animation. Refer to Fig. 16.43 for correct keyframe positioning.

The next step in creating the animation is to make the **form field** symbol grow in size. Select frame 20 of the **field** layer, which contains only the **form field** symbol. Next open the **Transform** panel from the **Window** menu. The **Transform** panel can be used to change an object's size. Check the **Constrain** checkbox to constrain the object's proportions as it is resized. Selecting this option causes the **scale factor** to be equal in the width and height fields. The **scale factor** measures the change in proportion. Set the **scale factor** for the width and height to **150%**, and press *Enter* to apply the changes. Repeat the previous step for frame 10 of the **field** layer, but scale the **form field** symbol down to **0%**.

The symbol's animation is created by adding a motion tween. Adding the motion tween to **field** layer frames 10 and 20 will cause the **form field** symbol to grow from 0% of the original size to 150%, then to 100%. Figure 16.43 illustrates this portion of the animation.

Next, you'll add text to the movie clip to help the user understand the purpose of the corresponding text field. You'll set text to appear over the **form field** symbol as an example to the user. The text that appears below the **form field** symbol tells the user what should be typed in the text field.

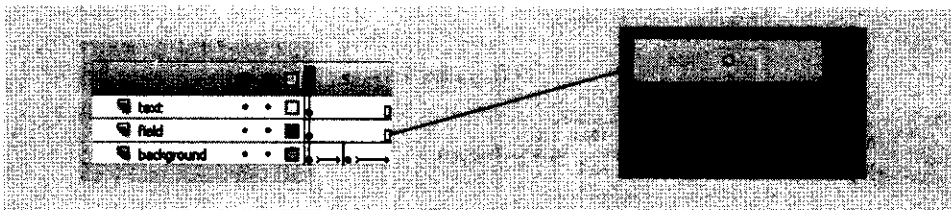


Fig. 16.42 | Adding the **field** symbol to the **nameWindow** movie clip.

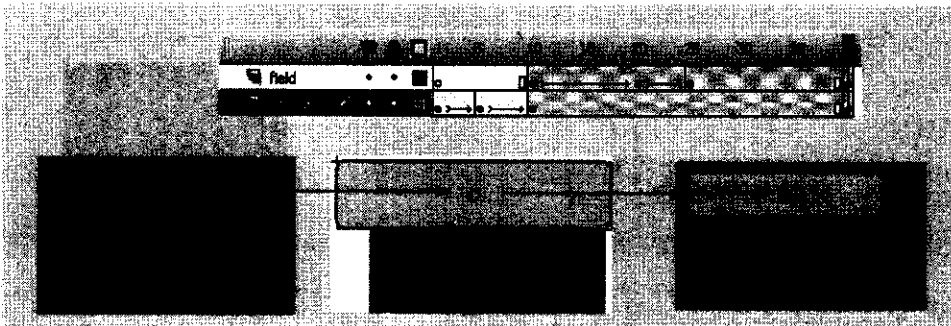


Fig. 16.43 | Creating an animation with the **form field** symbol.

To add the descriptive text, first insert a keyframe in frame 25 of the **text** layer. Use the Text tool (white, Arial, 16 pt text and 3 pt line spacing) to type the help information for the **Name** field. Make sure the text type is **Static Text**. This text will appear in the help window. For instance, our example gives the following directions for the **Name** field: **Enter your name in this field. First name, Last name.** Align this text with the left side of the rectangle. Next, add a keyframe to frame 40 of this layer, causing the text to appear throughout the animation.

Next, duplicate this movie clip so that it may be customized and reused for the other two help button animations. Open the **Library** panel and right click the **nameWindow** movie clip. Select **Duplicate** from the resulting menu, and name the new clip **passwordWindow**. Repeat this step once more, and name the third clip **memberWindow** (Fig. 16.44).

You must customize the duplicated movie clips so their text reflects the corresponding form fields. To begin, open the **memberWindow** editing stage by pressing the **Edit Symbols** button, which is found in the upper-right corner of the editing environment, and selecting **memberWindow** from the list of available symbols (Fig. 16.44). Select frame 25 of the **text** layer and change the form field description with the Text tool so that the box contains the directions **Enter your member number here in the form: 556677**. Copy the text in frame 25 by selecting it with the Text tool and using the shortcut **<Ctrl>-C**. Click frame 40 of the **text** layer, which contains the old text. Highlight the old text with the Text tool, and use the shortcut **<Ctrl>-V** to paste the copied text into this frame. Repeat these steps for the **passwordWindow** movie clip using the directions **Enter your secret password in this field**. [Note: Changing a symbol's function or appearance in its editing stage updates the symbol in the scene.]

The following steps further customize the help boxes for each form field. Open the **nameWindow** symbol's editing stage by clicking the **Edit Symbols** button (Fig. 16.44) and selecting **nameWindow**. Add a new layer to this symbol called **typedText** above the **text** layer. This layer contains an animation that simulates the typing of text into the form field.

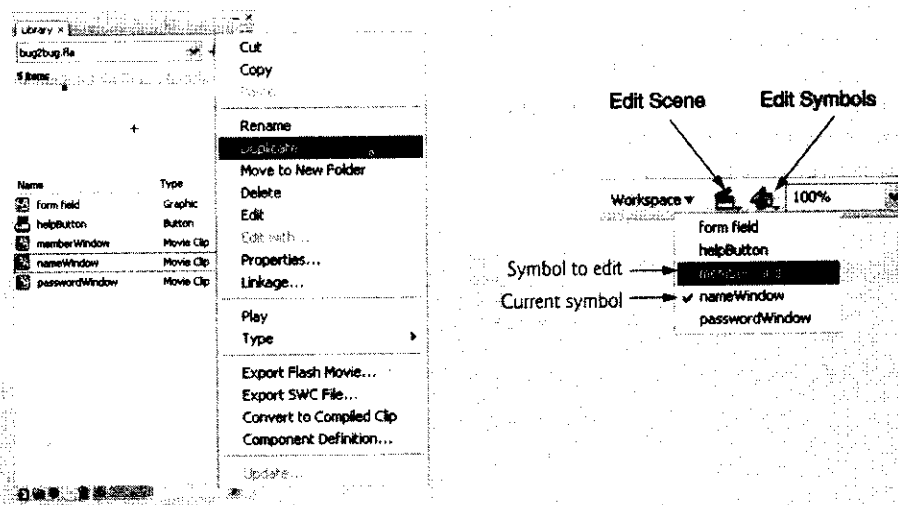


Fig. 16.44 | Duplicating movie-clip symbols with the **Library** panel.

Insert a keyframe in frame 25. Select this frame and use the Text tool to create a text box on top of the **form field** symbol. Type the name **John Doe** in the text box, then change the text color to black.

The following frame-by-frame animation creates the appearance of the name being typed into the field. Add a keyframe to frame 40 to indicate the end of the animation. Then add new keyframes to frames 26–31. Each keyframe contains a new letter being typed in the sequence, so when the playhead advances, new letters appear. Select the **John Doe** text in frame 25 and delete everything except the first **J** with the Text tool. Next, select frame 26 and delete all of the characters except the **J** and the **o**. This step must be repeated for all subsequent keyframes up to frame 31, each keyframe containing one more letter than the last (Fig. 16.45). Frame 31 should show the entire name. When this process is complete, press *Enter* to preview the frame-by-frame typing animation.

Create the same type of animation for both the **passwordWindow** and the **memberWindow** movie clips, using suitable words. For example, we use six asterisks for the **passwordWindow** movie clip and six numbers for the **memberWindow** movie clip. Add a stop action to frame 40 of all three movie clips so that the animations play only once.

The movie clips are now ready to be added to the scene. Click the **Edit Scene** button next to the **Edit Symbols** button, and select **Scene 1** to return to the scene. Before inserting the movie clips, add the following layers to the timeline: **nameMovie**, **memberMovie** and **passwordMovie**, one layer for each of the movie clips. Add a keyframe in frame 2 of the **nameMovie** layer. Also, add keyframes to frame 4 of the **form**, **text** and **button** layers, ensuring that the form and text appear throughout the movie.

Now you'll place the movie clips in the correct position in the scene. Recall that the ActionScript for each help button contains the script

```
function functionName( event : MouseEvent ) : void
{
    gotoAndStop( frameLabel );
}
```

in which *functionName* and *frameLabel* depend on the button. This script causes the movie to skip to the specified frame and stop. Placing the movie clips in the correct frames causes the playhead to skip to the desired frame, play the animation and stop. This effect is created by selecting frame 2 of the **nameMovie** layer and dragging the **nameWindow** movie clip onto the stage. Align the movie clip with the button next to the **Name** field, placing it half-way between the button and the right edge of the stage.

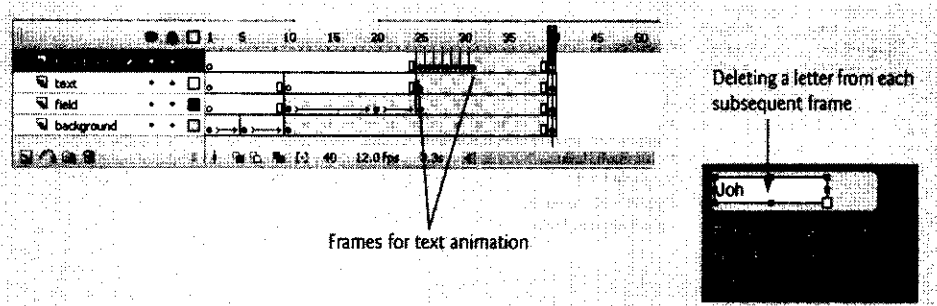


Fig. 16.45 | Creating a frame-by-frame animation.

The preceding step is repeated twice for the other two movie clips so that they appear in the correct frames. Add a keyframe to frame 3 of the **memberMovie** layer and drag the **memberWindow** movie clip onto the stage. Position this clip in the same manner as the previous clip. Repeat this step for the **passwordWindow** movie clip, dragging it into frame 4 of the **passwordMovie** layer.

The movie is now complete. Press **<Ctrl>-Enter** to preview it with the Flash Player. If the triggered animations do not appear in the correct locations, return to the scene and adjust their position. The final movie is displayed in Fig. 16.46.

In our example, we have added a picture beneath the text layer. Movies can be enhanced in many ways, such as by changing colors and fonts or by adding pictures. Our movie (`bug2bug.f1a`) can be found in the this chapter's examples directory. If you want to use our symbols to recreate the movie, select **Open External Library...** from the **Import** submenu of the **File** menu and open `bug2bug.f1a`. The **Open External Library...** option allows you to reuse symbols from another movie.

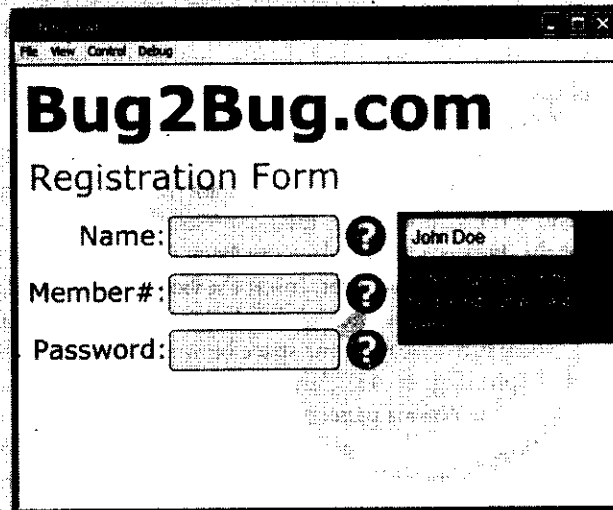


Fig. 16.46 | Bug2Bug.com help form.

16.6 Creating a Website Splash Screen

Flash is becoming an important tool for e-businesses. Many organizations use Flash to create website splash screens (i.e., introductions), product demos and web applications. Others use Flash to build games and interactive entertainment in an effort to attract new visitors. However, these types of applications can take a long time to load, causing visitors—especially those with slow connections—to leave the site. One way to alleviate this problem is to provide visitors with an animated Flash introduction that draws and keeps their attention. Flash animations are ideal for amusing visitors while conveying information as the rest of a page downloads “behind the scenes.”

A **preloader** or **splash screen** is a simple animation that plays while the rest of the web page is loading. Several techniques are used to create animation preloaders. The following

example creates an animation preloader that uses ActionScript to pause the movie at a particular frame until all the movie elements have loaded.

To start building the animation preloader, create a new Flash document. Use the default size, and set the background color to a light blue. First, you'll create the movie pieces that will be loaded later in the process. Create five new layers, and rename **Layer 2** to **C++**, **Layer 3** to **Java** and **Layer 4** to **IW3**. **Layer 5** will contain the movie's ActionScript, so rename it **actions**. Because **Layer 1** contains the introductory animation, rename this layer **animation**.

The preloaded objects we use in this example are animated movie clip symbols. Create the first symbol by clicking frame 2 of the **C++** layer, inserting a keyframe, and creating a new movie-clip symbol named **cppbook**. When the symbol's editing stage opens, import the image **cpphttp.gif** (found in the **images** folder with this chapter's examples). Place a keyframe in frame 20 of **Layer 1** and add a **stop** action to this frame. The animation in this example is produced with the motion tween **Rotate** option, which causes an object to spin on its axis. Create a motion tween in frame 1 with the **Properties** window, setting the **Rotate** option to **CCW** (counterclockwise) and the **times** field to **2** (Fig. 16.47). This causes the image **cpphttp.gif** to spin two times counterclockwise over a period of 20 frames.

After returning to the scene, drag and drop a copy of the **cppbook** symbol onto the stage in frame 2 of the **C++** layer. Move this symbol to the left side of the stage. Insert a frame in frame 25 of the **C++** layer.

Build a similar movie clip for the **Java** and **IW3** layers, using the files **java.gif** and **iw3.gif** to create the symbols. Name the symbol for the **Java** layer **jbook** and the **IW3** symbol **ibook** to identify the symbols with their contents. In the main scene, create a keyframe in frame 8 of the **Java** layer, and place the **jbook** symbol in the center of the stage. Insert a frame in frame 25 of the **Java** layer. Insert the **ibook** symbol in a keyframe in frame 14 of the **IW3** layer, and position it to the right of the **jbook** symbol. Insert a frame in frame 25 of the **IW3** layer. Make sure to leave some space between these symbols so that they will not overlap when they spin (Fig. 16.48). Add a keyframe to the 25th frame of the **actions** layer, then add a **stop** to the **Actions** panel of that frame.

Now that the loading objects have been placed, it is time to create the preloading animation. By placing the preloading animation in the frame preceding the frame that contains the objects, we can use ActionScript to pause the movie until the objects have loaded. Begin by adding a **stop** action to frame 1 of the **actions** layer. Select frame 1 of the **animation** layer and create another new movie-clip symbol named **loader**. Use the text tool with a medium-sized sans-serif font, and place the word **Loading** in the center of the symbol's editing stage.

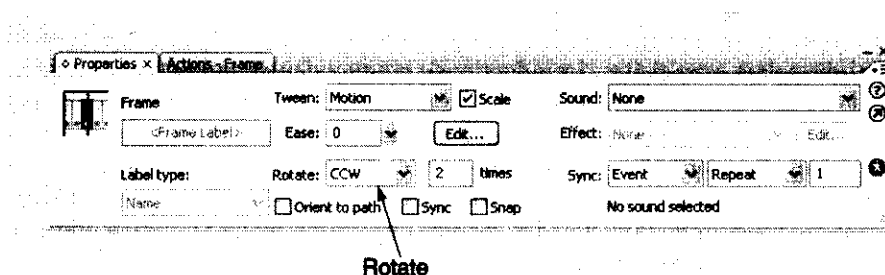


Fig. 16.47 | Creating a rotating object with the motion tween **Rotate** option.



Fig. 16.48 | Inserted movie clips.

This title indicates to the user that objects are loading. Insert a keyframe into frame 14 and rename this layer **load**.

Create a new layer called **orb** to contain the animation. Draw a circle with no stroke about the size of a quarter above the word **Loading**. Give the circle a green-to-white radial gradient fill color. The colors of this gradient can be edited in the **Color** panel (Fig. 16.49).

The block farthest to the left on the **gradient range** indicates the innermost color of the radial gradient, whereas the block farthest to the right indicates the outermost color of the radial gradient. Click the left block to reveal the **gradient color swatch**. Click the swatch and select a medium green as the inner color of the gradient. Select the right, outer color box and change its color to white. Deselect the circle by clicking on a blank portion of the stage. Note that a white ring appears around the circle due to the colored background. To make the circle fade into the background, we adjust its **alpha** value. Alpha is a value between 0 and 100% that corresponds to a color's transparency or opacity. An alpha value of 0% appears transparent, whereas a value of 100% appears completely opaque. Select the circle again and click the right gradient box (white). Adjust the value of the **Alpha** field in the **Color Mixer** panel to 0%. Deselect the circle. It should now appear to fade into the background.

The rate of progression in a gradient can also be changed by sliding the color boxes. Select the circle again. Slide the left color box to the middle so that the gradient contains more green than transparent white, then return the slider to the far left. Intermediate colors may be added to the gradient range by clicking beneath the bar, next to one of the existing

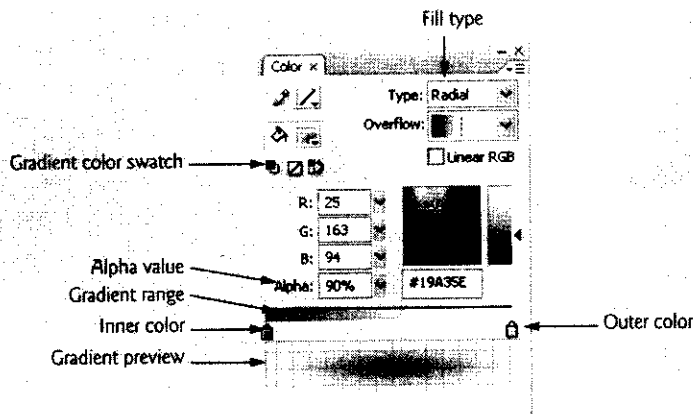


Fig. 16.49 | Changing gradient colors with the **Color** panel.

color boxes. Click to the right of the inner color box to add a new color box (Fig. 16.50). Slide the new color box to the right and change its color to a darker green. Any color box may be removed from a gradient by dragging it downward off the gradient range.

Insert keyframes into frame 7 and 14 of the **orb** layer. Select the circle in frame 7 with the selection tool. In the **Color** panel change the alpha of every color box to 0%. Select frame 1 in the **Timeline** and add shape tween. Change the value of the **Ease** field in the **Properties** window to **-100**. **Ease** controls the rate of change during tween animation. Negative values cause the animated change to be gradual at the beginning and become increasingly drastic. Positive values cause the animation to change quickly in the first frames, becoming less drastic as the animation progresses. Add shape tween to frame 7 and set the **Ease** value to 100. In frame 14, add the action `gotoAndPlay(1)` to repeat the animation. You can preview the animation by pressing **Enter**. The circle should now appear to pulse.

Before inserting the movie clip into the scene, we are going to create a **hypertext linked button** that will enable the user to skip over the animations to the final destination. Add a new layer called **link** to the **loader** symbol with keyframes in frames 1 and 14. Using the text tool, place the words **skip directly to Deitel website** below **Loading** in a smaller font size. Select the words with the selection tool and convert them into a button symbol named **skip**. Converting the text into a button simulates a text hyperlink created with XHTML. Double click the words to open the **skip** button's editing stage. For this example, we are going to edit only the hit state. When a button is created from a shape, the button's hit area is, by default, the area of the shape. It is important to change the hit state of a button created from text so that it includes the spaces between the letters; otherwise, the link will work only when the user hovers over a letter's area. Insert a keyframe in the hit state. Use the rectangle tool to draw the hit area of the button, covering the entire length and height of the text. This rectangle is not visible in the final movie, because it defines only the hit area (Fig. 16.51).

The button is activated by giving it an action that links it to another web page. After returning to the **loader** movie-clip editing stage, give the **skip** button the instance name **skipButton** and open the **Actions** panel for the first frame of the **link** layer. Invoke the `addEventListener` function using the **skipButton** instance to call function `onClick` whenever

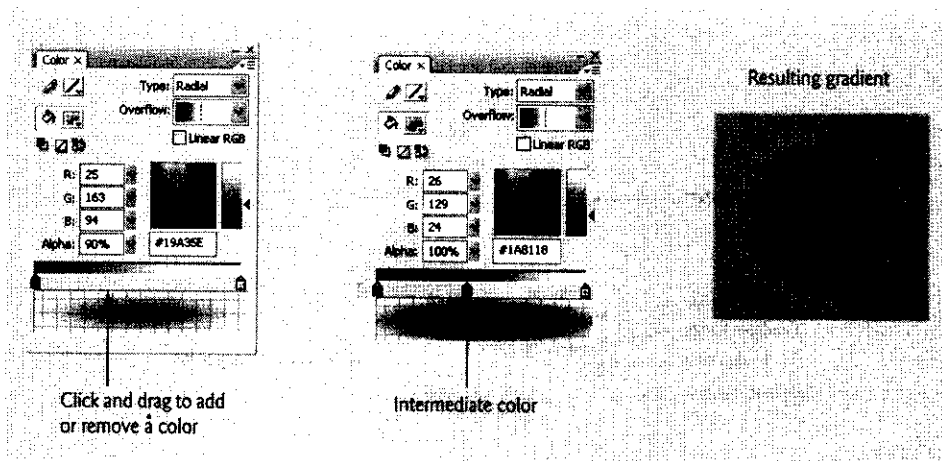


Fig. 16.50 | Adding an intermediate color to a gradient.



Fig. 16.51 | Defining the hit area of a button.

the button is clicked. Then, create an object of type `URLRequest` and give the constructor a parameter value of `"http://www.deitel.com"`. The function `onClick` employs Flash's `navigateToURL` function to access the website given to it. Thus, the code now reads

```
skipButton.addEventListener( MouseEvent.CLICK, onClick );
var url : URLRequest = new URLRequest( "http://www.deitel.com" );
function onClick( e : MouseEvent ) : void
{
    navigateToURL( url, "_blank" );
} // end function onClick
```

The `"_blank"` parameter signifies that a new browser window displaying the Deitel website should open when the user presses the button.

Return to the scene by clicking **Scene 1** directly below the timeline, next to the name of the current symbol. Drag and drop a copy of the **loader** movie clip from the **Library** panel into frame 1 of the **animation** layer, center it on the stage, and set its **Instance name** to **loadingClip**.

The process is nearly complete. Open the **Actions** panel for the **actions** layer. The following actions direct the movie clip to play until all the scene's objects are loaded. First, add a **stop** to the frame so that it doesn't go to the second frame until we tell it to. Using the **loadingClip** movie instance, use the `addEventListener` function to invoke the function `onBegin` whenever the event `Event.ENTER_FRAME` is triggered. The `ENTER_FRAME` event occurs every time the playhead enters a new frame. Since this movie's frame rate is 12 fps (frames per second), the `ENTER_FRAME` event will occur 12 times each second.

```
loadingClip.addEventListener( Event.ENTER_FRAME, onBegin );
```

The next action added to this sequence is the function `onBegin`. The condition of the `if` statement will be used to determine how many frames of the movie are loaded. Flash movies load frame by frame. Frames that contain complex images take longer to load. Flash will continue playing the current frame until the next frame has loaded. For our movie, if the number of frames loaded (`framesLoaded`) is equal to the total number of frames (`totalFrames`), then the movie is finished loading, so it will play frame 2. It also invokes the `removeEventListener` function to ensure that `onBegin` is not called for the remainder of the movie. If the number of frames loaded is less than the total number of frames, then the current movie clip continues to play. The code now reads:

```
stop();
loadingClip.addEventListener( Event.ENTER_FRAME, onBegin );
// check if all frames have been loaded
function onBegin( event : Event ) : void
{
    if ( framesLoaded == totalFrames )
    {
```

```

        LoadingClip.removeEventListener( Event.ENTER_FRAME, onBegin );
        gotoAndPlay( 2 );
    } // end if
} // end function onBegin

```

Create one more layer in the scene, and name the layer **title**. Add a keyframe to frame 2 of this layer, and use the Text tool to create a title that reads **Recent Deitel Publications**. Below the title, create another text hyperlink button to the Deitel website. The simplest way to do this is to duplicate the existing **skip** button and modify the text. Right click the **skip** symbol in the **Library** panel, and select **Duplicate**. Name the new button **visit**, and place it in frame 2 of the **title** layer. Label the instance **visitButton**, then create a keyframe in the second frame of the **actions** layer. Duplicate the code from the **Actions** panel of the first frame of the **link** layer in the **loader** symbol, and replace **skipButton** with **visitButton**. Double click the **visit** button and edit the text to say **visit the Deitel website**. Add keyframes to each frame of the **title** layer and manipulate the text to create a typing effect similar to the one we created in the **bug2bug** example.

The movie is now complete. Test the movie with the Flash Player (Fig. 16.52). When viewed in the testing window, the loading sequence will play for only one frame because your processor loads all the frames almost instantly. Flash can simulate how a movie would appear to an online user, though. While still in the testing window, select **56K** from the **Download Settings** submenu of the **View** menu. Also, select **Bandwidth Profiler** from the **View** menu. Then select **Simulate Download** from the **View** menu or press **<Ctrl>-Enter**. The graph at the top of the window displays the amount of bandwidth required to load each frame.

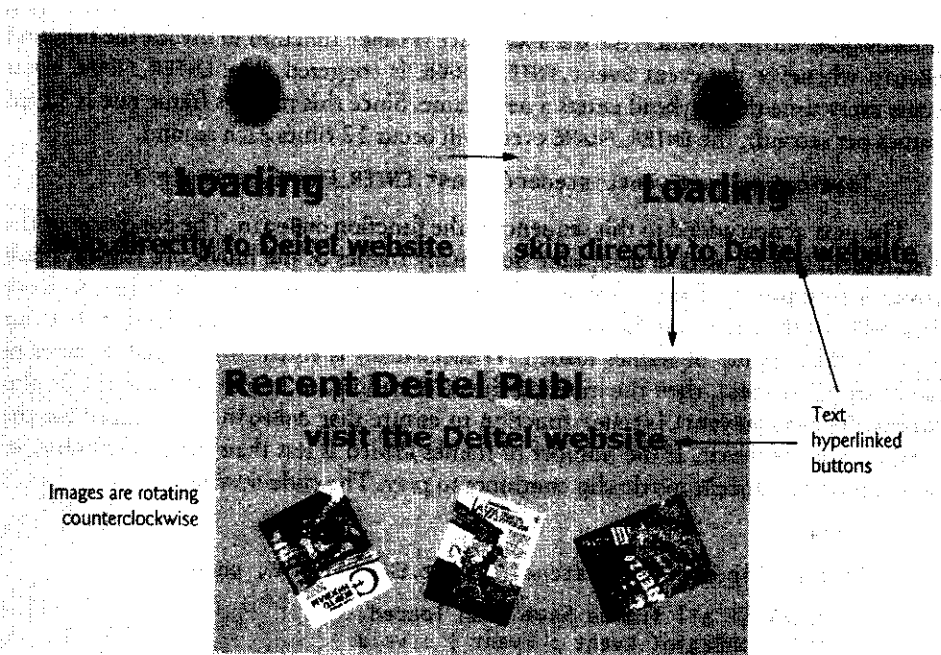


Fig. 16.52 | Creating an animation to preload images.

16.7 ActionScript

Figure 16.53 lists common Flash ActionScript 3.0 functions. By attaching these functions to frames and symbols, you can build some fairly complex Flash movies.

<code>gotoAndPlay</code>	Jump to a frame or scene in another part of the movie and start playing the movie.
<code>gotoAndStop</code>	Jump to a frame or scene in another part of the movie and stop the movie.
<code>play</code>	Start playing a movie from the beginning, or from wherever it has been stopped.
<code>stop</code>	Stop a movie.
<code>SoundMixer.stopAll</code>	Stop the sound track without affecting the movie.
<code>navigateToUrl</code>	Load a URL into a new or existing browser window.
<code>fscommand</code>	Insert JavaScript or other scripting languages into a Flash movie.
<code>Loader class</code>	Load a SWF or JPEG file into the Flash Player from the current movie. Can also load another SWF into a particular movie.
<code>FramesLoaded</code>	Check whether certain frames have been loaded.
<code>addEventListener</code>	Assign functions to a movie clip based on specific events. The events include <code>load</code> , <code>unload</code> , <code>enterFrame</code> , <code>mouseUp</code> , <code>mouseDown</code> , <code>mouseMove</code> , <code>keyUp</code> , <code>keyDown</code> and <code>data</code> .
<code>if</code>	Set up condition statements that run only when the condition is true.
<code>while/do while</code>	Run a collection of statements while a condition statement is true.
<code>trace</code>	Display programming notes or variable values while testing a movie.
<code>Math.random</code>	Returns a random number less than or equal to 0 and less than 1.

Fig. 16.53 | Common ActionScript functions.

16.8 Web Resources

www.deitel.com/flash9/

The Deitel Flash 9 Resource Center contains links to some of the best Flash 9 and Flash CS3 resources on the web. There you'll find categorized links to forums, conferences, blogs, books, open source projects, videos, podcasts, webcasts and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on Microsoft Silverlight (www.deitel.com/silverlight/) and Adobe Flex (www.deitel.com/flex/).

Summary

Section 16.1 Introduction

- Adobe *Flash CS3* (Creative Suite 3) is a commercial application that you can use to produce interactive, animated movies.
- Flash can be used to create web-based banner advertisements, interactive websites, games and web-based applications with stunning graphics and multimedia effects.
- Flash movies can be embedded in web pages, placed on CDs or DVDs as independent applications or converted into stand alone, executable programs.
- Flash includes tools for coding in its scripting language, ActionScript 3.0. ActionScript, which is similar to JavaScript, enables interactive applications.
- To play Flash movies, the Flash Player plug-in must be installed in your web browser. This plug-in has several versions, the most recent of which is version 9.

Section 16.2 Flash Movie Development

- The stage is the white area in which you place graphic elements during movie development. Only objects in this area will appear in the final movie.
- The timeline represents the time period over which a movie runs.
- Each frame depicts a moment in the movie's timeline, into which you can insert movie elements.
- The playhead indicates the current frame.
- The Tools bar is divided into multiple sections, each containing tools and functions that help you create Flash movies.
- Windows called panels organize frequently used movie options. Panel options modify the size, shape, color, alignment and effects associated with a movie's graphic elements.
- The context-sensitive Properties panel displays information about the currently selected object. It is a useful tool for viewing and altering an object's properties.
- You can access different panels by selecting them from the Window menu.

Section 16.3 Learning Flash with Hands-On Examples

- The .fla file extension is a Flash-specific extension for editable movies.
- Frame rate sets the speed at which movie frames display.
- The background color determines the color of the stage.
- Dimensions define the size of a movie as it displays on the screen.

Section 16.3.1 Creating a Shape with the Oval Tool

- Flash creates shapes using vectors—mathematical equations that define the shape's size, shape and color. When vector graphics are saved, they are stored using equations.
- Vector graphics can be resized without losing clarity.
- You can create shapes by dragging with the shape tools.
- Every shape has a stroke color and a fill color. The stroke color is the color of a shape's outline, and the fill color is the color that fills the shape.
- Clicking the Black and white button resets the stroke color to black and the fill color to white.
- Selecting the Swap colors option switches the stroke and fill colors.
- The Shift key constrains a shape's proportions to have equal width and height.
- A dot in a frame signifies a keyframe, which indicates a point of change in a timeline.

- A shape's size can be modified with the **Properties** panel when the shape is selected.
- Gradient fills are gradual progressions of color.
- The **Swatches** panel provides four radial gradients and three linear gradients.

Section 16.3.2 Adding Text to a Button

- Button titles communicate a button's function to the user. You can create a title with the **Text** tool.
- With selected text, you can change the font, text size and font color with the **Properties** window.
- To change the font color, click the text color swatch and select a color from the palette.

Section 16.3.3 Converting a Shape into a Symbol

- The scene contains graphics and symbols. The parent movie may contain several symbols that are reusable movie elements, such as graphics, buttons and movie clips.
- A scene timeline can contain numerous symbols with their own timelines and properties.
- A scene may have several instances of any given symbol.
- Symbols can be edited independently of the scene by using the symbol's editing stage. The editing stage is separate from the scene stage and contains only one symbol.
- Selecting **Convert to Symbol...** from the **Modify** menu or using the shortcut **F8** on the keyboard opens the **Convert to Symbol** dialog, in which you can set the properties of a new symbol.
- Every symbol in a Flash movie must have a unique name.
- You can create three different types of symbols—movie clips, buttons and graphics.
- A movie-clip symbol is ideal for recurring animations.
- Graphic symbols are ideal for static images and basic animations.
- Button symbols are objects that perform button actions, such as rollovers and hyperlinking. A rollover is an action that changes the appearance of a button when the mouse passes over it.
- The **Library** panel stores every symbol in a movie and is accessed through the **Window** menu or by the shortcuts **<Ctrl>-L** or **F11**. Multiple instances of a symbol can be placed in a movie by dragging and dropping the symbol from the **Library** panel onto the stage.

Section 16.3.4 Editing Button Symbols

- The different components of a button symbol, such as its fill and type, may be edited in the symbol's editing stage. You may access a symbol's editing stage by double clicking the symbol in the **Library** or by pressing the **Edit Symbols** button and selecting the symbol name.
- The pieces that make up a button can all be changed in the editing stage.
- A button symbol's timeline contains four frames, one for each of the button states (up, over and down) and one for the hit area.
- The up state (indicated by the **Up** frame on screen) is the default state before the user presses the button or rolls over it with the mouse.
- Control shifts to the over state (i.e., the **Over** frame) when the mouse moves over the button.
- The button's down state (i.e., the **Down** frame) plays when a user presses a button. You can create interactive, user-responsive buttons by customizing the appearance of a button in each state.
- Graphic elements in the hit state (i.e., the **Hit** frame) are not visible when viewing the movie; they exist simply to define the active area of the button (i.e., the area that can be clicked).
- By default, buttons only have the up state activated when they are created. You may activate other states by adding keyframes to the other three frames.

Section 16.3.5 Adding Keyframes

- Keyframes are points of change in a Flash movie and appear in the timeline as gray with a black dot. By adding keyframes to a button symbol's timeline, you can control how the button reacts to user input.
- A rollover is added by inserting a keyframe in the button's **Over** frame, then changing the button's appearance in that frame.
- Changing the button color in the over state does not affect the button color in the up state.

Section 16.3.6 Adding Sound to a Button

- Flash imports sounds in the WAV (Windows), AIFF (Macintosh) or MP3 formats.
- Sounds can be imported into the Library by choosing **Import to Library** from the **Import** submenu of the **File** menu.
- You can add sound to a movie by placing the sound clip in a keyframe or over a series of frames.
- If a frame has a blue wave or line through it, a sound effect has been added to it.

Section 16.3.7 Verifying Changes with Test Movie

- Movies can be viewed in their published state with the Flash Player. The published state of a movie is how it would appear if viewed over the web or with the Flash Player.
- Published Flash movies have the Shockwave Flash extension (.swf). SWF files can be viewed but not edited.

Section 16.3.8 Adding Layers to a Movie

- A movie can be composed of many layers, each having its own attributes and effects.
- Layers organize different movie elements so that they can be animated and edited separately, making the composition of complex movies easier. Graphics in higher layers appear over the graphics in lower layers.
- Text can be broken apart or regrouped for color editing, shape modification or animation. However, once text has been broken apart, it may not be edited with the Text tool.

Section 16.3.9 Animating Text with Tweening

- Animations in Flash are created by inserting keyframes into the timeline.
- Tweening, also known as morphing, is an automated process in which Flash creates the intermediate steps of the animation between two keyframes.
- Shape tweening morphs an ungrouped object from one shape to another.
- Motion tweening moves symbols or grouped objects around the stage.
- Keyframes must be designated in the timeline before adding the motion tween.
- Adding the stop function to the last frame in a movie stops the movie from looping.
- The small letter **a** in a frame indicates that it contains an action.

Section 16.3.10 Adding a Text Field

- **Static Text** creates text that does not change.
- **Dynamic Text** creates text that can be changed or determined by outside variables through ActionScript.
- **Input Text** creates a text field into which the viewers of the movie can input their own text.

Section 16.3.11 Adding ActionScript

- The `addEventListener` function helps make an object respond to an event by calling a function when the event takes place.

- `MouseEvent.CLICK` specifies that an action is performed when the user clicks the button.
- `Math.random` returns a random floating-point number from 0.0 up to, but not including, 1.0.

Section 16.4 Publishing Your Flash Movie

- Flash movies must be published for users to view them outside Flash CS3 and the Flash Player.
- Flash movies may be published in a different Flash version to support older Flash Players.
- Flash can automatically generate an XHTML document that embeds your Flash movie.

Section 16.5.1 Importing and Manipulating Bitmaps

- Once an imported image is broken apart, it may be shape tweened or edited with editing tools such as the Lasso, Paint bucket, Eraser and Paintbrush. The editing tools are found in the toolbox and apply changes to a shape.
- Dragging with the Lasso tool selects areas of shapes. The color of a selected area may be changed, or the selected area may be moved.
- Once an area is selected, its color may be changed by selecting a new fill color with the fill swatch or by clicking the selection with the Paint bucket tool.
- The Eraser tool removes shape areas when you click and drag the tool across an area. You can change the eraser size using the tool options.

Section 16.5.2 Creating an Advertisement Banner with Masking

- Masking hides portions of layers. A masking layer hides objects in the layers beneath it, revealing only the areas that can be seen through the shape of the mask.
- Items drawn on a masking layer define the mask's shape and cannot be seen in the final movie.
- The Free transform tool allows us to resize an image. When an object is selected with this tool, anchors appear around its corners and sides.
- Breaking text apart once converts each letter into its own text field. Breaking it apart again converts the letters into shapes that cannot be edited with the Text tool, but can be manipulated as regular graphics.
- Adding a mask to a layer masks only the items in the layer directly below it.

Section 16.5.3 Adding Online Help to Forms

- Use the Selection tool to align objects with their corresponding captions. For more precise alignment, select the desired object with the Selection tool and press the arrow key on the keyboard in the direction you want to move the object.
- An input text field is a text field into which the user can type text.
- Each movie clip should be created as a new symbol so that it can be edited without affecting the scene.
- Symbols may be embedded in one another; however, they cannot be placed within themselves.
- The Transform panel can be used to change an object's size.
- The Constrain checkbox causes the scale factor to be equal in the height and width fields. The scale factor measures the change in proportion.
- Changing a symbol's function or appearance in its editing stage updates the symbol in the scene.

Section 16.6 Creating a Website Splash Screen

- Many organizations use Flash to create website splash screens (i.e., introductions), product demos and web applications.

- Flash animations are ideal for amusing visitors while conveying information as the rest of a page downloads "behind the scenes."
- A preloader is a simple animation that plays while the rest of the web page is loading.
- Alpha is a value between 0 and 100% that corresponds to a color's transparency or opacity. An alpha value of 0% appears transparent, whereas a value of 100% appears completely opaque.
- The rate of progression in a gradient can also be changed by sliding the color boxes.
- Any color box may be removed from a gradient by dragging it downward off the gradient range.
- **Ease** controls the rate of change during tween animation. Negative values cause the animated change to be gradual at the beginning and become increasingly drastic. Positive values cause the animation to change quickly in the first frames and less drastically as the animation progresses.
- When a button is created from a shape, the button's hit area is, by default, the area of the shape.
- It is important to change the hit state of a button created from text so that it includes the spaces between the letters; otherwise, the link will work only when the user hovers over a letter's area.
- The "_blank" signifies that a new browser window should open when the user presses the button.
- Flash movies load frame by frame, and frames containing complex images take longer to load. Flash will continue playing the current frame until the next frame has loaded.

Terminology

ActionScript 3.0

active tool

addEventListener function

Adobe Flash CS3

alpha value

anchor

Bandwidth Profiler

bitmapped graphics

break apart

Brush Mode

Brush Tool

constrained aspect ratio

do while control structure

down state

duplicate symbol

Erase tool

.fla file format

frame

frame label

Frame Rate

frames per second

framesLoaded property

free transform tool

fscommand function

gotoAndPlay function

gotoAndStop function

gradients

Hand tool

hexadecimal notation

hit state

hypertext link

if control structure

input text field

instance

instance name

interactive animated movies

JavaScript

keyframe

Lasso tool

layer

Library panel

loader class

Magic wand

masking layer

math.random function

motion tween

movie clip

movie clip symbol

MP3 audio compression format

navigateToUrl function

Oval tool

over state

play function

playhead

preloader

radial gradient

raster graphic

raw compression

Rectangle tool

Sample Rate	Text tool
scenes	timeline
Selection tool	trace function
shape tween	tween
SoundFixer .stopAll function	up state
splash screen	vector graphic
stage	web-safe palette
stop function	while control structure
.swf file format	Zoom tool
symbol	

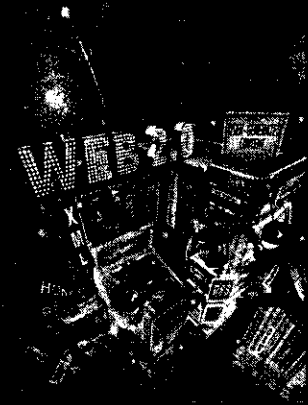
Self-Review Exercises

- 16.1** Fill in the blanks in each of the following statements:
- Adobe Flash's _____ feature draws the in-between frames of an animation.
 - Graphics, buttons and movie clips are all types of _____.
 - The two types of tweening in Adobe Flash are _____ tweening and _____ tweening.
 - Morphing one shape into another over a period of time can be accomplished with _____ tweening.
 - Adobe Flash's scripting language is called _____.
 - The area in which the movie is created is called the _____.
 - Holding down the *Shift* key while drawing with the Oval tool draws a perfect _____.
 - By default, shapes in Flash are created with a fill and a(n) _____.
 - _____ tell Flash how a shape or symbol should look at the beginning and end of an animation.
 - A graphic's transparency can be altered by adjusting its _____.
- 16.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- A button's hit state is entered when the button is clicked.
 - To draw a straight line in Flash, hold down the *Shift* key while drawing with the Pencil tool.
 - Motion tweening moves objects within the stage.
 - The more frames you give to an animation, the slower it is.
 - Flash's `math.random` function returns a number between 1 and 100.
 - The maximum number of layers allowed in a movie is ten.
 - Flash can shape tween only one shape per layer.
 - When a new layer is created, it is placed above the selected layer.
 - The **Lasso Tool** selects objects by drawing freehand or straight-edge selection areas.
 - The **Ease** value controls an object's transparency during motion tween.

Exercises

- 16.3** Using the combination of one movie-clip symbol and one button symbol to create a navigation bar that contains four buttons, make the buttons trigger an animation (contained in the movie clip) when the user rolls over the buttons with the mouse. Link the four buttons to www.nasa.gov, www.w3c.org, www.Flashkit.com and www.cnn.com.
- 16.4** Download and import five WAV files from www.coolarchive.com. Create five buttons, each activating a different sound when it is pressed.
- 16.5** Create a text "morph" animation using a shape tween. Make the text that appears in the first frame of the animation change into a shape in the last frame. Make the text and the shape different colors.

Adobe Flash CS3: Building an Interactive Game



Knowledge must come through action.

—Sophocles

It is circumstance and proper timing that give an action its character and make it either good or bad.

—Agesilaus

Life's but a walking shadow, a poor player that struts and frets his hour upon the stage and then is heard no more: it is a tale told by an idiot, full of sound and fury, signifying nothing.

—William Shakespeare

Come, Watson, come! The game is afoot.

—Sir Arthur Conan Doyle

*Cannon to right of them,
Cannon to left of them,
Cannon in front of them
Volley'd and thunder'd.*

—Alfred, Lord Tennyson

OBJECTIVES

In this chapter you'll learn:

- Advanced ActionScript 3 in Flash CS3.
- How to build on Flash CS3 skills learned in Chapter 16.
- The basics of object-oriented programming in Flash CS3.
- How to create a functional, interactive Flash game.
- How to make objects move in Flash.
- How to embed sound and text objects into a Flash movie.
- How to detect collisions between objects in Flash.