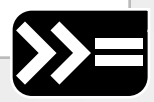
Objectives

- Basic communication in Flash MX:
  - `getURL()`
  - `loadVariables()`
- Introducing the communication objects:
  - The `LoadVars` object
  - The `XML` object
  - The `XMLSocket` object
- The `LoadVars` object and its associated methods
- Using `LoadVars` to communicate with server scripts



Introduction

To complete our study of ActionScript in Flash MX, a major piece of the puzzle is making Flash talk to the outside world through server communications. It's one thing to create a static Flash web site that never changes, but it's another thing to create a site that connects the user to the outside world, and to other users.

Imagine building a daily web log (or *blog*) on your site containing the latest sports scores, weather reports, news headlines, and perhaps messages from other users. To create this kind of functionality, we must take a look at the communication capabilities of Macromedia Flash MX, and the options that are available to us to make Flash really come to life.

Reaching outside of Flash

There are a few different techniques open to us for talking to the outside world. The first, `getURL()`, is mainly for opening up external web pages, and general browser level communication. The other methods deal with an open line of data communication without necessarily requiring the medium of a browser window:

- **`getURL()`** – This is one of the simplest and most commonly used methods of reaching outside of Flash. Essentially, what `getURL()` does is send an HTTP request to the browser. This means that we can make Flash open up other web sites, send emails, and activate JavaScript functions all by utilizing `getURL()`. The `getURL()` command doesn't actually return any data as such – it simply sends out HTTP requests.

- **`loadVariables()`** – Although `loadVariables()` is an old Flash 5 function, it's still highly useful today. With the `loadVariables()` command, we can call a web page, script, or text file, and retrieve data from it. The data comes back in a format that Flash can read, and we can use that data as variables. The data is sent in a format known as **name/value pairs**. A string of data in the name/value pair format might look something like this:

  ```
  variable1=data&variable2=moreData&variable3=lotsOfData&...
  ```

  Each name/value pair is separated by an ampersand (`&`). This data could be stored in a text file, or generated dynamically by a server script. Once a `loadVariables()` function is issued, the server responds and then the connection is closed.

- The **`LoadVars`** object – This is similar to `loadVariables()`, except that everything is neatly contained within an object, which allows for easy transmission and receipt of data. The `LoadVars` object has a number of associated send and receive methods, and data is also transmitted in name/value pairs. Once the send and response is complete, the connection is closed.

- The **`XML`** object – This object is similar to the `LoadVars` object, except it sends and receives its data formatted as an XML document rather than in name/value pairs. When an XML document is loaded, the connection is closed.

- The **`XMLSocket`** object – This object differs from the other data retrieval formats because once an `XMLSocket` connection has been established, the connection remains open, and any data can flow freely back and forth between server and Flash movie. Generally XML formatted data is sent across, but the `XMLSocket` object can actually transmit any format of data, as long as our Flash movie knows how to interpret and respond to it.

In this chapter we'll examine the `getURL()` and `loadVariables()` methods, along with the `LoadVars` object. We'll look at the specifics of XML, and the XML objects from Flash MX in the next couple of chapters.

getURL()

As mentioned above, the `getURL()` command allows us to perform various tasks at the HTTP browser level. Before we look at these in action, let's see how the `getURL()` function is actually structured:

```
getURL(URL, windowStyle, variableActions);
```

The first parameter here contains the URL you want to link to – this is the only required parameter (the other two are optional). The second parameter, `windowStyle`, describes how you want the window to open, and can take a few different values:

- `_blank` – Opens a new window. Flash will open up the web browser in a blank window, and display the URL there.
- `_top` – The top level frame in the current window (in this case the windows that are within a frameset).
- `_parent` – The parent of the current frame.
- `_self` – Specifies the current frame in the current window. That is, Flash will replace the SWF within the browser window with the URL we specify.

So, for example, to open up a web browser in a blank window, we could do this:

```
getURL("http://www.friendsofed.com", "_blank");
```

The third parameter, `variableActions`, is only necessary if we want Flash to pass variables to the URL that it's calling. These variables can be passed as either a `POST` or a `GET` standard of HTTP data transmission. Flash will automatically send *all* the variables of the timeline associated with the `getURL()` action. Sometimes, though, this is not what we want.

The `getURL()` command can also be sent as a method of a movie clip. For example, to send all the variables contained within the movie clip, `myMovieClip` via the `POST` method, we would write this:

```
myMovieClip.getURL("http://www.mysite.com/go.php", "_blank", "POST");
```

And to do the same via the GET method, we would do this:

```
myMovieClip.getURL("http://www.mysite.com/go.php", "_blank", "GET");
```

In both these instances, we're calling a PHP script called go.php. This script could be anything we want. As long as it knows how to handle our outgoing data, it will open up in a new browser window, with the results of our POST or GET.

The difference between POST and GET is the way the data is sent on the URL. With GET, the variables are all simply tacked on to the end of the URL string, as name/value pairs separated by ampersands. For example, if we had 3 variables in myMovieClip like this:

```
a = 5;
name = "Joe";
color = "Blue";
```

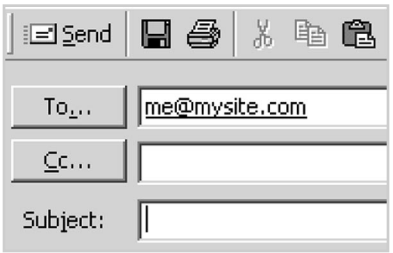
And we send the variables with getURL(), using the GET method shown above, then we'd see this URL string in our browser window:

The problem with this method is that we're limited to a maximum URL string of 256 characters. When we use the POST method, then our variables are sent embedded in something called the **HTTP Header**. This is a piece of data that we never see, which is sent to the server, and we're never made aware of. So the POST method allows us to send very long strings of data, much greater than 256 characters.

We can also use the variableActions parameter to make calls to built-in web browser functionality like mailto: and javascript:. These do not require windows to be opened, so we simply call them like so:

```
getURL("mailto:me@mysite.com");
```

This will open up an email window, with the To field filled in with the specified address:

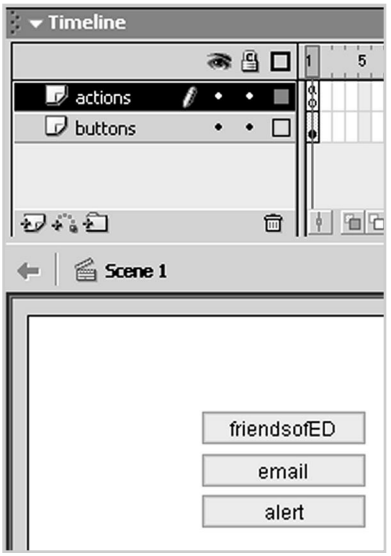


Let's look at the form of the JavaScript action now:

```
getURL("javascript:myFunction();");
```

This will call the JavaScript function `myFunction`, which would most likely be defined on the page that contains the Flash movie – this demonstrates how easy it is to link your ActionScript with JavaScript, perhaps to create a universally accessible site.

Let's see all this in action – open up `getURL.fla`, and look on the stage:



We see that there are three buttons, and an actions layer on the timeline. The buttons are labeled friendsofED, email, and alert, with corresponding instance names of `foedlink`, `mailbutton`, and `alertbutton`. The first button opens up the friends of ED web site, the second opens up an email with some specified details, and the third opens a JavaScript alert box. The code on the actions layer is as follows:

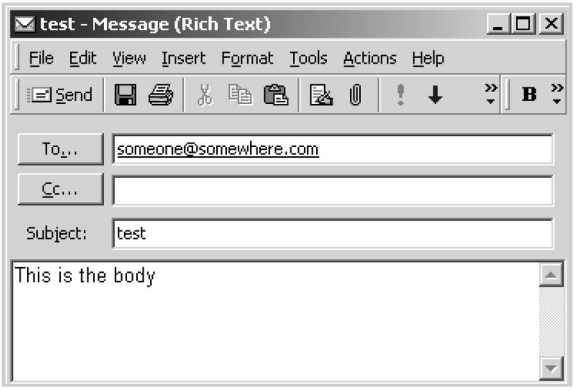
```
foedopen = function () {
      getURL("http://www.friendsofed.com", "_blank");
};
mailopen = function () {
      getURL("mailto:someone@somewhere.com?subject=test&body=This is the
      ➥body");
};
javaalert = function () {
      getURL("javascript:alert('hello');");
};
foedlink.setClickHandler("foedopen");
```

continues overleaf

```
mailbutton.setClickHandler("mailopen");
alertbutton.setClickHandler("javaalert");
```

Each function contains a simple `getURL()` call. The first one, referenced by `foedopen`, opens up www.friendsofed.com in a new browser window. The second function, `mailopen`, opens up an email addressed to someone@somewhere.com, with some text in the subject line and body. Finally, the third function opens up a JavaScript alert box. In the last three lines, we're tying each button to the appropriate function.

When we run this code, and click on the friends of ED button, we're taken to the friends of ED web site. Remember, you can publish your FLA file by selecting File>Publish (SHIFT+F12), and this will automatically create an HTML file for your SWF that you can open up in your web browser (alternatively, just hit CTRL+F12 when you're testing a movie to publish *and* open up the HTML file in one go!). If you now click on the email button, a new email should pop up from your email client ready to send:



We filled in the various fields by specifying the `mailto:`, `subject`, and `body` variables in the `getURL()` string, like a standard `GET` string:

```
getURL("mailto:someone@somewhere.com?subject=test&body=This is the
➥body");
```

The third button opens up a JavaScript alert window:

To achieve this, we have a line of simple JavaScript code directly inside the `getURL()` string:

```
getURL("javascript:alert('hello');");
```

We're calling the `alert` function, which opens up a window on the screen containing the word, hello, as instructed.

loadVariables()

The `loadVariables()` command simply sends and receives data in the form of ampersand delimited name/value pairs. Its syntax is as follows:

```
loadVariables(URL, target, variableActions);
```

The `URL` is the filename or script containing our variables; `target` is the name of a movie clip or level number that we want the loaded variables sent to; `variableActions` works in the same way that it does with the `getURL()` command – using `GET` or `POST` to send data.

So, say we have a file called `scores.txt` that, when opened in a text editor, you can see contains the following information:

```
score0=Houston 3 Denver 2&score1=Detroit 4 Boston 5&score2=New York 6 New
Jersey 3&score3=Toronto 7 Los Angeles 5&
```

This is essentially one long string containing four variables made up of name/value pairs: `score0` to `score3`. Each variable is a string of text detailing the results of a sports game. All we need to get the data is a `loadVariables()` call, like this:

```
loadVariables("scores.txt");
```

We needn't specify a `variableAction` because we're not sending any variables *to* the text file, and, since we're not specifying a target, the variables will be sent to the location from which this call was made. For the purpose of this simple example, the `scores.txt` file is local, so we don't need to specify an HTTP URL, we only need the filename. When the data comes back, Flash will automatically parse it and convert it into variables.

Take a look at `loadVariables.fla` to see this in action. In this movie, we have a dynamic text field on the stage called `scoredisp`, and in it we're going to display a scrolling tickertape of scores. The code to do this is attached to the actions layer on the main timeline, as follows:

```
_root.loadVariables("scores.txt");
_root.onData = function() {
      i = 0;
      if (score0 != undefined) {
            scoredisp.text = ".............................";
```

continues overleaf

```
                scoredisp.text += ".............................";
                scoredisp.text += ".............LATEST SCORES:  ";
                while (eval("score"+i) != undefined) {
                        scoredisp.text += "    "+eval("score"+i)+"    ";
                        i++;
                }
                scoredisp.text += ".............................";
                scoredisp.text += ".............................";
                scoredisp.text += ".............................";
        }
};
scoredisp.backgroundColor = 0;
_root.onEnterFrame = function() {
        scoredisp.hscroll += 50;
        scoredisp.hscroll %= scoredisp.maxhscroll;
};
```

In this code, first we issue the `loadVariables()` command as a method of the root timeline– because we're not specifying a target, the variables will be returned to the `_root`. We then have a function called `_root.onData`, which will be triggered when the data returns from the `loadVariables()` call. The relevant line to make sure of this is:

```
        if (score0 != undefined) {
```

So obviously the code in this function cannot run if the `score0` variable is undefined. The actions inside this `if` clause then go ahead and fill the `scoredisp` text field up with a list of scores derived from the `scores.txt` file.

Next, the `while` loop keeps adding on scores until we have no more score variables:

```
                while (eval("score"+i) != undefined) {
                        scoredisp.text += "    "+eval("score"+i)+"    ";
                        i++;
                }
```

Note the use of the `eval()` function here – this allows for the dynamic construction of an identifier based on a string (`score`) and an integer variable (`i`). With `eval()`, if the associated expression evaluates to a variable or property, the value of that variable will be returned. However, if the element named in the expression does not evaluate to a known object or variable, `undefined` is returned. After all the scores are printed, we send some *filler* dots to the text box.

Finally, we set the background color of the `scoredisp` text field to black, and then we've got an `onEnterFrame` function that is responsible for scrolling the text field horizontally, using the `hscroll` property of the TextField object, and then wrapping back around to the beginning when all is finished. When we test `loadVariables.fla`, we see the scores roll by in our text field, like so:

What we've done here is create a scrolling score ticker, where the data is retrieved from an external data source. The data in this example is hard coded in a text file, but it's also possible to call a script, in something like PHP or ASP, which would generate the results dynamically. All Flash really cares about is whether the data is sent back in the format it expects.

The LoadVars object

In its ability to load and send data, the `LoadVars` object is rather similar to the `loadVariables()` method. But now that we're dealing with an object, the load and send capabilities exist not as properties, but as part of a collection of methods that the `LoadVars` object offers us to make HTTP communications easy.

Creating a new instance of the `LoadVars` object is achieved in much the same way that we create other object instances:

```
myLoadVars = new LoadVars();
```

Once we've created our `LoadVars` object, we have two properties that we can access:

- `LoadVars.contentType` – This property contains the content type of the data, so we can ensure that the server knows what type of data we're sending. It's automatically set by Flash, and the default value is `application/x-www-form-urlencoded`.

- `LoadVars.loaded` – This property is simply a `true` or `false` Boolean variable. When we first send or load variables this property is set to `false`. Once the load has successfully completed, then `LoadVars.loaded` will be set to `true`. If the load failed for some reason (for example, if the data was missing or not returned), then this property will remain `false`.

LoadVars methods

Let's now examine all of the methods associated with the `LoadVars` object. First, we should note how we can add variables to our `LoadVars` object. For example, after creating a new `LoadVars` object called `myLoadVars`, we could write this:

```
myLoadVars.height = 44;
myLoadVars.aMessage = "Any Value";
myLoadVars.name = "Glen";
```

Then, if we were to issue a `send()` method, it would be these variables that would be sent. OK, on with our discussion of the methods we have available to us with this object.

send()

After defining our variables we can send them with the `send()` method, like so:

```
myLoadVars.send("http://www.mysite.com/go.php", "POST");
```

This will post all the variables associated with `myLoadVars` to a script called `go.php` at `www.mysite.com`. We can also use `GET` as the send type, but the default is `POST`.

Using this method, we don't expect anything back, as it's strictly for *sending* information to the server, and, assuming it's complete, Flash will ignore any responses from the server. However, if we want, we can tell Flash to redirect any responses from the server to a window, by adding a parameter to our code:

```
myLoadVars.send("http://www.mysite.com/go.php", "_blank", "POST");
```

If the server does send anything back, it will be sent to a new browser window. This can be helpful for debugging, and seeing whether or not our posts are working.

load()

This method just waits for a URL and then downloads all the variables at that URL. The code is as follows for a full URL:

```
myLoadVars.load("http://www.mysite.com/vars.php");
```

It's even simpler for a local file:

```
myLoadVars.load("vars.txt");
```

Once the load is complete, the `onLoad` event, if it has been defined, will be triggered, and the `loaded` property will be set to `true`. We'll look at the `onLoad` event in a moment.

First, let's look at a `LoadVars` version of our previous `loadVariables()` example. In the file `loadVars_load.fla` from the CD you'll find another score ticker. This time we'll load it up via the `LoadVars.load()` method. The code on frame 1 of the actions layer looks like this:

```
myLoadVars = new LoadVars();
myLoadVars.onLoad = function(success) {
      i = 0;
      if (this.loaded && success) {
            scoredisp.text = ".............................";
            scoredisp.text += ".............................";
            scoredisp.text += ".............LATEST SCORES:  ";
```

```
            while (this["score"+i] != undefined) {
                  scoredisp.text += this["score"+i]+"    ";
                  i++;
            }
            scoredisp.text += ".............................";
            scoredisp.text += ".............................";
            scoredisp.text += ".............................";
      }
};
myLoadVars.load("scores.txt");
scoredisp.backgroundColor = 0;
_root.onEnterFrame = function() {
      scoredisp.hscroll += 50;
      scoredisp.hscroll %= scoredisp.maxhscroll;
};
```

Everything is very similar to our previous example, except that here we're using the `LoadVars` object. We've created a new instance called `myLoadVars`, and then defined its `onLoad` function. In this function, we make sure that the data has completely loaded – this time by utilizing the `LoadVars.loaded` property, and that the load was successful. If everything is OK, we fill in the text field, as before.

We then issue the `load()` method of the `myLoadVars` object, and load the contents of `"scores.txt"`. Finally, we scroll the text field in the `onEnterFrame` event of the root timeline.

getBytesTotal() and getBytesLoaded()

The two `getBytes` methods return values once a load has been initiated. At that point, `getBytesTotal()` will return the total number of bytes in the entire data stream and `getBytesLoaded()` tells us how much of it has already been loaded.

We can determine a percentage like so:

```
perc = (myLoadVars.getBytesLoaded()/myLoadVars.getBytesTotal())*100;
```

This will set `perc` to be a number from 0 to 100, depending on how much has been loaded. If there is currently no load in progress, then `myLoadVars.getBytesTotal()` will return `undefined`. It's also worth noting that the number of bytes may sometimes be undetermined if, for example, the size was not sent in the HTTP header from the server. In such a case, `getBytesTotal()` will again return `undefined`.

sendAndLoad()

Here we have a method that performs the functionality of both `LoadVars.send()` and `LoadVars.load()` at once. It's works like this:

```
myLoadVars.sendAndLoad("http://www.mysite.com/go.php", retVar, "POST");
```

What we're doing here is specifying a URL to upload variables to, and then we're stating a `LoadVars`, `retVar`, object to receive the results. We do this because we don't always want the results to be returned to the same `LoadVars` object that initiated the call. For example, we could have a case where `LoadVars` sends some variables, gets some back, and then the next time it sends variables it also sends those variables it just received from the server. To avoid sending lots of redundant data, we specify another return `LoadVars` object, so all sent and received data is kept separately. Optionally, in this method we can set the send type, be it `POST` or `GET` – when neither is specified, `POST` is used by default.

If we want, we can specify the same `LoadVars` object for both sending and receiving, like so:

```
myLoadVars.sendAndLoad("http://www.mysite.com/go.php", myLoadVars);
```

toString()

We can output all the variables contained within a particular `LoadVars` object, by simply calling the `LoadVars.toString()` method, like so:

```
trace(myLoadVars.toString());
```

If we run this method within the `onLoad` function in `loadVars_load.fla` that we looked at earlier, it will trace out the following data string:

```
score3=Toronto%207%20Los%20Angeles%205&score2=New%20York%206%20New%20Jersey%
203&score1=Detroit%204%20Boston%205&score0=Houston%203%20Denver%202&onLoad=%
5Btype%20Function%5D
```

Note that the function `onLoad` is also part of the variable list, the final entry, because it is a property of the object (see below).

onLoad

Although it's not actually a method, the `onLoad` event handler of the `LoadVars` object is worth a mention of its own. This event is triggered when any load has completed and with it we can activate whatever function we like. The structure of this function should generally be like so:

```
myLoadVars.onLoad = function(success) {
      if (success) {
            trace("Done Loading");
      } else {
            trace("Failed Loading");
      }
};
```

Here, the `success` parameter is passed into the function, and it returns simply `true` if the variables were successfully loaded, or `false` if there was an error.

So that's how we use the `LoadVars` object to talk to our servers and local files. Before we finish this chapter, let's take a look at a simple, yet very powerful, script and how we interact with it using the `LoadVars` object.

Using LoadVars to communicate with server scripts

In this section, we'll see how to load variables, and submit them to a server script file. The script we'll be referring to is a PHP script, chosen because it's simple, powerful, and free! It's also very similar to Actionscript, so it should be easy to read and understand.

> *If you're unfamiliar with PHP, and would like to get to know it better, check out Foundation PHP for Flash from friends of ED, ISBN: 1-903450-16-0. However, you should be able to follow this example whatever your level of PHP knowledge.*

Check out the file `serverscripts.fla`. The idea behind this Flash movie is simple – when the movie starts up, it connects to a server and calls a script named `servertime.php`. This script gets the current time from the server, and then returns it as a single variable called `servertime`. In this example, we're calling the PHP script from our imaginary site, www.mysite.com/servertime.php (If you want to test this code from your own machine, you can just change the relevant code to a local URL, like `servertime.php`).

So, if you were to use this code as it is you'd need to set up this script on a PHP-enabled site. A useful application of such a script could be a clock on your website, displaying the correct local time to each visiting user, no matter how the clock was set on their computer.

> *To run this example, you'll need PHP and an appropriate web server installed on your machine – we recommend using Apache. You might be surprised to learn that both PHP and Apache are free for personal and commercial use. You can download the current version of PHP from* www.php.net/downloads.php*, and the latest version of Apache from* http://httpd.apache.org/dist/httpd/.

If we set up the PHP script on our site and test this movie, we see something similar to the following in the `timedisplay` dynamic text field:

Server Time:Fri Jul 5 02:31:44 GMT-0400 2002

A string is displayed in a text field, and it's retrieved from the server script using the following Actionscript, attached to frame 1 of the actions layer:

```
myLoadVars = new LoadVars();
myServerTime = new Date();
myLoadVars.onLoad = function(success) {
      myServerTime.setTime(this.servertime*1000);
      timeDisplay.text = "Server Time:"+myServerTime.toString()+newline;
};
myLoadVars.load("http://www.mysite.com/servertime.php");
```

In this script, we're firstly creating a new `LoadVars` object, and then a new `Date` object. The `Date` object is necessary in order to take the server time, and then convert it into readable time.

When the server returns the time, it returns it as a system time, which is a measure of the time in seconds from a fixed date in history (midnight, January 1, 1970). By using this as a universal reference date, all computers can communicate the time to each other. So, if the system time for midnight, January 1, 1970 is `0`, the system time at the time of writing is approximately `1025849135` seconds, and counting!

Flash uses this system of time calculation as well, but it actually counts the *milliseconds* since January 1, 1970, meaning the Flash system time will have three extra digits. That's why, in our `onLoad` event function, we're multiplying the variable `this.servertime` by 1000 – to transform the server time, set by our script in a variable called `servertime`, into a format that Flash understands. We set the value of our `Date` object, `myServerTime`, to the milliseconds time value with this line:

```
myServerTime.setTime(this.servertime*1000);
```

Then we display that value outputted as a nicely formatted time string:

```
timeDisplay.text = "Server Time:"+myServerTime.toString()+newline;
```

The result of this is that we see the full current time and date displayed in the text field `timeDisplay`.

Finally, we have the line of code that's responsible for actually calling the PHP script from which we'll obtain our original `servertime` value. We use the `load()` method of the `myLoadVars` object. Once this call is made to the server, the server returns a formatted name/value result representing the `servertime` variable. In fact, as far as Flash is concerned, we might as well be loading this from a text file because the results are exactly the same.

It's worth noting just how straightforward the PHP script that we've used is – open up `servertime.php` in a text editor:

```
<?php
      $t = time();
      echo "servertime=$t";
?>
```

That's all there is to it! We're basically getting the current time, storing it in a variable called `$t`, and using the echo command to output to the data stream `servertime=$t`, which will be filled out by PHP with the actual server time. Since this script is run on the *server side*, the time value will be that of the server's internal clock. The PHP script must be opened with `<?php`, and closed with `?>`.

In this section we've had only a brief look at the power of using scripting to create communication between Flash and the server. If we wanted to, we could send variables to the script using the `sendAndLoad()` method, and then PHP would make those variables available to itself immediately. So, for example, if we said in Flash:

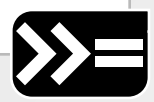
```
myLoadVars.country = "Canada";
```

then in our PHP script the variable `$country` would now exist, which we could then make use of. The variable would be set to `Canada`, because that was part of the information Flash passed using the `send()` or `sendAndLoad()` method. We'll expand upon our knowledge of server-side scripting in the next bonus chapter-which is on the CD- by taking a look at the ColdFusion MX technology as part of an example.

18 Understanding XML

Objectives

- Understanding the power of XML
- XML conventions:
  - Learning the language
  - Formatting XML documents
- Creating XML data:
  - Hierarchy and structure
  - Dissecting XML documents
- Server-side scripting with XML



Introduction

Integrating XML with Flash is a powerful way to handle content within your movies. Using XML makes it relatively easy to include dynamic content in an interface. This data is easy to manipulate, update, and even syndicate among numerous web sites. Many Flash designers and developers use XML and Flash for everything from menus, navigation, and articles, to news feeds, weblogs, and multi-user chat rooms. Using XML essentially means that you don't have to re-open an FLA file to update your content, thus simplifying site maintenance. This can be very useful for a client who wants to control his or her site after you, the designer, hand it over, particularly if the client doesn't know how to use Flash! It also means you can syndicate your content to be used remotely by other web sites.

Although we've been able to use XML data in Flash movies since Flash 5, an advantage of using Flash MX to integrate your XML, is that the parsing ability of the Flash Player 6 has been greatly augmented. XML is an easy language to understand and create after reaching only a basic understanding of its structure and usage.

In this chapter, we'll take a look at what XML can be used for and how to create and format XML documents. XML has a lot of terminology associated with the language – it's important to understand these terms and how they apply to XML, even if you are just using the XML object in Flash. Therefore, we will look at the concepts and language of XML to enable you to work with the data files together with Flash. You will learn how to structure data within an XML document itself, so you can create an XML file and use the information in a Flash interface. Finally, we'll look at how you can use server scripts to output XML formatted files. If you are already familiar with or experienced in writing XML, you will probably want to skip ahead to the next couple of chapters, where we'll discuss integrating Flash with XML: the XML object and using XML socket servers.

XML as a data concept

There are many different ways in which you can organize, contain, and include data in your Flash movies. XML is one of the languages you can use to dynamically include content in an interface. Understanding how to use XML to organize your data will help you immensely, particularly when you are working with dynamic movie content. In this first section, we'll take a look at what XML actually is, and what it is commonly used for in production.

What is XML?

XML stands for e**X**tensible **M**arkup **L**anguage, which is a language that describes data within a set structure. It is tag-based, like HTML, and most useful when you need to organize a lot of complex data. The term *extensible* is used because an XML file can be updated and customized by interface users. You can also modify the structure, by adding new content or even new tags manually or dynamically.

XML will look familiar to you if you've used HTML before, because they are both tag-based markup languages. However, XML does not use *pre-defined* tags to mark up data. Instead, you create *your own* tags to describe the content of your document. Therefore, an XML file will actually describe itself because the tags consist of both data and a description of the relationship of data in the document. Another

difference between the two is in the *purpose* of each language. HTML is a means of presentation – it formats and presents the data, sometimes in an interactive way. However, XML is a way of describing a data structure, rather than presenting it. The data can then be used in a presentation method of your choice (such as Flash).

We should be clear that XML is not a programming language like PHP or JavaScript. Instead, XML is a way for you to structure your data so that it can be used in other software or with other languages. Therefore, the purpose of XML is to provide a widely adoptable file format, which structures complex, hierarchical data. It is widely used because of its standardization and acceptance by some of the largest driving forces in the industry (Microsoft, IBM, Macromedia, and so on).

Good reasons to use XML

XML is a widely supported language and being standardized by the World Wide Web Consortium (W3C) ensures its universality. Standardization is necessary if you want a language to be supported on many platforms and by many different kinds of software. XML can be used as a great method for exchanging data, since it is understood by many different platforms. Since XML works with so many different pieces of software, you can develop one file that can be understood by numerous front ends on different kinds of machines. This immediately makes it easier to create, handle, and update your content source and develop diverse applications.

Add to this the fact that XML is a highly intuitive language – when you read an XML document, the tags can describe what is going on without you having to learn proprietary tags or coding lingo. This is because authors create the tags within the document, and thereby define the meaning of the data themselves. Also, as we mentioned earlier, this data is then transferable to almost any platform, and many different means of presentation. You could create an XML document that is syndicated and used in HTML pages or Flash movies at the same time. Also, because XML is so flexible, since you define the tags, many different kinds of information can be represented using this structure. The possibilities are both exciting and incredibly diverse when you use XML.

There are many different ways you can load external content into a Flash movie at runtime. As discussed in the previous chapter, it is possible to use the `loadVariables` method or the `loadVars` object to dynamically load data, including simple text files. XML makes it easier to import and use *complex* data structures within your interfaces, and does a much better job at importing nested data than `loadVars` can. XML is not always the best way to bring data into Flash, but when you have data which needs to be structured and handled in a complex structured way, it is typically the best solution. It is also a good idea to use XML if you need to work with different platforms. However, if you simply need to add text to a movie, a simple text file will suffice. A text file can be faster than an XML file, which needs to load and be parsed before being seen by the user.

Another good reason to use XML is the fact it's easy to understand and update. Also, imagine if you have both an HTML and Flash version of a site you need to update. If you are using the same XML file to generate content for both sites, you could potentially update both sites at the same time. XML saves you a lot of effort when is comes to content management.

The power of structured data

So, if XML is simply a bunch of data, what's all the hype about? Where XML becomes really powerful is in the way it *handles* the data. XML structures data so that it is understood in relation to other parts of the data. For example, if you had a piece of data, say `12 lbs`, it doesn't mean much on its own. These `12 lbs` could apply to a bag of apples, or a rock, or whatever. However, with XML you create a data structure so that this weight belongs to something specific, like a cat, as shown in the following example:

```
<cat>
   <cat-specs>
      <name>Kid</name>
      <breed>Siamese mix</breed>
      <breed-color>Black and white</breed-color>
      <weight>12 lbs</weight>
   </cat-specs>
</cat>
```

Being able to create hierarchical data, data *nested* within other pieces of data, is what XML is all about. Having such a useful format allows developers of all kinds of software, including Flash, to create applications using one data file in many different ways. Ultimately, the ActionScript in Flash MX will break the XML file down into usable portions. It separates data into pieces, depending on how the relationships are defined within the hierarchy. This is called **parsing**, which we will look at later in this chapter.

XML can also be used with XSL (e**X**tensible **S**tylesheet **L**anguage) and HTML to present data within an interface. When using XML documents, there are many options regarding what you can do in your applications, and how you format your end product.

XML in the real world

Many different web sites and applications use XML to display dynamic data. It is particularly useful when you need to display data that is continually updated, such as current events on a news server or products for sale on an e-commerce site. A highly dynamic situation is sometimes necessary – data is *pushed* to the end user, and *pulled* from that location on an open connection. This is somewhat different than the common form of data delivery where a user wants to see something, requests the content, and then receives it. XML is frequently used when information changed on the server at the *back end* needs to be sent to the user at the *front end*.

In simple terms, this means that when something is remotely updated, it is sent to the interface and updated immediately rather than just when the user requests it. In fact, this is the theory behind **socket servers**, which will be discussed in the final chapter of this book – **Chapter 20**. Socket servers are a complex, but extremely useful and exciting technology using XML for dynamic purposes. Chat rooms and other such multi-user applications can be based on a socket server that continually pushes data into an interface. At the time of writing, you can see this technology in use on Colin Moock's website – www.moock.org/chat.

However, XML is more commonly used in slightly less dynamic situations. Sometimes it is used simply as a means of organizing data to be included within a Flash site. As mentioned earlier, this data can be easily edited and updated outside of the Flash authoring environment. XML can also be used to provide clients with an easy way of controlling and updating the content of their web sites after a designer or developer has built them.

Sometimes XML is used when data needs to be dynamically added to through the interface itself. Therefore, you could have something such as a message board or guest book that are added to by your end users, and the XML document will be dynamic in the sense that it is added to, much like a database.

Newsfeeds typically use XML to syndicate data, like news headlines and stories, which are displayed as a part of any number of web sites. If you create or use newsfeeds in your web sites, it is useful to understand XML structures so you can work with the data in your Flash interface. You can display a newsfeed in many different ways. Check out the developer center at www.moreover.com for information about newsfeeds. We'll be looking specifically at how to connect to a newsfeed in the next chapter.

So, as you can see, XML can be used for something as simple as data within a text field, or things like dynamic multi-user environments. It is a diverse and robust language and it's not difficult to learn either. Since XML is so widely adopted by many different platforms and software, it's an ideal technology to integrate into your Flash design and development process.

Conventions in XML

Before we begin to use XML with Flash, it's important to understand some of the unique terminology used with the language. You'll see references to terms like **elements**, **content**, **attributes**, **nodes**, **CDATA**, and **whitespace**, which might not mean much until you understand the structure of an XML document. In this section, we will look at the many conventions and terms you will come across when learning and using XML.

Tags

If you're familiar with HTML, you probably know about tags – these are what are used in markup languages to format or define a piece of data. An opening tag will contain data in between the < and > characters. A closing tag will be the same as its corresponding opening tag, but begin with the / symbol. For example, a typical tag in HTML would be:

```
<html>
```

And its closing tag would be:

```
</html>
```

In HTML, tags are used simply for formatting and presentation. For example, a specifies that bold face text will follow it and a closing tag will end the bold face formatting. The important point is

that in HTML the tags are predefined (XHTML, e**x**tensible **HTML**, however, combines these predefined tags with custom-made tags).

XML, on the other hand, consists only of custom-defined tags. These tags are used to describe the data in your structures, where the name of the tags describes the data they contain. As we saw in our cat example earlier, the tag `<breed-color>` is used to describe the coat markings of the cat:

```
<breed-color>black and white</breed-color>
```

An important thing to note is that in XML you must remember to close all of your tags. When using HTML, browsers will allow you to use tags such as the `<p>` or `<br>` without closing them and no error will be thrown. XML will not allow you to get away with this, so remember to close every tag that you open, even if it does not contain any data.

Elements

In XML, elements are the pieces of data that build the structure within the file. They are represented by an opening and closing tag, and have optional content in between these tags. For instance, in the cat example, the element `breed-color` describes a particular feature (or element) of the cat:

```
<breed-color>black and white</breed-color>
```

The tags called `<breed-color>` and `</breed-color>` form the element, and the chosen name `breed-color` defines the content, which is `black and white`. Elements do not need to contain any content. It is completely acceptable and sometimes necessary to leave them empty. An empty element can be written in either of the following formats – firstly, as two tags with no content:

```
<breed-color></breed-color>
```

or as one tag with the closing slash at the end:

```
<breed-color/>
```

The root element

In an XML structure, you always have a root element. This element is the first one in the XML document, after which all sub-elements follow. The root element is formed by a pair of tags which contain the entire document: one at the beginning, and the other at the end. So in our cat example, cat was in fact the root element. The root element tags must encompass all of the sub-element tags within your document, and should describe the general topic of your XML file. In Flash MX, the root element is technically known as the **level 1 child node** (more on nodes shortly). In the following XML example, the root element, or level 1 child node is groceries:

```
<groceries>
   <dairy>
      <milk>2L of skim</milk>
```

```
        <cheese>cheddar</cheese>
    </dairy>
</groceries>
```

Sub-elements

Following the root element are **sub-elements**, which populate the XML file. Sub-elements are used specifically to describe the data and are *nested* within the root element tags. For example, let's elaborate upon our XML shopping list file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<groceries>
    <dairy>
        <milk>2L of skim</milk>
        <cheese>cheddar</cheese>
    </dairy>
</groceries>
```

The first line in this example `<?xml version="1.0" encoding="iso-8859-1"?>` is not an element, it is an **XML declaration** and **encoding attribute**. This declaration identifies the document as being of a certain version and encoding type. Since this is not an element, there is no need to close this declaration. So, as we mentioned in the last section, `<groceries></groceries>` forms our root element, and has been named to describe the document. The content of the `groceries` element are the sub-elements that, in turn, contain their own content. A root element and the nested sub-elements create a *hierarchy* within our document.

Attributes

Attributes are properties that are assigned to an element, and are given *within* the opening tag rather than between the opening and closing tags, like the content. Attributes in XML are very much like those in HTML. For example, an attribute in HTML is seen in the following example:

```
<body bgcolor="#99CCFF"> ... </body>
```

The attribute of a light blue color **`bgcolor="#99CCFF"`** is being assigned to the background of this web site. In XML, properties are given to your elements in the same way:

```
<milk brand="Happy Cow">2L of skim</milk>
```

You'll notice that we have now added an attribute of `brand` with a value of `"Happy Cow"` to the `milk` element from our grocerices list.

In fact, you could replace the content of your element entirely within an attribute. Sometimes it is simply easier to read your content as an attribute instead of as an element. For instance, we could restructure the `cheese` element in the grocery list from this:

```
<cheese>cheddar</cheese>
```

to this:

```
<cheese type="cheddar"></cheese>
```

We can also eliminate the content between the opening and closing tags of our `milk` element by representing it as attributes as well. We could restructure the earlier example as follows:

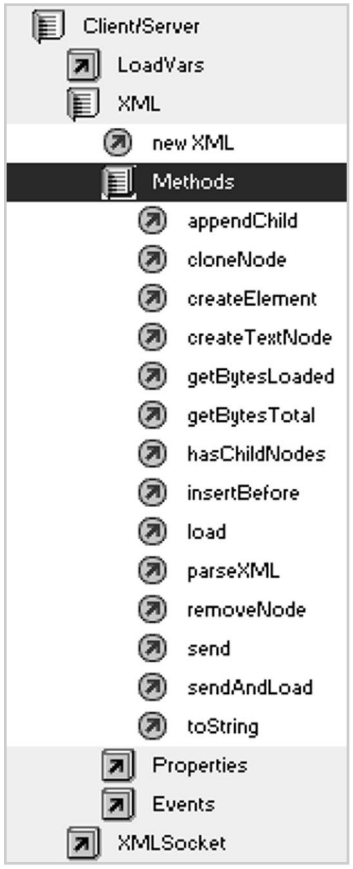
```
<milk brand="Happy Cow" amount="2L" type="skim"></milk>
```

In this example, we have more than one attribute within the tag. Note that what you choose as your attributes, `amount` or `type` in this case, is completely up to you. You could even combine these attributes into one string if you want - it basically depends on what best fits the intended purpose, and what is most usable for your Flash movie. Restructuring our data to use attributes instead of text content should also make your XML load faster when taking it into Flash. But the best way of figuring out what is best for you is by determining your use of the data, and then running speed tests with your movie. The format you use is up to you really. Sometimes you may find that using an attribute instead of content within an element will prove to be faster (or merely easier to read), and vice versa.

Finally, any special characters which you assign as an attribute or content to your element must be ***escaped***. For example, if you include an ampersand (`&`) in your attribute it should be written as `&` and a quotation mark (`"`), not including the ones that delimit the attribute value, is escaped as `&quot;`, and so on. You handle these characters in the same way you would if you were writing HTML.

XML Nodes in ActionScript

If you have looked at the `XML` object in ActionScript, you have probably noticed references to nodes. For example, take a look at all the methods available to the `XML` object in your Actions panel (F9):



Nodes simply represent the data within your XML document. They are the elements, text, whitespace, comments, and so on, within your data structures. Nodes come in various types, and have different properties. In fact, an element is one type of node.

It is imperative that you have an understanding of this terminology when working with the methods and properties of the `XML` object in ActionScript. In this subsection we will look at the different kinds and names of nodes within our document.

Navigating nodes can be somewhat lengthy and confusing when you are first dealing with them in Flash. Nodes are typically referred to as existing in a *tree* format. Essentially, you have a **level 1 child node** (the root element in XML) that resides at the top of the tree and, from this, more *child* nodes branch out. Let's take a look at how nodes are referenced – this will help you once you start navigating the data using ActionScript.

For this section we will revisit and dissect our old grocery list XML file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<groceries>
   <dairy>
      <milk>2L of skim</milk>
      <cheese>cheddar</cheese>
   </dairy>
</groceries>
```

The level 1 child node

The **level 1 child node** in our example file would be `groceries` (remember, this is the root element in XML).

Node names, node values, and node type

Using the `XML.nodeName` property in ActionScript will tell you the name of the node. This means that for our level 1 child node, the node name would be returned as `groceries`. The next sub-node name would be returned as `dairy`.

A node name is different to a node value, which is the actual content (text, symbols, and whitespace) between an opening and closing tag. A node value is seen in the `milk` element as `2L of skim`. It is important to be able to differentiate between these two different kinds of nodes. If you do not know which kind you're dealing with, you'll need to determine the node type.

All of these pieces of data are nodes, but they have different types: a **text node**, or an **element node**. In ActionScript, you can determine the node type with the `nodeType` property (pretty obvious, huh?), which will return a value based on which kind of node it is. If it is textual content, a value of `3` is returned; if the node is an XML element name, a value of `1` is returned. In Flash, these are the only two node types. Determining the type of a node (`1` or `3`) will tell you if you need to grab either the node name (if it is a `XML` element), or the node value (if it is a text node). Then you can use the ActionScript properties `nodeName` (for an element) or `nodeValue` (for a text node) to grab the data you need. Let's look at a fragment of our `grocery` example again:

```
<milk>2L of skim</milk>
```

In this element, `<milk>` is a type `1` element node and its name is `milk`. If you loaded it into a Flash movie, you could output `milk` in the trace window by using the following:

```
trace(groceriesXML.firstChild.childNodes[0].childNodes[0].nodeName);
```

On the other hand, `2L of skim` is a type `3` text node and its value is `2L of skim`. You could trace its value with this code:

```
trace(groceriesXML.firstChild.childNodes[0].childNodes[0].childNodes
➥[0].nodeValue);
```

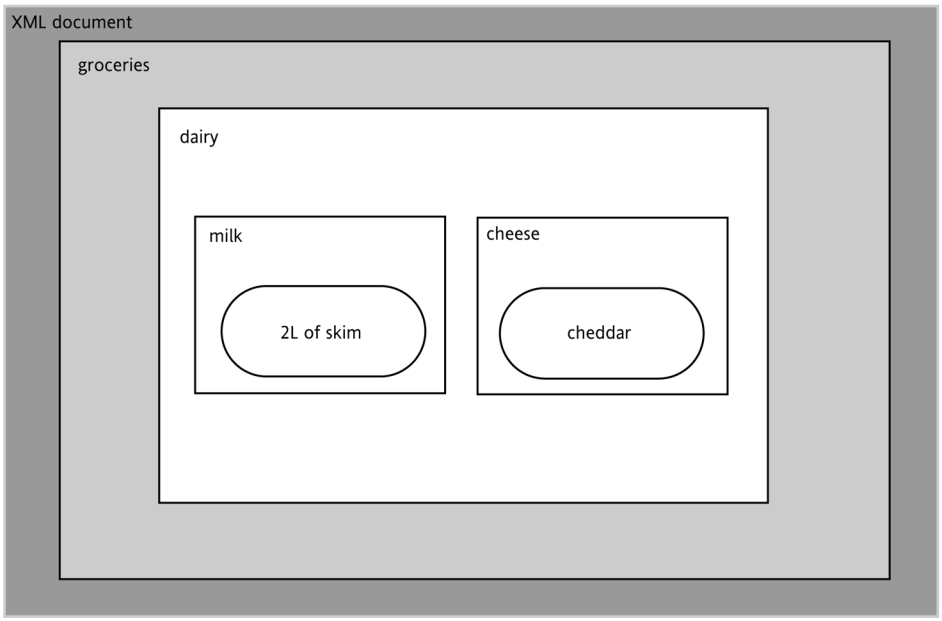
This would output `2L of skim`.. We'll look at structuring ActionScript to reference XML in detail in the next chapter.

Node properties

Sometimes you will also hear about node properties – these refer to parent and child relationships between nodes on the tree. So, the possible node properties would be the types of `nextSibling`, `previousSibling`, `firstChild`, `lastChild`, and `parentNode`. Node properties are used to navigate through the order of the child nodes within the node tree structure.

A ***parent*** node refers to a node that has sub-elements, or essentially nested ***child*** nodes within it. The name of the parent node should suitably describe the information held by it and its child nodes. A ***child*** node is then nested within the parent node. Child nodes can in turn have ***sibling*** nodes, which are on the same level as each other. These nodes can be navigated using the `previousSibling` and

`nextSibling` ActionScript properties. Children nodes can have further child nodes nested within them as well.



As you can see in this diagram of our `groceries` example, sibling nodes are on the same level as one another. Our `milk` and `cheese` elements are sibling nodes, and they are both children of the `dairy` node.

Note that attribute values can also be pulled from the nodes using Flash. The attributes are returned in an array using the `XML.attributes` property. You need to navigate to the appropriate node, and then call for the attribute array using `XML.attributes`.

Special characters and CDATA

`CDATA` stands for **C**haracter **Data**, which refers to special characters like `<`, `>`, `&`, `"`, and so on. `CDATA` is used in XML elements to escape the special characters (or *non-parseable* data) so it can be parsed correctly, and not throw an error. This is particularly useful when working with HTML within your XML document, which relies on character data. Let's take a look at how you would use `CDATA` within your XML document:

```
<breed-color><![CDATA[black&white]]></breed-color>
```

As you can see in this example, `CDATA` makes it very easy to escape the ampersand in our content. However, you should note that if you are retrieving the entire element `<breed-color>` in Flash using something like this:

```
myXML = new XML("<breed-color><![CDATA[black&white]]></breed-color>");
```

and then tracing the `firstChild`, you'd actually see the following data in your output window:

```
<breed-color>black&amp;white</breed-color>
```

As you can see, it does not send you back what you probably want to see: an ampersand character. However, if you trace the value of the content specifically:

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

Then you'll see `black&white` returned in the Output window, containing the ampersand. More information on extracting information from XML documents can be found in **Chapter 19**.

It is important to realize that you can use `CDATA` inside elements, but not in attributes. If you need to escape characters inside an attribute, you will need to manually escape them using escape characters. For example, escaping an ampersand `&` is accomplished by using `&` and `<` is escaped using `<` (less than):

```
<breed-color fur="black&amp;white"></breed-color>
```

As you can probably see, it could be time-consuming and confusing to read a document that needs a lot of special characters escaped. It is particularly useful to use `CDATA` for such pieces of data, like URLs, which only need one tag wrapped around all of the content to handle all of the special characters therein.

Whitespace

Before you start working with your XML file in Flash, you will need to make sure you strip the whitespace in the XML document by using a special routine. If you neglect to get rid of the whitespace, the level 1 child node is likely to be returned as empty (or `null`). Therefore, it's important to remember this crucial step while writing your ActionScript.

You have probably heard references to whitespace if you have looked into XML or ColdFusion previously. Whitespace refers to any carriage returns, spaces, or tabs between elements in an XML file. These spaces are necessary if you want to read your XML in a legible format, and not just one long continuous string of data. The problem is, Flash treats these spaces as empty nodes if they are not stripped from the document. Therefore it's necessary to get rid of any whitespace before Flash parses the XML data.

Removing the whitespace in an XML file is not difficult with the Flash Player 6. All you need to do is simply add the following line to your ActionScript:

```
myXMLObject.ignoreWhite = true;
```

This will also work for the Flash 5 Player release: 5,0,42,0 and later. With earlier releases of the Flash 5 Player, this property will not work. If it's important that your movies should work on earlier versions than

the Flash Player 6, you should find a way to strip the whitespace from the document. You may want to use Branden Hall's *XMLNitro* or *XML:ACK*, both of which you can download from http://chattyfig.figleaf.com. Colin Moock has also made a routine available on his site at www.moock.org/asdg/codedepot. You can include both of these files at the beginning of your Flash movie. Of course, you can always create your own routine to strip these characters from your XML documents as well.

We'll see the effects of whitespace, and how it interferes with parsing in Flash, in some examples in **Chapter 19**.

Parsing

When you load an XML file into Flash, it needs to be completely loaded before the data is seen in your movie. However, even before this happens, it must be fully parsed by the parser in Flash. Parsing refers to a part of a program that receives the XML file. It takes the elements and breaks them apart into separate pieces of data, such as the element name, the attributes, the content, and so on. After this is complete, the data can then be handled by the ActionScript functions in our Flash movie.

Luckily, parsing in the Flash Player 6 is much faster than it was in earlier versions of the Player. In Flash 5, XML parsing speed was definitely an issue. However, due to a vast improvement in XML handling, this is no longer much of a problem unless you are dealing with particularly large files. You should also be aware that parsing XML in Flash does use a lot of system resources. If you are trying to load a large XML file into Flash, you may want to consider breaking it up into smaller portions if performance is critical.

Formatting XML

XML is a very picky language because it doesn't let you get away with a lot of the coding faux pas that a browser will ignore with an HTML file. Therefore, it is very important to know what you can and cannot do when writing your XML documents. In this section, we'll discuss how to properly format an XML file so it does not throw errors when you use it with your Flash movies.

Elements and attributes

Elements must *begin* with a letter. However, they can contain numbers, underscores, colons, and hyphens *after* this initial letter. This means that *neither* of the following elements is valid in XML:

```
<_myFile>

<1myFile>
```

Both elements and attributes are case-sensitive, which means that the following element is *not* valid:

```
<cheese>cheddar</Cheese>
```

We have simply added an upper-case `C` to our `cheese` element, which will now not be considered a closing tag in XML. As attributes are also case-sensitive, the following element:

```
<my-movie extension="mov"></my-movie>
```

Is not the same as this one:

```
<my-movie Extension="mov"></my-movie>
```

It is also important that there are no spaces in an element or attribute name. If you need several words, it is a good idea to insert either an underscore or dash in between the words. This is what we have done with the above example `<my-movie>`. You could have separated the two words by inserting a capital letter, as in `myMovie`. However, if you use this method you need to take care to remember to capitalize it in each element because they are case-sensitive.

It is not necessary for an element to contain any content in XML. It is perfectly fine to have the following format: `<cheese></cheese>` or as we previously mentioned, to use the shortcut method of `<cheese/>`. You should also remember to escape all special characters (such as `<`, `>`, `"`, and `&`), or use `CDATA` sections where possible, in your elements, attributes, or content as discussed earlier.

Tags

The first tag you'll probably see in an XML document is the XML declaration and encoding specification:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

The encoding can be represented as it is above, or it can be written as `"UTF-8"` or `"UTF-16"`. `UTF-8` is the Unicode Transformation Format, and `8` is usually enough for Roman characters. However, `16` allows for many more characters to be encoded, useful if you are dealing with languages other than English. This tag is *not* an element, so there is no need to close it.

However, it is necessary to close every *element* tag in your document. If element tags are not closed, or even not closed in the correct order, an error will be generated. If we alter our groceries example like so:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<groceries>
   <dairy>
      <milk>2L of skim
      <cheese>cheddar
   </dairy></cheese>
</groceries>
```

A couple of errors will be thrown – firstly, there is no closing tag for `<milk>`, which is needed after the word `skim`. Secondly, the tags are being closed out of order. If you have worked with HTML, you will

know that closing tags in an incorrect order will still work when the file is viewed in a forgiving browser. For instance, if you have `<b><i>my text</b></i>` the text will still show as **bold** and *italic*. However, this is unacceptable in XML since precise and correct order is required in all cases.

Finally, as a matter of good form, your XML documents should include *nested* tags. The tags should *all* be nested within a single root element. Then, when applicable, these sub-elements should contain more elements nested within them. Each nested element should be more specific than the element it is nested within. This formatting is important in your documents, because hierarchical data representation is a primary purpose of the XML language.

Reserved words

There are some reserved words in the XML language, which you should avoid. As we said already, all element names must begin with a letter. However, you cannot start it with the *lower-case* letters **xml**. This is reserved for future versions of the language. So for example:

```
<xmlExample> ... </xmlExample>
```

is not a valid element name. In fact, it's probably best to steer clear of capitalized `XML` as well, to avoid any chance of errors.

Commenting

As you probably already know from writing or reading ActionScript, commenting your code is a good practice to adopt. As well as being useful to yourself, commenting can immensely help other designers to figure out the data structure within your documents – it's particularly recommended if you are working with code shared with others.

Commenting is done the same way in XML as in HTML. You format comments in your XML documents in the following way:

```
<!-- Your comment goes here -->
```

DTDs

You may notice DTD (**D**ocument **T**ype **D**efinition) elements near the beginning of a XML document, which will include a tag similar to the following:

```
<!ELEMENT title (#PCDATA)>
```

DTDs are used in XML to declare a format to which the document will validate (see below). A good example of a language using a DTD is HTML; this uses a browser to take the information and format it accordingly. It is also worthwhile to note that DTDs use a different syntax to XML. The DTD can be defined inline (as above), or in another location in a `.dtd` file. This would be formatted something like this:

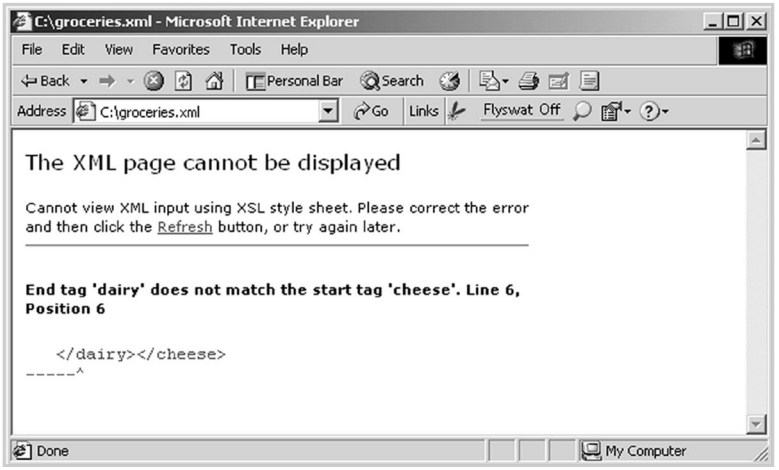
```
<!DOCTYPE grocery-list SYSTEM "grocery-list.dtd">
```

Flash MX does not really use the DTD, or perform much validation for that matter, so for our purposes it is only important to recognize that the DTD exists. You will certainly come across it if you ever load externally acquired XML documents into your Flash movies. However, you will not need to create or use a DTD with the XML you write yourself solely for Flash.

Validation

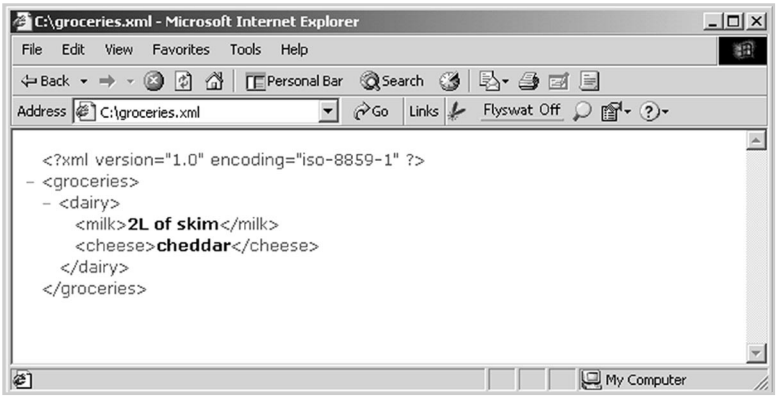
As we've stressed, XML is a strict language. Worse still, it doesn't allow for handling of errors, so it's very important to check your code works before attempting to work with Flash. A web browser that recognizes XML, such as Internet Explorer, will give us clues to any errors in our XML. Internet Explorer 5 was the first web browser to fully support XML. IE4 only included limited handling of XML, but it did ship with XML parsers and XML data source controls. A useful feature of IE5 is that it will display XML in the browser window, and validate your structures. After you have finished writing it, open your file in Internet Explorer, and it will inform you of any lines containing problems. You can also validate your files through the W3C website at http://validator.w3.org/.

The following screenshot shows the error message that arises if we try and validate our modified groceries example in Internet Explorer:



You will see a very similar error message if your XML does not validate. When Internet Explorer finds the first error it returns a message and stops validating. Once you have fixed that error, it will keep giving you hints as to what your problems are until your file does validate.

After your file validates, you will see the tree hierarchy in the browser window. As with the Windows Explorer tool on Windows machines, plus and minus signs allow you to collapse or expand the tree. This helps you see the levels of the hierarchy of the loaded XML file, as shown in our working groceries example:



Creating XML data

Now that we understand the terminology associated with XML and how to format it, we can start to write some XML. In this section, we will create three example XML files. The first XML file will be a simple menu hierarchy, which will give you an idea of how to create a very simple nested data structure. From there we will progress to more complicated concepts.

> *The sample files created in this chapter will also be used in the next chapter when we will integrate XML with Flash.*

Hierarchy in XML

As you've probably figured out by now, hierarchy is very important in an XML document. A good way to think of XML is that it is similar to a tree – which is a common analogy made when describing XML structures. An XML file will start with a root element, from which all child nodes will branch. More nested child and sibling nodes can then be placed within each of these child nodes, essentially branching out. Nesting elements within other elements creates a distinct hierarchy among the pieces of data, which reside on many different levels. The parent element of each child becomes more and more generalized. Sibling elements are on the same level, so should be somewhat similar in their relationship to the parent node. Moving up the tree gradually will bring you to the root level, or the most general and encompassing element. As we mentioned earlier, the nesting of elements is very important in XML, because one of the main purposes of this markup language is to describe complex data structures.

These nodes can be referenced in ActionScript by the properties `firstChild`, `nextChild`, and `nextSibling` or `previousSibling`. This referencing allows you to navigate through each node. In the following examples, we will look at how to construct XML using nested elements in order to create a hierarchy. We will also look at what each of the elements are, and what they are called in relation to one another.

In this first example, let's take a look at a menu which may be used on a web site for a movie theater. Our goal is to create a menu for this movie theater web site which needs three main areas. Each of these main areas then has sub-areas for the web site, and then in these areas, we could have further sub-areas. What is nice about XML is how the format is extensible, so we could add to any of these areas.

This file can be found on the CD, and it's called `menu.xml`:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<menu>
    <movies>
        <now-playing>
            <movie1/>
        </now-playing>
        <reviews>
            <review1/>
        </reviews>
        <coming-soon>
            <cs-movie1/>
        </coming-soon>
    </movies>
    <the-theater>
        <screen/>
        <sound-system/>
        <admission/>
        <map/>
    </the-theater>
    <about>
        <history/>
        <feedback/>
        <address/>
    </about>
</menu>
```

So, let's take a look at the code. In the first line, we are declaring the XML version and specifying the encoding type. This is not necessary when you are working solely with Flash, but it's a good form to adopt. Next we create a root element, which should describe the entire document. We have chosen to use `menu`, since that is what we are creating! All of our sub-elements will be nested within the root element `menu`. In Flash, `menu` is known as the level 1 child node:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<menu>
</menu>
```

Within `menu`, we want to have three main sections all on the same level, called `movies`, `the-theater`, and `about`. Therefore, the next step is to create sub-elements for each area, within our menu element:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<menu>
    <movies></movies>
    <the-theater></the-theater>
    <about></about>
</menu>
```

However, this isn't quite enough – we need to create further sub-elements, or child nodes, within each of our main areas. Let's first take a look specifically within our `movies` element. Here, we want elements for movies currently playing, movie reviews, and movies coming soon to the theater. So for each of these, we will make child nodes within our `movies` element, which will then be their parent:

```
<movies>
    <now-playing></now-playing>
    <reviews></reviews>
    <coming-soon></coming-soon>
</movies>
```

So now we have three sub-areas within our movies category. Again, each of these areas (`now-playing`, `reviews`, and `coming-soon`) are siblings to each other. But in each of these sibling areas, we need more sub-elements for each movie which is being played, reviewed or soon to be released. We simply continue with more nodes for each required movie, as follows:

```
<movies>
    <now-playing>
        <movie1/>
    </now-playing>
    <reviews>
        <review1/>
    </reviews>
    <coming-soon>
        <cs-movie1/>
    </coming-soon>
</movies>
```

The beauty with XML is how it is *extensible*, and it's therefore perfect for the purpose of a menu, which would be continually updated on such a site. As soon as a new movie is released, the site can update one of our elements or a new one can be added. You can update the menu according to what is current at the theater, simply by adding several more sibling nodes when necessary:

```
    <now-playing>
        <movie1/>
```

continues overleaf

```
            <movie2/>
            <movie3/>
        </now-playing>
```

The other sections are created in precisely the same way. `<the-theater>` and `<about>` are *sibling* nodes (along with the `<movies>` node), and those nested within them are their *child* nodes. Then these nested child nodes are siblings of each other. Finally, the root element is closed using the root element `</menu>` tag:

```
    <the-theater>
        <screen/>
        <sound-system/>
        <admission/>
        <map/>
    </the-theater>
    <about>
        <history/>
        <feedback/>
        <address/>
    </about>
</menu>
```

In this example, we have created an XML document that could be used in Flash for a menu system. We have created a distinct hierarchy, with menu items existing on several different levels.

Creating a complex data structure

Now that we have created a simple menu hierarchy, we will make a slightly more complicated XML document. In this example, we want a document that will list movies shown in two different theaters. In each of these movie theaters are different screens and each of these screens shows different movies at different times. Let's create a hierarchy to represent this system.

This XML file can be found on the CD as `movie-listings.xml`. First, the completed file is shown (validated in Internet Explorer), and following this we will step through the creation of this XML document:

This file has many different levels of nested elements, so let's start at the beginning and go from there. As usual, our document begins with an XML declaration and encoding specification, followed by our root element `movie-listings`:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<movie-listings>
</movie-listings>
```

Now we need to list the theaters in our movie listings. We have two different theaters, represented by the element name `movie-theater` with its attribute providing the theater names. These are nested within the root element:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<movie-listings>
    <movie-theater name="The Flix">
    </movie-theater>
    <movie-theater name="The Squeeze">
    </movie-theater>
</movie-listings>
```

The next step is to list the screens within each theater. Our first theater has two screens, and our second theater only has one screen. These screens are sibling nodes within their theater, and are child nodes of our movie theater:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<movie-listings>
    <movie-theater name="The Flix">
        <theater-one>
        </theater-one>
        <theater-two>
        </theater-two>
    </movie-theater>
    <movie-theater name="The Squeeze">
```

continues overleaf

```
        <theater-one>
        </theater-one>
    </movie-theater>
</movie-listings>
```

Now we need to create nodes representing our movies and their respective screening times. For this next piece of code, we will simply show the screening times for a movie in `theater-one` of our first movie theater. In `theater-one`, we have two different movies playing at various times. These two movie titles are given attributes that represent the rating of each movie. Sometimes, using attributes can make your information more usable, readable, and even speed up parsing times. However, how you structure your own XML documents is completely up to you, which is the beauty of such an open-ended language.

```
<theater-one>
    <movie-title rating="PG-13">Flashcoders</movie-title>
    <movie-title rating="R"><![CDATA[HTML&DW]]></movie-title>
</theater-one>
```

In this piece of XML, the second movie uses a `CDATA` section within the title. We use `CDATA` so we can use the `&` character without causing an error when we load our XML file (try it without the `CDATA` reference and see what happens!).

The next step is to nest our `showing-times` element within the movie title, and of course, add the times the movies are actually shown at. The elements representing movie showing times, `showing`, use a `time` attribute to list the time values. These child elements are easily extensible and updatable in XML:

```
<movie-title rating="PG-13">Flashcoders</movie-title>
    <showing-times>
        <showing time="6:30pm"/>
        <showing time="9:00pm"/>
    </showing-times>
<movie-title rating="R"><![CDATA[HTML&DW]]></movie-title>
    <showing-times>
        <showing time="7:00pm"/>
        <showing time="10:30pm"/>
    </showing-times>
```

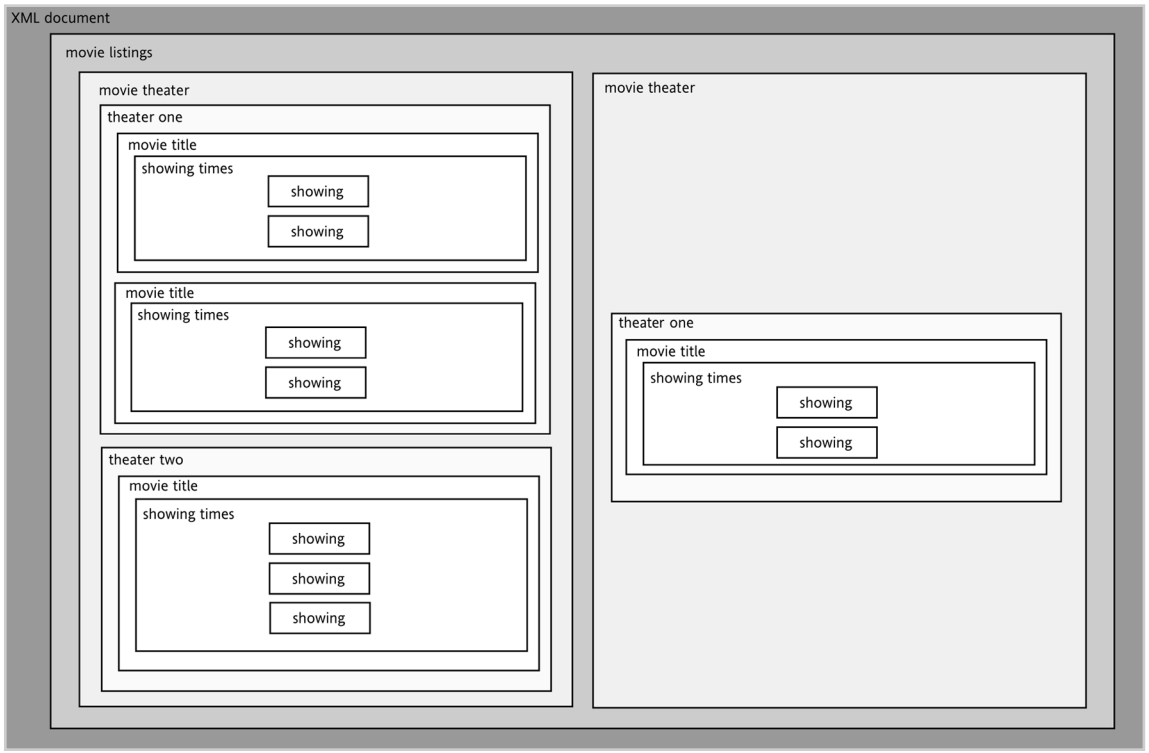
As you can easily see, the other theaters, screenings, and showing times are constructed in the same manner. After the last element of the final movie theater is closed, the entire XML document is completed by closing the root tag `</movie-listings>`.

You should notice the distinct heirarchy and number of levels within this XML document. By the time we reach the `<showing>` elements in our XML, we are actually many levels deep - but `<movie-listings>` remains the level 1 child node:

```
<movie-listings>
```

```
<movie-theater>
    <theater-one>
        <movie-title>
            <showing-times>
                <showing>
```

It often helps to visualize the relationships between nodes diagramatically:

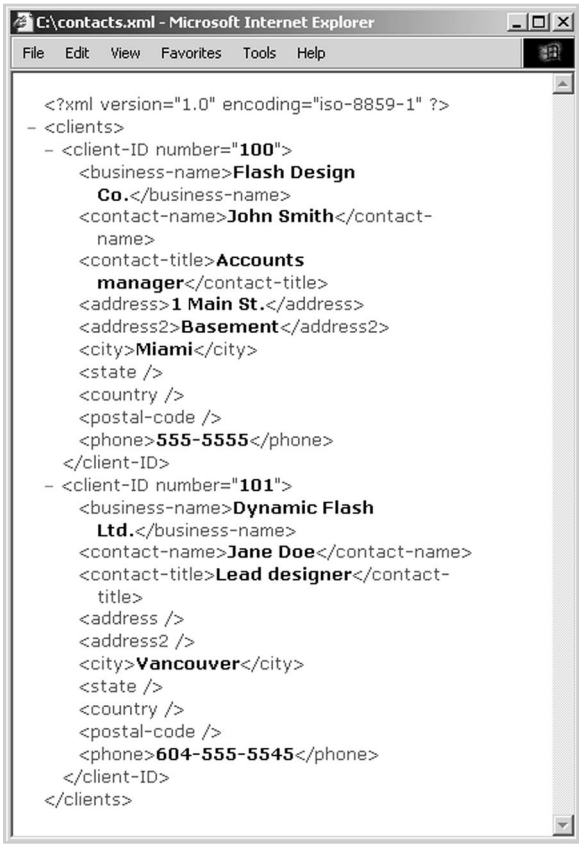


In the next chapter, when we come to use these XML documents with Flash MX, we'll have to carefully consider their different levels and relationships when referencing them with ActionScript.

An address list

For our final example in this section, we'll create an address list. Such a document is commonly used when information (or nodes) is constantly added to the file. In this file, you'll notice that many elements are left empty. You may wish for these to have empty representations in an interface, which a user can add to after the fact. We will also use this particular file in a further section in conjunction with Macromedia ColdFusion MX, where we will dynamically create this XML from a database using ColdFusion technology.

Here's the XML document, shown validated in Internet Explorer, representing our address list – it can be found on the CD as `contacts.xml`:



Note that in this file our elements are not nested to nearly the same degree as in our previous example, because it is unecessary given the content. Here we have created a fairly straightforward contact listing of clients who may be included in a database. We begin by setting our root element to clients, and then set the child element to reference the `client-ID`. Following this, our nested elements cover the content we need to gather or keep for each of our clients. You will probably notice that not all of our elements include text nodes. For example, our first client has information for the `address2` element, whereas the second client does not. In fact, it's worthwhile leaving these elements in the document despite the fact they do not contain text data - the empty elements can have this information filled in at a later date, or even use the element itself as data in your movie.

Looking at this file may remind you of a set of database records. In fact, what we will do in the following section is take records from a database and dynamically create an XML file from the data using Macromedia ColdFusion MX.

Server-side scripting with XML

In this section we will look at XML coupled with server-side programming and database interaction. This is useful when you need to take records from a database and convert them into an XML packet. In this section, we will use ColdFusion as a scripting language. However, the same principles can be applied to ASP, PHP, Perl, or any other web scripting language.

Server-side scripting is commonly used to create dynamic web applications by integrating web interfaces with back-end databases. Languages can vary considerably in how difficult they are to learn, but ColdFusion, which is what we will use for our example, is one of the easier technologies to pick up. It is actually very similar to XML in that both languages are tag-based.

Server-side scripting languages can be used to generate XML files from sources such as databases or newsfeeds. XML itself can be used to communicate with these scripts and dynamically pass data between the scripts and your Flash movie. Using this method of communication will enable you to incorporate content into your Flash movies from a wide variety of sources via XML.

Use of server-side scripting languages allows you to dynamically create XML files so you can change the XML returned to the user *on the fly*. A real-life example of this might be in an online bookstore – by allowing affiliates to pass variables to a server-side template (a ColdFusion MX file with `.cfm` extension), we could search our database of books and return only the matching records to the user. By filtering search results at the server, we reduce the amount of information being sent back and forth to the client, thus reducing the amount of bandwidth being used.

Let's look at another example of how dynamically created XML templates are used today: online web logs. If you are writing a blog, you might only want to show entries for the past two days on your front page, and automatically archive anything over two days. A scripting language could grab all entries from a database from the past two days, display these records on the front page, and then select all records older than two days for the archives. If you were *not* using server-side scripting, then you would have to manually edit two XML files (perhaps `current.xml` and `archive.xml`), and then copy old records from the current file to the archive on a daily basis. A different solution might be to have one single file for all of your entries. When a user comes to the front page of your site, you'd only show the entries from the last two days and ignore the rest. The downside of this approach is that you have to potentially transfer a large XML file from the server to the client in order to show a small portion of the records. By using server-side scripting, we only return the records from the past two days. Therefore, you aren't transferring a lot of unnecessary data back and forth between the server and the client.

As you can see, using scripting technologies with XML can be a great benefit when it comes to efficient site management and optimized content serving.

ColdFusion MX

In the latest version of ColdFusion, Macromedia has added XML support and made a few other additions to allow ColdFusion MX and Flash MX to integrate together seamlessly (current buzz words are *Flash remoting* and *server-side ActionScript*.). In this section we'll describe the basis of an example in which

ColdFusion MX is used to generate XML data. This example uses the `contacts.xml` file from the previous example, along with `coldfusion-xml.cfm`, which can both be found on the CD (skip ahead to **Chapter 19** for download details of the Macromedia ColdFusion MX free trial version).

Let's look the file `coldfusion-xml.cfm` – first we set the mime type for the current template using `cfcontent`:

```
<cfcontent type="text/xml">
```

Then the following tag will make sure that ColdFusion suppresses all extra whitespace that would be generated by the current template, and only returns data to the user's browser that is enclosed in the `<cfoutput>` tag:

```
<cfsetting enablecfoutputonly="Yes">
```

We then perform a query to grab all clients from our database:

```
<cfquery name="getClients" datasource="Contacts">
   SELECT *
   FROM Clients
   ORDER BY ContactName
</cfquery>
```

The `cfquery` tag allows us to let our ColdFusion template talk to a database. Before we are able to connect to our database we first have to set up a data source through the ColdFusion MX administrator - to access this, go to http://localhost:8500/cfide/administrator/ in your web browser, assuming a default installation (although if the ColdFusion server was installed on a remote server then you would substitute localhost for the IP address/URL of the server with ColdFusion installed). For more information on how to set up a data source, consult the ColdFusion documentation. We have included a simple database created with Microsoft Access that you can use for this example, you'll find it on the CD called `contactsdb.mdb`. For this example, we register this database with the data source name `Contacts`.

The `XMLDoc` variable declared next will hold the XML packet, which will ultimately be displayed in the browser. Note that since our XML declaration has a pair of double quotes in it, we will use a single quotes as our text delimiter in the next line:

```
<cfset XMLDoc = '<?xml version="1.0" encoding="iso-8859-1"?>'>
```

We use the `cfset` tag to set variables in ColdFusion. So for the above code we are setting a variable called `#XMLDoc#` and giving it a value of `<?xml version="1.0" encoding="iso-8859-1">`. All ColdFusion variables wrapped in a `cfoutput` tag should be enclosed in # signs, so that the ColdFusion server knows to process the variable instead of simply displaying the string in the browser.

The & character in the following line concatenates the existing value of XMLDoc with the new value, which is enclosed in double quotes:

```
<cfset XMLDoc = XMLDoc & "<clients>">
```

Next we'll loop over the query and, for each record returned by the query, loop over all the code between the opening <cfloop> and closing </cfloop>:

```
<cfloop query="getClients">
```

In this loop, the first line has a number attribute in the client-id element, so we put a pair of double quotes around the value. It's also worth pointing out that ColdFusion variables are enclosed in #, so in the following lines of code, whenever we see #*blah*#, ColdFusion will substitute the string #*blah*# with the value from the database:

```
    <cfset XMLDoc = XMLDoc & "<client-id number=""#ClientID#"">">
    <cfset XMLDoc = XMLDoc & "    <business-name>#BusinessName#
                                  </business-name>">
    <cfset XMLDoc = XMLDoc & "    <contact-name>#ContactName#
                                  </contact-name>">
    <cfset XMLDoc = XMLDoc & "    <contact-title>#ContactTitle#
                                  </contact-title>">
    <cfset XMLDoc = XMLDoc & "    <address>#Address1#</address>">
    <cfset XMLDoc = XMLDoc & "    <address2>#Address2#</address2>">
    <cfset XMLDoc = XMLDoc & "    <city>#City#</city>">
    <cfset XMLDoc = XMLDoc & "    <state>#State#</state>">
    <cfset XMLDoc = XMLDoc & "    <country>#Country#</country>">
    <cfset XMLDoc = XMLDoc & "    <postal-code>#PostalCode#</postal-code>">
    <cfset XMLDoc = XMLDoc & "    <phone>#Phone#</phone>">
    <cfset XMLDoc = XMLDoc & "</client-id>">
</cfloop>
```

The loop is finished so we close the root element:

```
<cfset XMLDoc = XMLDoc & "</clients>">
```

Next we display the value of our #XMLDoc# variable to the browser. It is important that whenever you display a ColdFusion variable on the screen that you wrap the variable in <cfoutput> tags:

```
<cfoutput>#XMLDoc#</cfoutput>
```

Finally we reset the enablecfoutputonly flag back to No:

```
<cfsetting enablecfoutputonly="No">
```

Now drop this file, `coldfusion-xml.cfm`, in the wwwroot folder (by default this should be at C:\CFusionMX\wwwroot\) and then view this file in a web browser by going to http://localhost:8500/coldfusion-xml.cfm - you should see a dynamically created XML tree structure. We'll see some more detailed examples involving server-side scripting with ActionScript and XML in the next chapter.

We've now finished studying the theory and concepts of XML, and the basics of creating an XML document that can be read by Flash. We've explored the many terms and ideas surrounding the XML language, the possibilities it offers to your Flash development and interfaces, as well as how to create and format XML documents correctly. We've also been introduced to using server-side scripts with XML and Flash.

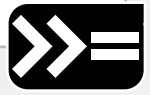
> *Although we've tried to provide as broad an introduction as possible in this chapter, it's clear that XML is a pretty large topic! So, if you're interested it might be worth checking out some XML-specific titles. For instance, although its cover is pretty scary, Beginning XML Second Edition from Wrox Press, ISBN: 1-861005-59-8, provides an excellent grounding in the use of XML. Also, from friends of ED, you could try Flash XML StudioLab, ISBN:1-903450-39-X* (www.friendsofed.com/books/studiolab/flashxml/).

So, where do we go from here? In the next couple of chapters, we'll look at integrating XML with Flash via the `XML` object and its predefined methods and properties. We'll also study socket servers, an exciting way of using XML to create dynamic applications.

19 Reading XML in Flash

Objectives

- Translating XML structures into ActionScript dot notation hierarchy
- Loading XML into Flash:
  - Understanding the `XML` object
  - Connecting to and loading XML
- Parsing XML in Flash:
  - Extracting data from XML files
  - Parsing and using XML data
- Creating new XML data:
  - Creating and adding data nodes
  - Working with a database
  - Sending and receiving data



Introduction

Using XML with Flash opens the door to adding real interactivity to your movies, not to mention the time an effort we save utilizing code that we can reuse in a variety of different interfaces. As discussed in the previous chapter, we can use XML to syndicate content, or just use it as a single data source for more than one movie. Data formatted in XML can also be used in conjunction with a variety of different scripts for data transmission.

XML data structures are brought into Flash, parsed in the Flash Player, and data in the XML tree structures is converted into the ActionScript parent/child relationships. This data is then loaded into an XML object that can then be processed further using ActionScript. The data extracted from the XML file can be used in many different ways in your Flash movies – such as for creating highly interactive environments, maintaining complex sets of data, or producing interesting interfaces using movie clips. In this chapter we'll explore some of the ways that we can process XML data, and provide several different examples of how you can integrate it into a movie. It's important throughout to understand the parent/child structures of your files when extracting the data using ActionScript. You'll notice a heavy dependency on arrays and looping through each *node* (defined in **Chapter 18**) of the XML data.

Some of the common uses of XML and Flash include content syndication (such as newsfeeds), blogs, MP3 play lists, database interaction, and site content management. The popularity of integrating these two technologies has been increases since the release of Flash MX, and is likely to rise even more so with the release of the Flash Communication Server MX. In this chapter we will learn how to load XML data into a Flash object, and process the data so that it is usable in an interface. We will also look at creating new XML data in a movie, and transferring it to and from a database.

Translating XML structures into a dot notation hierarchy

When working with XML data in Flash, we need to translate the markup tags into a dot notation hierarchy. Now that you're familiar with tag-based structures, we need to turn it around and rework these structures in a completely different way. By using such a hierarchy in ActionScript, we're able to access and process XML data within its structure. Let's begin with a segment from the `contacts.xml` file from the previous chapter:

```
<clients>
   <client-ID number="100">
      <business-name>Flash Design Co. </business-name>
      <contact-name>John Smith</contact-name>
      <contact-title>Accounts manager</contact-title>
   </client-ID>
</clients>
```

Looking at this structure we can see that it's analogous to the structure of nested movie clips in Flash, and the hierarchy is just as important as the scope of our variables and functions in ActionScript. We simply need to start with the root node, which is known as `firstChild`. Then we look through the nodes within the structure by looping through the child nodes of an element. For example, if we want to

access client-ID with our myXML Flash XML object we need to use the following hierarchy in our ActionScript:

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

This hierarchy is needed because firstChild references the root element, and client-ID is the nodeName of the first child of this node array. Remember that arrays are 0-based, so childNodes[0] actually refers to the first child, childNodes[1] refers to the second node, and so on. Then, nodeName extracts the name of this element.

To extract the number from the client-ID element, 100 in this case, you would need to use the following:

```
myXML.firstChild.childNodes[0].attributes.number
```

If we wanted to dig deeper into our structure, and access John Smith from the contact-name element we would need to use the following:

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

Note that because contact-name is the second sub-element of client-ID we must refer to it as childNodes[1].

As you can see, this works the same way as the dot notation that you are used to when object referencing in ActionScript, except we are extracting child nodes within parent nodes from a tag based structure. This is instead of, for example, targetting a child movie clip nested within a parent movie clip. We'll return to this topic later on in the chapter called when we come to look specifically at extracting data from XML. For the moment, however, we'll shift our focus to loading up XML documents into the Flash MX XML object.

Loading XML files into Flash with the XML object

Before anything can be done with our XML files, we must load the data into our Flash movie. This is accomplished using the XML.load() method, which loads the data into the XML object we create in Flash. In this section we will look at the XML object constructor, and how we can load information into it using XML.load() method. We'll also learn how to check that it has loaded by using the XML.onload event handler and the status of the loaded data using the XML.status property.

XML objects and variables in a Flash movie are contained within an XML object. You use the standard object constructor to create a new object, and then load local or remote XML data into it. The XML object constructor is as follows:

```
myXMLObject = new XML([source]);
```

Where `source` is the optional XML text to parse that forms the new XML object. Once created, you can then process and manipulate the data of `myXMLObject`. In fact, there are two different ways we can create an XML object. The other way creates an empty XML object, as follows:

```
myXML = new XML();
```

After you create a new empty XML object, you can load XML data into this new object from a document on the server (local or remote) using server-side script. In order to do this, you use the `XML.load()` method, which we'll look at next.

Returning to the first technique for creating an XML object, we could enter data within the `source` parameter of the object constructor, like so:

```
foodXML = new
➥XML("<food><group>vegetable</group><type>carrot</type></food>");
```

If you construct the XML object in this way, Flash will parse the data inside the `source` parameter. To check whether or not this has been formed correctly, you can use the XML.status method, discussed later in this chapter.

Importing data with XML.load() and XML.onLoad

The `XML.load()` method is used to load data from a particular URL into an `XML` object:

```
myXML.load(url);
```

The data can be loaded from a local or remote file location. If you are loading XML data from a remote location, you must use middleware (server scripts) in order to access the information (we'll explore this shortly).

Note that there are many potential problems that could occur when loading this data (such as a faulty transmission, for one), so it is important to use an event handler to call a function when the data has finished loading. Accordingly, we typically use the `XML.load()` method and the `XML.onLoad` event handler together when bringing XML data into a movie. When it's included in a function, `XML.onLoad` essentially checks whether or not the data has been successfully loaded, so you can run further ActionScript based on this information.

`XML.onLoad` is used in callback functions with both `XML.load()` and `XML.sendAndLoad()`. It sends a Boolean value back to Flash once an XML object has been loaded into a movie. If the XML data has been completely loaded without a problem, `true` is returned into the `success` parameter (see next example). If there was an error when loading the data, then `false` is returned. You can also use the `XML.status` property to check for a specific error encountered when parsing XML data – we'll look at this shortly.

Creating an XML object

In this example, we are simply going to load XML data into a Flash movie. We will use the XML object constructor, the `onLoad` handler, and the `load()` and `toString()` methods (remember, the `LoadVars.toString()` was discussed in **Chapter 17**, and it's used just the same as the `XML` object).

Open up `loadMenu.fla` from the CD, and make sure you copy it to the same directory as `menu.xml` – this is the XML document that we're going to load into Flash. Check out the code on frame 1 of the actions layer:

```
menuXML = new XML();
menuXML.onLoad = function(success) {
      if (success) {
            trace("file is loaded, and here is the file: ");
            trace(this.toString());
      } else {
            trace("error in loading file");
      }
};
menuXML.load("menu.xml");
```

In this code, first we create a new empty XML object using the standard object constructor:

```
menuXML = new XML();
```

In the following steps we will load data into our `menuXML` object. The second step is to create an inline function that is triggered when a file is loaded into a Flash movie:

```
menuXML.onLoad = function(success) {
```

Next up, assuming the load is a success, we trace a suitable text message to the output window, and then convert the contents of our `menuXML` object to a string, and trace the result (just to make sure it's there!):

```
      if (success) {
            trace("file is loaded, and here is the file: ");
            trace(this.toString());
```

Note the use of the `this` statement here meaning that we're attaching the `toString()` method directly to the `menuXML` `XML` object, as determined by the scope of the function.

Additionally, if the load is not successful, perhaps due to data transmission failure of some sort, we trace out an error message:

```
      } else {
            trace("error in loading file");
```

```
        }
};
```

Finally, we actually load the XML document with the following line:

```
menuXML.load("menu.xml");
```

Now if you test the movie, the contents of the XML document are displayed in the Output window (note that we'll use this file again later on in the chapter):

It's important to remember that this only works because the XML document is saved in the same location as the SWF file. If you were displaying this movie online, the XML file would need to be located on the same domain as the movie file – obviously in this scenario you'd be using dynamic text fields to display the text instead of a simple trace. You'd need to use a server-side technology such as ColdFusion, ASP, or PHP if you wanted to load an external XML file. This will be covered using ColdFusion MX when we look at an example demonstrating how to connect to a newsfeed, later in this chapter.

Using the XML.status property

You can use the `status` property to check how the XML file has loaded. With this property you have a numeric value returned that provides information regarding whether or not an XML document was successfully parsed into the XML object. If it wasn't, it'll offer clues as to what the problem might be.

```
myXML.status;
```

Let's take some XML data we have already seen, and change it in order to create an error. Remember the `foodXML` object we created earlier? If we trace out the status of it, we'll see whether or not it's valid XML:

```
foodXML = new XML("<food><group>vegetable</group><type>carrot</type></food>");
trace(foodXML.status);
```

When you test this code, your Output window will display a big fat `0`. ***What does this mean? Well, as we'll soon see, it means everything's fine!***

Now try removing one of the tags. Below we've deleted the opening <type> tag, to make the XML into something that will not parse correctly:

```
foodXML = new XML("<food><group>vegetable</group>carrot</type></food>");
trace(foodXML.status);
```

This time when we run the following code we'll see an output of `-10`. This refers to the specific error scenario in which an end-tag is encountered without its associated start tag. How do we know this? Well, the following table gives details of all of the possible status codes:

XML.status code	Description
0	No error - successful parse
-2	`CDATA` section not terminated correctly
-3	XML declaration not terminated correctly
-4	DOCTYPE declaration not terminated correctly
-5	Comment not terminated correctly
-6	XML element incorrectly formed
-7	Out of memory
-8	Attribute value not terminated correctly
-9	Start-tag not matched with end-tag
-10	End-tag encountered without matching start-tag

We can use this property to check that the XML parses correctly before we manipulate the loaded data. Ultimately, this means that we can write our code such that it can *catch* errors before they can do too much damage, or waste processor and transmission time – this leads to truly robust code.

Connecting to an external XML source

An extremely common way of using XML is to pick up external data feeds and display them in a Flash movie, or on an HTML based web site. These data feeds can help a site stay regularly updated with fresh content automatically. It can also help make a site easy to maintain. Data feeds may come from news sites, portals, or even your developer friends in the community. They might be used to display dynamic data which is continuously fed into an interface. Alternatively, you could load the information into a movie once, and simply display the information in a more static format.

To get started with accessing data feeds you need to have an understanding of XML, parsing, and know a little bit of know-how about security and a server-side scripting language. Obviously, because you'll be working with server-side scripting languages, you'll also need to have the proper servers and middleware installed wherever you're uploading your Flash movies to. You can even use a server on your own personal computer at home, such as Microsoft IIS or the Apache server on.

In this chapter we'll be using Macromedia ColdFusion MX in a couple of examples, which has a built-in stand alone web server. In conjunction with this, we'll be getting to grips with the language used with the ColdFusion server – ColdFusion Markup Language (CFML). So, to test any of the examples involving middleware in this chapter, you can download a 30-day trial version of ColdFusion MX from Macromedia at: www.macromedia.com/software/coldfusion/downloads/

CFML is perhaps one of the easiest server-side scripting languages to get started with when working with this kind of data transfer. You may be interested not only in using it with XML, but trying some of the other possibilities of integrating this language and server with Flash MX. If you're more familiar with another language, like ASP or PHP, you may be reluctant to start over with yet another technology. If this is the case, don't worry, it's only the general concepts that we're really interested in – each example should offer you some highly transferable methodologies. That said, you might share the opinion that adding another tool to your tool belt of design skills can't really hurt!

Connecting to a newsfeed

In our earlier XML examples, we've loaded files from a local source only. If you load a file from an external source (on someone else's server, for instance) you won't notice any problems when you test it in the testing environment (using CTRL+ENTER). However, if you publish your movie in a web browser you'll find that the XML document does not load. XML data cannot be loaded directly from another server because of the security features built into Flash.

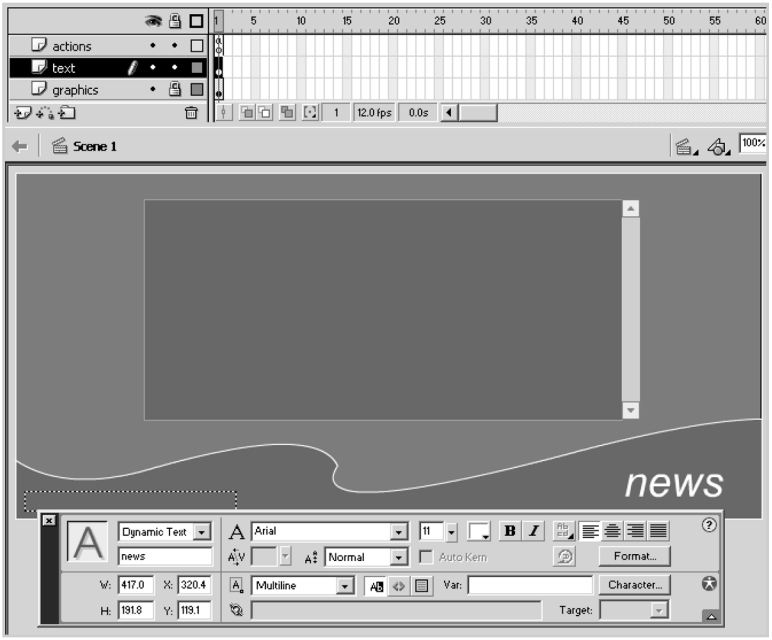
> *For more information on security and Flash, refer to the following article on Macromedia's website:* www.macromedia.com/desdev/mx/flash/whitepapers/security.pdf.

To work around this problem, we need to use a server-side scripting language and middleware to load the XML. It is not a difficult process, and you can use just about any middleware language. We will detail how to do so in the following example by connecting our Flash movie to an external newsfeed.

A popular and extensive newsfeed resource is that of Moreover Technologies, a major online source of real-time news, information, and even business rumors. You can find the XML newsfeed listings of Moreover at the following location: http://w.moreover.com/categories/category_list_xml.html**.**

In this example, we are going to load an external newsfeed relating to the graphics industry into a Flash movie. The example files for this exercise can be found on the CD as `newsfeed.fla`, and the associated ColdFusion template is called `graphic_industry_news.cfm`.

OK, let's get down to it – open up `newsfeed.fla` to see how it has been constructed:



In this FLA we've got a few layers separating our graphics, dynamic text fields, and ActionScript layers: *text*, and *actions*. Note that the large text field has an instance name of `news` and it's set to Multiline in the Property inspector (CTRL+F3). You should also remember to sure your font color is in contrast to your background color. A scroll bar component has been added to this dynamic text field, and its Target TextField parameter is set to `news`, again via the Property inspector. Additionally, there is another text field in the bottom left corner of our movie to display any error messages – this called `error`.

Now open up the Actions panel (F9) on frame 1 of the actions layer, where you'll see the following code:

```
myXML = new XML();
myXML.load("http://p.moreover.com/cgi-
Xlocal/page?c=Graphics%20industry%20news&o=xml");
myXML.onLoad = function(success) {
      if (success) {
            _root.news.text = myXML;
                } else {
            _root.error.text = "loading has failed.";
      }
};
```

The first thing we do is create a new XML object, and load a newsfeed using the direct URL, like we have already done in earlier examples. Test the movie now by pressing CTRL+ENTER. You will see a large amount of XML data in the `news` text field:

Now if we publish our movie, upload the file to a web server and re-test it, we'll see that the movie doesn't work any more – the newsfeed is no longer displayed in the text field! We *cannot* load external content into the movie yet without some server-side technology. So, we need to change our code a bit and add an extra layer in the data transmission process to work around the security settings in Flash. As you probably already know, Flash can only load content from a local file.

For our middleware layer we're going to use ColdFusion MX. However, as we've stressed, it's up to you what server-side scripting you opt for. We need to create a ColdFusion MX template (a `.cfm` file) to load the URL into. This is what we do in `graphic_industry_news.cfm`, and our movie will grab the newsfeed from this location:

```
<cflocation url="http://p.moreover.com/cgi-local/page?c=Graphics%20indus-
try%20news&o=xml">
```

The `<cflocation>` tag is a server-side redirect that forwards the web page to a new location, which is the content we need from the moreover site. In the finished web page we'll be loading content from the external newsfeed *via* the local `graphic_industry_news.cfm` – so Flash only needs to *see* the local file.

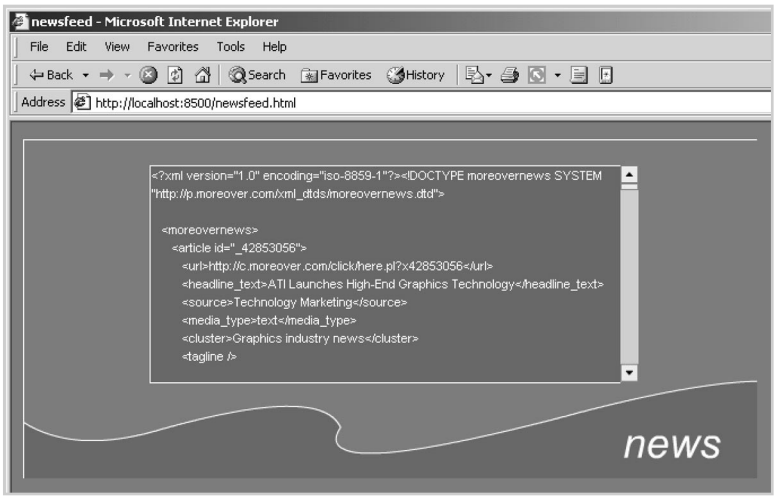
For other languages, the code is practically the same. For example, in PHP the equivalent code would be written as follows:

```
<?php header("location:http://p.moreover.com/cgi-
local/page?c=Graphics%20industry%20news&o=xml");?>
```

Now, return to your Flash movie and modify the code so that it knows to look for the ColdFusion file:

```
myXML = new XML();
myXML.load("graphic_industry_news.cfm");
myXML.onLoad = function(success) {
      if (success) {
            _root.news.text = myXML;
      } else {
            _root.error.text = "loading has failed.";
      }
};
```

Now we need to upload all of the files, the ColdFusion template and the modified SWF, this time to a web server that recognizes ColdFusion. We're using the ColdFusion MX server (trial version download details were given earlier in this chapter) – and we can deploy our web page by simply by copying the relevant files to the ***C:\CFusionMX\wwwroot*** directory and accessing the file in your browser with ***http://localhost:8500/newsfeed.html*** (assuming you've published the HTML file for your SWF in Flash MX). This time our Flash movie will have no problem grabbing the newsfeed from the Moreover server, via our middleware:



Flash is again accessing a file on the same server, so there is no longer a security issue. You could use many different server-side languages to achieve this; the code is only slightly different in each case. We'll come back to sending and loading data, specifically turning to the `XML.sendAndLoad()` method, in the final section in this chapter.

Note that this example isn't quite complete as it is. In any web site containing news headlines, stories, and links, you'll find the information nicely formatted for easy reading. Here, we're just presenting the raw XML data. Obviously, we need to take it that one step further by parsing the XML data and presenting it in some Flashy way. We'll learn how to do so very soon.

The power of RSS

What's RSS? Well, you may have heard about it when looking into XML, newsfeeds, or any number of the Flash developer web sites out there. RSS stands for RDF Site Summary (where RDF is the Resource Description Framework) and was originally developed by Netscape. It is a particular format of XML which is commonly used when a website author wants to easily share headlines. RSS is typically used to share content between several different sites as a *channel*, and syndicate headlines from a website.

Parsing RSS data feeds in Flash is popular because of the wealth of blogs and newsfeeds in the designer and developer communities. RSS is used because it sets a standard for syndicated content. For more information on RSS and creating RSS files, refer to the following link, which is an excellent resource for tutorials, examples, and links to newsfeeds using RSS:

www.webreference.com/authoring/languages/xml/rss/

Some developers are starting to create systems to display multiple RSS feeds in centralized locations. The following websites provide information on parsing RSS feeds, and great examples of how RSS feeds can be used in a Flash interface:

www.samuelwan.com/information/

www.philterdesign.com/dev/flashFeeds/

Parsing and extracting XML data using Flash

Now that we can load XML data into Flash, we need to manipulate it so that it's useful in our movies. So far we have simply viewed raw XML data in text fields or the output window, which isn't terribly useful. Parsing XML data will sort the elements and attributes from the XML structure into usable data, contained within the object. Once you have brought the data into Flash, you can use ActionScript to manipulate it within your interface in any number of creative ways.

In **Chapter 18** we learned how to create XML files in a proper format. In this section, we are going to take the XML files we created in that chapter and learn how to process the data using ActionScript.

Flash uses the Document Object Model (DOM) API to parse XML content, as opposed to an event-based parser, like PHP uses. The DOM is extremely powerful and effective in its job of navigating and

manipulating the object-oriented data it contains – for more information, go to www.w3.org/DOM/. One pitfall of using this kind of parser is that it typically doesn't handle large XML files very well. Large files sometimes take up too much of a computer's processor power to build the model, since the data is read into memory. Accordingly, it's advisable to break large XML files into smaller pieces before parsing them in Flash. That said, it should be noted that the XML parser in the Flash Player 6 has dramatically improved its performance over the Flash 5 Player.

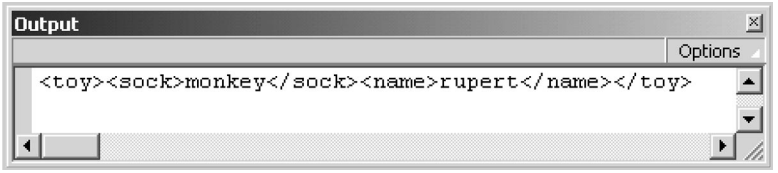
Let's look at a simple example of how to extract data from some XML information. Here's a simple `XML` object in Flash:

```
monkeyXML = new XML("<toy><sock>monkey</sock><name>rupert</name></toy>");
```

To begin, we'll extract the first child of our data (remember, the first child of an XML data structure refers to the root node):

```
monkeyXML = new XML("<toy><sock>monkey</sock><name>rupert</name></toy>");
trace(monkeyXML.firstChild);
```

On testing this code, you'll see the following output:



If you only wanted the *name* of the first child node, you could use the following trace to see what data would be extracted:

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

This would output toy. On the other hand, if you added the following code:

```
trace(monkeyXML.firstChild.firstChild);
```

You'd see <sock>monkey</sock> in the Output window. Now let's use a `for..in` loop to extract the child nodes of the XML structure. If we examine our XML data we'll find we have two child nodes: `sock` and `name`. We can use the following loop to grab these two child nodes:

```
for (i in monkeyXML.firstChild.childNodes) {
trace(monkeyXML.firstChild.childNodes[i].nodeName);
}
```

Now you will see sock and name in the output window. Understanding complex structures may take a little practice at first – it's important to dissect the XML files and understand the relationships between

the nodes. As we'll soon see, looping, recursion, and arrays are used heavily when parsing XML files. Before that, we need to make a return visit to the issue of whitespace in our XML documents.

Avoiding whitespace issues

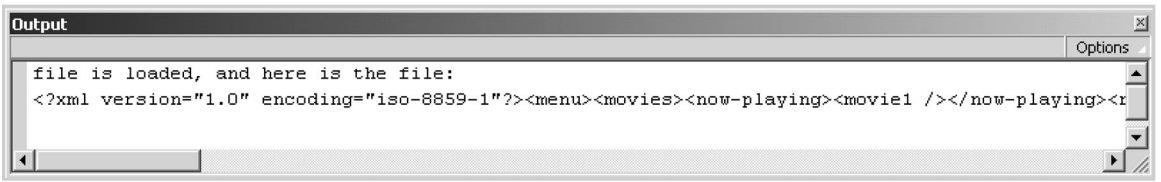
In our first example in this chapter we loaded `menu.xml` into a Flash movie, and output the XML data using the `toString()` method. You may have noticed that this string of XML data was formatted pretty much the same as how it was in the text editor, like Notepad. It was relatively easy to read, and you could easily see the nested structure. Clearly, this XML data still included whitespace, meaning that it wasn't quite ready to parse in Flash - XML creates empty nodes where there are spaces or carriage returns in the file.

As discussed in **Chapter 18**, we need to strip all the whitespace out of XML documents loaded into Flash so that we can correctly parse the node trees. We haven't had any trouble with our examples so far because we haven't tried to parse any XML with whitespace in the data structure. However, if we tried to work with one of our external files from Chapter 18, we would have been unsuccessful in our attempts because of the whitespace in these documents.

Let's take the whitespace out of the first example from this chapter. Open up `loadMenu_whitespace.fla` – this is very similar to our earlier example (`loadMenu.fla`), but the following line has been added to the code:

```
menuXML = new XML();
menuXML.ignoreWhite = true;
menuXML.onLoad = function(success) {
      if (success) {
            trace("file is loaded, and here is the file: ");
            trace(this.toString());
      } else {
            trace("error in loading file");
      }
};
menuXML.load("menu.xml");
```

If you test this movie (CTRL+ENTER) you'll notice quite a difference in the Output window:



Now all of your XML is output in one continuous line. By setting the `ignoreWhite` property to `true` we remove all of the carriage returns, spaces, and tabs from the XML document so it can be parsed correctly. If you neglected to do this, each *white space* would be treated as a different node and your XML would therefore be extremely difficult to handle in Flash. For instance, the very first node of your document would be empty (refer to the first screen image in this chapter to see why). Removing whitespace makes it much more difficult for people to read, but it's ideal for a machine to parse. In the next section we'll find out why it's so important to learn how to navigate the structure of XML documents.

Using onLoad to parse XML data

As we've seen, when loading XML data we need to use the `onLoad` event handler to tell us whether or not the data was fully loaded without a problem. It is necessary to use this method before we begin to parse our data, so we use it to initiate processing data once the data has completed loading. A parsing function can then be called based on the `onLoad` event handler. When loading XML, we've simply been tracing the results of our load to the Output window. Generally speaking, to parse the XML data after it loads successfully we'd set up the `onLoad` handler as usual, and add the specific parsing code like this:

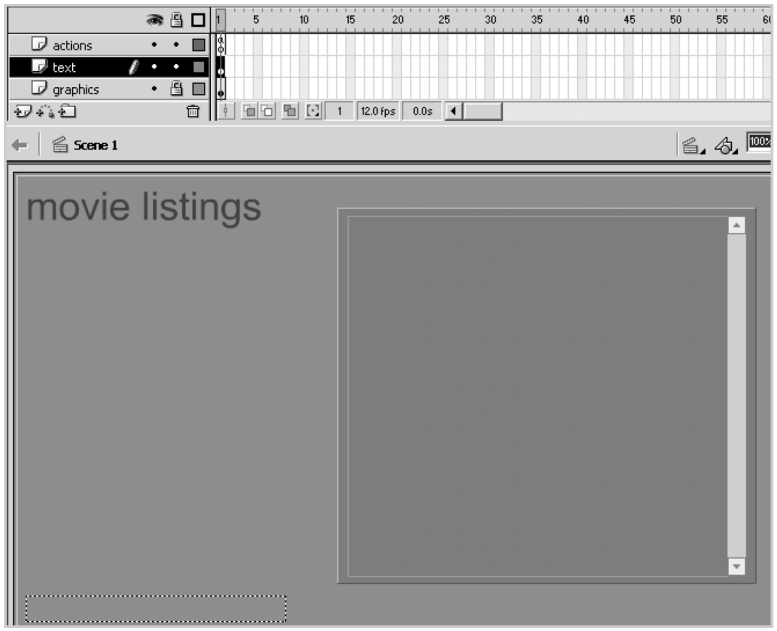
```
myXML = new XML();
myXML.onLoad = function(success) {
      if (success) {
            parseXMLdoc();
      } else {
            trace("error loading XML");
      }
};
myXML.load("myFile.xml");
function parseXMLdoc() {
      // Code to parse XML file goes here.
}
```

The function `parseXMLdoc` will handle parsing the data in the loaded XML document `myFile.xml`. In the next section we'll look at a real world example of parsing and formatting XML in Flash using this model.

Parsing and formatting XML in Flash

In **Chapter 18** we created an XML document of movie listings. In this example we'll load the `movielistings.xml` file into a FLA file, and then parse the data and list it on the stage. The FLA is called `movielistings.fla` and can be found on the CD. Remember, you'll need to save these two files into the same folder for the XML to be accessible.

Open `movielistings.fla` and have a look at the stage in the Flash MX authoring environment - here we see 3 layers: actions, text, and graphics. As in our earlier example, on the text layer we have a couple of dynamic text fields for our listings content and an error message:



Now take a look at the code attached to frame 1 of the Actions panel. There's quite a big chunk of code here, so we'll break it down into nice digestible pieces to describe it.

In the first segment of our ActionScript, data from the `movielistings.xml` file is loaded into a new XML object named `listingsXML`. If the load is successful, the `parseXMLdoc` function is called (described next). If the XML does not load correctly, an error message is displayed in the `error` text field:

```
listingsXML = new XML();
listingsXML.ignoreWhite = true;
listingsXML.onLoad = function(success) {
      if (success) {
            parseXML();
      } else {
            _root.error.text = "error loading XML";
      }
};
listingsXML.load("movielistings.xml");
```

The `parseXMLdoc` function that actually parses the XML file is the most interesting thing here, so we'll step through its code piece by piece. It's probably useful to open up the `movielistings.xml` file and follow along with the nodes that we are looping through. Let's look at the first part of this function:

```
function parseXML() {
      _root.listings.html = true;
```

```
// Parsing code goes in here.
for (i=0; i<listingsXML.firstChild.childNodes.length; i++) {
      var movieTheater = listingsXML.firstChild.childNodes[i];
      _root.listings.htmlText += "<b><font color='#000000'
      ➥size='15'>"+movieTheater.attributes.name+"</font></b><br>";
```

First of all, we add some HTML formatting to the text displayed in the `listings` text field, which will make it much easier to read. Then we set the variable name `movieTheater` to `listingsXML.firstChild.childNodes[i]` because this will simplify our code later on in the function (it's essentially used as a shortcut). Next, the first `for`-loop loops through `firstchild.childNodes` (these are the `movie-theater` elements in the XML file) and extracts the attributes from each one. So, this segment of code displays the name of each theater, and formats it using HTML.

The next piece of code loops through the movie screens (`theater-one`, `theater-two`) and changes these strings into upper case characters:

```
for (j=0; j<movieTheater.childNodes.length; j++) {
      var theater = movieTheater.childNodes[j];
      _root.listings.htmlText += "<font color='#BDD1BC'>"
      ➥+theater.nodeName.toUpperCase()+"</font><br>";
```

The line `var theater = thisMovieTheater.childNodes[j]` creates another shortcut variable, so we don't have to fully scope our hierarchy. As you may notice, it also contains `movieTheater`, another shortcut we created in the previous step.

The next couple of lines of ActionScript loop through each of the movies within the theaters we listed above:

```
for (k=0; k<theater.childNodes.length; k++) {
      var movie = theater.childNodes[k];
```

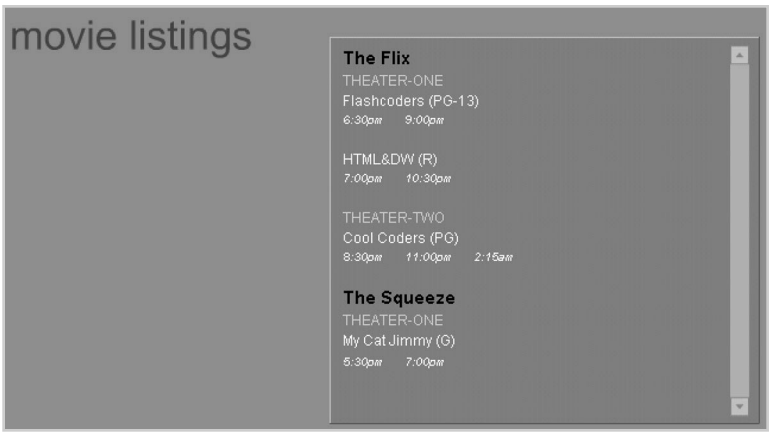
Now for our final piece of code:

```
switch (movie.nodeName) {
case 'movie-title' :
     _root.listings.htmlText += movie.firstChild.nodeValue+"
➥ ("+movie.attributes.rating+")<br>";
     break;
case 'showing-times' :
     var showTimes = "";
     for (m=0; m<movie.childNodes.length; m++) {
          showTimes += movie.childNodes[m].attributes.time+"    ";
     }
     _root.listings.htmlText += "<i><font
size='10'>"+showTimes+"</font></i><br><br>";
```

```
                                    break;
                                }
                            }
                        }
                    }
                }
```

This `switch` statement evaluates the name of the current node (either `movie-title` or `showing-times`), and processes each element differently. If it is a `movie-title`, it will append the name of the movie and the rating to our `listings` text field. If it is a show time, then it loops through the showings and grab the time attributes. This text is added to a temporary string called `showTimes`, which is then appended to the `listings` text field. The switch statement is useful when there are different child nodes at the same hierarchical level, and they need to be processed in different ways.

Test the movie to see the finished effect – our listing details has been loaded into Flash, parsed, and then nicely formatted for our movie listings page:



Understanding recursion in parsing

XML is recursive, meaning that it has a tree-like data structure composed of repetitive nested patterns. For example, or an element can have other elements within it, and in turn they may have their own child elements. When parsing, recursion refers to a function that calls *itself* in order to complete a task.

Using recursive functions is often helpful when dealing with XML – you might find it is useful when you are working with files that you don't know what to expect with regard to the depth of the data. We should note, however, that the downside of a recursive function is that it will generally be slower than a single function with loops.

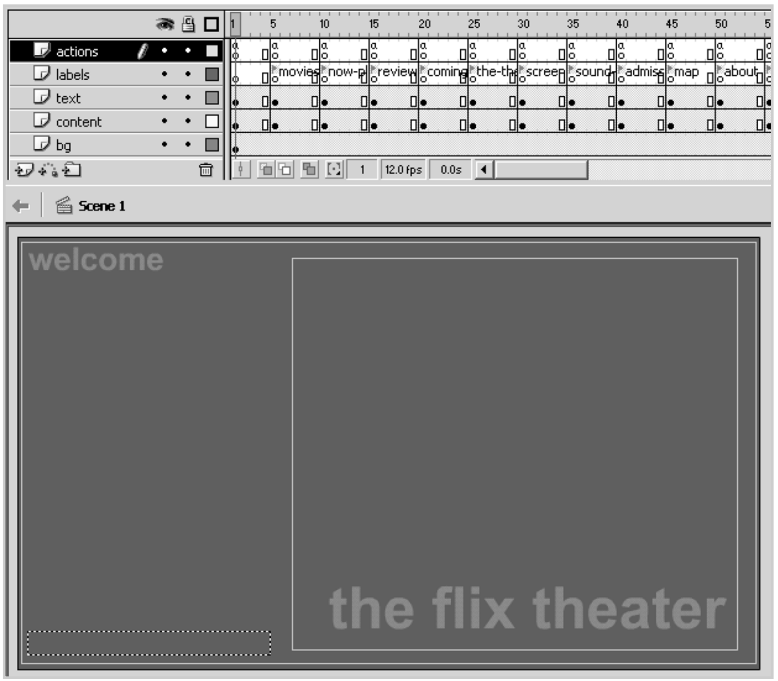
A recursive function will look at each part of the XML document, and determine if it is a text node or an element. Elements will need to be parsed, whereas text nodes will likely be placed somewhere as content – your ActionScript will determine what needs to be done. This is where you may need to use the

`XML.nodeType` property (discussed in **Chapter 18**) in your code, which will tell you if you are dealing with a text node (returns type `3`) or an element (returns type `1`). Essentially, a recursive function breaks your XML data into continually smaller pieces until it is fully analyzed.

Manipulating and presenting XML data

In this example we will parse our `menu.xml` file (developed in **Chapter 18**, and used earlier in this chapter) and manipulate the data to suit our needs. This should be a relatively easy document to parse, although we will make some of the nodes part of the interface instead of simply displaying the data as plain text in our movie – we can turn the text (our menu) into hyperlinks.

Take a look at `menu.fla` and `menu.xml` from the CD – there's a lot going on in the main timeline of this FLA file, so it's worthwhile familiarizing ourselves with the general layout before we go on to study the code. We have different pages for each area of this web site. The stage has been set up with a very basic layout for each page as a labeled frame that will be used in our navigation system:



What we want to do with the ActionScript in frame 1 of the Actions layer is load the XML data, and format it so the text is converted into buttons that can be used to navigate to every area of the web site. Let's step through the code – we begin by setting up some variables to help position our text fields, and then we load the XML file and write the `onload` function:

```
stop();
var XOffset = 10;
var YOffset = 19;
var currentY = 32;
menuXML = new XML();
menuXML.ignoreWhite = true;
menuXML.onLoad = function(success) {
      if (success) {
            parseXML();
      } else {
            _root.error.text = "error loading menu";
      }
};
menuXML.load("menu.xml");
menuid = 1;
```

The `menuid` variable here is a unique ID for each text field that ensures every field is placed on a different level in Flash, so that no instance will be overwritten.

The next step is to parse the `menu.xml` document and manipulate the data so it becomes part of our interface. What we want to do is create text fields which can be used to navigate to different parts of a website.

```
function parseXML() {
      var tempXML = menuXML.firstChild.childNodes;
      for (i=0; i<tempXML.length; i++) {
            _root.createTextField("menu"+menuid, menuid, XOffset, currentY,
            ➥300, YOffset);
            _root["menu"+menuid].html = true;
            _root["menu"+menuid].htmlText = "<font color = '#99CCFF'><a
            ➥href=\"asfunction:gotoAndPlayLabel,
            ➥"+tempXML[i].nodeName+"\">"+ tempXML[i].nodeName+"</a></font>";
            currentY += YOffset;
            menuid++;
            for (j=0; j<tempXML[i].childNodes.length; j++) {
                  _root.createTextField("menu"+menuid, menuid, XOffset*2,
                  ➥currentY, 300, YOffset);
                  _root["menu"+menuid].html = true;
                              _root["menu"+menuid].htmlText = "<font color =
                                                            ➥'#FFFFFF'><a
              href=\"asfunction:gotoAndPlayLabel,"+tempXML[i].childNodes[j].
➥nodeName+"\">"+tempXML[i].childNodes[j].nodeName+"</a></font>";
                  currentY += YOffset;
                  menuid++;
            }
      }
}
```

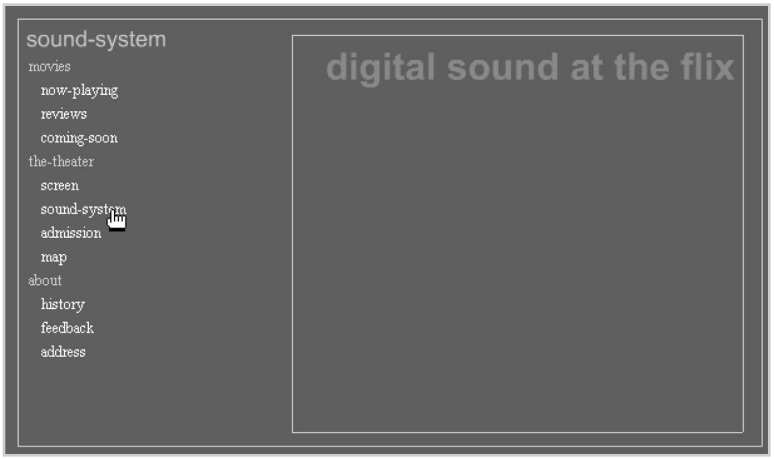
The code above is creating our simple text field buttons. At the beginning of the `parseXMLdoc` function, we create a `for`-loop that loops through the main menu buttons in our XML document (`movies`, `the-theater`, and `about`). It creates a text field for each button and assigns it a unique name and level. They are set to HTML, and a link pointing to a frame label in our movie is created for each button. The line `currentY += YOffset` increments each button so they are placed vertically on the stage. Then, `menuid++` increases the counter variable which is used to keep track of each used level.

The next `for`-loop does exactly the same task as the previous one, but it displays and places the sub-menu links on the stage. These links are indented slightly further from their parent nodes.

The final piece of code creates a function called `gotoAndPlayLabel` that is used in the `asfunction` to move the playhead to the specified frame label. Each frame label is the same as our XML data, so we only need to find the node name.

```
function gotoAndPlayLabel(myLabel) {
     gotoAndPlay(myLabel);
}
```

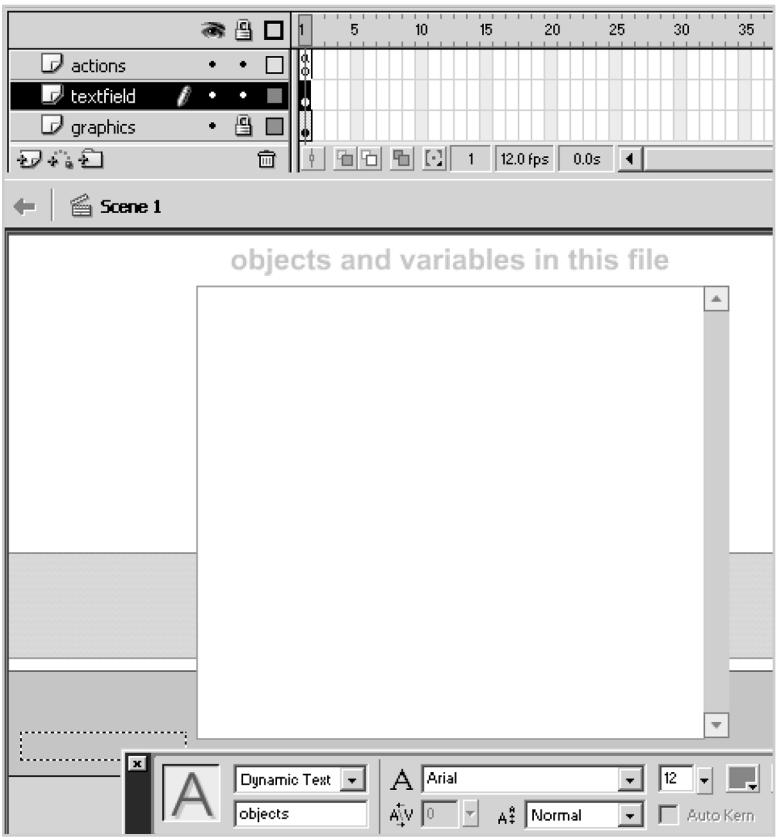
And that's all there is to it! The final result shows a navigational menu consisting of hyperlinked text pulled out of our XML file that can be pressed to navigate the website:



Parsing a file into objects and variables

It's possible to use ActionScript to process an XML document, and parse it into objects and variables that we can use in Flash. In the following example, we'll use data from the `contacts.xml` file (constructed in **Chapter 18**) – we'll take our client listings and separate the nodes into objects and variables.

Open up the FLA file called `clients.fla` from the CD and save it in the same directory as the `contacts.xml` file. As usual in these simple-but-effective demonstrations, we have several layers, and a couple of dynamic text fields on the stage (with instance names `objects` and `error`):



Let's now step through the code and explain what is happening along the way. Look at first part of the ActionScript on frame 1:

```
myXML = new XML();
myXML.ignoreWhite = true;
myXML.onLoad = function(success) {
      if (success) {
            parseXMLdoc();
      } else {
            _root.error.text = "error loading file";
      }
};
myXML.load("contacts.xml");
```

This part of our code should be familiar to you by now; we're simply loading our XML content, `contacts.xml`, into a new XML object called `myXML`. If our file does not load correctly, an error message is displayed in the `error` text field.

If our file loads without error, we will move on to the `parseXMLdoc` function, which follows:

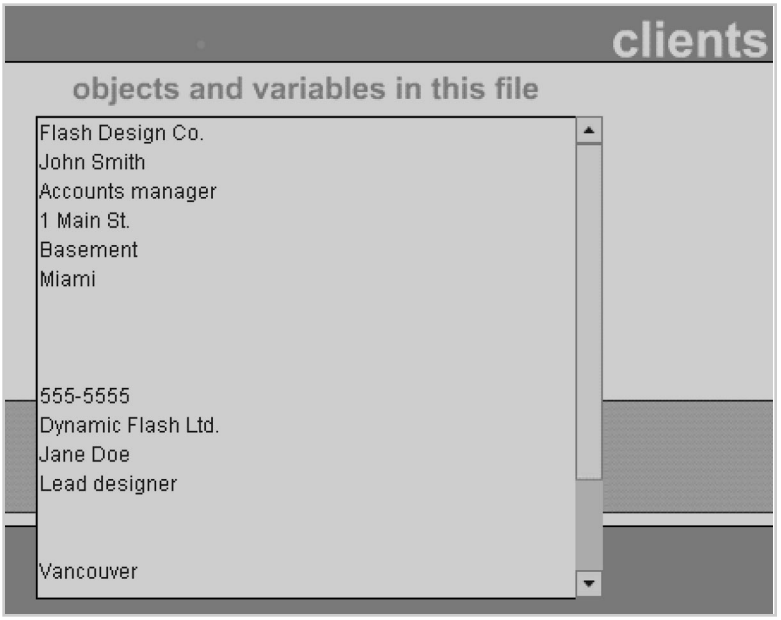
```
myXML.load("contacts.xml");
function parseXMLdoc() {
      var XMLdoc = myXML.firstChild.childNodes;
      newDoc = new Array();
      for (i=0; i<XMLdoc.length; i++) {
            newDoc[i] = new Object();
            newDoc[i]['number'] = XMLdoc[i].attributes.number;
            for (j=0; j<XMLdoc[i].childNodes.length; j++) {
                  newDoc[i][XMLDoc[i].childNodes[j].nodeName] =
ÂXMLDoc[i].childNodes[j].firstChild.nodeValue;
            }
      }
      writeXML();
}
```

This function creates an array of objects from the processed XML data. We begin by creating a new array called `newDoc`, and then we loop through each of the child nodes of the root element `clients`. The `client-ID` attribute number is then added to the array object. Then we loop through the child nodes of each client, and for each element a new property is added to the object. We do this so it is easy to reference the information in our `writeXML` function.

The `writeXML` function is the final piece of code to consider:

```
function writeXML() {
      var output = "";
      for (i=0; i<newDoc.length; i++) {
            output += newDoc[i]['business-name']+"\n";
            output += newDoc[i]['contact-name']+"\n";
            output += newDoc[i]['contact-title']+"\n";
            output += newDoc[i]['address']+"\n";
            output += newDoc[i]['address2']+"\n";
            output += newDoc[i]['city']+"\n";
            output += newDoc[i]['state']+"\n";
            output += newDoc[i]['country']+"\n";
            output += newDoc[i]['postal-code']+"\n";
            output += newDoc[i]['phone']+"\n";
      }
      _root.objects.text = output;
}
```

In the `writeXML` function we are outputting the properties from the `newDoc` array into the dynamic text field called `objects`. We do this by looping over our array and, for each element, we display the object properties of the current client. The ultimate result of this code is a movie that cleanly outputs the text data from our XML document to the text field:



Hopefully you are now more comfortable with some of the common coding practices when working with XML files. You will find that there are several different ways to parse XML files. Since XML is a very open markup language, it usually requires different parsing methods depending on the code you are provided with. In the previous examples, we parsed several different formats. This should give you an idea of how to handle different XML structures when you need to process them in Flash. In the next section, we will actually create our XML data in Flash itself, and then interact with a server using the `send` and `sendAndLoad` methods.

Creating new XML data

In previous examples we've been working with static XML files, meaning the files were created and not changed during runtime. With Flash MX it's possible to *create* XML data using ActionScript methods within the `XML` object. To do so, we have the `appendChild()`, `createElement()`, and `createTextNode()` methods at our disposal. The latter two of these methods are known as *constructor* methods and will define a new element or a new text node, while `appendChild()` simply adds a node to a specified XML object.

Adding nodes and data using constructors

The `XML.createElement()` and `XML.createTextNode()` methods will both *prepare* the node by setting up tag names and textual content, and then the `myXML.appendChild()` method can be used to *add* it as a child node of and XML object. We'll see how this is done in this section.

An element is created and appended to an XML object like this:

```
myElement = myXML.createElement("myname");
myXML.appendChild(myElement);
```

Here we create a new element called `myname`, and then we add this element to the `XML` object `myXML`. Now your XML data would look like the following:

```
<myname/>
```

You would create a text node in almost the same way. For example, if you wanted to add a `textNode` to the element that we just created, you could use the following code:

```
myText = myXML.createTextNode("bob");
myXML.childNodes[0].appendChild(myText);
```

Now if you trace the `myXML` object, you'd find the following data in the Output window:

```
<myname>bob</myname>
```

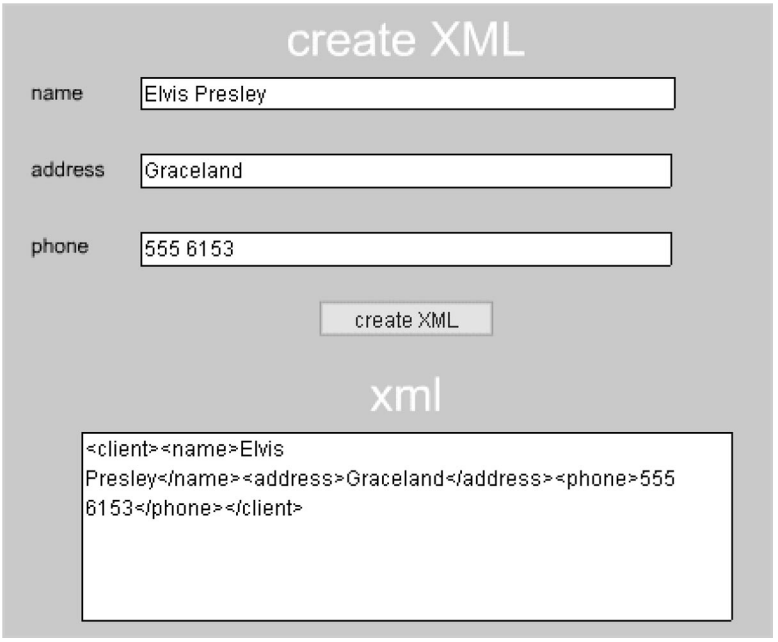
As you can see, you can create both element nodes (type `1`) and text nodes (type `3`). You can create attributes too, but the technique is slightly different – you take an existing element, such as our earlier example, and use the following code:

```
myXML.childNodes[0].attributes["id"] = "superhero";
```

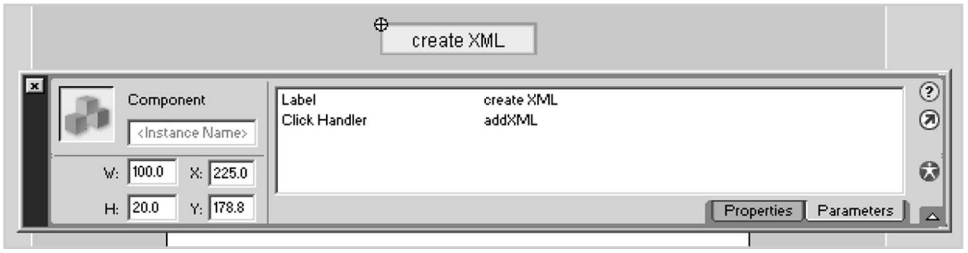
Now our XML structure would look like this:

```
<myname id="superhero">bob</myname>
```

It's not difficult to get started in creating new XML data. Take a look at `create_xml.fla` on the CD – this is an example of how to create XML based on user input. Elements are created using ActionScript, but the end user will input the text node data within the XML by adding data into input text fields on the screen. When a PushButton component is pressed, the XML is generated in a text field below. Test this FLA right now to get a feel for the kind of functionality it offers:



Now return to the authoring environment and look at the create XML PushButton component. A click handler called addXML is attached to this button so that the addXML function (described below) is called when the button is pressed:



New XML data is created from the text that is entered the text input fields on the stage. This is then passed to the myXML text field as new XML data in the myNewXML object.

Let's take a look at the addXML function now - you'll find its definition on frame 1 of the main timeline:

```
function addXML() {
      myNewXML = new XML();
      // Create a client element.
      newElement = myNewXML.createElement("client");
      myNewXML.appendChild(newElement);
      // Create the name element.
      newElement = myNewXML.createElement("name");
```

```
        myNewXML.childNodes[0].appendChild(newElement);
        // User will add a name.
        newText = myNewXML.createTextNode(_root.name01.text);
        myNewXML.childNodes[0].childNodes[0].appendChild(newText);
        // Create an address.
        newElement = myNewXML.createElement("address");
        myNewXML.childNodes[0].appendChild(newElement);
        // User will enter their address.
        newText = myNewXML.createTextNode(_root.address.text);
        myNewXML.childNodes[0].childNodes[1].appendChild(newText);
        // Create phone element.
        newElement = myNewXML.createElement("phone");
        myNewXML.childNodes[0].appendChild(newElement);
        // User enters phone number.
        newText = myNewXML.createTextNode(_root.phone.text);
        myNewXML.childNodes[0].childNodes[2].appendChild(newText);
        // Add the text to the myXML text box.
        _root.myXML.text = myNewXML;
}
```

A new XML object called `myNewXML` is created. We add new elements or text nodes to this XML object using the `createElement()` or `createTextNode()` methods. However, when using either of these methods, we must also use the `appendChild()` method to actually add the node to our object. In this example we are naming the elements ourselves (for example, `client`, `name`, or `address`). However, instead of adding content to the text nodes, we are adding the text entered into the text fields. We'll come across these methods again in the next example.

Integration of server-side technology

As we saw earlier in this chapter, when loading XML files from a remote server, server-side scripts are needed to work around security issues. Server-side scripts and databases are also commonly used when you want to add information to an XML document. For example, you may want users to register with your site – registration information could be added to a database using XML. In such a scenario, the XML data needs to be created on-the-fly, and then understood by the server-side language and database before it is added. Then, as long as information is properly formatted when it is returned to Flash, it can be understood and integrated back into your Flash site.

The `XML.send()` method is used to convert an XML source to a string, which is then sent to an application or a script at a specified URL location. Then, the application or script will process this information and is able to respond to it. This response is sent to a browser window and usually takes the form of a pop-up window. The response can also be sent to Flash via the `sendAndLoad()` method – this is discussed towards the end of this chapter.

Sending XML data to the server or a database

Integrating a database and middleware with our Flash MX designs might be used for any number of purposes: displaying an online guest book, remembering high scores details in a Flash game, storing the results of a survey, and so on. Let's look at a general way of using the `XML.send()` method to send XML data to a server.

This first snippet of code represents some ActionScript from a FLA file:

```
/* echo.fla */
myXML = new XML("<?xml version=\"1.0\" ?><message text=\"Hello world.\"
/>");
myXML.send("echo.cfm", "_blank");
```

This next code segment is from a ColdFusion MX `.cfm` template:

```
<!--- echo.cfm --->
i received a packet from flash!<br>
<cfoutput>#XMLFormat(getHTTPRequestData().content)#</cfoutput>
```

So, the FLA contains XML data in the `myXML` object. This data packet is sent to `echo.cfm`, which is opened in a new target window. The `echo.cfm` file then displays the XML packet that was sent by Flash. As usual, this can also be accomplished using several different server-side languages (PHP, ASP, and so on).

Setting up a guest book using XML.send()

Now let's look at a more complicated example of sending information to a database. This example will take text which is typed in input text fields on the stage, and send it all to a server. This is similar to our earlier example on using constructors to add nodes, except this time the new XML object will be sent to a server and entered into a database.

For the purposes of this example, we'll be using ColdFusion MX to process the data, which is then added to a Microsoft Access database - this database is an easy solution for smaller scale web sites. You can find an example of a database made using Access on the CD, `guestbook.mdb`, and we'll use this file in the example. If you are interested, there are many online tutorials about working with Microsoft Access and setting up data sources.

Open `guestbook.fla` from the CD - it's set up much like all the other examples in this chapter. What we're most interested in is the code for the guest book:

```
_root.submitbutton.onRelease = function() {
      myXML = new XML();
      myXML.xmlDecl = "<?xml version=\"1.0\" encoding=\"iso-8859-1\"?>";
      myXML.appendChild(myXML.createElement("feedback"));
      myXML.firstChild.appendChild(myXML.createElement("name"));

myXML.firstChild.childNodes[0].appendChild(myXML.createTextNode(_rooXt.name.
```

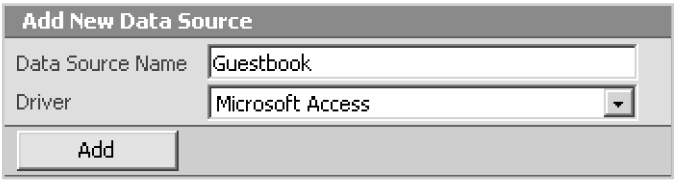
```
text));
      myXML.firstChild.appendChild(myXML.createElement("comments"));

myXML.firstChild.childNodes[1].appendChild(myXML.createTextNode(_rooXt.com-
ments.text));
      myXML.send("guestbook.cfm", "_blank");
};
```

This example creates a skeleton framework of XML data: a root node with two child nodes. When a user enters data into the text input fields in the Flash movie, this data is added to the XML object, `myXML`. Then, when the `submitbutton` instance is released, this function is called. Note the use of the `XML.xmlDecl` property - this is used to set and return details of a document's XML declaration.

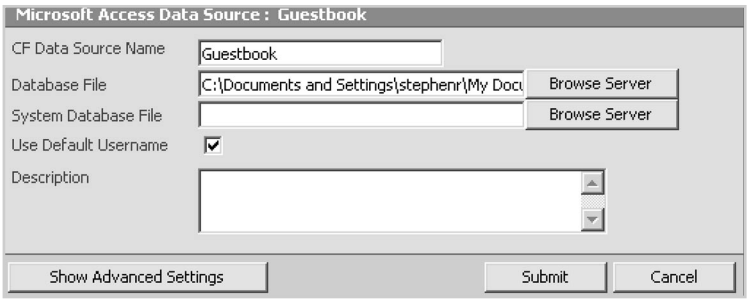
So, when you press the send button, XML elements are created based on the user's input and the element names we give the nodes. This is the same as what we did in the earlier example when we created new XML data. The `myXML` object is sent to `guestbook.cfm`, using the `XML.send()` method, and targets a new window.

Now let's look at the server-side script that we need for our guest book - open up `guestbook.cfm` in your favorite text editor. In the first block of code, the `guestbook.cfm` file checks whether or not XML data has been received. If data is received, it parses the XML and inserts it into a database, which is `guestbook.mdb`. The target HTML window then displays this new entry, along with other recent entries. If data is not received, it simply grabs the most recent entries and displays them in the target window in a very simple HTML table.

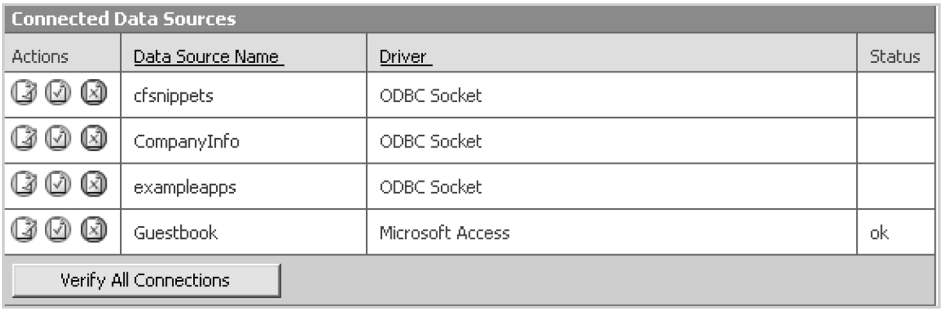
First of all, we'll need to set up a data source in ColdFusion MX called `Guestbook`, so let's go through this process step by step. In your web browser window, go to http://localhost:8500/CFIDE/administrator/index.cfm (assuming that your ColdFusion installation is using the internal web server running on port 8500). Then, log in and go to Data & Services>Data Sources. Under Add New Data Source, type in a name for your data source, and select a ***driver*** type - we're using a data source called `Guestbook` and the Microsoft Access driver. Press the Add button to register this name:



We are now able to modify the details of our data source, and we need to specify the database file that this data source name will be linked to. So, press the Browse Sever button and you will be shown a directory tree where you can browse through your local computer for the required Access database file. Press Apply once you have located and selected `guestbook.mdb`. You'll then be taken back to the original screen. Make sure that Use Default Username is selected, and enter a description if you want (this step is not mandatory). Finally, press Submit when you are finished:

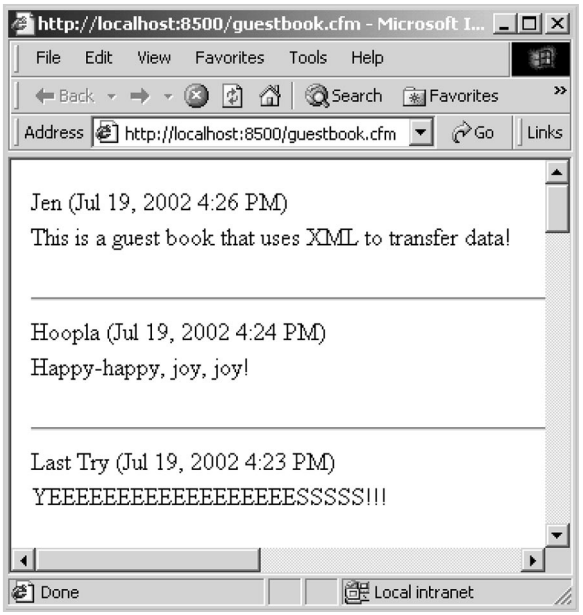


Now you are taken back to the list of data sources and you should see data source updated successfully with an ok in the Status column beside the name of the newly created data source:



You're now ready to upload all the files, `guestbook.swf`, `guestbook.html`, and `guestbook.cfm` to the ColdFusion server (drop these files in the ***wwwroot*** directory of your ColdFusion MX installation). Finally, browse to the guest book to test it:

Press the send button to process the text field data, and the text entered, along with previous entries stared in the database, will pop up a new window:



This example could easily be extended to display the guest book in a different Flash interface, or use a different server-side language you may be more comfortable with.

Sending and receiving data using sendAndLoad()

Sending data to a server, and receiving XML formatted data back is a useful way to communicate with site visitors. You can take data from the user, send it to a database, and send a response back to the user – perhaps regarding the status (success/failure) of the transmission. `XML.sendAndLoad()` is used to *serialize* XML into a source of XML code, and then send it to an application or script residing on a server. The XML is then processed and sent back to the Flash movie into a target XML object specified in the send method. This response is parsed and processed by Flash, which can then put the new data into your interface.

Let's look at some code that uses the `sendAndLoad()` method to send XML data to a server where it will be processed and sent back to Flash. The following ActionScript uses a server-side script in ColdFusion MX, `redirect.cfm`, to grab an XML feed from a remote server; a variable is set to the URL of the feed we want to grab. A new `XML` object is created (`myXML`) with XML data set to the URL we want to grab. A second empty XML object is created (`newXML`), to hold the XML data returned to the movie by the ColdFusion server – `newXML` is our XML feed.

Take a look at the code in `sendandload.fla` from the CD:

```
var XmlUrl =
```

```
"http://www.macromedia.com/desdev/resources/macromedia_resources.xml";
myXML = new XML("<?xml version=\"1.0\" ?><redirect url='"+XmlUrl+"' />");
newXML = new XML();
newXML.ignoreWhite = true;
myXML.sendAndLoad("redirect.cfm", newXML);
newXML.onLoad = function(success) {
      if (success) {
            _root.output.text = newXML.toString();
      } else {
            _root.output.text = "transmission error";
      }
};
```

Now we should examine the CFML code – the code below represents the entirety of our `redirect.cfm` document. This is used to parse the incoming XML object, `myXML`, and extract the URL attribute within it:

```
<cfset newXML = XMLParse(getHTTPRequestData().content)>
<cflocation url="#newXML.redirect.XmlAttributes.url#">
```

This URL is then returned to Flash and is loaded into the new `XML` object that we've created, `newXML`. The `onLoad` event handler is used to check that it is loaded without any problems. This is useful for implementing dynamic newsfeeds, or when you want to access many different newsfeeds, but do not want to set up individual ColdFusion templates for each one. If you were using the first newsfeed example, `newsfeed.fla`, to access several different newsfeeds, you would need to create individual templates for each URL. Using the `sendAndLoad()` method, you can get around this potential hassle.

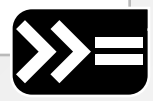
Using XML in your movies can be an effective way of handling dynamic content. As this chapter has shown, there are several different ways of processing XML data and incorporating it into the interface of a Flash movie. You might want to use XML for syndication, or for handling many interfaces that need to display the same content. You might instead use it as a way to transmit data between Flash and a server or database. XML can also be used to handle dynamic data that has a definite structure and might change often, such as a list of clients or an MP3 play list.

Understanding how to use XML within your Flash interfaces largely involves recognizing how the tag-based markup structure translates into a dot notation hierarchy. Once you can reference the nodes, it's simply a matter of processing them using ActionScript to extract the data in a usable form. Just as there are many ways of using the data, there are also several ways to process it, as we've seen from the examples in this chapter.

20 XML Sockets

Objectives

- Understanding the basics of socket communication:
  - Learning about connections and protocols
  - Bandwidth concerns
- Socket server software
- The Flash MX `XMLSocket` object:
  - The `XMLSocket` constructor
  - `XMLSocket` event handlers
  - Sending and receiving data
- Connecting to socket server
- Handling connections:
  - Parsing the XML data
  - Creating a simple chat room



Introduction

Now that we have covered how to load, send, and process XML data in our Flash movie (see **Chapters 18** and **19**), we're ready to tackle a slightly more complicated topic: **socket servers**. A socket server is a piece of software that is placed on a server and used to connect many users together in almost real-time Flash environments. Instead of ActionScript, these servers are written in languages such as Java, C++, Visual Basic, PHP, Perl, or Python. A socket server is used to send data back and forth between Flash movies and a server in a continuous, cyclical basis. The communication is based on events: when new information arrives at the server, connected users will *hear* it, and a response is triggered. The data is sent, and will be processed in a Flash movie.

Socket servers are used to create multi-user applications in Flash such as chat rooms, online meetings, instant messengers, and multi-player games. Socket servers vary greatly in complexity and what they are capable of. Some will simply *echo* data passed to them back to the connected users, while others are robust enough to integrate with back-end database technology. A socket server is great for reducing the overhead incurred when you use HTTP (Hypertext Transfer Protocol) transfer methods. The connection between a Flash movie and the socket server remains open, instead of being repeatedly opened and closed as it is when using HTTP to transfer data. The data (which can be formatted as XML, but does not need to be) is **persistent**, and remains so until the server or a client closes the connection.

In this chapter we'll delve into the world of socket communication. We won't actually *create* a socket server here, this would be beyond the scope of both this chapter and many Flash designers (and developers!), but we will, however, create a chat room using ActionScript that can use a number of existing (downloadable) socket servers to create a multi-user environment.

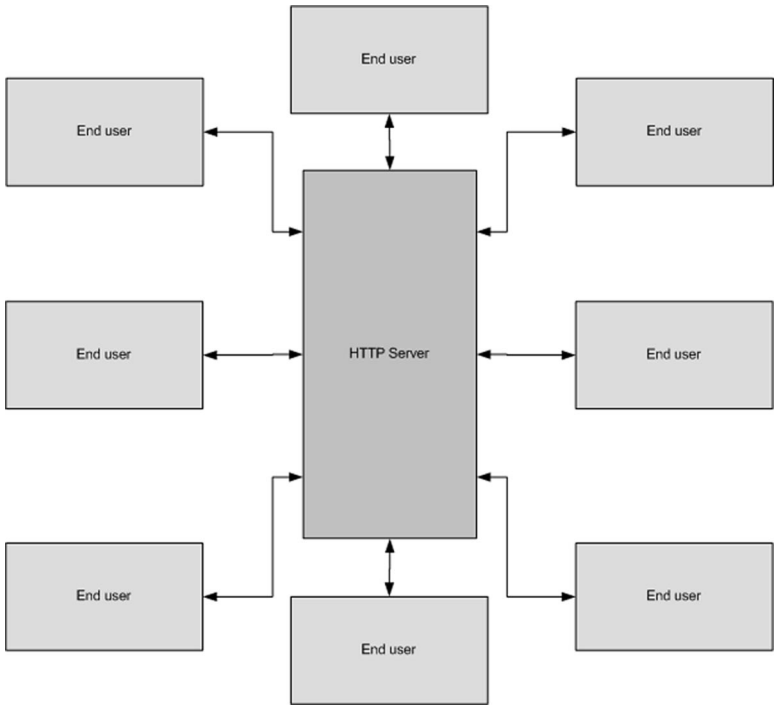
Understanding the basics of socket communication

Socket servers are a complex but powerful method of data transmission and communication and play a large part in online communication between computers. A socket itself is the end point of a network connection between two machines (for instance, a client and a host). Therefore, there will be a socket on each end of this kind of communication. A connection is opened by a user, whose computer connects to the socket at a particular server, and this connection remains open and active, with constant data flow between these two locations. The connection is then closed by either of the parties involved.

Sockets are commonly used when small and continuous data communication is required. You're probably already familiar with the kinds of technology that use sockets. ICQ, and other popular instant messengers and peer-to-peer networks all use sockets for data transfer. Sockets are used all the time for communication on the Internet because it is a much better way of transferring data in (almost) real-time. Sockets are used to connect people on the Internet for many different purposes. They facilitate connecting many people to a single host in a continuous data feed between the two computers. The host will connect all of the users listening to the port together. Sockets are also commonly used in gaming communities. A quick response time is required to achieve the effect of instantaneous interaction in multi-user games. Instead of a few seconds of latency, socket servers reduce this to mere milliseconds!

An XML socket also makes a continuous feed of data between two computers possible. This is very different from the kind of XML data transfer that we studied in the previous chapter, when a connection is closed after the XML has been loaded from, or sent to, the server. In those kinds of transfer methods a connection is opened, data is requested and sent, and then the connection is closed. With socket servers, instead of pushing data to the user in one direction and then closing the connection, they send a *stream* of data continuously over a connection that remains open. As soon as there is new data on the server, everyone connected to the socket server will have their Flash movies updated. Because of this, a much more dynamic and useful system can be developed.

When thinking of the kind of systems we might create with our socket servers, we need to look at a couple of different models. Systems can be made either in a peer-to-peer architecture, or what is commonly called a *hub-and-spoke* architecture. You might be familiar with peer-to-peer data transfer systems already, but Flash does not allow these kind of connections because of security risks – we'll discuss these later in the chapter. The following schematic diagram shows the kind of architecture that socket servers create when used with Flash:

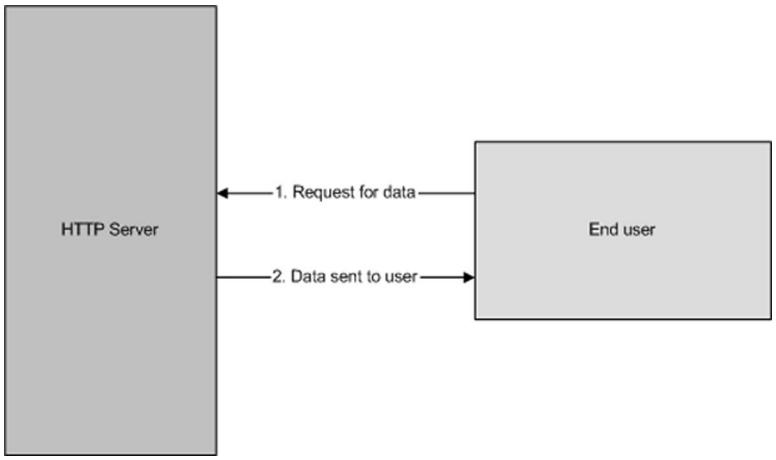


As you can see, in this diagram many users are connected to one central host in continuous data feeds. Each arrow represents a continuous flow of data. In the following section, we will look at the types of connections that are used, and what kind of terminology is involved with socket servers for Flash.

Socket servers vs. HTTP connections

When working with socket server technology, it is very important to understand Internet connections and how they work. As we have discovered, socket servers are a somewhat unique way of transmitting data. This can potentially benefit your Flash movies greatly, but even more so if you fully understand the power of sockets in relation to the other methods you might use.

There are many different ways of transferring data between computers. You're probably most familiar with the HTTP method of data transfer. When using HTTP transfer, a user will request data, which will then be sent to the user from the server. This way of transferring information is best suited for large and infrequent amounts of data, as there is a good deal of overhead in each transfer:



After these steps are completed, the data transmission is closed. Therefore, there are essentially four steps involved:

- The connection is opened
- The data is requested
- The data is sent
- The connection is then closed

In **Chapter 17** we learned that we can transfer data using either the `POST` or `GET` parameter in ActionScript when using HTTP to sending information. Data is sent within the HTTP header if you decide to send it using `POST`, which is useful for sending long strings of data, while data sent using `GET` are actually sent *within* the URL. For example, if you wanted to send a name and a color using `GET`, it would be sent in the URL as a name/value pair, like this:

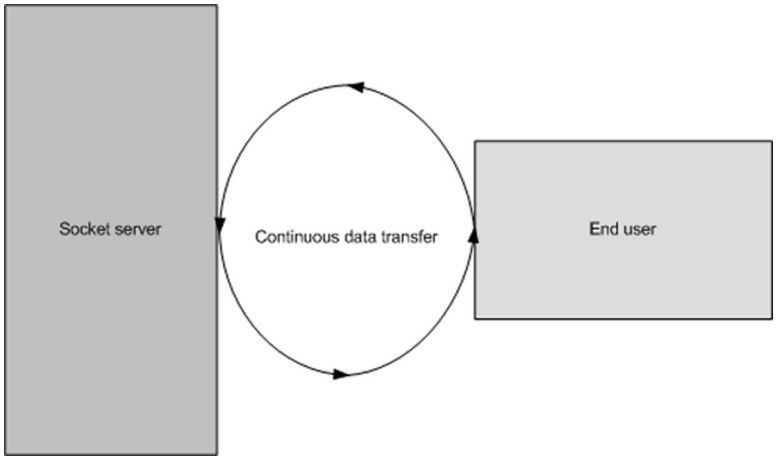
```
http://www.yourdomain.com/mydata.asp?name=bob&color=blue
```

This is usually considered most useful for sending small amounts of data (up to 256 characters). Using HTTP is great for transferring infrequent, or larger, packages of data. However, using HTTP for frequent data transfers creates a lot of needless overhead, which is where socket servers come into play.

Although HTTP transfer was used in the previous chapter on XML data transfer, and is the most common kind of transfer you will find on the Internet, if you're looking for a solution for fast, real-time communication, you need to look at socket servers.

Advantages of using a socket server

Sockets are great for short, continuous amounts of data that needs to be presented in an almost real-time environment. As we've established, using a socket server means that you do not have the *latency* (the time it takes to send the data) that HTTP inevitably creates. Our graphical representation of the communication flow now looks like this:



Protocols

A **protocol** is a set of rules used for communications, such as between computers on the Internet (this series of protocols is referred to as TCP/IP – Transmission Control Protocol/Internet Protocol). Protocols set standards for the industry to adhere to. For instance, HTTP gives us a standard set of rules for transferring files such as text, audio, images, and other forms of multimedia.

You may also recognize the term SSL, which refers to Secure Sockets Layer. This is a standard protocol for sending secure data, represented by HTTPS when you connect with it, which you've probably noticed in your browser's address bar when making an online payment or entering personal information – this is a secure socket layer over HTTP. Flash socket servers transfer XML data (although, it does not *have* to be formatted as XML) across a dual TCP/IP streamed connection. This connection is not limited in the number of messages that are sent back and forth.

Ports and IP addresses

When connecting your movie to a server by way of sockets, you will need some understanding of ports and IP addresses before getting started. Generally speaking, ports are a place on a computer where a connection is made. The kind of port we are concerned with is commonly referred to as a *logical connection*.

Numbered ports help a computer determine what kind of connection is being made. They range from 0 to 65536, and numbers below 1024 are reserved for certain kinds of *services*. These services include the web (HTTP at port 80), and FTP (default set to 21). Of course, we can change the specific port numbers because you do not have to use these particular services. New services can be bound to new port numbers – for example, if you install a new file transfer program, you might set the port it uses to 5800, and so on. Note that once a port is used, another service cannot share it, but remote users can still listen to it.

Socket servers communicate through a port of your choosing. However, the port number must be greater than 1024 for security reasons. Any port number between 1024 and 65536 is fine as long as it is not being used by another service. Firewalls usually block ports above 1024, which means if you are considering hosting your applications from behind a firewall, you may need to configure the firewall to allow the data to pass through (*punch a hole* through it!). We'll cover installing third party socket servers shortly.

Limiting your data overhead

HTTP transfers, as we previously discussed, are notorious for latency and bogging down a transfer with a lot of data overhead. Luckily, XML data transfer using a socket server solves the problem of a bulky overhead of headers and the like. You are able to streamline the communication between clients and the host. Reducing the transfer time is the key in these communications, which are meant to replace the lengthy transfer time in HTTP communication. Only one connection and disconnection ever occurs, and a continuous stream of information flows between the two locations. When there is new data on a server, that data is sent to all of the connected users, negating the need for data requests and connections constantly being opened and closed.

When it comes to transferring data, you will also want to try to keep the amount of data being sent as small as possible. A good way of limiting your data overhead is by using as few headers and tags as you can. You can also use attributes instead of text nodes, which can help even more. You can create an attribute by manually adding it to an existing element.

```
theXML.childNodes[0].attributes["cat-name"] = "whiskers";
```

Using the fewest number of text nodes as possible will cut down on your overall data overhead, and will help speed up the transfers between the movie and the socket server. Another notable point is that you do not need to transfer XML formatted data either, which can also help reduce the overhead in data transfer. For example, you can send data as a string, or format it as an array of numbers. Using these methods reduces the amount of data sent to the server, and in turn received by others connected to the socket server. This will be of great benefit to your users when there is a lot of traffic that the server has

to handle. This will reduce the both the stress on the servers and latency, thus creating an environment much closer to real-time communications.

Socket server software

In this section we are going to take a look at some of the popular socket server software available to us. There is a huge variety of servers available to use with your Flash interfaces, and these sockets differ in a number of characteristics, such as what they are intended to be used for, their cost (are they free?!), their capability, and so on. Some servers are more robust than others, with the ability to handle a great number of connections; others are meant for development, and therefore have fewer connections. This section will cover what you need to *know* to determine what you *need* to get the job done.

So you want to build a socket server...

Socket servers are complicated creatures, and building one from scratch is certainly not for the faint hearted. Coding a socket server capable of handling a multi-user environment would not only mean the need to write a large amount of code, but also you'd need to build a server using a language supporting socket communication like Java, C++, and so on. Some socket servers can integrate with databases too. In fact, any language accessing sockets – if it can open ports and recognize a null termination character – can be used to build a server for Flash.

If you are interested in writing a socket server, and you have not directly worked with any of the relevant languages, it's advisable to choose a language similar to one you already know. For example, the Java programming language is not too distant from ActionScript. As it happens, it's also an excellent language for creating socket servers, with significant support for sockets. You might also choose to try building a short and simple trial socket server using a language like PHP. Perhaps you're already familiar with the language? In which case, you could create a very simple *echo server* that can bounce the information you send to the server back to your movie. Building a simple echo server could very well be within your programming range (it may seem somewhat advanced, but is more than likely quite attainable with your now-solid understanding of ActionScript!).

That said, there seems to be a greater number of free socket servers available online than tutorials on how to build them for Flash! If you are thinking of building a server, it is at least advisable to download some of the open source servers, to dissect and study the code within them. As noted earlier, it's beyond the scope of this book to create a socket server – we're going to focus on what you need to know in order to get Flash working with a socket server, which you can download for free from many online sources (links in the next section). It's easy to get it up and running locally on your machine for development purposes.

As a Flash designer and ActionScript coder, you will no doubt want to know about the differences and abilities of socket servers. There are many options available to you, both for commercial and personal endeavors. As we've emphasized previously, socket servers range in how many connections they can accept and the kind of commands you can send to them. They also differ in the sorts of tasks they support, and the commands you can send to them. In the following section, we will list a few of the

servers available for download so you can search around and find a suitable piece of software to integrate with your movies.

Commercial and open-source socket servers

It's very fortunate that the Flash community is so active and supportive – because of this, there are a number of socket servers available online for download. Generally speaking, if you need a socket server that has many features and supports many concurrent connections, you'll probably have to purchase it. If you're interested, you'll certainly want to carefully read the capabilities and the licenses for the socket server on each web site, and choose one that best suits your projects and can handle the number of connections necessary. They usually (but not always!) come with instructions as to how to set up the server, and sometimes instructions on commands that can be used to track connections, and so on.

The following socket servers are available for immediate download:

- *CommServer* from Colin Moock and Derek Clayton – www.moock.org/unity. This includes a link to the free Java CommServer socket server. *CommServer* is a helpful tool for learning about socket servers or using with small web sites. Also available from this web page are trial, non-profit/academic, and commercial versions of the robust and professional *Unity* socket server.

- *AquaServer* from Branden Hall at Figleaf Software – www.figleaf.com/development/flash5.

- *nexusJava* is a free, open-source socket server also written using Java for Flash multi-user experiments – www.flashnexus.com.

- *Swocket* is another open source socket server, this time written in Python – swocket.sourceforge.net. This one works on multiple platforms, and the page includes demos using the socket server, and the developer notes that it might soon be updated for Flash MX.

- *FlashNow* – www.nowcentral.com. Here you'll find both free and commercial versions of the *FlashNow* software. This server socket supports multiple platforms, and is easy to install on Windows machines using an `.exe` file for installation and XML configuration. It also supports a number of special features, and there is extensive documentation on the web site.

These are only a few of the commercial and open-source socket server options available. If you are having difficulties with your server, try using an `.exe` or `.bat` file to start up or configure your server. Something else you will want to watch out for is that some of the socket servers will require certain data to be sent back to the socket server, or will send XML data to your Flash movie. For example, when connections are made to the *CommServer*, XML data is sent to your movie. You may have to work these differences into your Flash movies, but these setups are usually in place to help benefit your applications.

Also, there are mailing lists and message boards dedicated to these (and other) socket servers. It's a good idea to find an active community discussing the servers, particularly the one you are using, because some of the code involved with this kind of communication can get quite complex. Checking out pre-built socket servers is also a great way of learning a bit more about how they are constructed. Studying these

files, and getting involved with the communities around them, may even assist you in customizing and expanding your own multi-user environments and applications.

Hosting your socket servers

One of the earliest challenges you may encounter when working with socket servers and movies that you want to develop for an audience is finding a place to host your socket server applications. There are security issues associated with installing socket servers on a server, which means it can be difficult to find a hosting business that will allow the (custom) software. Also, you probably won't be able to access the servers themselves, meaning it would be tricky to control or start up the socket server. You also need to find a server that never shuts down, and a company willing to custom-configure their server for you. Finding a hosting company which allows the security risk, gives you this kind of access, and has adequate hardware to run a fast socket server can be a difficult, and often pricey, task.

This is why many developers decide to use their own personal servers (or even their development computer) for running a socket server on. This means you can have instant access to control and test your applications at will, and you can try out as many of the servers as you want. One of the most important things to remember when working with socket servers is that for the same security reasons that were explored in **Chapter 19**, you will need to be running your movie and socket server in the same domain (`www.mydomain.com`) or sub-domain (`flash.mydomain.com`) as one another. In the following section, we'll discuss how to set up your socket server, and start it up from a Command Prompt. However, it should be noted that it's not likely you'll have the same reliability and bandwidth that a proper hosting company would be able to provide. Reliability, bandwidth, and hardware should all be taken into consideration when deciding how you are going to host your applications, and what kind of conditions you will need for them to run smoothly.

Testing your movies when working with a socket server

You can test your movies locally by placing the socket server files within your web root directory. Obviously, to do so you will need to have a web server installed, such as IIS (Internet Information Services), PWS (Personal Web Server), Apache, and so on. If you only want to test your application locally with the socket server (just on your own PC as opposed to a local network or live), you don't *need* to place the files in the web root to test your movies with a socket server. You can test them on a completely local basis by running the socket server from any location on your hard drive.

Not all socket servers require you to start them up using a Command Prompt. Some come with `.bat` or `.exe` files that will start and configure them automatically for you. This is usually the easiest way to set up a socket server. Usually, they will include `README` files that will provide exact instructions on how to configure the servers (for example, set which port you want the socket to listen to).

We'll be using Colin Moock's CommServer for the demonstrations in this chapter - this is a socket server implemented in Java, so if you want to use it you'll need to download the Java runtime environment (if you haven't already got it) from http://java.sun.com/j2se/.

In the following example, although we can use any web server really, we'll be using the server built into ColdFusion MX (see **Chapter 19** for trial download details). Using this server we simply drop files for upload into the wwwroot ***directory***. This demonstration will be useful for when we need to start up the socket server in our chat room application.

Using the Command prompt to start a socket server

If you download a Java socket server, like *AquaServer*, which includes Java Class files (those with the `.class` extension), you'll probably need to start the server up via your Command Prompt window. Let's briefly outline the necessary steps using DOS from a command line to start your socket server.

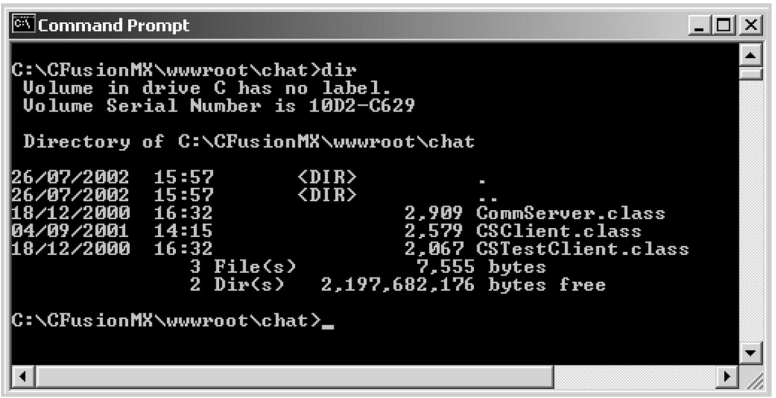
In Windows, you can open up the DOS Command Prompt from Start>Programs>Accessories, or just type CMD in the Start>Run window (if this doesn't work, try command if you are running Windows 98). This will open up a fresh Command Prompt window:

Now you'll need to find the directory that you're working in – in our case we've placed all of the *CommServer* Java Class files in their own folder called chat, and just dropped this in the ColdFusion MX wwwroot directory. So, type in cd to change directory, followed by the full path and name of this directory:

```
cd C:\CFusionMX\wwwroot\chat
```

This path should look exactly the same as it does in your Windows Explorer address bar. Hit ENTER and you'll see that the current directory changes to the one you've specified:

To double check that you've got the right directory, you can type in `dir` to display its contents:



Here we see all the downloaded files associated with the *CommServer* socket sitting happily in the `chat` folder, as expected. The final step is to start the socket server – this will of course vary slightly depending on which server you are using, but the typical method for a Java socket is to use the following line:

```
java ServerName 12345
```

`ServerName` is the name of your server, and `12345` is the port number that you want the socket server to listen to for data. We use the `java` command to run the Java class files. Some socket servers might have more text to type following this command. The port number can be anything above `1024`. Be sure to remember your port number for later in this chapter, because you will need to insert it into your Flash movie to enable the communication.

So, for *CommServer*, we type in the relevant commands, press ENTER, and get a message returned indicating that the socket server has started:

If it throws an error, check the capitalization of the server name (it's case-sensitive). If this is correct and there is still an error, try changing your port number – if the port you choose is actively being used, the server will not run.

You can leave this DOS window open, and you may get messages to the window depending on the activity of your testing. Some servers accept command lines, as stated in the window above, which you could use to change the configuration or see connections and so on. All you need to do now in your Flash movie is use `localhost` as your host parameter, and the port number you specified above (`12345` in our case),

in the `XMLSocket.connect()` method. As you can see, it's pretty easy to get your socket server up, running, and ready for connections!

Possible alternatives

In the last couple of sections we've looked at how socket servers work in comparison to the standard data transfer methods that we used in earlier chapters. As you already know, you can use `LoadVars`, `XML.load()`, `XML.send()`, and `XML.sendAndLoad()` to transfer data between end users and a server. All of these methods use the slower HTTP data transfer and in this chapter we're discussing how you can also use socket servers to improve the efficiency of this communication. But what if you either want an alternative, or maybe you find socket servers may not be the best solution for your movie? There are other ways you can use to transfer data between a server and a Flash front-end.

The Macromedia Flash Communication Server MX (www.macromedia.com/software/flashcom/) is a brand new option for transferring data, which allows you similar functionality to a socket server. Using this server, you can connect many users to a host computer and transfer text, video, and audio. It does not, however, replace socket servers outright because professional editions of the Communication Server MX are not as economically competitice as the average commercial socket server. Other alternatives include using Macromedia Director and the Multiuser Server, (www.macromedia.com/support/director/multiuser.html) which has supported multi-user environments and gaming for some time now.

Despite the fact that there are other options available, this doesn't necessarily make the use of socket servers redundant. Indeed, some computers cannot handle the overhead of these other technologies. For example, devices like Pocket PCs and PDAs can't handle some of the added overhead of something like the Flash Communication Server MX (at this time!). In fact, at the time of writing, the Flash Player 5 was only recently released for handheld devices. It might be some time until such devices are powerful enough to fit in well with the new servers that are available, and offer support for video compression and an updated Player. For now, socket server communication is still a great solution for this kind of set up.

That's enough background to socket server technologies – it's time to learn how we can work with them using our ActionScript.

Using the XMLSocket object

The `XMLSocket` object contains methods and event handlers that can be used with functions to process information and create dynamic multi-user environments. The handlers will automatically send information to the movie when data arrives on the server, or when a connection is made or lost.

Creating an XMLSocket object

Creating an `XMLSocket` object will probably be familiar to you after using the `XML` object constructor in the previous chapter – using the `XMLSocket` constructor is very similar. In this case, we are creating an object to facilitate communication between a socket in the client's computer and one in the server. The domain name (or IP address) for this connection and the listening port are then defined in the `XMLSocket.connect()` method.

As with the XML object, you create a new `XMLSocket` object in the following way:

```
mySocket = new XMLSocket();
```

...where `mySocket` is the name you are giving your new `XMLSocket` object. This simply creates the object with which you connect to a server using the appropriate method.

Connecting to an XML socket stream

After creating the new `XMLSocket` object, you need to connect it to a stream. This is done using the `XMLSocket.connect()` method, as follows:

```
mySocket.connect(host, port);
```

Here, `host` is a domain name or an IP address (or a loop back like `localhost` if we're just testing locally). Keep in mind that you must have your socket server and Flash movie within the same domain or sub-domain for security reasons and, as we have already mentioned in the earlier section on ports, the port number specified in this method must be greater than 1024.

If testing the movie locally, as we will in the example featured shortly, we will either have to use the loop back address of `127.0.0.1` or `localhost` instead of an external IP address or domain name. This points us to the local server on our own machine:

```
mySocket.connect("localhost", 55667);
```

Then, after testing, if you had your socket server installed on a web server with a domain, you would write the method like this:

```
mySocket.connect("www.mydomain.com", 55667);
```

We must use this method to connect with the server because the server cannot start this communication itself. This is also true when a connection is lost between the movie and the server - the Flash movie must reconnect to the server using this method.

Security issues with connecting to a server

As you already know, the `XMLSocket` object establishes an open connection to a server. Because of this, `XMLSocket` lends itself to security risks and a number of measures are in place to combat malicious use of the connection. This is why you can only connect to a port greater than 1024, and within the same domain or sub-domain using the connect method. Remember, ports below 1024 are used for some of the most common web services, such as FTP and the web. Not allowing socket servers to use these ports limits potential abuse when using this object and Flash.

Using the XMLSocket.send() method

The `XMLSocket.send()` method is used to transmit data to the server that is specified in `XMLSocket.connect()`. The method sends data via the `XMLSocket` object created in your movie, and is written in the following way:

```
myXMLSocket.send(object);
```

The `object` parameter in this case is either an XML object or some other data to be sent. If you create a new `XML` object populated with some data, then you can enter the `XML` object name as the `object` parameter here and the XML data is converted to a string that is sent to the server. The data string sent to the server is terminated by a 0 byte, but does not have to be formatted as XML.

You can also format XML data or a string directly within the object parameter, for example:

```
myXMLSocket.send("<shouts>hello there world!</shouts>");
```

It is important to remember a couple of things when using the `send()` method. Firstly, note that nothing is returned to the movie about whether or not the data was successfully sent or received by the server. Also, if the connection has failed in the first place, the `send()` will also fail (pretty obvious!). This is when event handlers and callback functions are necessary – we'll discuss these soon.

Disconnecting from an XML stream

An `XMLSocket` object will remain connected to a Flash movie, continually streaming data between it, and will stay open, at least until one of two things happens: the browser or player window is closed, or the server disconnects the transmission of data. A connection to the socket server stream can also be terminated by using the `close()` method, as is seen in the following line:

```
mySocket.close();
```

In this case, the connection opened by the `mySocket XMLSocket` object will be closed. As soon as this occurs, all subsequent attempts at data transfer will fail. The server cannot reinitiate contact with Flash – the Flash movie must always start communication with the server.

It should also be noted that the `onClose` event handler only concerns the server's end of the connection. If the connection terminates, then the callback function associated with the `onClose` handler is triggered. We will discuss this in the following section.

The XMLSocket event handlers

The event handlers for the `XMLSocket` object are triggered in Flash when the particular event in question occurs, and we can write functions based on these callbacks. These are very useful event handlers because data transfer will fail unless there is a connection between the socket server and Flash. Our functions can tell the movie what to do if and when the connection has died, or what to do when data or XML is

received, and so on. Therefore, we are able to monitor what happens (regarding errors and so forth) during the transmission and call functions based on this constant checking.

onConnect

Using the `onConnect` event handler is important because it can alert the user as to whether or not they are connected to the server. The following code is commonly used to inform the user of the status of their connection with the server (we'll also see similar code in action in the chat room example, later in this chapter):

```
theSocket = new XMLSocket();
theSocket.onConnect = connectionHandler;
function connectionHandler(theStatus) {
      if (theStatus) {
            _root.myAlert.text = "you are connected";
      } else {
            _root.myAlert.text = "you cannot connect";
      }
}
theSocket.connect("localhost", 12345);
```

This code will execute whether the connection succeeds or fails, and will display the status text in a text field called `myAlert`. The Boolean value of `true` or `false` is passed, depending on the outcome of the connection attempt. If the attempt is successful and a connection is made then the `true` value is passed; if not, then `false` is passed. The actual connection errors (timeout, network problems, and so on) are not specified with this callback handler.

onXML

The `onXML` event handler is executed when a string of data is received by Flash (it's terminated by the null character). Flash can then receive and parse the data when `onXML` is triggered, via the `XMLSocket` object initially specified. Using this handler we can call a function when the data is received. Usually the XML data is processed after this event triggers (i.e: data is received) – the data is processed (perhaps placed somewhere on the stage, and so on).

```
theSocket = new XMLSocket();
thesocket.connect("localhost", 12345);
theSocket.onXML = function(thedata) {
      _root.mytextfield.text += "data has arrived:::"+newline;
      _root.mytextfield.text += thedata+newline;
};
```

We'll cover this in more depth in the following section.

onData

The `onData` handler is invoked when new data is received from the server, similar to the `onXML` handler.

The `onData` event handler can be overridden with a custom function, in which case the `onXML` handler will not be executed. An example of this is as follows:

```
theSocket = new XMLSocket();
thesocket.connect("localhost", 12345);
thesocket.onData = function(thedata) {
      _root.mytextfield.text += "data arrived"+newline;
      _root.mytextfield.text += thedata+newline;
};
```

This will display the raw unprocessed data in the text field `mytextfield`.

This handler is different from `onXML` because it can be used to grab incoming strings, and use them in your movie *without* parsing the data. Handling data in this way can potentially decrease the number of tags needed, and improve your transfer times by reducing latency. It can therefore be used to optimize multi-user environments (particularly multi-user gaming environments). The `onData` handler is used in pretty much the same way as the `onXML` handler, only differing in the way each one handles the data received by the movie.

onClose

This event handler is triggered when the connection on the server side is lost, either by the server simply losing the connection, or terminating it on purpose. Your callback can be invoked as soon as the connection is lost from the server side. This handler is not invoked when the user purposefully closes the movie, or with the `XMLSocket.close()` method. You can send a message to the user when the server terminates the connection in the following way:

```
theSocket = new XMLSocket();
theSocket.onClose = function() {
      _root.mytext.text = "The server has lost the connection!";
};
```

You can then reconnect the user to the server, but it must be done *manually* by using the `XMLSocket.connect()` method. You could use a PushButton component with a click handler called `openConnection` and write a function on the main timeline as follows:

```
function openConnection() {
      theSocket.connect("localhost", 12345);
}
```

Otherwise, the end user would have to refresh the page containing the movie.

Handling incoming XML socket data

Incoming data is continuous when you have an XML socket. Luckily, we can use the `onXML` handler to alert and parse the incoming data to Flash. In the following code snippet example, we use the `onXML` event handler to call a function that processes the data coming from the server to our movie:

```
theSocket = new XMLSocket();
theSocket.onXML = XMLhandler;

function XMLhandler(thedata) {
      var data1 = thedata.firstChild.childNodes[0].nodeName;
      var data2 = thedata.firstChild.childNodes[0].childNodes[0].nodeValue;
      ➥_root.textField.text += (data1.toString()+data2.toString());
      // Can add other actions here if you need anything
      // to occur when data is received...
}
```

As you can see, in this code when the XML data arrives in our movie we process it to extract two individual pieces of data. Then, the processed data is then placed in a text field (called `textField`). You can handle other actions at this point – for instance, you might want a graphic to appear each time some data comes into the movie, your text to scroll, or perhaps a sound to play. These actions will fall within the `onXML` handler function.

Parsing incoming data

As you can see from the previous example, we process and extract information from incoming data in exactly the same way as was covered in **Chapter 19**. Parsing occurs when new data is received from the server, and is processed by the functions triggered by the `onData` or `onXML` event handlers. We'll see this occurring in the following example, where we will construct a very simple chat room that uses a socket server to transmit data between connected users.

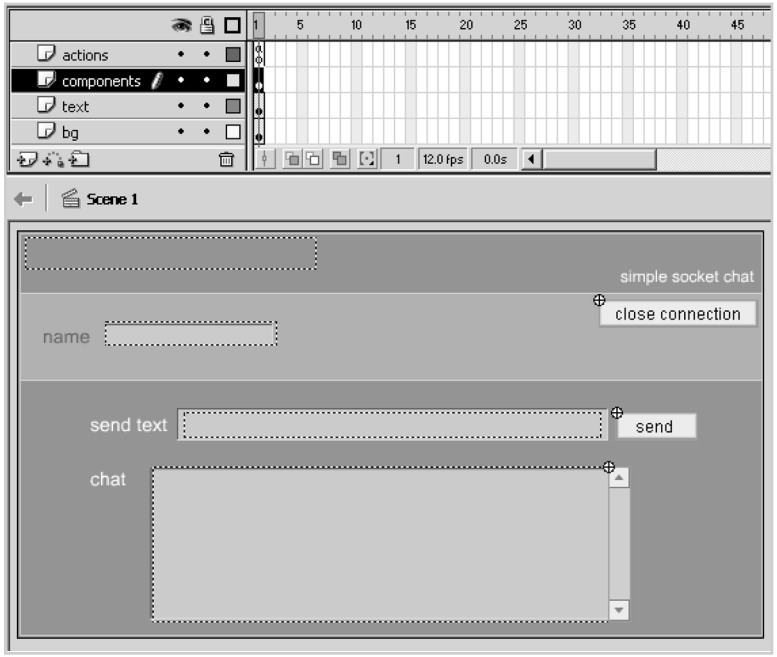
A simple socket-based chat room application

In this exercise we are going to go through the creation of an extremely simple socket server chat room. Although this application is stripped to its barest bones in functionality for the purpose of demonstration, it can readily be expanded upon with a login process, or more complicated measures to handle what happens when users are connected or disconnected from the sockets, and so on.

You can build and test the chat room on your local web server or hard drive, but you will need a socket server to be running on your computer. As we discussed earlier on in the chapter, there are many free socket servers you can download from a number of different places. If it's applicable to the server you are using, go back to the instructions on how to get your socket server fired up using the DOS Command Prompt, and start it up locally, on a port above `1024` (in our example we are using port `12345`). Keep this window open while testing the chat room application, and you can watch for messages sent to the Command Prompt window.

Although the following example will work without modification if you use Colin Moock's *CommServer* from www.moock.org/unity, you might have to make minor changes to the code of this example if you're working with a different socket server. Some sockets require you to send data to the server to initiate a connection, and others may send objects back to the chat app. These can obviously interfere with functionality. If you use this chat application with the *CommServer*, you will notice a colon appear in the chat window when a connection is made. This is because XML data is being sent back to the application when this occurs. Obviously, you could modify your application to account for this if you use this particular server for testing.

The first stage of the design of a chat room is setting up the interface. The chat room we will be creating is very simple, mainly relying on ActionScript and the transmission of data between the users connected to it. Therefore, we only need layers for graphics, text fields, components, and script. Open up `socketchat.fla` from the CD and take a look at the contents of the stage:



In the text layer we have four text fields on the stage:

- A *single line input* text field for users to enter a username, with an instance name of `username`.
- A *single line input* field for the user to enter their chat messages, with an instance name of `sendtext`.

- A large *multiline dynamic HTML* text field for the chat messages to be displayed in, with an instance name of chat (note that the Render text as HTML button has been selected). This text field has a ScrollBar component attached to it.

- A *single line dynamic* text field, with an instance name of connection.

You will also notice two PushButton instances on the stage: close connection (with a Click Handler of closeIt) and send (with a Click Handler of sendXML).

With the stage fully set up, we're ready to look at the fun stuff – the ActionScript. Go to frame 1 of the actions layer, and look at the first chunk of code:

```
//tab index
username.tabIndex = 1;
sendtext.tabIndex = 2;
sendbutton.tabIndex = 3;
//key object
pressEnter = new Object();
Key.addListener(pressEnter);
```

This sets the tab index of the instances on the stage so that when a user presses the TAB button to navigate through the text fields, the cursor will move from the username to the sendtext field, and then it will highlight sendbutton (which can be pressed to send the text, when the movie is within a browser). After this, we create a new object called pressEnter, and add a Key listener to it. This will help us when we add the code that allows the user to press ENTER from the sendtext field, which then sends the XML data to the server and displays the text in the chat field.

Following the above text, enter the following code into frame 1 of the actions layer:

```
//create socket object
theSocket = new XMLSocket();
theSocket.onConnect = connectHandler;
theSocket.onXML = XMLhandler;
theSocket.onClose = closeHandler;
//connect to a site and an empty port (above 1024).
//enter localhost or 127.0.0.1 for testing locally.
//for live server enter your domain www.yourdomain.com.
theSocket.connect("localhost", 12345);
```

In this important section of code, we are creating a new XMLSocket object called theSocket. This object will handle all of our data transmission with the socket server. Following this, we set names for the event handlers onConnect, onXML, and onClose. Later on in our code, we will create functions to be invoked when any of these events occur. Following this is the most important line of all – the theSocket.connect() method in which we declare localhost as our host and 12345 as our port number. If you have made your socket server listen to a different port, be sure to adjust this accordingly.

Now that we are connected to a server, we need to make sure this connection actually works. We use the following callback for the `onConnect` handler:

```
//displays text depending on user connecting to socket
function connectHandler(myStatus) {
      if (myStatus) {
            _root.connection.text = "you have made a connection";
      } else {
            _root.connection.text = "you cannot connect to the server";
      }
}
```

So, if a connection is not made (perhaps the socket server hasn't been started up), an error text message will be displayed in the `connection` text field at the top left of the stage. We use a simple `if..else` statement, since our `onConnect` method will return a Boolean value of `true` (connected) or `false` (connection failed). Using this handler to display messages is useful if you aren't sure whether your socket server is running properly.

Next we define our `sendXML()` function – this will create new XML data to be sent to the socket server when the send button is pressed. We create a new `XML` object (*not* a new `XMLSocket` object, note) called `myNewXML`, and create some new XML for it, based partly on what is entered in the `username` and `sendtext` text fields:

```
//the XML to send
function sendXML() {
      //create the XML. can also create a string and parse it into XML
      myNewXML = new XML();
      //create root node
      newElement = myNewXML.createElement("whole-message");
      myNewXML.appendChild(newElement);
      //create the user element
      newElement = myNewXML.createElement("user");
      myNewXML.childNodes[0].appendChild(newElement);
      //username text entered
      newText = myNewXML.createTextNode(_root.username.text);
      myNewXML.childNodes[0].childNodes[0].appendChild(newText);
      //create the chat element
      newElement = myNewXML.createElement("chat");
      myNewXML.childNodes[0].appendChild(newElement);
      //chat text entered
      newText = myNewXML.createTextNode(_root.sendtext.text);
      myNewXML.childNodes[0].childNodes[1].appendChild(newText);
      //send the text
      theSocket.send(myNewXML);
      _root.sendtext.text = "";
```

```
}
```

You probably recognize and hopefully immediately understand this part of the code; it's very similar to the example on creating XML data in the previous chapter. You may decide it is a better idea to use attributes instead of text nodes, but you need to be sure to `escape()` and `unescape()` the data within the attribute. You could create an attribute in the following way, which will reduce the overhead of the data sent to the server:

```
myNewXML.childNodes[0].attributes["theusername"] = _root.username.text;
```

This would create an attribute from the text entered into the `username` text box, and you would obviously have to restructure the processing of data received by the movie. You could also restructure the data to be used with the `onData` handler as well.

The new XML is created, based on the stage and the elements we specify when the `sendXML` function is called. This function is called when our `send` button is pressed. At the end of this function, the XML data is sent to the server using the `XMLSocket.send()` method. Then the `sendtext` text field is cleared, ready for the next string of data to send.

The next few lines of code will enhance the functionality of our chat room. Most chat rooms and some instant messengers allow you to press ENTER to send text. To add this intuitive feature, we will need to use the `Key` listener:

```
//so you can press enter to send text as well...
pressEnter.onKeyDown = function() {
      if (Key.getCode() == Key.ENTER && _root.sendtext.text != "") {
            sendXML();
      }
};
```

This code will call the `sendXML` function if the ENTER key is pressed when there is text in the `sendtext` text field.

Next we have the following function, `XMLhandler`:

```
//handles incoming XML data, and formats it
function XMLhandler(memessage) {
      var user = memessage.firstChild.childNodes[0].childNodes[0].nodeValue;
      var chattext=
      ➥memessage.firstChild.childNodes[0].nextSibling.childNodes[0]. nodeValue;
      _root.chat.htmlText += ("<b>"+user.toString()+":</b>"+"
      ➥"+chattext.toString()+"\n");
      //handles the scrolling chat window when XML changes
      _root.chat.scroll = _root.chat.maxscroll;
}
```

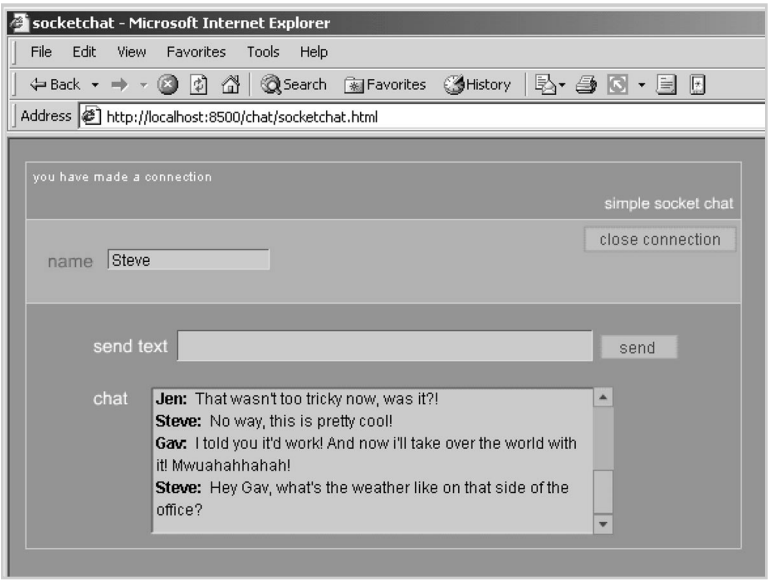
The XMLhandler function first formats the XML data returned to the chat message box. The function listens for incoming XML data (with the onXML method). It will boldface the username portion of the XML data, then insert a colon, and finally append the body of the message followed by a new line (\n or newline). Following our text formatting is what causes our ScrollBar and chat text field to scroll to the bottom each time data is sent from the server. This also means that we can still scroll up using the cursor buttons. Each time new text is sent to the movie, the scroll bar will return to the bottom of the chat field again.

The final pieces of code are as follows:

```
//close functions
function closeHandler() {
      _root.chat.htmlText = "Connection lost.";
}
function closeIt() {
      _root.chat.htmlText = "server connection is closed";
      _root.connection.text = "disconnected from server";
      theSocket.close();
}
```

These two functions handle closing the connection manually, and displaying a message if the connection is closed from the server side. If you remember, our click handler for the close connection button was called closeIt. This invokes the close() method, and disconnects the movie from the server. If the connection is lost or dropped from the socket server end, the onClose handler will trigger closeHandler function. Then the Connection lost string is displayed in the chat window.

Remember, this example can be found on the CD; it's called socketchat.fla. As we've emphasized, you'll need to download one of the many socket servers available (some options were listed earlier in this chapter) and start it up before testing your movie. If you have the socket server running on a live server, you can upload the SWF into the same sub-domain and chat with anyone who goes to the URL containing your movie. If not, you can simulate multiple connections if you open this chat in several browsers at once, and can then be easily entertained for hours by chatting to yourself!



Additionally, take a look in the Command Prompt window of your socket server after opening/exiting the chat room – it'll let you know exactly what's been going on:

In this example we've learned that using socket servers on the Flash end of things is not very different than just using the `XML` object. There is no need to be intimidated when using these servers and, once you are comfortable with handling connections, there is a wide range of possibilities when using this technology. It's extremely easy to set up a very simple communication application as soon as you have your server up and ready to go. Socket servers are readily available for download, and the smaller or non-profit servers are usually free for your use. Not only this, but more and more socket servers are being developed and popping up around the Flash community. Accordingly, it's well worth your time getting acquainted with socket servers – they can greatly advance the interactive nature of your web interfaces for little or no cost. This chapter will hopefully help you on your way towards truly interactive and stimulating multi-user Flash movies!

Index

The index is arranged hierarchically, in alphabetical order, with symbols preceding the letter A. Many second-level entries also occur as first-level entries. This is to ensure that users will find the information they require however they choose to search for it.