

Foundation XML and E4X for Flash and Flex

Sas Jacobs



Foundation XML and E4X for Flash and Flex

Copyright © 2009 by Sas Jacobs

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1634-6

ISBN-13 (electronic): 978-1-4302-1635-3

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

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 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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

Credits

Lead Editor

Ben Renow-Clarke

Production Editor

Ellie Fountain

Technical Reviewer

Kevin Ruse

Composer

Patrick Cunningham

Editorial Board

Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham,
Tony Campbell, Gary Cornell, Jonathan Gennick,
Michelle Lowman, Matthew Moodie, Jeffrey Pepper,
Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft,
Matt Wade, Tom Welsh

Proofreader

Lisa Hamilton

Indexer

Broccoli Information Management

Project Manager

Beth Christmas

Artist

April Milne

Copy Editor

Marilyn Smith

Cover Image Designer

Corné van Dooren

Associate Production Director

Kari Brooks-Copony

Interior and Cover Designer

Kurt Krames

Manufacturing Director

Tom Debolski





Chapter 3

ACTIONSCRIPT 3.0 AND XML

So far in this book, you've been introduced to XML and seen some different ways to generate XML content for SWF applications. In this chapter, you'll learn about the role of XML in ActionScript 3.0. If you have been coding with ActionScript 2.0, be prepared to forget any previous experience you have with ActionScript and XML, because things have changed radically.

ActionScript 3.0 completely reworks the XML class and also introduces a new XMLList class. Both represent a leap forward for developers, as the changes will speed up the development process greatly. The new functionality is based on ECMAScript standards and is referred to as E4X.

Another major change is that XML is now a native data type. This means that you can declare and explicitly assign content to XML objects, in much the same way that you would with strings and numeric types.

You'll learn about these new ActionScript 3.0 features in this chapter. You can download the resources for this chapter from <http://www.friendsofed.com>.

Differences between ActionScript 2.0 and 3.0

If you've worked with earlier versions of ActionScript, you'll be familiar with the ActionScript 2.0 XML class. Unfortunately, you'll need to forget what you've previously learned, because ActionScript 3.0 changes things quite significantly. The good news is that the new approaches make life much easier for developers.

ActionScript 3.0 includes completely new XML functionality based on the E4X specification, ECMA-357. E4X is an extension to ECMAScript (JavaScript) specifically targeted at working with XML documents. It is a standard, managed by an international body called the European Computer Manufacturers Association (ECMA), which allows an XML document to be defined as a JavaScript object or, in our case, an ActionScript object.

E4X is implemented in a number of different areas, including a full implementation within ActionScript 3.0. Unfortunately, there is only limited support for E4X in JavaScript within current web browsers.

In ActionScript 3.0, you can target content in an XML object by using methods of the XML class to write E4X expressions. You can also use some shorthand expressions that are similar to XPath, and I'll cover those in the next chapter.

E4X expressions allow you to target the content in an XML object by writing paths that refer to node and attribute names. If you've ever had to write paths or loop through XML content using ActionScript 2.0, you'll welcome this functionality with open arms!

The introduction of E4X to ActionScript 3.0 has the following advantages:

- ActionScript 3.0 now uses a standardized approach.
- E4X usually produces far less code to parse XML content compared with ActionScript 2.0.
- E4X expressions are simple and are generally easier to understand than the equivalent ActionScript 2.0 expressions.
- E4X is easy to learn, especially for developers with experience in XPath.

You can find out more about E4X by reading the specification at <http://www.ecma-international.org/publications/standards/Ecma-357.htm>. If you want to know more about the XPath specification, you can find it at <http://www.w3.org/TR/xpath>, although it's not essential for using E4X in Flash and Flex.

E4X introduces several new classes to ActionScript 3.0, as follows:

- XML: This class works with XML objects.
- XMLList: This class works with an ordered collection of XML content and may involve more than one XML object.
- XMLListCollection: This class is a wrapper collection class for the XMLList class.
- QName: This class represents the qualified name of XML elements and attributes.
- Namespace: This class works with namespaces.

The old ActionScript 2.0 XML class has been renamed to the XMLDocument class for support of legacy applications. Using this class for new SWF applications is not recommended. XMLNode is also available as a legacy class.

We'll take a closer look at these new classes in this chapter. Before we start though, let's look at XML as a native data type in ActionScript 3.0.

XML as an ActionScript data type

ActionScript 3.0 includes XML and XMLList as new complex data types. This means that you can write XML inline within ActionScript code. If you are working with Flex, you can also use a tag-based approach to create literal XML content.

Writing XML inline within ActionScript

If you've worked with ActionScript 2.0, you're probably familiar with passing a string to the XML constructor to create an XML object, as shown here:

```
var strXML:String = "<phoneBook><contact>Sas</contact></phoneBook>";
var phoneBookXML:XML = new XML(strXML);
```

While you can still create XML content in this way, ActionScript 3.0 allows you to write XML inline within ActionScript code using a literal value, as shown here:

```
var phoneBookXML:XML = <phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Red Street, Redland, Australia</address>
    <phone>123 456</phone>
  </contact>
</phoneBook>;
```

In this case, I've created an XML object with `<phoneBook>` as the root element. The object contains a single `<contact>` element. Notice that I don't need to use quotes around the XML content, and it can be split onto different lines. The `phoneBookXML` object has a data type of XML, not String.

The only requirement for creating an XML object in this way is that the content must be well-formed. This requirement also applies if you're passing a string to the XML constructor method. (If you're not sure what *well-formed* means, refer to Chapter 1.)

You can write a path to locate the root element in an XML object by using the object name. In this example, I can use the name `phoneBookXML` to reference the root of the XML object, `<phoneBook>`. The root node is the starting point for most of the XML methods covered in this chapter.

You normally create an XMLList object by applying an E4X expression to an XML object. You'll find out more about these expressions later in this chapter and in the next chapter.

Writing XML with the XML tag in Flex

You can use the `<mx:XML>` tag in Flex to work with literal XML content in an XML data model. This tag compiles the data into an XML object, which you can work with using E4X expressions.

You can either add the content directly to the `<mx:XML>` tag or specify an external document source. The following block shows how to declare the content of an `<mx:XML>` element explicitly:

```
<mx:XML id="phoneBookXML">
  <phoneBook>
    <contact id="1">
      <name>Sas Jacobs</name>
      <address>123 Red Street, Redland, Australia</address>
      <phone>123 456</phone>
    </contact>
  </phoneBook>
</mx:XML>
```

This example is equivalent to the ActionScript variable created in the previous section. As with the ActionScript 3.0 equivalent, the content inside an `<mx:XML>` tag must be well-formed XML.

One advantage of using the `<mx:XML>` tag is that you can specify an external document as the content using the `source` attribute, as shown here:

```
<mx:XML id="phoneBookXML" source="assets/address.xml"/>
```

This `<mx:XML>` element loads content from the `address.xml` file in the `assets` folder of the Flex project.

Be careful with this approach though, as the `<mx:XML>` tag compiles the data into the SWF file, rather than loading it at runtime. You can't modify the external XML document and expect the application to update the content automatically. If you need that functionality, you must load the content using one of the methods shown in Chapter 5. For simplicity, in this chapter we'll keep working with static XML content stored in the SWF application.

Next, let's look at the new ActionScript 3.0 E4X classes.

Overview of the new ActionScript 3.0 classes

Before we explore the new classes in detail, it's useful to understand the general purpose of each one, as well as the way the classes interact with each other. I'll provide a brief summary of the new classes here.

The ActionScript 3.0 XML class

The ActionScript 3.0 XML class allows you to work with XML content. As mentioned earlier, this class implements the E4X standard for working with XML documents. In the previous section, you saw that it was possible to declare an XML object and assign a value using ActionScript.

An XML object contains a single root node with some optional child nodes. An `XMLList` object is a collection of XML content or nodes. It doesn't have a root node as a container for the other elements.

The XMLList class

You can create an XMLList through various E4X expressions and methods of the XML class. For example, you can use the children() method of an XML object to return a collection of all child nodes of an element. You can also use an E4X expression to create a list of nodes matching certain criteria, with the expression acting as a filter for the XML content.

The XMLList class represents an ordered collection of XML elements. Many of the XML class methods that you'll see shortly return an XMLList. One useful feature of an XMLList is that you can assign it as a dataProvider for Flex components.

The difference between XMLList and XML objects is that the XML object is a single object containing any number of child nodes, whereas an XMLList is a collection of one or more objects. However, the distinction blurs somewhat, because an XMLList object containing a single item is treated in the same way as an XML object.

So how can you tell whether you're working with an XML object or an XMLList object? If you call the length() method, the value will always be 1 in the case of an XML object. An XMLList can have any other value, although an XML class method returning an XMLList with a single element will also show a value of 1.

The XMLListCollection class

If you're working in Flex, it's useful to assign the XMLList to an XMLListCollection object and use that as the basis for data binding. The XMLListCollection provides additional methods for working with the XMLList content, and the binding is monitored for changes. This isn't the case if you bind an XMLList.

You'll usually use an XMLListCollection object as the dataProvider property for another component. In addition to XMLList methods, it contains methods for sorting data, adding items, and editing items.

The QName and Namespace classes

You'll use the QName and Namespace classes when you work with XML content that falls within a specific namespace. Remember that namespaces associate XML elements with an owner, so that each element name is unique within the XML document. You can create a QName object to associate an element with a specific namespace or to identify the element uniquely with a string.

Now let's look at the new classes in more detail, beginning with the XML class.

Working with the XML class

The XML class class allows you to work with XML documents. XML documents must have a single root node and be well-formed before they can be parsed by either Flash or Flex.

You can either declare the XML content within your SWF application or load it from an external source. You saw the first option earlier in this chapter. You'll see how to load external content in Chapter 5.

Each XML object can include any or all of the following node types, called *node kinds*:

- An element
- An attribute
- A text node
- A comment
- A processing instruction

To understand the XML class a little better, let's look at its properties and methods, and work through some examples of how they might apply in both Flash and Flex.

Properties of the XML class

The XML class class has five static properties that determine how the XML content is treated. These properties are listed in Table 3-1. Because they are static properties, they determine the overall settings for XML objects in your SWF applications, rather than applying to a specific instance.

Table 3-1. Static properties of the XML class

Property	Type	Description	Default value
<code>ignoreComments</code>	Boolean	Determines whether to ignore comments when the source XML content is parsed	<code>true</code>
<code>ignoreProcessingInstructions</code>	Boolean	Determines whether to ignore processing instructions while parsing the XML document	<code>true</code>
<code>ignoreWhitespace</code>	Boolean	Determines whether to ignore whitespace when parsing the XML document	<code>true</code>
<code>prettyIndent</code>	int	Determines the amount of indentation, in spaces, when <code>prettyPrinting</code> is set to <code>true</code>	2
<code>prettyPrinting</code>	Boolean	Determines whether whitespace is preserved when the XML document displays with either the <code>toString()</code> or <code>toXMLString()</code> method	<code>true</code>

We'll look at the effect of these properties in an example using the following XML object:

```
<phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Red Street, Redland, Australia</address>
    <phone>123 456</phone>
  </contact>
</phoneBook>
```

We'll work through examples in both Flash and Flex, starting with Flash.

Working with XML properties in Flash

Follow these steps in Flash:

1. Open Flash and create an XML object called phoneBookXML. Assign the preceding XML content to this object, as shown here:

```
var phoneBookXML:XML = <phoneBook>
  <contact id="1">
    <name>Sas Jacobs</name>
    <address>123 Red Street, Redland, Australia</address>
    <phone>123 456</phone>
  </contact>
</phoneBook>
```

2. The `toXMLString()` method of the XML class provides a string representation of the XML object. Add the following lines below the XML object:

```
XML.prettyPrinting = false;
trace(phoneBookXML.toXMLString());
```

This example turns off the `prettyPrinting` property so you can see unformatted XML content.

3. Test the SWF file using the Ctrl/Cmd+Enter keyboard shortcut. Figure 3-1 shows the output from this simple ActionScript code. As you can see, the content renders without indentation or any spacing that would make it easier to read.

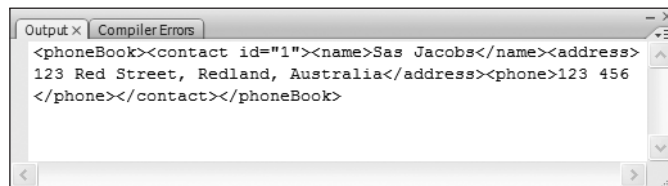


Figure 3-1. Tracing the XML object without comments or pretty printing

4. Modify the code as follows:

```
XML.prettyIndent = 4;
XML.prettyPrinting = true;
trace(phoneBookXML.toXMLString());
```

You've turned on the `prettyPrinting` setting and set an indent of four spaces. Figure 3-2 shows how this content appears in the Output panel. The output now uses pretty printing to lay out the content. It includes four characters of spacing between indented lines because of the `prettyIndent` and `prettyPrinting` property settings.



Figure 3-2. Tracing the XML object with pretty printing

You can find the file used for this example saved as `xmlObjectStaticProperties.fla` with this chapter's resources.

Working with XML properties in Flex

You can re-create the preceding example in Flex by creating a project and application file, as follows:

1. Add the XML content to the application in an `<mx:XML>` element:

```
<mx:XML id="phoneBookXML">
  <phoneBook>
    <contact id="1">
      <name>Sas Jacobs</name>
      <address>123 Red Street, Redland, Australia</address>
      <phone>123 456</phone>
    </contact>
  </phoneBook>
</mx:XML>
```

2. Modify the `<mx:Application>` element to add a `creationComplete` attribute, as follows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" creationComplete="initApp(event)">
```

3. Add the following `<mx:Script>` block:

```
<mx:Script>
  <![CDATA[
    import mx.events.FlexEvent;
    private function initApp(e:FlexEvent):void {
      XML.prettyPrinting = false;
      trace(phoneBookXML.toXMLString());
    }
  ]]>
</mx:Script>
```

The script block imports a `FlexEvent` as that object is passed within the `initApp()` function. It creates the `initApp()` function, which sets the `prettyPrinting` static property to `false`. As with the previous example, it displays the XML content using `toXMLString()`.

4. Debug the application to see the output in the Console view, which will look similar to the output shown in Figure 3-1.
5. Replace the `prettyPrinting` line with the following code:


```
XML.prettyIndent = 4;
XML.prettyPrinting = true;
```
6. Debug the application again. This time, you'll see the same effect as in Figure 3-2, with the XML content formatted and easy to read.

You can find the MXML application saved in the resource file `xmlObjectStaticProperties.mxml`.

In addition to the static properties, the `XML` class also has a number of methods.

Methods of the XML class

You can access content within an `XML` object by using E4X expressions, as discussed in the next chapter, or by using methods of the `XML` class. Many of these methods return an `XMLList` object. There are also a number of methods that developers can use to create and modify XML content, which are covered in Chapter 8.

The most common methods of the `XML` class can be divided into the following categories:

- Methods that help to locate XML content
- Methods that find out information about XML content
- Methods that modify XML content

We'll start by looking at methods for locating XML content.

Locating XML content

The `XML` class includes many methods that allow you to parse an XML object to locate specific information. This information is always treated as a `String` data type, so you'll need to cast any values that should be treated as numbers or dates.

As I mentioned, there are two ways to access content from an XML object: use the `XML` class methods or use E4X expressions as a kind of shorthand. In this chapter, we'll focus on using XML methods, and the shorthand E4X expressions are covered in Chapter 4. In most cases, you can use either approach to locate values within your XML content. However, the following methods of the `XML` class don't have an equivalent E4X expression:

- Locating the parent with the `parent()` method
- Using `comments()` to access the collection of comments in the XML object
- Using `processingInstructions()` to access the collection of processing instructions

Table 3-2 shows some of the most common methods that allow you to access content in an XML document. These methods also apply to the `XMLElement` class.

Table 3-2. Methods of the XML class that assist in locating XML content

Method	Description
<code>attribute(attributeName)</code>	Returns the value of a specified attribute as an <code>XMLElement</code> . The attribute name can be a string or a <code>QName</code> object.
<code>attributes()</code>	Returns a list of attribute values for a specified node as an <code>XMLElement</code> .
<code>child(propertyName)</code>	Returns the child nodes of a specified node matching the supplied name.
<code>children()</code>	Returns all children of the specified node as an <code>XMLElement</code> .
<code>comments()</code>	Returns all comments within an XML object.
<code>descendants(name)</code>	Returns all descendants of an XML object as an <code>XMLElement</code> .
<code>elements(name)</code>	Lists the elements of an XML object as an <code>XMLElement</code> . This method ignores comments and processing instructions.
<code>parent()</code>	Returns the parent of the specified XML object as an <code>XMLElement</code> .
<code>processingInstructions(name)</code>	Returns processing instructions.
<code>text()</code>	Returns all text nodes.

Let's look at examples of some of these methods.

Instructions for the code samples

The code samples for both Flash and Flex are identical, but you need to add the code at different locations in the Flash and Flex files.

For the Flash examples, follow these steps:

1. Open up a new Flash document.
2. Add the contents of the `authors.xml` document to an XML object called `authorsXML` on an Actions layer. Here are the first two lines, to get you started:

```
var authorsXML:XML = <allAuthors>
    <author authorID="1">
```

3. Add the ActionScript lines shown in the following examples below the declaration for the XML object.

Follow these steps for the Flex Builder files:

1. Create a new project and application file using File ► New ► Flex Project.
2. Create an assets folder for the project and add the XML document authors.xml.
3. Add an <mx:XML> element and set this XML element as the source for the XML object, as shown here:

```
<mx:XML id="phoneBookXML" source="assets/authors.xml" />
```

4. Add the following creationComplete event to the <mx:Application> element:
creationComplete="initApp(event)"
5. Add the following <mx:Script> block containing the initApp() function above the <mx:XML> object:

```
<mx:Script>
  <![CDATA[
    import mx.events.FlexEvent;
    private function initApp(e:FlexEvent):void {
      //add testing lines here
    }
  ]]>
</mx:Script>
```

6. In the examples that follow, replace the //add testing lines here content with the code samples. Because you'll be using trace() statements, you'll need to debug rather than run the Flex application.

The complete Flex code follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" creationComplete="initApp(event)">
  <mx:Script>
    <![CDATA[
      import mx.events.FlexEvent;
      private function initApp(e:FlexEvent):void {
        //add testing lines here
      }
    ]]>
  </mx:Script>
  <mx:XML id="authorsXML" source="assets/authors.xml" />
</mx:Application>
```

The authors.xml file contains information about several authors and the books that they've published. We'll use the content as an XML object to demonstrate the most important methods listed in Table 3-2.

To start, we'll look at the attribute() and attributes() methods.

Working with attribute() and attributes()

You can use the `attribute()` method to locate a specific attribute in an element. You need to provide the path through the document tree before calling the `attribute()` method. Remember that the name of the XML object is equivalent to its root node.

```
trace(authorsXML.author[1].attribute("authorID"));
```

This expression finds the `authorID` attribute of the second `<author>` element. The expression starts by referencing the `<authorsXML>` root element using `authorsXML`. It then locates the second author using `author[1]`. You use the number 1 because the list of elements is zero-based, so the first author is `author[0]`. The expression uses the `attribute()` method, passing the name of the attribute, `authorID`. When you test the SWF application, the Output panel should show the value 2, as shown in Figure 3-3.

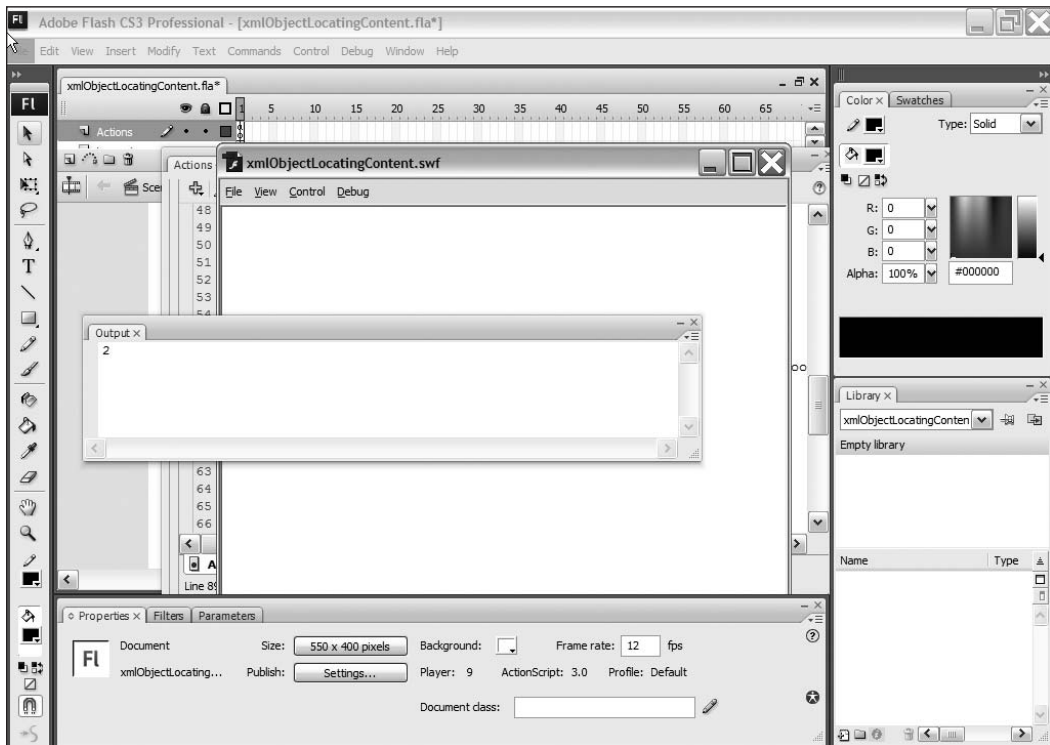


Figure 3-3. The Flash Output panel showing the result of using the `attribute()` method

If you need to access all of the attributes of an element or collection of elements, you can use the `attributes()` method, as follows:

```
trace(authorsXML.author.attributes());
```

This expression will display the output 1234, because these are all of the attribute values in the `<author>` elements of the `authorsXML` object.

You can access individual attributes from this collection by specifying an index. For example, to find all attributes of the first <author> element, use the following:

```
trace(authorsXML.author[0].attributes());
```

Testing that expression will display the value 1.

Finding child elements

You can find the child elements of a parent element by calling the `child()` method and specifying the name of the child element to locate. Here is an example:

```
trace(authorsXML.author[2].books.book.child("bookName"));
```

In this case, the expression finds all of the <bookName> elements associated with the children of the third <author> element. Remember that the first author is at position 0. Notice that you need to specify the element structure from the root element, including the <books> and <book> elements. The expression produces the following output:

```
<bookName>Bikes as an alternative source of transport</bookName>
<bookName>Entertaining ways</bookName>
<bookName>Growing radishes</bookName>
<bookName>Growing tulips</bookName>
```

The output provides a collection of all of the <bookName> elements for this author. You could assign this expression to an `XMLList` object to work with it further.

The `children()` method accesses all children of a specific element. The following expression finds all children of the fourth author:

```
trace(authorsXML.author[3].children());
```

It produces the following output:

```
<authorFirstName>Saul</authorFirstName>
<authorLastName>Sorenson</authorLastName>
<books>
  <book ID="2">
    <bookName>Bike riding for non-bike riders</bookName>
    <bookPublishYear>2004</bookPublishYear>
  </book>
</books>
```

Notice that it finds the <authorFirstName>, <authorLastName>, and <books> child elements.

Finding descendants

The `descendants()` method allows you to access all of an element's descendants, including text nodes. The returned content includes the child elements, grandchildren, and every element at lower levels in the XML object. Here is an example of using this method:

```
trace(authorsXML.author[3].books.descendants());
```

This expression finds all descendants of the <books> element of the fourth author. It produces the following output:

```
<book ID="2">
  <bookName>Bike riding for non-bike riders</bookName>
  <bookPublishYear>2004</bookPublishYear>
</book>
<bookName>Bike riding for non-bike riders</bookName>
Bike riding for non-bike riders
<bookPublishYear>2004</bookPublishYear>
2004
```

Notice that each descendant is listed in turn, starting with the complete <book> element, then the <bookName> element and its text, followed by the <bookPublishYear> element and its text.

It's also possible to pass the name of an element to match within the descendants() method, as shown here:

```
trace(authorsXML.author[3].books.descendants("bookPublishYear"));
```

This expression returns the value 2004.

Finding elements

The elements() method lists all of the elements of the XML object, ignoring comments and processing instructions. Unlike the descendants() method, elements() doesn't return the text nodes separately. You can see how this works by testing the following expression:

```
trace(authorsXML.author[3].books.elements());
```

This time, you'll see the following output:

```
<book ID="2">
  <bookName>Bike riding for non-bike riders</bookName>
  <bookPublishYear>2004</bookPublishYear>
</book>
```

Compare it with the earlier output from the descendants() method to see the difference.

Finding the parent element

The parent() method returns the complete parent element for the child path specified. Here is an example:

```
trace(authorsXML.author[0].books.parent());
```

This expression returns the first `<author>` element, complete with all of its children:

```
<author authorID="1">
  <authorFirstName>Alison</authorFirstName>
  <authorLastName>Ambrose</authorLastName>
  <books>
    <book bookID="1">
      <bookName>Shopping for profit and pleasure</bookName>
      <bookPublishYear>2002</bookPublishYear>
    </book>
    <book ID="4">
      <bookName>Fishing tips</bookName>
      <bookPublishYear>1999</bookPublishYear>
    </book>
  </books>
</author>
```

Locating text

The `text()` method returns all of the text nodes in an element. Note that these must be children of the specific element, rather than descendants. Here is an example of using this method:

```
trace(authorsXML.author[0].books.book.bookName.text());
```

If you test the expression, you will see the text nodes under all `<bookName>` elements for the first author:

```
Shopping for profit and pleasureFishing tips
```

It's possible to specify a single node by adding an index, as in this expression:

```
trace(authorsXML.author[0].books.book[1].bookName.text());
```

In this case, the output would display as follows:

```
Fishing tips
```

You can also combine the `text()` method with other methods, such as `children()`, as in this example:

```
trace (authorsXML.author[0].books.book.children().text());
```

Testing this expression produces the following output:

```
Shopping for profit and pleasure2002Fishing tips1999
```

You see all of the text inside the child elements of this author's `<book>` elements.

You can find all of the expressions discussed in this section in the resource files `xmlObjectLocatingContent fla` and `xmlObjectLocatingContent mxml`. Uncomment the expressions you wish to test. Run the file by pressing `Ctrl/Cmd+Enter` in Flash, or by clicking `Debug` or pressing `F11` in Flex Builder.

Finding information about XML content

Other methods of the `XML` class provide information about the content in an XML object. Table 3-3 summarizes these methods.

Table 3-3. Methods of the `XML` class that provide information about XML content

Method	Description
<code>childIndex()</code>	Identifies the position of the child within its parent node, starting from zero.
<code>hasComplexContent()</code>	Determines whether an XML object contains complex content.
<code>hasSimpleContent()</code>	Determines whether an XML object contains simple content.
<code>length()</code>	Determines the number of nodes in the object. For an XML object, it always returns 1.
<code>localName()</code>	Returns the local name portion of a qualified name of an XML object, without the namespace reference.
<code>name(prefix)</code>	Returns the qualified name of the XML object.
<code>namespace()</code>	Returns the namespace associated with a qualified name.
<code>nodeKind()</code>	Returns the node kind: text, attribute, comment, processing-instruction, or element.
<code>toString()</code>	For complex content, returns XML content as a string containing all tags. It returns text only for simple content.
<code>toXMLString()</code>	Returns all XML content as a string including all tags, regardless of whether the content is simple or complex.
<code>XML ()</code>	Constructor method that creates a new XML object.

Most of these methods are self-explanatory—they return information about the specified XML element. Let’s see how some of these methods work in Flash and Flex. Follow the instructions in the “Instructions for the code samples” section earlier in this chapter to set up the examples.

Finding an object’s position within its parent

You can find the position of an object within its parent by using the `childIndex()` method, as in this example:

```
trace(authorsXML.author[3].books.childIndex());
```

Testing this line in the Flash document will produce the output 2. Because this value is zero-based, it indicates that the `<books>` element is the third child element of the fourth `<author>` element.

Determining content type

You can determine what type of content an XML object contains using the methods `hasComplexContent()` and `hasSimpleContent()`. Simple content contains only text; complex content contains child nodes. Both methods return a Boolean value.

The following example returns a value of `true` because the first `<author>` element has child nodes:

```
trace(authorsXML.author[0].hasComplexContent());
```

The next example returns a value of `true` because the `<bookName>` element contains only text.

```
trace(authorsXML.author[0].books.book[0].bookName.  
    hasSimpleContent());
```

Determining the number of elements

The `length()` method determines the number of elements in an XML object or `XMLList`. In the case of an XML object, the method will always return a value of 1, because an XML object contains a single item. This method returns the number of elements in an `XMLList`. It is typically used to loop through all of the elements of the list, as shown in the following code sample:

```
for (var i:int=0; i < authorsXML.author.length();i++) {  
    trace (authorsXML.author[i].authorLastName);  
}
```

This loop displays each `<authorLastName>` from the `authorsXML` object. When you test your sample file, this ActionScript block returns the following values:

```
Ambrose  
Donaldson  
Larcombe  
Sorenson
```

Displaying the name of an element

Both the `localName()` and `name()` methods return the name of an element. The difference is that the `name()` method returns the qualified name, which will occur if an element is part of a namespace.

The following shows the use of the `name()` method:

```
trace(authorsXML.author[0].children()[1].name());
```

In this case, testing the code will return the string `authorLastName`. Using the `localName()` method would display the same output, as the element is not within a namespace.

Determining the type of node

You can determine the type of node within an XML object using the `nodeKind()` method. This method will return one of the following values: `text`, `comment`, `processing-instruction`, `attribute`, or `element`. Here is an example of using this method:

```
trace(authorsXML.author[0].books.nodeKind());
```

The expression returns the value `element`, indicating that `<books>` is an element.

Displaying a string representation of XML

The `toString()` and `toXMLString()` methods deserve some special attention. You can use both methods to return a string representation of an XML object. However, there is a slight difference. Consider the following XML object:

```
var simpleXML:XML = <contact>Sas Jacobs</contact>;
```

If you're working in Flash, add the XML object to the Actions layer. For Flex, add the following `<mx:XML>` element:

```
<mx:XML id="simpleXML">
  <contact>Sas Jacobs</contact>
</mx:XML>
```

Call both the `toString()` and `toXMLString()` methods on this XML object, as follows:

```
trace(simpleXML.toString());
trace(simpleXML.toXMLString());
```

When you apply the expression `simpleXML.toString()`, it returns the text `Sas Jacobs`. The expression `simpleXML.toXMLString()` returns `<contact>Sas Jacobs</contact>`. In this case, the XML object contains only simple content; that is, it is a single node containing text. It means that the `toString()` method will return only the text content inside the `<contact>` element. The `toXMLString()` method returns the full element, including the opening and closing tags.

If the XML object contained child elements or complex content, both the `toString()` and `toXMLString()` methods would return the same content. It would include all elements and text.

If you use the `trace()` method to display an XML or `XMLList` object without using either `toString()` or `toXMLString()`, the data will display using the `toString()` method.

You can find the examples in this section within the resource file `xmlObjectFindingInformation fla`.

In addition to the methods mentioned previously, there are a number of methods that allow you to manipulate the content in XML and `XMLList` objects.

Modifying XML content

Several methods of the XML class can be used to modify an existing XML object. These are listed in Table 3-4.

You can use these methods to manipulate content within an XML object. For example, you may want to do the following:

- Change the values of nodes and attributes
- Add new nodes
- Duplicate nodes
- Delete nodes

You'll see how to carry out these tasks later in the book, in Chapter 8.

Table 3-4. Methods of the XML class for modifying XML content

Method	Description
<code>appendChild(child)</code>	Inserts a child node at the end of the child nodes collection of the specified node
<code>copy()</code>	Creates a copy of a node
<code>insertChildAfter(child1, child2)</code>	Inserts a child node after a specified child node
<code>insertChildBefore(child1, child2)</code>	Inserts a child node before a specified child node
<code>prependChild(value)</code>	Inserts a child node at the beginning of the child nodes of the specified node
<code>replace(propertyName, value)</code>	Replaces a specified property with a value
<code>setChildren(value)</code>	Replaces children of an XML object with specified content

Working with the XMLList class

An XMLList object is an ordered list of XML objects. It could be part of an XML document or a collection of XML objects. The XMLList class is useful because, being ordered, you can loop through the content. In fact, you saw an example of this a little earlier in the chapter. An XML object is the same as an XMLList that contains a single XML object.

If you're working with an XMLList containing a single XML object, you can use any of the methods of the XML class. Methods such as `childIndex()` wouldn't make sense if you were working with more than one XML object in the XMLList. You can also apply XML class methods on single elements of an XMLList.

If you try to use XML class methods on an XMLList containing more than one XML object, you'll cause an error. Instead, you'll need to loop through the XMLList and apply the methods on each individual XMLList element. For example, you can't find out if an entire XMLList contains simple content. Instead you need to apply the `isSimpleContent()` method on each XMLList element individually.

You can loop through an XMLList in several different ways. Here, I'll show you examples of each type of loop. Again, follow the instructions in the "Instructions for the code samples" section earlier in this chapter to set up the examples.

The first code example shows a simple for loop using a counter variable called `i`.

```
for (var i:int=0; i < authorsXML.author.length();i++) {
    trace (authorsXML.author[i].authorFirstName + " " + authorsXML.
        ↪author[i].authorLastName);
}
```

The expression `authorsXML.author` returns an `XMLList` of all `<author>` elements. You can use the `length()` method to return the number of items in this `XMLList`. In this type of loop, you refer to each item in the list using its position, a zero-based number. The first item in the list is at position 0.

Testing the loop produces the following output:

```
Alison Ambrose
Douglas Donaldson
Lucinda Larcombe
Saul Sorenson
```

You can also use a `for in` loop to work through the `XMLList`. The following code block produces the same output as the previous example when tested:

```
for (var node:String in authorsXML.author) {
    trace(authorsXML.author[node].authorFirstName + " " + authorsXML.
        ▶author[node].authorLastName);
}
```

You declare `node` as a `String` variable because you're using it as a placeholder for the name of the node. This time, you pass the variable to the path `authorsXML.author` to locate each separate element.

A third alternative is to use a `for each` loop, as shown here:

```
for each(var aXML:XML in authorsXML.author) {
    trace(aXML.authorFirstName + " " + aXML.authorLastName);
}
```

This example produces the same output as the previous two examples. You define a placeholder variable called `aXML` to refer to each `authorsXML.author` element in the `XMLList`. You can then use the placeholder in a path with the child node values, as in `aXML.authorFirstName`.

You can find examples of all of these loop types in the resource files `xmlListObjectLoops.fla` and `xmlListObjectLoops.mxml`.

Working with the `XMLListCollection` class in Flex

If you're working with Flex, you can access the `XMLListCollection` class. This class is a wrapper for working with the `XMLList` class. It adds extra collection functionality to an `XMLList`. For example, you can access methods like `removeItemAt()` and `removeAll()`, which aren't available to the `XMLList`.

You would use the `XMLListCollection` object if you wanted to use the content as a data provider, because the collection will update when the XML content changes. In fact, Adobe recommends that you use the `XMLListCollection` object each time you assign an `XMLList` as a data provider. If you bind an `XMLList` object directly, the binding isn't monitored for changes.

To work with an `XMLListCollection` object in ActionScript code, you'll need to import the class first:

```
import mx.collections.XMLListCollection;
```

You can create an `XMLListCollection` object by passing an `XMLList` object when you call the constructor method, `XMLListCollection()`:

```
var myXML_LC:XMLListCollection = new XMLListCollection(myXMLList);
```

You can also create the object without passing the `XMLList` as an argument. You do this by setting the source property of the `XMLList` after it is created, as follows:

```
var myXML_LC:XMLListCollection = new XMLListCollection();
myXML_LC.source = myXMLList;
```

Table 3-5 shows some of the additional properties that the `XMLListCollection` class makes available to an `XMLList`.

Table 3-5. Some of the properties of the `XMLListCollection`

Property	Type	Description
<code>filterFunction</code>	Function	A function that filters items in the list
<code>length</code>	int	The number of items in the list
<code>sort</code>	Sort	Specifies the order for the list
<code>source</code>	<code>XMLList</code>	The underlying <code>XMLList</code>

Two important tasks for developers are filtering and sorting content, so let's see how the `filterFunction` and `sort` properties work with an `XMLListCollection`.

Setting up the Flex application

Follow these steps to create the new Flex application that you'll use to test these examples:

1. Create a new Flex application in the project you set up earlier. You normally would have only one application file per project, but for simplicity, you can keep the files together in the same project.
2. Add the following interface elements:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" creationComplete="initApp(event)">
  <mx:XML id="authorsXML" source="assets/authors.xml"/>
  <mx:VBox x="10" y="10">
    <mx:TextArea id="txtAuthorsXML" width="400" height="100"/>
    <mx:Button id="btnClick" label="Click Me!"
      click="clickHandler(event)"/>
  </mx:VBox>
</mx:Application>
```

The interface contains a `VBox` that holds `TextArea` and `Button` components. The `TextArea` will display a string representation of the XML object. Clicking the button will apply a filter and a sort.

3. Add the following ActionScript code block to the file:

```
<mx:Script>
  <![CDATA[
    import mx.events.FlexEvent;
    import mx.collections.XMLListCollection;
    private var myXMLListCollection:XMLListCollection
    private function initApp(e:FlexEvent):void{
      myXMLListCollection = new XMLListCollection();
      myXMLListCollection.source = authorsXML.child("author")[0].
        child("books").child("book");
      txtAuthorsXML.text = myXMLListCollection.toXMLString();
    }
    private function clickHandler(e:MouseEvent):void {
    }
  ]]>
</mx:Script>
```

The `initApp()` function sets up the object `myXMLListCollection`, which is of the type `XMLListCollection`. The application calls this function after the interface has been created. The function sets the source of the `XMLListCollection` object to an `XMLList` of all `<book>` elements inside the first `<author>` element.

```
authorsXML.child("author")[0].child("books").child("book")
```

The function then displays a string representation of the list in the `txtAuthorsXML` `<mx:TextArea>` element. You'll use the `clickHandler` function to demonstrate the `filterFunction` and `sort` properties of the `XMLListCollection` object.

4. Run the application, and you should see something like the output shown in Figure 3-4.

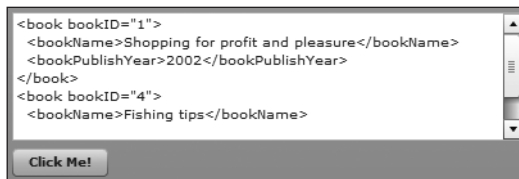


Figure 3-4. Displaying an `XMLListCollection` object

The `TextArea` displays an `XMLListCollection` containing books associated with the first author.

Next, you'll use the `filterFunction` property to apply a filter to this list.

Using a function to filter an `XMLListCollection`

The `filterFunction` property creates a function that filters the items in an `XMLListCollection`. Calling the `refresh()` method of the `XMLListCollection` applies the filter.

The `filterFunction` determines whether a data item matches a filter expression and returns either `true` or `false`. If it returns `true`, the item appears in the filtered list.

You'll add a filter function that filters the XMLListCollection to display the book with an ID of 4. The application will call the function when a user clicks the Click Me! button.

1. Modify the `clickHandler()` function as shown here. The new lines appear in bold.

```
private function clickHandler(e:MouseEvent):void {
    myXMLListCollection.filterFunction = filterBooks;
    myXMLListCollection.refresh();
    txtAuthorsXML.text = myXMLListCollection.toXMLString();
}
```

The function sets the `filterFunction` property of the XMLListCollection object to the `filterBooks` function. It then calls the `refresh()` method to apply the filter, and finishes by displaying the filtered XMLListCollection in the `<mx:TextArea>` element.

2. The `filterBooks()` function needs to be added to the `<mx:Script>` block, as follows:

```
private function filterBooks(item:Object):Boolean {
    var booMatch:Boolean = false;
    if(item.attribute("bookID")== 4){
        booMatch = true;
    }
    return booMatch;
}
```

This function works through all items in the XMLListCollection. It uses a Boolean variable `booMatch` to determine whether the list item should appear in the filtered list. It compares the `bookID` attribute with the value 4. If there is a match, it sets the `booMatch` variable to `true`; otherwise, the variable contains the default value `false`.

It's a common naming convention to prefix the name of a variable with its data type. This example uses a Boolean variable, so I've used the name `booMatch`.

The function returns the value of the `booMatch` variable. Items with a return value of `true` remain in the XMLListCollection when the `refresh()` method is called.

3. Run this application. The TextArea initially displays the entire XMLListCollection object.
4. Click the Click Me! button. The TextArea will display only the book with the `bookID` value of 4, as shown in Figure 3-5.

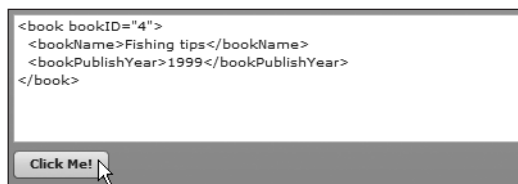


Figure 3-5. Filtering an XMLListCollection object

You can remove the `filterFunction` from an XMLListCollection by setting the value to `null` and applying the `refresh()` method.

Sorting an XMLListCollection

Another useful feature of the `XMLListCollection` class is the ability to sort content. We'll work through an example that changes the `clickHandler()` function to sort the `XMLListCollection` in order of the `<bookName>` element.

1. Before you add the `ActionScript`, you need to import the `Sort` and `SortField` classes. Add the following lines under the other `import` statements:

```
import mx.collections.Sort;
import mx.collections.SortField;
```

2. You'll create a `Sort` object and use the `SortField` class to specify the sort order. Modify the `clickHandler()` function as shown here. The changed lines appear in bold.

```
private function clickHandler(e:MouseEvent):void {
    var mySort:Sort = new Sort();
    mySort.fields = [new SortField("bookName", true)];
    myXMLListCollection.sort = mySort;
    myXMLListCollection.refresh();
    txtAuthorsXML.text = myXMLListCollection.toXMLString();
}
```

The function starts by declaring a new `Sort` object called `mySort`. It then sets the `fields` property of the sort, using a `SortField` that specifies the `<bookName>` element. The `true` parameter indicates that the sort is not case-sensitive. Note that if you had only one field to sort in the `XMLListCollection`, you would pass `null` instead of the element name.

The function sets the `sort` property of the `XMLListCollection` to the `Sort` object and calls the `refresh()` method. It finishes by displaying a string representation of the sorted list in the `TextArea`.

3. Run the application. Figure 3-6 shows the application in a web browser. The order of the `<book>` elements has changed so that they appear in alphabetical order of `<bookName>`.

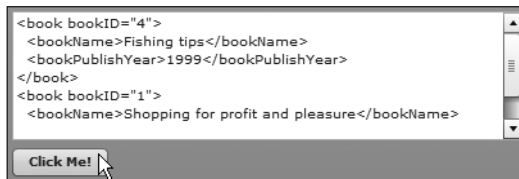


Figure 3-6. Applying a sort to the `XMLListCollection`

The `XMLListCollection` class also makes a number of additional methods available to an `XMLList` object. These methods are listed in Table 3-6.

Table 3-6. Some methods of the XMLListCollection

Method	Description
<code>addItem(item)</code>	Adds an item at the end of the list
<code>addItemAt(item, index)</code>	Adds an item at a specified position
<code>getItemAt(index, prefetch)</code>	Gets the item at the specified position
<code>getItemIndex(item)</code>	Gets the index of the item
<code>removeAll()</code>	Removes all items from the list
<code>removeItemAt(index)</code>	Removes the item at the specified position
<code>setItemAt(item, index)</code>	Places an item at a specified position

These methods add, move, and delete content from the XMLListCollection. The methods work in the same way as with an ArrayCollection object.

Next, let's look at working with the Namespace class.

Understanding the Namespace class

The Namespace class defines or references namespaces that are included in an XML object. It also has other uses in ActionScript 3.0 that we won't touch on here.

Namespace objects aren't usually required when you're working with simple XML content in your SWF applications. You need to work with namespaces only when you have XML content that refers to a namespace, perhaps because there's more than one element of the same name. This can happen when you refer to XML content from more than one source in the same XML object. The namespace provides the context for each element so it can be uniquely identified.

The following example shows an XML object containing two namespaces in the root element. The namespace attribute appears in bold.

```
var phoneBookXML:XML = <phoneBook
  xmlns:foe="http://www.foe.com/ns/"
  xmlns="http://www.sasjacobs.com/">
  <contact id="1">
    <foe:name>Sas Jacobs</name>
    <address>123 Red Street, Redland, Australia</address>
    <phone>123 456</phone>
  </contact>
</phoneBook>
```

The first namespace in the root element has the prefix `foe`. Any elements that start with this prefix fall within the `http://www.foe.com/ns/` namespace. The code identifies that they are associated with that source.

The second namespace doesn't have a prefix. It is the default namespace for all elements inside the <phoneBook> element. The <contact>, <address>, and <phone> elements all fall within the default namespace.

To refer to any of the elements in this XML object, you need to qualify them with their namespace. In order to do that, you need to define a Namespace object for each namespace. You can define a Namespace object by referring to its URI, as shown in the following code:

```
var myNS:Namespace = new Namespace("http://www.foe.com/ns/");
var defaultNS:Namespace = new Namespace(http://www.sasjacobs.com/);
```

This code creates two Namespace objects, which I will refer to using the prefixes myNS and defaultNS.

As I mentioned previously, the URI that you use in a Namespace object doesn't actually need to house a document. The only requirement is that the location is unique among all of the namespaces in your SWF application.

The <name> element in the phoneBookXML object is prefixed by the text foe. This element has a local name of <name> and a qualified name of <foe:name>. If you had other <name> elements, you would need to work with the qualified name to distinguish them from this one.

Because the other elements are within the default namespace, they aren't written with a prefix. However, when you want to work with them in ActionScript, you'll need to qualify them with the default namespace.

You can access an element within a namespace by using the scoping operator ::. The scoping operator indicates that the local name is qualified by that namespace. XML documents use a colon to indicate the namespace. As the colon is reserved in ActionScript, you use the next best thing—two colons.

```
phoneBookXML.myNS::name
```

In this example, the <name> element, is qualified by the myNS Namespace object, which was declared earlier.

The next example shows an element in the default namespace.

```
phoneBookXML..defaultNS::contact
```

Even though the element doesn't display with a prefix in the XML object, you still need to work with the default Namespace object.

If all XML objects are within the same namespace, you can set a default namespace by using the statement's default xml namespace, as shown here:

```
var myNS:Namespace = Namespace("http://www.foe.com/ns/");
default xml namespace = myNS;
```

Let's look at an example so you can see where the Namespace class might come in handy. We'll work with a simple document that describes an XHTML <table> element as well as a furniture <table> element. I'll show this example in Flash; feel free to try to re-create it in Flex.

1. Create a new Flash document. Add the contents of the `sample.xml` document to an XML object called `sampleXML`, as follows:

```
var sampleXML:XML = <sample
  xmlns:html="http://www.w3.org/1999/xhtml">
  <table>Wood with four legs</table>
  <html:table>Text for an XHTML table</html:table>
</sample>;
```

2. Set a reference to the `html` namespace in the constructor of the `Namespace` class, as shown here:

```
var htmlNS:Namespace = new Namespace("http://www.w3.org/1999/xhtml");
```

3. Add the following `trace()` statements to view the contents of the XML object. You need to use the scoping operator to reference the `htmlNS` namespace.

```
trace(sampleXML.table);
trace(sampleXML.htmlNS::table);
```

4. Test the application. It should produce the following output:

```
Wood with four legs
Text for an XHTML table
```

In this example, you can see that the XML object contains two elements of the same name, `<table>`, one within the namespace prefixed with `html`. Tracing the local name `table` shows the contents of the `<table>` element that has no namespace assigned. If you had defined a default namespace, the second `<table>` element would be associated with it.

Using the scoping operator to add the `htmlNS` namespace to the `<table>` element produces a different result. It displays the contents of the `<table>` element associated with the `http://www.w3.org/1999/xhtml` namespace.

You'll need to understand this technique if you're working with elements that fall within declared namespaces. You can find the example used here saved in the resource file `namespaceObject fla`.

Namespace objects are also used with the `QName` class.

Understanding the QName class

The `QName` class is really an abbreviation for the term *qualified name*. This class provides a mechanism for identifying qualified names for elements and attributes in an XML object. Qualified names are necessary when you are working with elements that fall in namespaces.

A `QName` has two parts: the local name for the element or attribute and a namespace URI to associate the element with a namespace. The namespace URI is optional, and you can omit it. If you do this, you'll map the element or attribute to the default global namespace for the XML object.

A QName object has two properties:

- `localName`: Returns the local (unqualified) name of the object.
- `uri`: Returns the URI of the namespace associated with the QName object.

You can create a QName object using the `QName()` constructor method. If you're using a namespace in your QName, you can either create a Namespace object or pass a string representing the namespace. The following example shows the use of a namespace in creating the QName object:

```
var myNS:Namespace = new Namespace("http://www.foe.com/ns/");
var myQName:QName = new QName(myNS, "localName");
```

This example is equivalent to the following line, which uses a string representation of the namespace:

```
var myQName:QName = new QName("http://www.foe.com/ns/", "localName");
```

The following steps demonstrate how to create QName objects and access their properties. We'll do this with a Flash file. Again, you might want to re-create this example yourself in Flex.

1. Create a new file and add the `authorsShort.xml` file contents to an XML object called `authorsXML`, as shown here:

```
var authorsXML:XML = <foe:allAuthors xmlns:foe=
  ↳"http://www.friendsofed.com/ns/"
  ↳xmlns="http://www.sasjacobs.com/ns/">
  <author authorID="1">
    <authorFirstName>Alison</authorFirstName>
    <authorLastName>Ambrose</authorLastName>
    <books>
      <foe:book bookID="1">
        <bookName>Shopping for profit and pleasure</bookName>
        <bookPublishYear>2002</bookPublishYear>
      </foe:book>
      <book ID="4">
        <bookName>Fishing tips</bookName>
        <bookPublishYear>1999</bookPublishYear>
      </book>
    </books>
  </author>
</foe:allAuthors>;
```

The XML document contains two namespaces in the root element. One is the default namespace, `http://www.sasjacobs.com/ns/` and the other is the namespace `foe`, which has the URI `http://www.friendsofed.com/ns/`. The default namespace has no prefix. The first `<book>` element is within the `foe` namespace. The second is within the default namespace. You will create QName objects from these elements.

2. You need to map the two namespaces in the XML object. Use the following code to create two Namespace objects:

```
var foeNS:Namespace = new Namespace("foe",
  "http://www.friendsofed.com/ns/");
var defaultNS:Namespace = new Namespace("sas",
  "http://www.sasjacobs.com/ns/");
```

3. Create two QName objects to reference the <book> elements, as follows:

```
var foeBookQName:QName = new QName(foeNS, "book");
var defaultBookQName:QName = new QName(defaultNS, "book");
```

Using a QName object allows you to provide a unique reference for each of the qualified element names.

4. Add the following trace() actions to view the URI associated with each QName:

```
trace(foeBookQName.uri);
trace(defaultBookQName.uri);
```

5. Test the code. You should see this output:

```
http://www.friendsofed.com/ns/
http://www.sasjacobs.com/ns/
```

This example is simplistic, but it shows that you've mapped two different elements of the same name to different namespaces, and you have two different QName objects that you can use to reference the two <book> elements. You can find this example saved in the file QNameObject.fla.

Incidentally, if you were trying to locate elements in this XML object, you would need to use both Namespace objects with the scoping operator. For example, to get to the <foe:book> object, you would need to use this E4X expression:

```
trace (authorsXML.defaultNS::author[0].defaultNS::books.foeNS::book);
```

It could get very confusing if you didn't understand that the default namespace qualifies the <author> and <books> elements so it must be included in the expression. This issue is particularly important when working with web services and RSS feeds. You'll see this later, in Chapter 10.

As you've seen, the new E4X classes provide a more streamlined approach to working with XML content in SWF applications. However, you need to be aware of their limitations.

Limitations of working with the XML class

While the XML and XMLList classes are great additions to the ActionScript language, there are a couple limitations to their use. The first is within the ActionScript parser itself. The second relates to the relationship of the XML class to external XML documents.

The ActionScript 3.0 parser is nonvalidating, which means that if the XML content refers to an XML schema or DTD, the parser is not able to check that the content is valid. If you need to check that your XML content conforms to the rules of its vocabulary, you must do this outside your SWF application. This could prove cumbersome when using dynamically generated content from a database.

The second issue relates to the loading of external, dynamically generated XML documents into SWF applications. In order to load external content into a SWF application, the application must first request the XML document. If the external content changes, the SWF application must request the document again in order to access the changed content.

When working with the XML class, the only solution is for the SWF application to poll the server continually, in case the content has changed. It's simply not possible for the external XML data source to push content into an XML object in a SWF application. I'm sure you'll agree that this is not a very practical approach. An alternative is to use the XMLSocket class. We won't cover that approach in the book, as it's an advanced topic.

Summary

This chapter introduced the ActionScript 3.0 classes that fall within the E4X specification. You learned how to work with the XML object as a data type, and you saw examples of the XML class properties and methods. You also learned how to work with the XMLList, XMLListCollection, Namespace, and QName classes.

In the next chapter, we'll look at some other types of E4X expressions and see how they can be used to locate content in an XML object.

