Foundation XML for Flash

Sas Jacobs

Foundation XML for Flash

ISBN (pbk): 1-59059-543-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The source code for this book is freely available to readers at www.friendsofed.com in the Source Code section.

Credits

Lead Editor
Chris Mills

Technical Reviewer
Kevin Ruse

Editorial Board
Steve Anglin, Dan Appleman
Ewan Buckingham, Gary Cornell
Tony Davis, Jason Gilmore
Jonathan Hassell, Chris Mills
Dominic Shakeshaft, Jim Sumser

Associate Publisher
Grace Wong

Project Manager
Pat Christenson

Copy Edit Manager
Nicole LeClerc

Copy Editor
Liz Welch

Assistant Production Director
Kari Brooks-Copony

Production Editor
Kelly Winquist

Compositor
Katy Freer

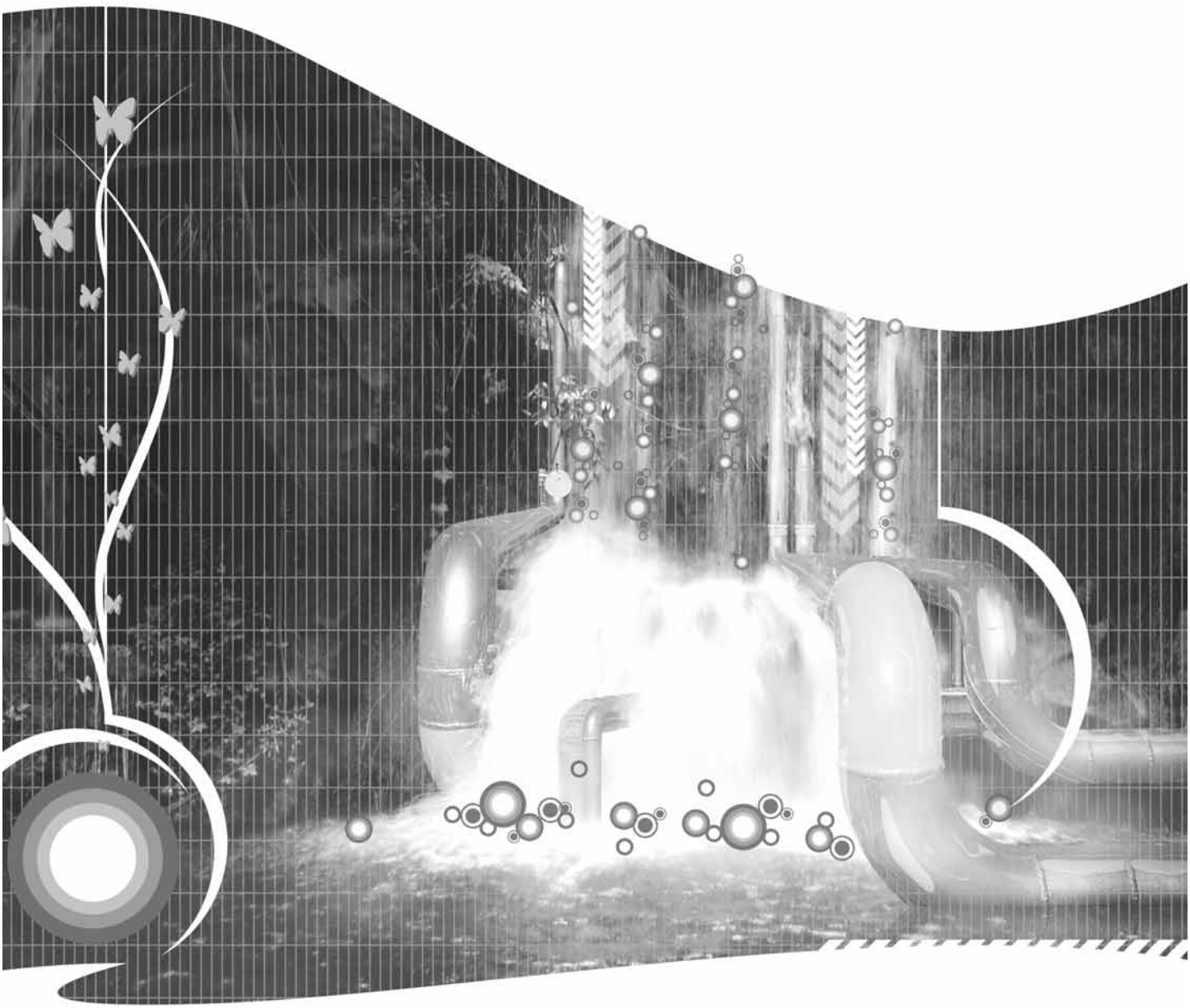
Proofreader
Lori Bring

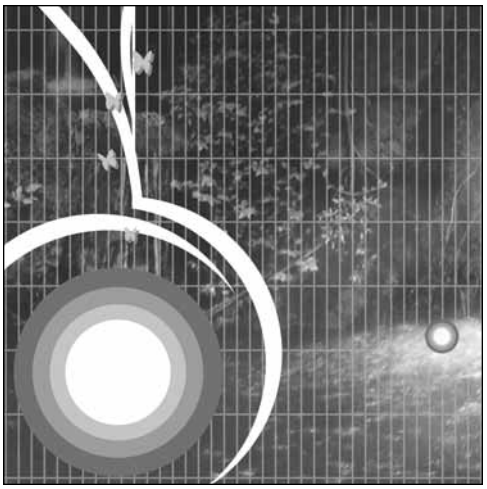
Indexer
Broccoli Information Management

Artist
Katy Freer

Cover Designers
Corné van Dooren, Kurt Krames

Manufacturing Director
Tom Debolski





Excerpt from Chapter 4

USING THE XML CLASS

In the earlier chapters, you learned about XML and some of the related technologies. We covered the uses for XML and the advantages that it provides over other forms of data storage. In this chapter, we'll look at how to work with XML in Flash. You'll create Flash movies that include data from external XML documents. You'll also create and modify XML content within Flash and learn how to send it to other applications. We'll work through several examples so that you can practice what we cover.

This chapter will introduce you to the **XML** class—the most common way to work with XML documents in Flash. In Chapters 8 and 9, we'll look at another way to work with XML documents: by using data components.

If you're not familiar with object-oriented programming, the term *class* refers to the design of an object. It specifies the rules for the way the object works and lists all the methods and properties for the object. The XML class contains all the information needed for working with XML objects in Flash.

The XML class was introduced in Flash 5, and since that time, the ActionScript associated with it hasn't changed significantly. The XML class became a native object in Flash 6, which increased its speed compared with Flash 5. The XML class stores XML content in document trees within Flash. The class allows you to

- Create new XML documents or fragments
- Load external XML documents
- Modify XML content
- Send XML information from Flash

The XML class includes methods and properties to work with XML document trees. You'll also use a related class, the **XMLNode** class, which allows you to work with specific nodes in an XML document tree. You can find a summary of the methods, properties, and events of both classes in the tables at the end of this chapter.

When we create a new XML object from the XML class, we call the object an *instance* of the class, and the process of creating the object is known as *instantiation*. One way to view the difference is to see the XML class as a template that you use to create XML objects.

We'll start this chapter by learning how to load an external XML document into a Flash XML object. We'll look at how you can extract information from the document tree so you can add it to your Flash movie. You could do this with either a physical document saved with an `.xml` file extension or a stream of XML information. If you're working with an information stream, you can load XML content generated by a PHP, ColdFusion, or ASP.NET server-side page or from a web service. We'll cover web services in Chapter 9.

Loading an XML document into Flash

The process of loading an XML document into Flash involves the following steps:

1. Create an XML object.
2. Specify what happens after the XML document loads, that is, identify a function to deal with the loaded XML information.
3. Load the external document into the XML object. Flash parses the data into the XML document tree.

After the document has been loaded and parsed into an XML document tree, an event handler calls the function specified in step 2. Usually, this function checks first to see if the XML document has been loaded successfully. If so, the function extracts the information from the XML document and adds it to the Flash movie. It's common to use the XML document to populate UI components.

Using the load method

The following code demonstrates one way to load in an external XML document. It uses ActionScript version 2.0.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = functionName;
myXML.load("filename.xml");
```

On the first line, I create a new XML object called `myXML`. The second line sets the `ignoreWhite` property to true. This means that Flash will ignore white space such as tabs and returns in the XML document. If you forget to set this property, blank lines will be treated as nodes, which can cause problems when you're trying to extract information.

On the third line, I use the onLoad event handler to refer to the function that will be called after the file has loaded. In the example, I've used the name functionName. The last line loads the file called filename.xml. If this were a real example, I'd have to create the function called functionName to process the XML content from filename.xml after the file loads.

I can also load the XML document from a subdirectory of the current directory:

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = functionName;
myXML.load("foldername/filename.xml");
```

Be aware that if your XML document includes file names and file paths, storing the XML document in a subfolder might cause you problems. When you try to use this information in a movie, Flash calculates the relative paths from the location of the .swf file. As this may be in a different location from the XML or .swf file, the movie may not be able to find the files at those locations. To avoid these potential problems, it's much easier if you keep your XML document in the same folder as your Flash movie.

In the code you've seen so far, I've used a file name ending with the .xml extension. However, you aren't limited to this type of file. You can load any type of file providing it results in an XML document. This means that you can load a server-side PHP, ColdFusion, Java, or ASP.NET file as long as it generates XML content.

If you do load a server-side file into your XML object, you'll need to include the full path so that the web server can process the server-side code. I've shown an example here; we'll learn more about working with server-side documents a little later in this chapter.

```
var myXML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = functionName;
myXML.load("http://localhost/webfolder/filename.aspx");
```

Understanding the order of the code

You may have noticed that, in the code samples above, I specified the load function before I loaded the XML document. On the surface, this doesn't make sense. Shouldn't I wait until the XML file loads and then specify which function to call?

In Flash, some ActionScript code doesn't run in a strict order. Lines don't necessarily wait for other lines to complete before they run. Much of the code that we write is in response to a specific event such as a mouse click. In this case, the event is the loading of an external XML file into an XML object. We call this *asynchronous* execution.

If we specified the load function after the load line, the XML file might have already finished loading before we set the function that should be called. This would mean that the onLoad function would be skipped completely. The only way to be sure that you call the function correctly is to set a reference to it before you load the XML file.

Understanding the onLoad function

After the XML document has finished loading, Flash calls the function that you specified in the onLoad line. The function is called by the onLoad *event handler*, and it runs after the XML document has been received by Flash and parsed into the document tree. You can assign the onLoad function with the following line:

```
myXML.onLoad = functionName;
```

If you've worked with ActionScript before, you'll notice that the function doesn't include those brackets that you're used to seeing after the function name, for example, functionName(). That's because we're not actually calling the function in this line; instead, we're assigning it to the onLoad event handler. The call will happen after the XML document is loaded and parsed.

Any onLoad function that you create automatically includes a parameter that tells you whether or not the file loaded successfully. The onLoad function should always test this parameter first before you start to process the XML document. You can also check the status property of the loaded document to see if there were any problems.

You can see this process in the following sample code:

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = myFunction;
myXML.load("filename.xml");
function myFunction(success:Boolean):Void {
  if (success) {
    //process XML content
  }
  else {
    //display error message
  }
}
```

In these lines, I've defined the function myFunction with a parameter called success. You can use any name you like for this parameter. The important thing to remember is that the parameter is Boolean, so it can only have one of two values: true or false.

The first line in the function checks to see if the value of the parameter is true; that is, if the XML document has loaded successfully. If so, the function would normally then include lines that process the XML content. Otherwise, if the document didn't load successfully, we'd probably want to display an error message.

Using the line

```
if (success) {
```

is equivalent to using

```
if (success == true) {
```

I could also have used an inline function as shown here:

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = function (success:Boolean):Void {
  if (success) {
    //process XML content
  }
  else {
    //display error message
  }
};
myXML.load("filename.xml");
```

Either of the two approaches shown here is acceptable. If you create the onLoad function separately, rather than inline, you will be able to reuse it when you load other XML documents. After all, if your XML documents have the same structure, you'll probably want to process them in the same way. Using the same function allows you to reuse the code and means that you'll only have to maintain one block of ActionScript. If you're loading a single XML document into your movie, it doesn't matter which method you choose.

The code shown so far uses the onLoad event handler. Flash provides another event handler for the XML object: onData. Both events trigger after the content has been loaded into Flash. The difference is that the onLoad event happens after the XML content has been parsed by Flash and added to the XML document tree. The onData event takes place before parsing, so you can use this event to access the raw XML from your external document. In most cases, you'll use the onLoad event handler.

> *In case you're wondering what the* `:Void` *means in* `myFunction(success:Boolean):Void`*, it indicates what type of information the function returns. In this case, nothing is returned, so I've used the word* `Void`*. You could also specify the datatype for the value returned by the function, for example,* `String` *or* `Number`.

After you load the XML content into an XML object, you'll need to add the data to your Flash movie. Before you do this, you should check that the document has loaded successfully.

Testing if a document has been loaded

You can test whether an XML document has been loaded by checking the loaded property of the XML object. The property returns a Boolean value and will display either true or false when traced in an Output window. Here's an example:

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = function (success:Boolean):Void {
  trace (this.loaded);
};
myXML.load("filename.xml");
```

In the sample code, `this` refers to the XML object. I can use the keyword `this` because I'm inside the `onLoad` function for the XML object.

Note that the `loaded` property may return a value of `true` even when errors have occurred in parsing the XML content. Flash provides a mechanism for finding errors.

Locating errors in an XML file

When you load an external XML document into Flash, it's possible that it may not be well formed. Remember that well-formed documents meet the following requirements:

- The document contains one or more elements.
- The document contains a single root element, which may contain other nested elements.
- Each element closes properly.
- Start and end tags have matching case.
- Elements nest correctly.
- Attribute values are contained in quotes.

Where a document is not well formed, Flash may have difficulty in parsing it into the document tree. Flash may still indicate that the document has loaded successfully, even if it wasn't parsed correctly.

The XML class has a `status` property that indicates any problems that occurred when parsing the XML document. This property returns a value between `0` and `-10`; the values for each are shown here:

- `0`—No error; the parse was completed successfully.
- `-2`—A CDATA section was not properly terminated.
- `-3`—The XML declaration was not properly terminated.
- `-4`—The DOCTYPE declaration was not properly terminated.
- `-5`—A comment was not properly terminated.
- `-6`—An XML element was malformed.
- `-7`—The application is out of memory.
- `-8`—An attribute value was not properly terminated.
- `-9`—A start tag was not matched with an end tag.
- `-10`—An end tag was encountered without a matching start tag.

Where a document contains multiple errors, the `status` property will only return one error value. Even though Flash may detect an error during parsing, it may still be possible to find information from all or part of the document tree.

To show you an example of the status numbers, let's load the resource file `address1.xml` file into Flash. The document is missing an ending </phoneBook> tag, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Some Street, Some City, Some Country</address>
    <phone>123 456</phone>
  </contact>
```

I've saved this example in the resource file `simpleloadstatus.fla`. Figure 4-1 shows the status code that displays when I test the file. In this case, the code is –9, indicating a mismatch between start and end tags within the document.

Figure 4-1. The Output window displaying the status property

So far, we've covered the theory behind loading external XML documents into Flash. We create an XML object and load a file, and Flash parses the contents into a document tree. We can then use a function to add the XML content to our movie.

Each time you load an external XML document, your code will probably start with the same steps:

1. Create the XML object.
2. Set the `ignoreWhite` property to true.
3. Specify the name of the function that will deal with the loaded XML document.
4. Load the XML document.
5. Within the load function, test whether the XML file has loaded successfully.
6. Display the document tree with a `trace` action to check the loaded contents.

We'll work through an example to illustrate these steps. We'll load an external XML document into Flash and display it in an Output window. When you do this, you should see the same content that is in the external document.

Exercise 1: Loading an external XML document

In this example, we'll create a simple Flash movie that loads the `address.xml` file and displays it in an Output window. You can see the completed example in the resource file `simpleload.fla`.

1. Create a new Flash movie and click frame 1 on the Timeline.
2. Save the document in the same folder as the `address.xml` file.
3. Add the following code into the Actions panel. You can open the panel by using the *F9* shortcut key.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = processXML;
myXML.load("address.xml");
function processXML(success:Boolean):Void {
  if (success) {
    trace(this);
  }
  else {
    trace ("Error loading XML file");
  }
}
```

In this example, I call the function `processXML` after the file `address.xml` loads. The function checks to see if the XML document loads successfully by checking the variable `success`. If so, I use the `trace` action to display the XML document tree in an Output window. If not, I display an error message in the Output window.

Because the function has been called by the `onLoad` event of the XML object, I can use the word `this`. In fact,

```
trace(this);
```

is the same as

```
trace(myXML);
```

4. Save the movie and test it with the *Ctrl-Enter* shortcut (*Cmd-Return* on a Macintosh). You should see an Output window similar to the one shown in Figure 4-2. The Output window displays the document tree from the XML object. I've saved the sample file `simpleload.fla` with your resources.

Figure 4-2. The Output window displaying XML content

5. If you see an error when you test the movie, it's most likely to be due to a misspelling. Check the XML document file name and the spelling of your success parameter within the function. A sample error is shown in Figure 4-3.

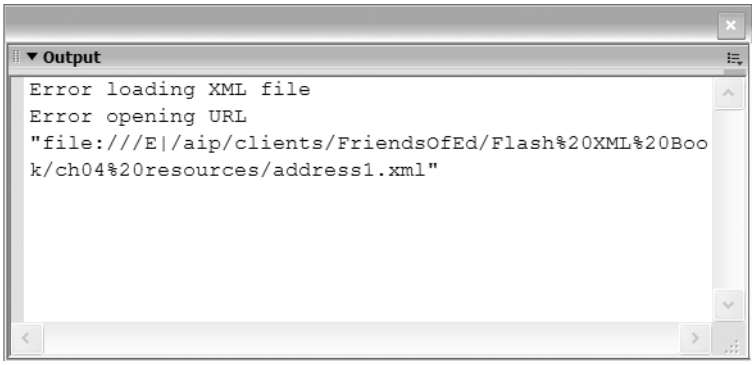


Figure 4-3. The Output window displaying an error message

In the previous exercise, we loaded the file `address.xml` into Flash. We tested that the document loaded successfully, and then we displayed the contents of the XML document tree in an Output window. Tracing the document tree can be a useful way to test that you've loaded the XML document correctly.

> *With most objects in Flash, you'll see* `[object Object]` *when you trace them in the Output panel. If you trace an* `XML` *or* `XMLNode` *object, Flash uses the* `toString` *method to display a text representation of the object. This is very useful when you want to view the XML document tree. Using*
>
> ```
> trace(myXML);
> ```
>
> *is the same as using*
>
> ```
> trace(myXML.toString());
> ```

Points to note from exercise 1

- It's possible for you to change the XML content within Flash independently of the external XML document. Flash doesn't maintain a link back to the original document. If the external document changes, it will have to be loaded again before the updated content displays within Flash.
- Flash can't update external documents. To update an external XML document, you'll have to send any changes you make to the document tree out of Flash to a server-side page for processing.
- Earlier in the chapter, I discussed inline onLoad functions. I've re-created the preceding example using an inline function. The code is shown here, and it's saved in the resource file `simpleload2.fla`:

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = function(success:Boolean):Void {
  if (success) {
    trace(this);
  }
  else {
    trace ("Error loading XML file");
  }
};
myXML.load("address.xml");
```

Testing for percent loaded

Loading a large XML document might take some time, and it's useful to let the user know what's happening. You may want to display a message showing the user that the XML file is loading. The XML class allows you to find out how big the XML file is and how much has loaded, using the `getBytesLoaded` and `getBytesTotal` methods.

You can use these methods with the `setInterval` action to check progress at given time intervals. You can also use them in the `onEnterFrame` event handler of a movie. Loading small XML files is very quick, so these methods are only going to be useful when you're working with large XML files.

The following example is based on the code provided within the Flash help file. It shows how you can test for the percentage of an XML file loaded using the `setInterval` action. You can find the example in the resource file `simpleloadpercent.fla`.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = processXML;
var intervalID:Number = setInterval(checkLoad, 5, myXML);
myXML.load("large.xml");
function processXML(success:Boolean):Void {
  clearInterval(intervalID);
  if (success) {
    trace("loaded: " + this.getBytesLoaded());
    trace ("total: " + this.getBytesTotal());
  }
```

```
  else {
    trace ("Error loading XML file");
  }
}
function checkLoad(theXML:XML):Void {
  var loaded:Number = theXML.getBytesLoaded();
  var total:Number = theXML.getBytesTotal();
  var percent:Number = Math.floor((loaded/total) * 100);
  trace ("percent loaded: " + percent);
}
```

On my computer, despite loading an XML file of over 3 MB in size, the loading process only displays the total file size.

The following example shows the same methods used with the onEnterFrame event handler. You can see the example saved in the resource file simpleloadEnterFrame.fla.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = processXML;
var intervalID:Number = setInterval(checkLoad, 5, myXML);
myXML.load("large.xml");
this.onEnterFrame = function():Void {
  var loaded:Number = myXML.getBytesLoaded();
  var total:Number = myXML.getBytesTotal();
  var percent:Number = Math.floor((loaded/total) * 100);
  trace ("percent loaded: " + percent);
  if (percent == 100) {
    delete this.onEnterFrame;
  }
}
function processXML(success:Boolean):Void {
  if (success) {
    trace("loaded: " + this.getBytesLoaded());
    trace ("total: " + this.getBytesTotal());
  }
  else {
    trace ("Error loading XML file");
  }
}
```

Again, I am not able to get the percentage loaded value to display each time the movie enters a new frame. You may wish to use these methods with caution. In my work, using a preloader for XML documents has rarely been necessary. If required, an alternative approach might be to show and hide a movie clip that displays a loading message.

After you've loaded an external XML document, you'll need to display the contents within your Flash movie. The next section shows how you can extract information from an XML object. A little later, we'll use the techniques to populate UI components.

Navigating an XML object

In Chapter 2, I talked about XML parsers. You might remember that we use a parser to process an XML document. Once parsed, the software can work with the content from the XML document.

There are two types of parsers: validating and nonvalidating. The difference is that validating parsers compare an XML document against a schema or DTD to make sure that you've constructed the document correctly. Nonvalidating parsers don't do this.

Flash contains a nonvalidating parser. When you load an XML document into Flash, it processes the contents and creates an XML document tree. If you include a reference to a schema or DTD within an XML document, Flash won't check the document for validity before it is loaded.

The Flash document tree includes all elements from the XML document. Flash uses a family analogy to refer to different branches within the tree. Elements can be *children* of another *parent* element, or *siblings*.

Each element includes a collection of child elements called `childNodes`. The collection is an array so we can use the standard array methods. For example, we can determine how many elements are in the collection using the `length` property. Elements with no child elements will have a `childNodes` length of `0`. We can also loop through the collection when we're processing an XML object.

In the `address.xml` file, shown here, the root element `<phoneBook>` has three child `<contact>` elements. We could programmatically work through each of the child elements, for example, adding them to a List component.

```
<?xml version="1.0" encoding="UTF-8"?>
<phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Some Street, Some City, Some Country</address>
    <phone>123 456</phone>
  </contact>
  <contact id="2">
    <name>John Smith</name>
    <address>4 Another Street, Another City, Another Country</address>
    <phone>456 789</phone>
  </contact>
  <contact id="3">
    <name>Jo Bloggs</name>
    <address>7 Different Street, Different City, UK</address>
    <phone>789 123</phone>
  </contact>
</phoneBook>
```

You can refer to a child element by its position in the collection. Because the `childNodes` collection is an array, the first element is at number `0`. The following line refers to the first child element of the `myXML` object. You can also refer to children of an `XMLNode` object.

```
myXML.childNodes[0];
```

The firstChild and lastChild properties allow you to refer to the first and last items in the childNodes collection.

```
myXML.firstChild;
myXML.lastChild;
```

When you are processing an XML document tree, you usually start by referencing the root node of the document. This is the parent of all other elements and is the firstChild or childNodes[0] of the XML object.

All elements are children of the XML object, so you can refer to each one using its position within the XML object. It's a bit like a map. Start with the root node and move to the first child. Go to the second child node and find the third child. Finish at the first child of this node. You can end up with long paths as shown here:

```
myXML.firstChild.childNodes[1].childNodes[2].firstChild;
```

You'll learn a bit more about locating specific child nodes in a document later in this chapter. In the next section, I've shown the Flash notation for the XML elements in the file address.xml.

Mapping an XML document tree

I've shown the complete address.xml document here. Table 4-1 shows how you can refer to specific parts of this document once it's loaded into Flash. The table assumes that we've created an XML object called myXML.

```
<?xml version="1.0" encoding="UTF-8"?>
<phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Some Street, Some City, Some Country</address>
    <phone>123 456</phone>
  </contact>
  <contact id="2">
    <name>John Smith</name>
    <address>4 Another Street, Another City, Another Country</address>
    <phone>456 789</phone>
  </contact>
  <contact id="3">
    <name>Jo Bloggs</name>
    <address>7 Different Street, Different City, UK</address>
    <phone>789 123</phone>
  </contact>
</phoneBook>
```

Table 4-1. Mapping the XML document tree for address.xml

Element	Flash XML Element Path
`<?xml version="1.0" encoding="UTF-8"?>`	`xmlDecl` property
`<phoneBook>`	`myXML.firstChild` or `myXML.childNodes[0]`
`<contact id="1">`	`myXML.firstChild.firstChild` or `myXML.childNodes[0].childNodes[0]`
`<name>`	`myXML.firstChild.firstChild.firstChild` or `myXML.childNodes[0].childNodes[0].childNodes[0]`
`<address>`	`myXML.firstChild.firstChild.childNodes[1]` or `myXML.childNodes[0].childNodes[0].childNodes[1]`
`<phone>`	`myXML.firstChild.firstChild.lastChild` or `myXML.childNodes[0].childNodes[0].childNodes[2]`
`<contact id="2">`	`myXML.firstChild.childNodes[1]` or `myXML.childNodes[0].childNodes[1]`
`<name>` (within contact 2)	`myXML.firstChild.childNodes[1].firstChild` or `myXML.childNodes[0].childNodes[1].childNodes[0]`
`<address>` (within contact 2)	`myXML.firstChild.childNodes[1]. childNodes[1]` or `myXML.childNodes[0].childNodes[1].childNodes[1]`
`<phone>` (within contact 2)	`myXML.firstChild.childNodes[1].lastChild` or `myXML.childNodes[0].childNodes[1].childNodes[2]`
`<contact id="3">`	`myXML.firstChild.childNodes[2]` or `myXML.childNodes[0].childNodes[2]`
`<name>` (within contact 3)	`myXML.firstChild.childNodes[2].firstChild` or `myXML.childNodes[0].childNodes[2].childNodes[0]`
`<address>` (within contact 3)	`myXML.firstChild.childNodes[2].childNodes[1]` or `myXML.childNodes[0].childNodes[2].childNodes[1]`
`<phone>` (within contact 3)	`myXML.firstChild.childNodes[2].lastChild` or `myXML.childNodes[0].childNodes[2].childNodes[2]`

You can replace the myXML references with this if you're including the references in the onLoad function of the XML object. Placing any of the paths in a trace statement will display the complete element in an Output window. For example, the following onLoad function traces the second contact's <name> element from the address.xml file, as shown in Figure 4-4.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = processXML;
myXML.load("address.xml");
function processXML(success:Boolean):Void {
  if (success) {
    trace(this.firstChild.childNodes[1].firstChild);
  }
  else {
    trace ("Error loading XML file");
  }
}
```

Figure 4-4. Tracing an element from the document tree

Understanding node types

The XML class stores XML content in a document tree. Earlier in the book, we learned that XML documents can contain

- Elements
- Attributes
- Text
- Entities
- Comments
- Processing instructions
- CDATA

Within an XML document tree, Flash recognizes only two types of nodes—XML elements and text nodes. You can access the attributes within an XML element but Flash ignores comments, processing instructions, and CDATA.

You can use the property nodeType to identify which type of element you're working with. The property returns a value of 1 for element nodes and 3 for text nodes. It's important to know which type you're working with because some properties of the XML class are specific to certain node types. This code shows how you can use the nodeType property to display the node type:

```
trace(myXML.nodeType);
trace(myXMLNode.nodeType);
```

You can find the name of an element node by using the nodeName property. This is the name of the tag included within the element, and you can use the property with an XML object or an XMLNode object.

```
trace(myXML.firstChild.nodeName);
trace(myXMLNode.nodeName);
```

Text nodes don't have a tag name, so the nodeName property will return a value of null.

A text node is the child of the parent element node. Instead of a nodeName, text nodes have a nodeValue, which displays the text content. To display the text inside an element, you can use

```
trace(myXML.firstChild.firstChild.firstChild.nodeValue);
```

The nodeValue property for an element node will display null.

Table 4-2 shows some examples of how to access the text from the file address.xml.

Table 4-2. Locating the text nodes within XML document tree for address.xml

Text	Flash XML Element Path
Sas Jacobs	myXML.firstChild.firstChild.firstChild.firstChild.nodeValue or myXML.childNodes[0].childNodes[0].childNodes[0].childNodes[0].nodeValue
7 Different Street, Different City, UK	myXML.firstChild.childNodes[2].childNodes[1].firstChild.nodeValue or myXML.childNodes[0].childNodes[2].childNodes[1].childNodes[0].nodeValue
456 789	myXML.firstChild.childNodes[1].childNodes[2].firstChild.nodeValue or myXML.childNodes[0].childNodes[1].childNodes[2].childNodes[0].nodeValue

Again, you can replace the myXML references with this if you're including these references in an onLoad function. Adding any of the paths shown in Table 4.2 in a trace statement will display the complete element in an Output window.

The statements in the preceding table appear a little confusing. The paths are long, and it's not easy to figure out which element we're targeting with paths like firstChild.childNodes[1].childNode[2]. Your code will be much easier to read if you create XMLNode variables. These variables can act as signposts to specific parts of the XML document and make it easier to navigate the document tree.

Creating node shortcuts

As you've seen, writing a path to a specific element within the document tree can be an arduous process. It's much easier to use an XMLNode variable to provide a shortcut to a specific position in the document tree, as shown here:

```
var myXMLNode:XMLNode = myXML.firstChild.firstChild.firstChild;
```

By writing this line, you can use myXMLNode to refer to the element instead of the full path. If you use descriptive names for the XMLNode objects, you'll find it much easier to understand your code:

```
var NameNode:XMLNode = myXML.firstChild.firstChild.firstChild;
trace(NameNode);
```

You'd normally start this process by locating the root node of the document. Remember that each file has a single root node that contains all of the other elements.

Finding the root node

The root node of the tree is always the first child of the XML object, so you can locate it with the following code. Both lines are equivalent.

```
myXML.firstChild;
myXML.childNodes[0];
```

If you are referring to the first child within the onLoad function, you can also use

```
this.firstChild;
this.childNodes[0];
```

Displaying the firstChild of the XML document in an Output window is almost the same as displaying the entire XML document tree. The difference is that the firstChild doesn't include the XML declaration.

In the resource file simpleload.fla, replacing the line trace (this); with trace(this.firstChild); will show the document tree without the XML declaration.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = processXML;
myXML.load("address.xml");
function processXML(success:Boolean):Void {
  if (success) {
    trace(this.firstChild);
  }
  else {
    trace ("Error loading XML file");
  }
}
```

It can be useful to assign the root node to a variable so that you don't have to keep writing this.firstChild each time.

Setting a root node variable

It's often useful to set a variable for the root node. I like to use the variable name RootNode. You can use the variable type XMLNode so that you'll get code hints each time you type the variable name. These two lines are equivalent, and you can use either:

```
var RootNode:XMLNode = myXML.firstChild;
var RootNode:XMLNode = myXML.childNodes[0];
```

If you're referring to the root node from within the onLoad function, you can also use the word this:

```
var RootNode:XMLNode = this.firstChild;
var RootNode:XMLNode = this.childNodes[0];
```

Setting a variable provides a shortcut each time you want to refer to the root node. It saves you from having to write myXML.firstChild or myXML.childNodes[0].

To get to the first child of the root node, you could use either of these two lines:

```
this.firstChild.firstChild;
this.childNodes[0].childNodes[0];
```

You could also write

```
var RootNode:XMLNode = this.firstChild;
RootNode.firstChild;
```

or

```
var RootNode:XMLNode = this.childNodes[0];
RootNode.childNodes[0];
```

Using the descriptive name RootNode makes it much easier to identify your position within the XML document. You can use the same approach with other elements within the XML document tree.

Displaying the root node name

You can find out the name of the root node by using its nodeName property:

```
var RootNode:XMLNode = myXML.firstChild;
trace(RootNode.nodeName);
```

This is equivalent to the single line

```
trace (myXML.firstChild.nodeName);
```

You can try this with your resource file simpleload.fla. Modify the onLoad function as shown here:

```
function processXML(success:Boolean):Void {
  if (success) {
    var RootNode:XMLNode = this.firstChild;
    trace(RootNode.nodeName);
  }
```

```
    else {
      trace ("Error loading XML file")
    }
}
```

When you test the movie, you should see an Output window similar to that shown in Figure 4-5.

Figure 4-5. Displaying the root node name

You can see this example in the resource file `simpleprocess.fla`.

When you first start working with the XML class, it can be a very useful to trace the name of the root node as a first step. Making sure that the name is correct will help you to identify simple errors such as forgetting to set the `ignoreWhite` property value to true.

Once you've located the root node, you can start working your way through the document tree to find specific child nodes. Again, it's useful to create variables for positions within the document tree to make your code easier to understand.

Locating child nodes

Earlier in the chapter, you saw some examples of how to locate the child nodes within an XML object. Tables 4-1 and 4-2 provide some useful summaries. You started with the root element and used properties to find a specific node.

Working with specific child nodes

To refer to a specific node in your document tree, you need to construct a path. You can refer to each section of the path using properties like `firstChild` or a position in the `childNodes` collection such as `childNodes[2]`.

For example, in the XML fragment that follows, the <contact> element is the firstChild of the <phoneBook> root element, which is the firstChild of the XML object. The <name>, <address>, and <phone> elements are childNodes[0], childNodes[1], and childNodes[2], respectively, of the <contact> element.

```
<phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Some Street, Some City, Some Country</address>
    <phone>123 456</phone>
  </contact>
</phoneBook>
```

To refer to the <address> element, I could use the path

```
myXML.firstChild.firstChild.childNodes[1];
```

or

```
myXML.childNodes[0].childNodes[0].childNodes[1];
```

I could combine this with a root node variable to achieve the same result:

```
var RootNode:XMLNode = myXML.firstChild;
RootNode.firstChild.childNodes[1];
```

You can see an example of this in the resource file simpleprocess.fla.

I could use the following code to refer to the <phone> element:

```
myXML.firstChild.firstChild.lastChild;
```

or

```
myXML.childNodes[0].childNodes[0].childNodes[3];
```

The childNodes collection and the firstChild and lastChild properties are read-only. This means you can't use them to change the structure of the XML object.

Text elements are always the firstChild of the element that contains them. To refer to the text inside the <address> element, I could use the expression

```
myXML.firstChild.firstChild.childNodes[1].firstChild.nodeValue;
```

or

```
myXML.childNodes[0].childNodes[0].childNodes[1].firstChild.nodeValue;
```

I could also use the RootNode variable as shown here:

```
var RootNode:XMLNode = myXML.firstChild;
RootNode.childNodes[0].childNodes[1].firstChild.nodeValue;
```

Again, you can see an example of this in simpleprocess.fla. You may need to uncomment the relevant lines in the file.

All of the child nodes of an element live within the childNodes collection. This is an array of all the child nodes. As you'll often want to treat each childNode in a similar way, it makes more sense to work with the collection as a whole.

Working with the childNodes collection

It's more common to work with all childNodes in a collection rather than finding single nodes within the document tree. You can loop through the collection and perform similar actions on all of the nodes. The code that follows shows how to use a for loop in this way. We can determine how many children are in the collection of childNodes by using the childNodes.length property. This is the same as the length property of an array.

```
for (var i:Number=0; i < myXMLNode.childNodes.length; i++) {
  //do something
}
```

You can determine if an element has child nodes by testing the length property of the collection or by using the hasChildNodes method. You may want to perform one action for elements with child nodes and another for elements without children. Using the hasChildNodes method returns a value of either true or false, so it is often used within if statements, as shown in this code snippet:

```
if (RootNode.hasChildNodes()) {
  //do something with the child nodes
}
else {
  //do something else
}
```

The following example shows how we could display all of the names of the children of a specific node, in this case, the first <contact> element. I've shown the relevant lines in bold. You can also open the resource file simpleprocess.fla to test the example.

```
function processXML(success:Boolean):Void {
  if (success) {
    var RootNode:XMLNode = this.firstChild;
    var ContactNode:XMLNode = RootNode.childNodes[0];
    for (var i:Number=0; i < ContactNode.childNodes.length; i++) {
      trace (ContactNode.childNodes[i].nodeName);
    }
  }
  else {
    trace ("Error loading XML file");
  }
}
```

This code assigns the first contact node to an XMLNode variable called ContactNode. We can then loop through each of the child nodes of that variable and display their names.

If you test the movie, you should see an Output window similar to the one shown in Figure 4-6.

Figure 4-6. Displaying the child node names

Notice that I used an XMLNode variable called ContactNode to refer to the first <contact> element. The expression

```
ContactNode.childNodes[i].nodeName;
```

is much easier to understand than

```
myXML.firstChild.firstChild.childNodes[i].nodeName;
```

I could modify the function to display the text within each of the childNodes. Remember that the text within a node is always the firstChild of that node and that you can find the text using nodeValue. I've shown an example here; you can also see it in the simpleprocess.fla resource file.

```
function processXML(success:Boolean):Void {
  if (success) {
    var RootNode:XMLNode = this.firstChild;
    var ContactNode:XMLNode = RootNode.childNodes[0];
    for (var i:Number=0; i <ContactNode.childNodes.length; i++) {
      trace (ContactNode.childNodes[i].firstChild.nodeValue);
    }
  }
  else {
    trace ("Error loading XML file");
  }
}
```

Working your way through a complicated XML document can take some time. You have to understand the document structure and write code accordingly. An alternative way to work with the entire document tree is to use recursive functions. This can also be useful if you don't know the structure of the file or the names of the nodes.

Creating recursive functions

A *recursive function* is a function that calls itself. You can use a recursive function to extract the contents from the whole document tree. By calling the function again and passing the next branch of the tree, you can work your way through the entire XML object. You start by calling the function and passing the root node. If you find child nodes, you call the function again with each of the child nodes. You repeat the process until you've moved through the entire document tree.

This concept can be a little difficult to grasp, so I'll work through an example to help you understand it better.

Exercise 2: Processing an XML object with a recursive function

1. Create a new Flash file and save it in the same folder as the `address.xml` file.
2. Enter the following code. Instead of processing the XML object with the `processXML` function, I've used it to call another function called `showChildren`. The `showChildren` function takes one parameter, the root node of the XML object, which I've specified using `this.firstChild`.

```
var myXML:XML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = processXML;
myXML.load("address.xml");
function processXML(success:Boolean):Void {
  if (success) {
    showChildren(this.firstChild);
  }
  else {
    trace ("Error loading file");
  }
}
```

3. Add the `showChildren` function in the Actions panel, underneath the `processXML` function:

```
function showChildren(startNode:XMLNode):Void{
  if (startNode.nodeType == 1) {
    if (startNode.hasChildNodes()) {
      trace (startNode.nodeName + " has child elements:");
      for (var i:Number = 0; i < startNode.childNodes.length; i++) {
        if (startNode.childNodes[i].nodeType == 1) {
          trace ("element: " + startNode.childNodes[i].nodeName);
        }
        else {
          trace ("text: " + startNode.childNodes[i].nodeValue);
        }
        showChildren(startNode.childNodes[i]);
      }
    }
  }
}
```

The function looks confusing at first. It takes an XMLNode variable as a parameter and only proceeds if the XMLNode is an element node, that is, nodeType == 1. Text nodes can't have children.

The second if statement determines whether there are any childNodes of the current element node. If there are, the function traces the name of the node and the words has child elements.

Next, the function loops through the childNodes of the starting node. If the childNode is an element, the function traces the word element with the node name. Otherwise, for text elements, it traces the word text with the text content.

Finally, the function calls itself and passes the current childNode as a parameter. This repeats the process at the next level in the document tree. The function stops when it encounters a text node or when the current node has no childNodes.

4. Save the Flash file and test the movie. You should see an Output window similar to the one shown in Figure 4-7. You can find the completed file in your resources saved under the name recursive.fla.

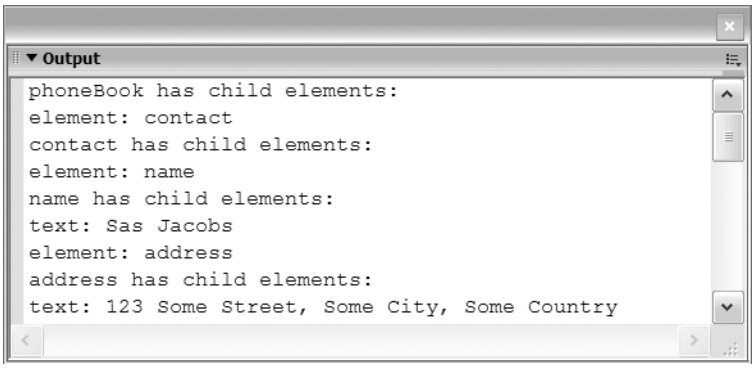


Figure 4-7. Displaying the contents of an XML document using a recursive function

Using a recursive function allows you to process the contents of the document tree without understanding the structure. It can also be a more efficient way to write code that processes the document tree.

So far, we've worked with child nodes, but it's useful to know that you can use ActionScript to find sibling nodes. These are nodes that share the same parent node.

Locating siblings

Flash provides two properties for dealing with siblings: nextSibling and previousSibling. These properties allow you to locate elements that share the same parent as the current node. You can refer to the previous and next siblings of the current node using

```
myXMLNode.previousSibling;
myXMLNode.nextSibling;
```

If there is no previous or next sibling, the property will return undefined, so you can't find the previousSibling of the first child node or the nextSibling of the last child node. As both of these properties are read-only, you can't use them to move nodes within the document tree.

The following example shows the processXML function modified to return the next and previous siblings of the second <address> element:

```
function processXML(success:Boolean):Void {
  if (success) {
    var RootNode:XMLNode = this.firstChild;
    var AddressNode:XMLNode = RootNode.childNodes[0].childNodes[1];
    trace ("current node: " + AddressNode.nodeName);
    trace ("previous: " + AddressNode.previousSibling.nodeName);
    trace ("next: " + AddressNode.nextSibling.nodeName);
  }
  else {
    trace ("Error loading XML file");
  }
}
```

I've included this example in the simpleprocess.fla resource file. If you test this file, you should see something similar to the Output window shown in Figure 4-8.

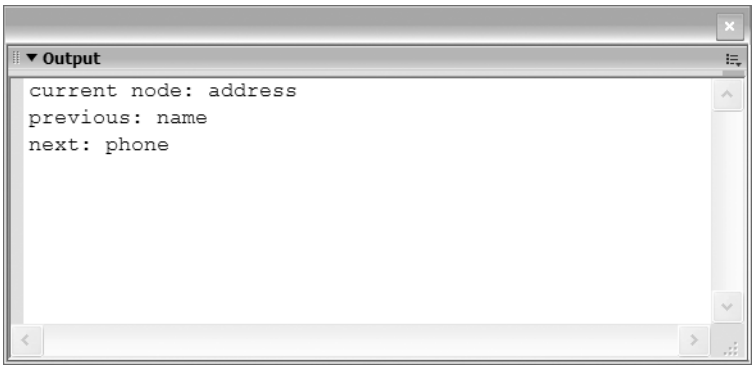


Figure 4-8. Displaying the previous and next sibling node names

As well as working with siblings, you can find the parent element of a node. This might be a quicker way to locate a node rather than writing a full path starting from the root node.

Locating parent nodes

You can refer to the parent of a current node using the parentNode property, as shown here:

```
myXMLNode.parentNode;
```

In this example, the processXML function displays the name of the parent of the second <address> element. I've made the relevant lines bold.

```
function processXML(success:Boolean):Void {
  if (success) {
    var RootNode:XMLNode = this.firstChild;
    var AddressNode:XMLNode = RootNode.childNodes[0].childNodes[1];
    trace ("parent node is " + AddressNode.parentNode.nodeName);
  }
  else {
    trace ("Error loading XML file");
  }
}
```

You can see the example in the simpleprocess.fla resource file. If you test the file, you should see an Output window similar to that displayed in Figure 4-9.

Figure 4-9. Displaying the parent node name

Note that the parentNode property is read-only, so you can't use it to change the structure of a document tree.

So far in this chapter, we've looked at how to extract information from both element and text nodes within an XML document. In the next section, I'll explain how you can work with attributes.

Extracting information from attributes

You refer to attributes differently compared with elements and text. Attributes aren't children of element. Rather, they are a collection, or array, within an element. Unlike the childNodes collection, the attributes collection is an associative array. This means that you can't use a position number. You have to refer to each attribute using its name.

The following lines show you how you can refer to the value of an attribute using its name. The two examples are equivalent. The first uses dot notation while the second uses associative array notation.

```
myXML.firstChild.attributes.attName;
myXML.firstChild.attributes["attName"];
```

In both examples, I'm finding the attribute called `attName` within the first child of the XML object called `myXML`.

In this XML fragment,

```
<?xml version="1.0" encoding="UTF-8"?>
<phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Some Street, Some City, Some Country</address>
    <phone>123 456</phone>
  </contact>
</phoneBook>
```

I could display the value of the `id` attribute of the `<contact>` element using the following code lines. Both the second and third lines are equivalent.

```
var RootNode:XMLNode = myXML.firstChild;
trace(RootNode.firsChild.attributes.id);
trace(RootNode.firstChild.attributes["id"]);
```

It's easy to refer to attributes when you know their names. However, there may be occasions when you don't know their names. In those cases, it can be useful to loop through the attributes collection, as shown here:

```
for (var theAtt:String in myXMLNode.attributes) {
  //reference the attribute name using theAtt
  //reference the value using myXMLNode.attributes[theAtt])
}
```

This code is equivalent to saying *for each attribute in the attributes collection*.

The next example shows the `processXML` function modified to show the attributes within the first `<contact>` element of the `address.xml` file. The function displays the name and value of each attribute. Unfortunately, the element only has one attribute so the loop repeats only once.

```
function processXML(success:Boolean):Void {
  if (success) {
    var RootNode:XMLNode = this.firstChild;
    var ContactNode:XMLNode = RootNode.childNodes[0];
    for (var theAtt:String in ContactNode.attributes) {
      trace(theAtt + " = " + ContactNode.attributes[theAtt]);
    }
  }
  else {
    trace ("Error loading file");
  }
}
```

In this example, we create a new XMLNode variable called ContactNode to refer to the first <contact> element. We use a for loop to move through the collection of attributes. Because I'm working with an associative array, I have to refer to the value of the attribute using ContactNode.attributes[theAtt]).

You can see this example in the simpleprocess.fla resource file. Uncomment the relevant lines and test the movie. You should see an Output window similar to the one shown in Figure 4-10.

Figure 4-10. Looping through the attributes collection

You've learned a lot about loading external documents and extracting their values within Flash. We covered the various properties that you could use to move through the document tree. I showed you how to find the name of a node and the value of text within a node. The theory we've covered so far will make more sense when you work through an exercise.

Putting it all together

In this section, we'll put together everything we've covered so far in the chapter and create a simple XML application. We'll use the photoGallery.xml file that you created in Chapter 3. The application will load the content from the XML document and add the contents to create a photo gallery.

We'll work through the following steps to create our application. These steps are likely to be the same ones you use each time you load an external XML document.

1. Create the XML object.
2. Set the ignoreWhite property to true.
3. Specify the name of the function that will deal with the loaded XML document.
4. Load the XML document.
5. Within the load function, test whether the file has loaded successfully.
6. Display the document tree with a trace action to check that you've loaded the contents correctly.
7. Set a variable referring to the RootNode of the XML document.
8. Work through the document tree, adding content to the Flash movie.

The completed file gallery_completed.fla is included with your resources for Chapter 4 in case you want to see how the finished application works. Note that I've used ActionScript version 2.0 and version 2.0 components in the files, so you'll need at least Flash MX 2004 to complete the exercise.

Exercise 3: Creating an XML photo gallery

In this exercise, we'll create a simple Flash photo gallery that loads external images. We'll take the images from the photos folder included with the Chapter 4 resources. You can also use your own images if you'd prefer. Just add them to the photos folder and the XML document.

Setting up the environment

1. Move the photos folder, the starter file gallery.fla, and the photoGallery.xml file from the resources to the same directory on your computer.
2. Open gallery.fla. Figure 4-11 shows the interface.

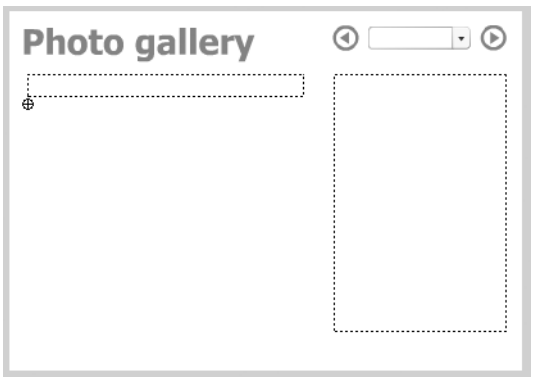


Figure 4-11. The gallery.fla interface

The interface includes a static text field containing the text Photo gallery. There are two dynamic text fields, one on the left for the caption and one on the right for the comments about the image. They have the instance names caption_txt and comment_txt, respectively.

There is an empty movie clip called empty_mc below the caption. We'll use this to display the photos. The top right of the interface includes a ComboBox component, gallery_cb, and two buttons, back_btn and forward_btn. Users will choose the gallery from the ComboBox component and navigate through the photos with the two buttons.

The XML document
We created the XML document photoGallery.xml in Chapter 3. The following code shows a summary of the document structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<photoGallery>
  <location locationName="galleryName">
    <photo filename="filename.jpg" caption="caption content">
      Text
    </photo>
    <photo filename="filename.jpg" caption="caption content">
      text
    </photo>
  </location>
  <location locationName="galleryName">
    <photo filename="filename.jpg" caption="caption content">
      Text
    </photo>
  </location>
</photoGallery>
```

The root element <photoGallery> contains one or more <location> elements. Each <location> has a single attribute locationName and contains one or more <photo> elements. The <photo> elements contain a filename and caption attribute as well as some descriptive text.

Feel free to update the XML document and photos folder with your own contents. The Flash movie is set up for landscape images with a width of up to 290 pixels, but you can change the movie if you're using differently sized images.

Loading the XML document into Flash
We'll load the XML document into the gallery.fla movie.

3. Create a new layer in the gallery.fla file and name it actions.
4. Click the first frame of the actions layer and open the Actions panel with the *F9* shortcut key.
5. Add the code shown here. This code loads the document photoGallery.xml into the photoXML object. When the loading is completed, the loadPhotos function displays the contents of the XML object.

```
var photoXML:XML = new XML();
photoXML.ignoreWhite = true;
photoXML.onLoad=loadPhotos;
photoXML.load("photoGallery.xml");
stop();
function loadPhotos(success:Boolean):Void{
  if (success) {
    trace (this);
  }
  else {
    trace("Error in loading XML file");
  }
}
```

6. Save the movie and test it with the *Ctrl-Enter* shortcut key (*Cmd-Return* on a Macintosh). You should see something similar to the image shown in Figure 4-12.

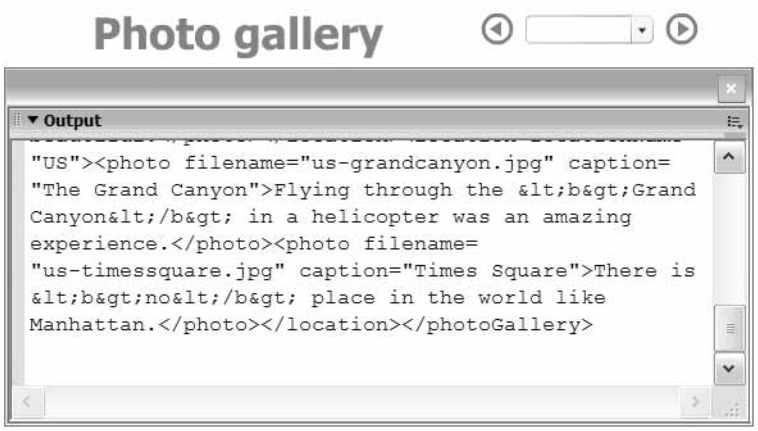


Figure 4-12. Testing that the XML document has been loaded into Flash

You'll notice that Flash has ignored the CDATA block from the XML document. Instead, the <b> tag has been converted to the HTML entities < and >. As we discovered earlier, Flash doesn't recognize CDATA.

Once the XML document has been loaded into the photoXML object, it's time to start populating the interface.

Loading the ComboBox component

We'll start by adding the names of each location to the ComboBox component. The name comes from the locationName attribute in the <location> element.

7. Create a new XMLNode variable called RootNode at the top of the actions layer. We create it outside of the loadPhotos function so that we can refer to the root node of the XML object anywhere within the Flash movie.

```
var RootNode:XMLNode;
```

8. Modify the loadPhotos function as shown here. I've indicated the new lines in bold. The function calls the loadCombo function after successfully loading the XML file.

```
function loadPhotos(success:Boolean):Void{
  if (success) {
    RootNode = this.firstChild;
    loadCombo();
  }
  else {
    trace("Error in loading XML file");
  }
}
```

9. Add the `loadCombo` function below the `loadPhotos` function. I've used the `addItem` method of the ComboBox component to add the `locationName` attribute values. Notice that I've done this inside a loop so that I can process all child elements of the root node in the same way. I have also added an item --Select-- at the beginning of the ComboBox.

```
function loadCombo():Void {
  var galleryName:String;
  gallery_cb.addItem("-- Select --");
  for (var i:Number=0; i< RootNode.childNodes.length; i++) {
    galleryName = RootNode.childNodes[i].attributes.locationName;
    gallery_cb.addItem(galleryName);
  }
}
```

10. Test the movie again. You should see the gallery names in the ComboBox component as shown in Figure 4-13.

Figure 4-13. Testing that the ComboBox component has been populated

At the moment, when we select a value from the ComboBox component, nothing happens. We actually want the first image from the selected gallery to be displayed on the Stage. To achieve that, we'll need to add an event listener to the ComboBox component.

Adding an event listener to the ComboBox

ActionScript is an event-driven language. We use it to respond to events that occur in a movie, for example, the click of a button or selecting a value from a ComboBox. Because we want something specific to happen when these events occur, we can use an event listener. Event listeners listen for specific events and respond by calling a function.

We want an image to display when the value of the item in the ComboBox changes. We can only do this with an event listener that listens for the change event of the ComboBox. When the listener detects that event, it will call a function to deal with the changed value in the ComboBox.

11. Add the following code above the `loadPhotos` function. The code creates an object called `CBOListener`, which listens for the change event. When the event fires, the listener calls the `loadGallery` function.

```
var CBOListener:Object = new Object();
CBOListener.change = loadGallery;
gallery_cb.addEventListener("change", CBOListener);
```

12. Add the `loadGallery` function below the `loadCombo` function. The function receives the object that called it as a parameter, that is, the listener. It traces the label of the selected option using `evtObj.target` to locate the target of the event listener.

```
function loadGallery(evtObj:Object):Void {
  trace (evtObj.target.selectedItem.label);
}
```

13. Test the movie. You should see something similar to Figure 4-14 when you make a selection in the ComboBox.

Figure 4-14. Testing the ComboBox listener

Once we have detected the selected gallery, we can move to that section of the document tree and load the first image.

Loading the photos

The first task is to move through the document tree and locate the selected gallery. We'll create some variables to help out.

14. Add new variables called `selectedGallery`, `photoPosition`, `GalleryNode`, and `PhotoNode` below the `RootNode` variable at the top of the Actions panel. Set the types to `String`, `Number`, and `XMLNode`, as shown here. These variables will have scope throughout the Flash movie because we haven't created them within a function.

```
var RootNode:XMLNode;
var selectedGallery:String;
var photoPosition:Number;
var GalleryNode:XMLNode;
var PhotoNode:XMLNode;
```

15. Modify the `loadGallery` function as shown here. The new lines appear in bold. The function finds if we've selected a gallery and sets the variable `selectedGallery`. It then finds the correct gallery and sets the position in the document tree within the variable `GalleryNode`. Finally, it calls the `loadPhoto` function, passing a value of `0` to indicate that the first image should display. The `break` statement ends the loop.

```
function loadGallery(evtObj:Object):Void {
  var galleryName:String;
  if (evtObj.target.selectedItem.label !="– Select –") {
    selectedGallery = evtObj.target.selectedItem.label;
    for (var i:Number=0; i< RootNode.childNodes.length; i++) {
      galleryName = RootNode.childNodes[i].attributes.locationName;
      if (galleryName ==  selectedGallery) {
```

```
            GalleryNode = RootNode.childNodes[i];
            photoPosition=0;
            loadPhoto(photoPosition);
            break;
          }
        }
      }
    }
```

16. Add the function `loadPhoto` below the `loadGallery` function. This function traces the file name of the first image in the gallery.

```
function loadPhoto(nodePos:Number):Void {
  trace (GalleryNode.firstChild.attributes.filename);
}
```

17. Test the movie. Select a gallery. You should see something similar to the image shown in Figure 4-15.

Figure 4-15. Testing the loadPhoto function

Now we need to use the file name to display the image from the `photos` folder in the empty movie clip. We also need to add text from the XML document to the caption and comment fields.

18. Modify the `loadPhoto` function as shown here. The function sets the `PhotoNode` variable and finds the file name, caption, and comments from the document tree. The `loadMovie` action loads the image from the `photos` folder into `empty_mc`. The `text` and `htmlText` properties display the caption and comments. Notice that we set the `html` property of the `comment_txt` field to true so that it can render the HTML tags from the CDATA section of the XML document.

```
function loadPhoto(nodePos:Number):Void {
  PhotoNode = GalleryNode.childNodes[nodePos];
  var filename:String = PhotoNode.attributes.filename;
  var caption:String = PhotoNode.attributes.caption;
  var comments:String = PhotoNode.firstChild.nodeValue;
  empty_mc.loadMovie("photos/" + filename);
  caption_txt.text = caption;
```

```
    comment_txt.html = true;
    comment_txt.htmlText = comments;
}
```

19. Test the movie again and select an image gallery. You should see something similar to Figure 4-16.

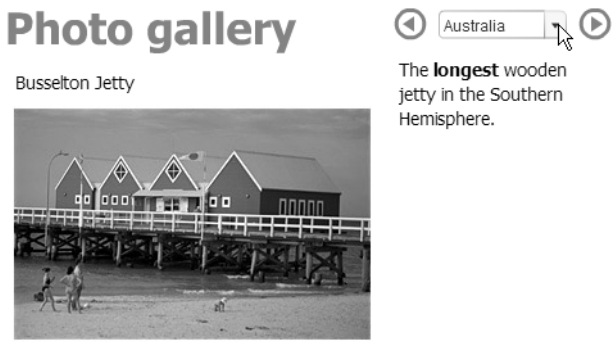


Figure 4-16. Testing that the first image loads

If you select the other galleries, the first image from each will load. A problem arises when we choose the `-- Select --` option. The current image, caption, and comment remain. It would be better if this selection cleared the image and the text fields.

20. Change the `loadGallery` function as shown here. If we choose the `-- Select --` item, `empty_mc` is unloaded and the text fields are cleared.

```
function loadGallery(evtObj:Object):Void {
  var galleryName:String;
  if (evtObj.target.selectedItem.label !="-- Select --") {
    selectedGallery = evtObj.target.selectedItem.label;
    for (var i:Number=0; i< RootNode.childNodes.length; i++) {
      galleryName = RootNode.childNodes[i].attributes.locationName;
      if (galleryName ==  selectedGallery) {
        GalleryNode = RootNode.childNodes[i];
        photoPosition=0;
        loadPhoto(photoPosition);
        break;
      }
    }
  }
  else {
    empty_mc.unloadMovie();
    caption_txt.text = "";
    comment_txt.htmlText = "";
  }
}
```

21. Test the movie again. Select an image gallery and then choose the first option in the ComboBox. The interface should clear.

So far, we can view the first image in each image gallery. The next step is to configure the buttons so we can navigate through all of the photos in each gallery.

Configuring the buttons

The gallery movie includes two buttons: back_btn and forward_btn. When the back button is pressed, we should move to the previous image, if one exists. Conversely, the forward button should move us to the next image. We'll start by disabling the buttons so that we can't navigate until after we've selected an image gallery.

22. Enter the following lines above the photoXML variable declaration at the top of the actions layer:

```
back_btn.enabled = false;
forward_btn.enabled = false;
```

23. Change the loadGallery function. We'll need to enable the buttons after we have selected a gallery. We'll also need to disable the buttons when the -- Select -- option is chosen and clear the selectedGallery variable.

```
function loadGallery(evtObj:Object):Void {
  var galleryName:String;
  if (evtObj.target.selectedItem.label !="-- Select --") {
    selectedGallery = evtObj.target.selectedItem.label;
    for (var i:Number=0; i< RootNode.childNodes.length; i++) {
      galleryName = RootNode.childNodes[i].attributes.locationName;
      if (galleryName ==  selectedGallery) {
        GalleryNode = RootNode.childNodes[i];
        photoPosition=0;
        loadPhoto(photoPosition);
        back_btn.enabled = true;
        forward_btn.enabled = true;
        break;
      }
    }
  }
  else {
    empty_mc.unloadMovie();
    caption_txt.text = "";
    comment_txt.htmlText = "";
    selectedGallery = "";
    back_btn.enabled = false;
    forward_btn.enabled = false;
  }
}
```

24. Test the movie. Check that the buttons are enabled and disabled as you choose different gallery options.

Finally, we'll need to make the back and forward buttons work. We created the variable photoPosition earlier so we could keep track of the current photo number and the childNode within the selected gallery.

25. Enter the following lines above the loadPhotos function at the top of the actions layer. The onRelease functions test whether there is a previousSibling or nextSibling of the current PhotoNode. If so, we either decrement or increment the photoPosition variable and call the loadPhoto function.

```
back_btn.onRelease = function():Void {
  if (PhotoNode.previousSibling.nodeName != undefined) {
    photoPosition--;
    loadPhoto(photoPosition);
  }

forward_btn.onRelease = function():Void {
  if (PhotoNode.nextSibling.nodeName != undefined) {
    photoPosition++;
    loadPhoto(photoPosition);
  }
}
```

26. Test the movie for the last time and check that the gallery is functioning properly. Congratulations on completing the exercise. You can find the completed file saved as gallery_completed.fla in your resources for this chapter.

Points to note from exercise 3

- It's important to define variables in the appropriate place. If you'll only use a variable inside a function, you should define it inside that function using a var statement. When the function has finished running, the variable will cease to exist. This type of variable has *local* scope.

  If you want to use a variable in more than one function, you'll need to define it outside the functions. The variable will then have *timeline* scope, and it will be available to every block of code on the current timeline. I normally list these variables at the top of a layer for convenience.

  You can set, retrieve, and change the values of timeline variables within functions. The RootNode variable is a perfect example of a timeline variable. We declare the variable at the top of the actions layer but its value isn't set until we call the loadPhotos function.

- There is probably a more elegant way of dealing with the back and forward action of the buttons. For example, I could have disabled the back button if we were at the first photo and the forward button if we were at the last photo in the collection. Instead, I chose to use the previousSibling and nextSibling properties so you could see how they work.

- If you followed the example, you'll notice that, in each step, I wrote a little bit of code and then tested the movie. It's very important that you test your movie regularly. If you leave it too long before testing, it will be much harder to debug than if you have made only small changes each time. Sometimes, errors can compound and create strange results, making it hard to track down the cause of the problem.

- You probably also noticed that I was fairly specific about where you should place the code on the actions layer. In fact, you could have put the code in just about any order and the gallery would still have worked. I wanted you to keep your code in a logical order so you could compare your content with the completed file. That way it'll be easier for you to locate any errors in your code.