Introduction to XML in Flash

by senocular

XML Basics

Before using XML in Flash, lets first learn what XML is, at least at its basic form. Entire books have been written on XML and the use of XML in Flash. Here, I intend to keep things simple enough to get you started yet attempt to provide you with enough information to at least make you feel comfortable when working with XML in Flash.

So, XML; what is it and why does it exist? Basically, XML (eXtensible Markup Language) is a *tagged markup* format that was created to serve as an easily usable, readable method to format information within a basic text file. One of its primary advantages is that it allows you to store this information in a hierarchical manner. This means you can store information within other chunks of information much like a folder hierarchy on your computer. In fact, you can think of XML just as that, as a text write-out of a hierarchical file system which stores textual information. XML is NOT really a script and does NOT really "do" anything; it's basically just a fancy text file (though based on what you are using to *interpret* the XML it could potentially be script-like). Where you may normally use a text file, you can instead use XML where XML will provide you with greater structure and clarity for storing the content within.

For more information on XML beyond laymen's terms, see:
http://www.xml.com/
http://www.w3.org/XML/

Basic XML Structure

In looking at an example of XML, you'll notice it looks a lot like HTML. In a way, HTML is a type of XML (though HTML was around before XML - both are based off of SGML). Their structure is, in many, cases the same. Here's a simple example of an XML file:

```
<letter>
    <to>Sandy</to>
    <from>Peter</from>
    <body>I love you!</body>
</letter>
```

If you're familiar with HTML, you can see the similarities. Like HTML, just from looking at the above text, it's fairly easy to see what information the file holds and how it's laid out. Just like the italic tags <i></i> may encase italic text in HTML, the ambiguous <to></to> tags above surround Sandy to indicate that Sandy is to whom this particular letter (<letter></letter>) is to be sent.

The information enclosed in tags in XML (and HTML) allow for a hierarchical structure of information. For instance, above, we knew that Sandy is getting the letter because "Sandy" was placed within the opening and closing to tags and the to tags were in between the opening and closing letter tags.

In the file-folder analogy, the letter tag represents a folder which contains 3 other folders to, from and body, all which are representative of the contents or aspects of a letter. Each one of those folders contains the given text respective to their types (i.e. to, from or body) which could be considered the files of those folders.

The contents of an XML document, tags and information within those tags, are considered to be *nodes*. These nodes are the basic building blocks XML defining all that it is. There are two types of nodes at play in the letter example: *elements* and *text nodes*. Letter, to, from and body are elements; Sandy, Peter and I love you! are text nodes.

Element Nodes
Elements are the actual XML tags (or folders). Each letter, to, from and body are all XML elements.

```
<letter>
    <to>Sandy</to>
    <from>Peter</from>
    <body>I love you!</body>
</letter>
```

The letter element represents the XML *document root element.* There should only be one root element tag for every XML document (though, actually, Flash specifically could care less). All other elements should be placed within this root element to define the entire contents of the document.

When one node is in another node, it is said to be a *child* of that node. A node with children is considered to be the *parent* of its children - almost like we're dealing with a family tree. The letter element has 3 children, to, from and body. The parent node of each of those three is letter.

All element nodes in XML must have a beginning and an end - an opening and closing tag. Elements with no content within an opening or closing tag, or having no children, like <to></to> for example, are called "empty elements" and can be alternatively written as <ELEMENTNAME /> or in the case of to, <to />, where you have one tag basically closing itself. Having just <to> without any closing would be incorrect. The following is the same example from before, just without any text within the to element making it empty.

```
<letter>
    <to />
    <from>Peter</from>
    <body>I love you!</body>
</letter>
```

Apparently Peter just got dumped.

When naming elements, there are certain rules you should follow. Element names must contain valid only characters. These include letters, numbers, underscores (_), hyphens (-), periods (.), and colons (:). Colons are typically only used as a namespace delimiter, a topic which won't be covered here. Though the characters just mentioned are all valid, element names can only start with letters or underscores (or colons). On top of that, you should not begin element names with "xml" as it is also, technically, restricted. Naming is also case sensitive. So if you start a non-empty element with <tag> it must end in </tag> to properly close it and not, for instance, </Tag>. Typically XML tags are written in lowercase, and should you place HTML within XML, the HTML is often uppercase.

Unlike folders in a directory on your computer, elements can exist within the same parent element and share the same name. It is quite ok to have more than one to element in the letter example. Each element still remains unique in its own right even though it's name is shared among one or others within the parent (or document for that matter). This makes it much easier to send a letter to more than one person!

```
<letter>
    <to>Lisa</to>
    <to>Jenna</to>
    <to>Karen</to>
    <from>Peter</from>
    <body>I love you!</body>
</letter>
```

Text Nodes
Text nodes are what make up the bulk of your text, especially body text, in XML. Unlike elements, they themselves do not need an opening or closing tag, mainly because, well, they simply have no tags at all. They're just text. The previous example had three text nodes, Sandy, Peter, and I love you!

```
<letter>
    <to>Sandy</to>
    <from>Peter</from>
    <body>I love you!</body>
</letter>
```

Though all single lined above, text nodes can span multiple lines and contain most types of text. However, because of the very form of XML and how it's structured using tags (with elements), there are certain characters in a text node that cannot be used or it will break the document formatting. If this happens, the document is said not to be *well formed*.

These certain characters will have to be replaced by *character entity references*. Strictly speaking only "<" and "&" are restricted (in their typical use and that can depend on the environment you're working in), though there are 5 predefined entity references for XML that you should be aware of. They are as follows:

Character	Replace with
<	&lt;
>	&gt;
&	&amp;
'	&apos;
"	&quot;

Example:

```
<incorrect>
It is clearly stated that variable A < variable B & variable A > variable C
</incorrect>
<acceptable>
It is clearly stated that variable A &lt; variable B &amp; variable A > variable C
</acceptable>
```

Notice that it was not *completely* necessary to replace the ">" character, only < and &, though it may be in your best interest to do so for good practice and consistency and, you know, just because it's the right thing to do.

```
<best>
```

```
It is clearly stated that variable A &lt; variable B &amp; variable A &gt; variable C
</best>
```

One thing that is easily forgotten is that text nodes are nodes - they are distinctly separate from element nodes and not a part of them. In fact, they are children of element nodes. Sandy is a child node of the to element and the to element is Sandy's parent. When getting into extracting content from XML, this is easily overlooked as, often, some elements' sole purpose is to contain a text node.

Attributes

XML can have another type of data structure called attributes. Attributes are additional (generally simple) values which can be included in element nodes. They consist of a name (whose naming restrictions pretty much follow that of elements) and a quoted value included within the opening tag of an element. Unlike, elements, however, they have no child nodes or hierarchies of their own.

So far we haven't seen any attributes as they are optional parameters. Though, should you so desire, you can have as many or few as you desire for each element in your XML. Here is a previous example that now uses attributes:

```
<letter to="Sandy" from="Peter">
    I love you!
</letter>
```

This takes two text nodes which were before in elements and places them as attribute values in the letter element. Since the rest of the letter is pretty much assumed to be its body, we can leave the "I love you!" text node directly within the letter element as a child. Such text, since it has the potential to be long and fairly complicated, would probably not be proper as an attribute anyway.

It is important to understand that attributes aren't meant to contain complex information. They must exist on a single line, in a single element and have a value defined in quotes. Since attribute values are defined in quotes they themselves cannot contain quotes - at least not the same kind of quotes used in their definition. Attributes can be defined with either single or double quotes so you can have single quotes in an attribute defined with double and vise versa, but you can't have double within an attribute defined with double. Attempting to escape quotes with a backslash (\") will not work either. And this not only applies to quotes, but also other characters you may wish to include such as tab (\t) or new line (\n). The attribute will treat each as plain text.

Ok, you got me! The above is not *entirely* accurate and all is not *entirely* lost if such aspects are required in your attributes. Like elements, attributes can use the aforementioned character references as well as others in the ISO 8859-1 (Latin-1) character set. This also includes the original 7-BIT ASCII standard. So yes, quotes can be added and yes, tabs and new line characters too. Want to know what they are?

Character	**Replace with**
tab (\t)	&#09;
line feed (\n)	&#10;
carriage return (\r)	&#13;

Find out more here.

Regardless, my position still remains; attributes should be kept simple. They are used to provide properties for elements, not relay your life story. The simpler the better as too many attributes or too much information within attributes can create unreadable XML, and that's not what we're after here, is it?

Ok, now take a look at this XML:

```
<a href="http://www.senocular.com">senocular.com</a>
```

Oh wait, its an anchor tag in HTML. No, its XML I say! You got yourself the element a with an attribute href whose value is "http://www.senocular.com" and a text node child of the a element whose value is senocular.com. Notice the similarities. XML isn't so bad. If you've been working with HTML then you've practically been working in XML. Just remember to keep those line break tags closed.

```
<br />
```

(Now you're making XHTML!)

CDATA
There is one more type of XML node that should be mentioned. That element is a CDATA node. A CDATA (standing for Character DATA) node is a special node that looks somewhat like a cross between an element and a text node, though, it really exists as more of a resilient text node. It has a type of opening and closing tag but is meant to store body text inside the two. This text can be multi-lined and contain just about anything including those terror causing characters which would otherwise have to be replaced with character references in text nodes. The following XML has a CDATA section:

```
<text>
<![CDATA[ Here is a bunch of text.
And here is a little more: <, >, &, " and ' with no problem.
Expecting a third line? Neither was I!]]>
</text>
```

The opening tag of a CDATA section is <![CDATA[. Anything, and I mean anything, between that and the closing]]> is part of that section. The only exception to this is another]]>. You can't necessarily have a]]> in a CDATA node because it would effectively close that node. Everything else, no matter what it is, between <![CDATA[and]]> is acceptable. The idea is that anything within a CDATA section is ignored by the XML parser, almost like a comment in the XML (XML comments are like HTML comments <!-- *comment* -->), but something from which you can still extract information from.

CDATA is often used for body text because of this (again, the only caveat here is that you cannot have]]> in a CDATA section - this also prevents you from nesting CDATA sections - so you may need to watch for this if you are allowing others to dictate CDATA content). It's also especially useful in keeping markup like HTML or even other XML where replacing illegal characters (namely < and >) with their respective entity references causes an ugly mess.

The thing is that Flash doesn't *directly* provide much support for using them. There's no problem with using them, it's just that Flash will just treat them as normal text nodes. Any illegal characters passed into Flash via a CDATA section will be replaced with character references and the CDATA node itself will be replaced with a basic text node as a result. Don't worry, however, since should you need either versions (original or with character references) you will be able to get both.

Additional XML Content
As you work with XML, especially that which is not of your own creation (if you're just starting out), you may notice other bits of information outside the XML document root element. Such information can exist before (in the XML Prolog) and/or after (the XML Epilog) the document root.

Most XML documents come with *declarations* that provide more information about the XML. These include XML declarations as well as doctype declarations.

The optional XML declaration specifies certain properties about the XML page itself such as version and character encoding used. This is placed at the very beginning of an XML document before anything else, including comments and even white space. Most XML documents should at least be declared with one of these that at least specifies XML version, though Flash typically doesn't care whether an XML file it receives has one or not. An example of a XML declaration would be:

```
<?xml version="1.0"?>
```

Doctype declarations give information about how an XML document is defined. They usually consist of the name of the document root of the XML file, its availability and a url to a DTD (Document Type Definition) file that defines the file's *vocabulary*. An example would be

```
<!DOCTYPE root PUBLIC "http://www.kirupa.com/root.dtd">
```

where root is the the name of the document root element and root.dtd is the

DTDs and *Schemas*, a successor to the DTD (usually saved as .xsd), are files or instructions within the doctype that outline how an XML behaves and what it means. They define the vocabulary that an XML file (or any of many that use it) must adhere to to become valid.

Processing Instructions can also be used in XML. These can go in the prolog, epilog or within the XML body within the document root. These provide special instructions to the application processing the XML and are typically defined with <? and >. For example, a PHP tag in HTML is a processing instruction and can be placed anywhere within an HTML (or PHP, rather) document.

```
<HTML>
    <HEAD>
          <TITLE>My Page</TITLE>
    </HEAD>
    <BODY>
      <?php
          print "Welcome to my page!";
      ?>
    </BODY>
</HTML>
```

Most of these are of no importance when it comes to Flash, however. The Flash interpreter for XML will, for the most part, ignore these though not without giving you access to their raw values (via some XML properties to be covered shortly). So unless otherwise needed, you wouldn't have to spend too much time worrying about what they all mean. Plus, as an *introduction* to XML in Flash, they are out of the scope of what is to be covered here in the first place.

Evaluating An Example of XML
Let's look at a small example of XML and check it for errors... if there are any.

```
<thursday>
```

```
    <Meeting time="10:00">
        This meeting was boring. >_< I made paper airplanes with a memo I received earlier.
    </Meeting>
    <lunch time="1:30">
        Peter came over to talk to me. He seems depressed.
        <served maincourse="Steak"
            side1="Apple"
            side2="Mustard"
            drink="Tang" />
        <spent mealcard="0" cash="12.00" />
        Peter came over to talk to me. He seems depressed.
    </lunch>
    <Meeting time="3:00" time="5:00">
        These meetings were even more boring than the first.
        I slept straight through both.
    </meeting>
    <18holes_of_golf />
</thursday>
```

From looking at the root element, you can tell that this XML document contains information about thursday. The first child element of thursday is a Meeting element that contains an attribute time. This attribute's value is10:00. So far so good.

Within that Meeting element is a text node. It has text referencing the meeting, how boring it was, and has one of those silly ASCII faces in there. Uh oh, that marks a problem - the use of < and > within a text node. They need to be replaced with their respective character references.

```
<Meeting time="10:00">
        This meeting was boring. &gt;_&lt; I made paper airplanes with a memo I received earlier.
</Meeting>
```

Other than that, this Meeting element looks fine.

Next up is the lunch element. Notice that it contains both a text *and* element nodes. No problem there. A text node can coexist with other elements within any given parent node. The only problem you might have is with the second text node (below spent). Why is it exactly the same as the first? Well, there's nothing technically wrong with it. I think this person was just very adamant about that particular thought. One thing to get from this is that a text node starts at an element tag (an opening or closing) and ends where the next one picks up (which too can be an opening or closing tag). So, despite the similarities of the two text nodes in the Meeting node here, they are distinctly separate because of the elements between them.

You may have noticed that the spent element within lunch is a little funky looking. There's nothing wrong with that either. White space within elements is negligible so long as you don't divide attribute values with new lines.

Following the lunch element you can see another Meeting element. It has the same name as another element, but there's nothing wrong with that. It also has two attributes, time and... time. Red flag! That won't work. Attributes can't have the same name as other attributes in the same element. Better change one of those. Lets use time and duration instead of two time elements.

```
<Meeting time="3:00" duration="2:00">
```

Within meeting is a text node. It is a single text node consisting of two divided lines. It's content is innocent, so there it's not a problem. The following tag, however, the closing tag of the Meeting element, is. What's wrong? Its name is all lowercase, whereas the opening tag started with a capital M. Better fix that bad boy and make sure the closing tag also has a capital M.

```
<Meeting time="3:00" time="5:00">
        These meetings were even more boring than the first.
        I slept straight through both.
</Meeting>
```

Last but not least it looks like thursday was wrapped up with a little golf (this must have been the agenda for the boss?). Again, small naming problem. Elements cant have names starting with numbers. We'll have to switch that around a little bit. How about putting the 18 in an attribute? Never know when you'll play less than 18.

```
<golf holes="18" />
```

Though we've gotten rid of the underscores (_) they were not a problem.

That brings us back to the closing tag of the single root element which, itself, is just fine. With our corrections, the XML now looks like the following.

```
<thursday>
    <Meeting time="10:00">
        This meeting was boring. &gt;_&lt; I made paper airplanes with a memo I received earlier.
    </Meeting>
    <lunch time="1:30">
        Peter came over to talk to me. He seems depressed.
        <served maincourse="Steak"
            side1="Apple"
            side2="Mustard"
            drink="Tang" />
        <spent mealcard="0" cash="12.00" />
        Peter came over to talk to me. He seems depressed.
    </lunch>
    <Meeting time="3:00" duration="2:00">
        These meetings were even more boring than the first.
        I slept straight through both.
    </Meeting>
    <golf holes="18" />
</thursday>
```

Looks good! If you were able to catch all those mistakes then you should be on your way to being able to make your own valid XML!

XML With Respect to Flash

So what's the big deal about XML in Flash? Nothing really. Its just XML... and Flash. There is no huge bond or special relation that the two entities have with each other. Flash does, however, provide you with a simple means of collecting data from an XML source and interpreting it in a way that can be understood in ActionScript. Simply, you've got yourself a way of loading information into Flash. It's easy to get carried away with the current buzz surrounding XML at this point as its becoming so widely used for so many things, but really, when it comes down to it, and in the simplest terms, all you get in the end is just some formatted text that you can load into Flash during runtime. Not to say that's a bad thing. No, not at all. And XML is good, but it can easily be touted to be more than it really is. So before you get too excited, realize that this is not rocket science; its just text and Flash loading it in. That's nothing new, right?

XML vs Variable Strings

Ok, so XML *can* get much more complicated than *just* text, but for now, that's probably all you need to think of it as being. As such, the big comparison you can make with XML is to urlencoded text which you can load into Flash to obtain external text variables; this via the LoadVars object (or using loadVariables). The XML object in Flash, the ActionScript construct used to load and manage XML, is very similar to the LoadVars object used for urlencoded text variables. Lets take a look at an example of urlencoded text so that we can make a quick comparison between it and XML:

```
subject=Flash in the News&date=07.25.04&body=Macromedia just announced
their support for flashers around the world, those with overcoats and
those without.&
```

The above shows a urlencoded string that would be contained in a text file - a text file which can be loaded into and utilized by Flash. When this happens (via a LoadVars instance), you get 3 variables: subject, date, and body with their respectively assigned strings as values. It's pretty simple and straight forward. You have your basic variable string defined with equal signs (=) and separated with ampersands (&). Not too hard to follow, right? A respective XML file may look like following:

```
<news>
    <entry date="07.25.04">
        <subject>Flash in the News</subject>
        <body>
          Macromedia just announced their support for flashers around the world; those with overcoats and
          those without.
        </body>
    </entry>
</news>
```

The above consists of the same content as the variable string before. Only this time, as XML, it's just laid out a little differently and is organized using tags (XML elements). This, as you can probably tell (color coding aside), makes it a little easier to follow compared to the variable string. It may be a little longer, and thereby requiring slightly more hard drive space to store and making for slightly more bandwidth consumption, but the added clarity is definitely an improvement over the previous format.

When this is brought into Flash, however, you get more than just three simple variables like those which you get with the variable string. Instead, you have a complex data object (contained within an instance of the XML object) harboring not only the content but also the structure of the XML and the hierarchy that defines it. And that is where XML gets scary and confusing. Because it's a markup language, you get that added structure intertwined in with your data.

Why deal with all that mess when you can have three simple variables? Given the example above, it would certainly be easier to use a variable string and LoadVars. Structure and organization may suffer, but that particular instance doesn't really use much of it, right? There are many cases where that is right, where variable strings may be advantageous over XML. These include cases where structure may be irrelevant or where you're dealing with fairly simple, straight-forward data that wouldn't be all that confusing as a variable string.

Structure and clarity are the key elements that XML offers text-based data. Given the circumstances of your situation, you will need to decide whether or not your data requires this or whether you might be better off just using urlencoded variables. You get more speed with the variables but you lose structure and clarity XML provides that structure and clarity but can take longer to load depending on that structure and takes extra effort to parse when interpreted into usable data.

Loading XML Into Flash
When you decide to use XML to load data into Flash, the next step is figuring out how exactly that XML file makes it there and what commands are needed to make it happen. Well, don't worry. It's not hard at all. If you've ever loaded in a variable string with loadVariables then you've pretty much already loaded XML too. It's the exact same process.

Loading XML revolves around 2 functions. One of these functions is a pre-existing function that you simply call yourself. However, the other is a *callback* function that you have to define which will automatically be called by Flash depending on the occurrence of a certain event. The event we're dealing with here is the event of the XML being fully downloaded and introduced into Flash movie. This is the called the *onLoad* event.

Each of of these functions are used on an XML instance. XML instances are created using the XML object, or class, and provide a construct in Flash that lets you manage your XML. If you're working at all with XML in Flash, then you're pretty much guaranteed to be using an XML instance.

So first, before anything, when loading XML into Flash, you have to create that XML instance. The XML instance for this example, and most you will see from now on despite the fact that naming is arbitrary, will be called "my_xml." Note: using "_xml" at the end of your XML variable name in Flash MX or MX04 will give you code hints. In MX04, typing a new XML instance will provide hints without the suffix (i.e. *var life:XML = new XML();* suffices).

```
var my_xml = new XML();
```

In creating an XML instance in this manner, you do have the option of passing in an XML string to the XML constructor (the constructor being the function that creates the XML, here new **XML()**). That string would consist of XML which will be immediately defined within the XML instance created. For example:

```
var my_xml = new XML("<some>stuff</some>");
```

Though, when about to load in external XML, there's really no point since whenever you load XML into an XML instance in Flash, all XML contained within that instance is replaced with that which is loaded. Passing text like that is optional, so for this example it will just be omitted.

Once an instance exists, you can define the onLoad callback function. The callback function, whenever you make it, always has to be called onLoad. Just like other event handlers, this is how Flash knows to call it when its needed, i.e. when the XML content has been loaded and parsed. Additionally, a *success* argument is passed into each onLoad when it fires. This will let you know if your XML has actually successfully loaded or not. For example, when you try to load XML from the URL "http://get a life," success will be false - "http://get a life" is obviously not a valid URL (hey, we all have lives here!). However, use a valid url (with a rock-solid internet connection) and success will be true signifying the completion of XML being loaded into Flash and ready for use.

Here, we'll make an onLoad function that simply traces the XML object when successfully loaded. Tracing an XML object directly will reveal the XML in text format.

```
var my_xml = new XML();
my_xml.onLoad = function(success){
    if (success){
        trace(this);
    }
}
```

Since onLoad is defined in the XML instance, *this* inside the function references the instance directly.

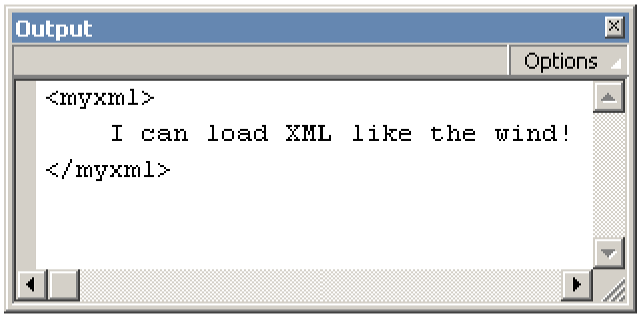
Now that the onLoad has been defined, it's now time to request the XML to load in an external XML document. This is handled through the load method, the second of the 2 functions. The load method accepts one argument, the external XML document's URL (this can be relative or absolute).

```
var my_xml = new XML();
my_xml.onLoad = function(success){
    if (success){
        trace(this);
    }
}
my_xml.load("my_document.xml");
```

Now, supposing my_document.xml contained the following:

```
<myxml>
    I can load XML like the wind!
</myxml>
```

When the ActionScript above is run and the XML is loaded, you would receive a trace that would resemble the following.



```
<myxml>
    I can load XML like the wind!
</myxml>
```

[output of loaded xml trace]

Bear in mind that the XML *does* have to *load* into Flash. This is not an immediate process. It takes time, often many seconds or Flash frames before any of the loaded XML is accessible through the XML instance.

This means that any attempt to access that information in the same script which the load method is used will end in failure. That is, of course, unless you do so within the onLoad function. Though the onLoad is *defined* in the same script as everything else, it doesn't actually get executed until the XML is fully loaded and parsed - some time after the rest of the script has already completed running, So, in other words, don't do this:

```
var my_xml = new XML();
my_xml.onLoad = function(success){
    if (success){
        trace(this);
    }
}
my_xml.load("my_document.xml");
trace(my_xml);  ← too early, not loaded yet
```

It's in the onLoad function where you pretty much do everything it is you need to do with your loaded XML content. You need it to populate a menu? Do it in the onLoad. Want to display your family tree? Do it in the onLoad (someone has to have a nice XML family tree floating around). The onLoad is the key to handling loaded XML since it is at that point you actually have access to it. Anywhere else and you just may not have any XML to reference.

Preloaders With XML
Just like anything else loaded into Flash, you can also create preloaders for XML. Generally, however, since XML is usually light on the loading side, preloaders aren't needed. A general "Now Loading" message usually suffices. But sometimes, for those real hefty files, you may want a preloader to show the status of the XML loading.

If you're worried about learning a whole new way of making preloaders for XML, don't be. Making a preloader for XML is exactly like making one for a loaded SWF or JPEG. The only difference is that instead of calling getBytesLoaded and getBytesTotal from a movie clip, you're calling it from your XML instance. So really, you can use preloaders created for movie clips with XML so long as you reference the XML object to get bytes loaded and bytes total. Here's a quick example of a preloader that can work with XML as well as movie clips:

```
preloadbar_mc.onEnterFrame = function(){
    if (!this.target) return (0);
    var loaded = target.getBytesLoaded();
    var total = target.getBytesTotal();

    var scale = 0;
    if (loaded && total){
        var percent = loaded/total;
        scale = 100 * percent;
    }

    this._xscale = scale;
}
```

```
preloadbar_mc.target = my_xml;
```

Where preloadbar_mc is a horizonally scaling movie clip representing the preloader and my_xml is the XML instance you wish to show the preloader for. The onEnterFrame event of the preloadbar_mc runs the preloader using a generic target to get bytes loaded and bytes total. Whether this is a movie clip or an XML instance, as long as getBytesLoaded and getBytesTotal work, it doesn't matter.

White Space in Loaded XML
There is a certain XML option concerning loaded XML that should not go without mention. That is the option to ignore extraneous white space between elements in an XML document. This is determined by an *ignoreWhite* property of an XML instance.

```
my_xml.ignoreWhite = true;
```

What this does is prevents white space such as tabs and spaces used in formatting in your XML to be interpreted as text nodes which Flash likes to do (white space is text too right?). For example. How many child nodes does the happy element have here:

```
<happy>
	<joy />
</happy>
```

If you said three, then you were right! The happy element has one child element, joy, and two text nodes; a *newline* + *tab* text node before joy and another *newline* text node following it. This effect is generally not desired as such white space is meant solely for formatting purposes. By default, the ignoreWhite property for any XML instance in Flash is set to false, so you may want to get into the habit of setting it to true immediately after creating an XML instance if you don't want such white space to be considered text elements. Here it is applied to the previous example used to load my_documents.xml:

```
var my_xml = new XML();
my_xml.ignoreWhite = true;
my_xml.onLoad = function(success){
    if (success){
        trace(this);
    }
}
my_xml.load("my_document.xml");
```

The ignoreWhite property should be set before XML is loaded as it effects the parsing process. It won't change existing XML within an XML instance.

Note: ignoreWhite only removes white space *between elements*, this doesn't not include any white space that makes up valid text nodes, even that which is used for formatting them.

Tracing XML
If you've tried the above script, you'll notice that the XML instance traces out the entire XML document it contains when passed to a trace command. Many objects in Flash put "[object Object]" in the output window when traced. The XML object, however, has a unique *toString* method (the method used by objects represent themselves when trace needs to show it in the output window) that overrides the "[object Object]" with the string representation of the actual XML contents. When you set ignoreWhite to true, you can see in

a trace how the XML's structure has changed as a result of that property. For example, the happy-joy XML with ignoreWhite set to true would trace "<happy><joy /></happy>." Later, we'll show you how you can make a toString-esque method that can effectively reverse this process.

XML to XML Object
Immediately following the process of loading XML into Flash, there is a behind the scenes process which converts the original XML text into a usable ActionScript object that assumes the identity of what you know to be your XML instance. This process is called *parsing*. With a LoadVars urlencoded variable string, parsing converts the string into variables with their respective values. For XML, the parsing process creates a usable XML instance.

This sounds wonderful and easy at first, after all, you don't have to do anything yourself during this process. And making usable Flash objects out of the XML text? What's better than that? At second glance, however, and especially when you actually start working with this new ActionScript representation of XML, you start to realize that XML in ActionScript is far more complicated than it had previously seemed to be. Sure, conceptually, its easy enough to understand - elements, text nodes, CDATA; we've already covered that with no problem. However, once you take that and then shove it into an alternative programmatic structure (ActionScript), you have a whole new layer of complexity to deal with, and this especially when trying to code your way through that structure. XML alone may be easy. ActionScript alone may be easy. Put them together and it suddenly isn't looking so easy. Fear not; that's what I'm here for.

The trick to mastering your way through XML via ActionScript is through knowledge (whoda thunk?). Yes, as GI Joe has been telling us for years, "Knowing is half the battle." You will just need to know *what* in ActionScript represents what in XML and *how* to get to it so that you can retrieve what you need to retrieve (or perform whatever operation you need to perform). People are afraid of the dark because they don't know what dangers it may contain. People are afraid of XML because no one clearly explained to them what they need to know to fully understand it's representation in Flash.

Since Flash objects and timelines are hierarchically structured much in the same way as XML is, such a conversion from XML to ActionScript object would seem simple enough as it could be translated fairly directly. However, because of the way XML is defined, there ends up being some complications. Lets take a look back at what the basic XML structure looks like including common node types:

```
<root>
    <child attribute="value" attribute2="value2">
        Text Node: Child of child.
    </child>
    <child>
        Text Node 2: Child of second child.
    </child>
    <child attribute="value2" />
</root>
```

What we have are a collection of hierarchically related nodes, some element nodes (some with attributes and some with not) and a couple of text nodes. This adheres to the basic structure and rules of an XML document. The structure is hierarchical with element nodes containing other nodes which they themselves can contain more nodes. The rules to keep in mind here are that elements can share similar names whereas attribute names must be unique.

If we were to take the above and convert it into a similarly structured Flash object, you may get something like the following:

```
var parent = new Object();
parent.child = new Object();
parent.child.attribute = "value";
parent.child.attribute2 = "value2";
parent.child = new Object();
parent.child.text = "Text Node: Child of child.";
parent.child = new Object();
parent.child.text = "Text Node 2: Child of second child.";
parent.child = new Object();
parent.child.attribute = "value2";
```

Immediately, you should be able to see at least one apparent problem - the assignment of child. Because XML elements don't need to have unique names, when a new child object is added to the parent object above in Flash, it effectively replaces whatever object was defined there under the same name (the original child).

Also, though far less apparent, is that you have no preservation of order in using object properties to define elements. Given the original XML layout, it's easy to tell which child is first, second, and third - information which could be important to the content (in XML node order is not redundant). With Flash objects, you have no real control of object property order, they exist just as properties. So then, what *would* facilitate objects in a specific order whose name's don't have to be unique? Hmm... arrays would, right? Keeping each element in an array will allow it not only to have whatever name it wants (it can be stored as a property of an object element in the array - a property under something like "nodeName" perhaps?) but also maintains the order as it is specified within the original XML document. A good array name for holding child nodes of any element may be... "childNodes," don't you think?

What about attributes? Is there anything horribly wrong with them in the Flash object above? Well, for the most part, no. But being assigned directly to an element object could cause confliction with already predefined element values and methods such as those provided by Flash. You shouldn't be restricted from using a certain attribute name just because Flash might use it as an XML property in an XML instance. To keep these separated a bit to avoid such confusion, attribute definitions can be kept in a single object within the element object called... how about "attributes?" The fact that they all need unique names means that they can remain defined under a variable of a similar name instead of needing an array (attribute order is not a factor).

All that remains are text nodes. They seem fine enough. But remember, text nodes are nodes and separate entities of the elements in which they exist. They, like other elements, would be children of their parent element. As such, they too will need to be placed in the childNodes array of the element containing them. Also, in being a *node* these entities should be created as objects in order to facilitate node properties and methods as Flash may seem fit to provide (as opposed to just being String variables). The actual text can go in a property of that node object called, lets say, oh, I don't know, "nodeValue?"

Wow, things just got a little more complicated. Let's revise the Flash object from above to work with the problems we just solved:

```
var parent = new Object();
parent.childNodes = new Array();
parent.nodeName = "parent";
parent.childNodes[0] = new Object();
parent.childNodes[0].nodeName = "child";
parent.childNodes[0].attributes = new Object();
parent.childNodes[0].attributes.attribute = "value";
parent.childNodes[0].attributes.attribute2 = "value2";
parent.childNodes[0].childNodes = new Array();
parent.childNodes[0].childNodes[0] = new Object();
parent.childNodes[0].childNodes[0].nodeValue = "Text Node: Child of child.";
parent.childNodes[1] = new Object();
parent.childNodes[1].nodeName = "child";
parent.childNodes[1].attributes = new Object();
parent.childNodes[1].childNodes = new Array();
parent.childNodes[1].childNodes[0] = new Object();
parent.childNodes[1].childNodes[0].nodeValue = "Text Node 2: Child of second child.";
parent.childNodes[2] = new Object();
parent.childNodes[2].nodeName = "child";
parent.childNodes[2].attributes = new Object();
parent.childNodes[2].attributes.attribute = "value2";
parent.childNodes[2].childNodes = new Array();
parent.childNodes[3] = "I'm tired of typing...";
```

Suddenly that simple XML file isn't looking so simple in its ActionScripted version anymore. And in case you were wondering, the structure above is pretty much exactly how that XML would be laid out in an instance of the XML object in ActionScript. The parent variable here actually represents the first child of an XML instance (the XML instance itself acts like a node containing all XML of that instance within it as a child). Everything else is as it would be within that object. Daunting, isn't it?

Fear not, you're half-way in the know now. We just went through the reasons why this complexity exists which is a large step in helping to understanding it. In summary Flash translates an XML document's structure into an ActionScript object - an XML instance - through the following:

- The base XML object is an object containing the entire XML hierarchy beneath it (as a child).
- All children of any given node (including those in the base XML object which is itself a node) are contained within an Array in that object called childNodes.
- Each Element Node contains a property nodeName (among others) giving its name.
- Each Text Node node contains a property nodeValue (among others) containing the text.
- Attributes of an Element Node object are contained with an attributes object in that node object under their respective variable names.

Finding Your Way Around An XML Object

Once you know (or think you know) what is what in a Flash XML instance, it's time to figure out how to extract that what so that you can use it for your evil conquests of world domination... and... other Flash needs too. Navigating an XML object can be just as hard, if not harder, than simply understanding its structure. Though once you understand the structure, navigation becomes a bit easier.

Because arrays are used heavily in storing elements, looping will start to become your favorite pastime when working with XML. Loops (*for* and *while*) in Flash allow you to easily cycle through all elements of an array, or, in the case of XML, all the children of an element, allowing you access each one of those nodes individually within each iteration of the loop. Depending on the structure of your XML, for loops may need to be nested in order to loop through elements within elements already being looped through. That's always fun right?

	Note: Design XML for Accessibility
	If you are in direct control of the structure of your XML file, you may want to try, if possible, to minimize the number of levels in the XML hierarchy. The fewer nested loops you need to implement, the better. There are other ways around nested loops which will be brought up later. But it's important that if you are designing XML to be used primarily in Flash that you design it to be easily navigable.

Navigation Through Helpful XML Properties
So far we've seen that XML nodes in Flash form have attribute objects and child nodes arrays stored as properties represent their structure. Flash ActionScript provides some additional properties added to XML nodes to help make navigation through it a little more easier. Here's a list of XML structural properties you can reference off of XML nodes.

Property	**Represents**
XML Nodes	
attributes	An object containing attribute variables assigned to this element.
childNodes**	An array containing all child nodes belonging to this node.
parentNode*	This node's parent node.
firstChild*	The first child in this element's childNodes, or `childNodes[0]`
lastChild*	The last child in this elements childNodes, or `childNodes[childNodes.length-1]`
nextSibling*	The node after this node in the parent's childNodes array.
previousSibling*	The node before this node in the parent's childNodes array.

* Read-only and can only be checked, not directly set.
**Altering element order in this array will not be reflected in the XML instance.

Together, these properties provide the groundwork for you being able to navigate through your XML.

Of course seeing the properties, and reading and understanding what they represent is one thing. Being able to use them is a whole new slice of cheese. Making proper use of these properties is really what makes XML so difficult to deal with in Flash. We'll work through a little of that now but the Flash examples given later on will provide a better understanding of using them more effectively and in context.

The first thing to remember is that an XML instance represents a single node in which the rest of the XML is defined. That means that your XML instance and the document root node of your XML are not the same thing. The root node is actually a child of the XML instance. Since there should be only one root node

(DOCTYPE and XML declarations are not considered nodes and are not accessible as children), that means your root node would be the first child of the XML instance. Given the XML instance my_xml, the root node would be:

```
my_xml.firstChild
```

From here, you can then start accessing your XML as needed.

Now, you'll notice that even just getting to the document root of an XML document through an XML instance requires using firstChild. What do you think happens when you want to get to the first child of the first element in your document root? Consider the following XML:

```
<mydocuments>
    <mypictures>
        <family>
            <image title="Sister laughing" />
            <image title="Brother laughing" />
            <image title="Mother beating me" />
        </family>
        <vacation location="Myrtle Beach">
            <image title="Sun bathing" />
            <image title="Walking the dog" />
            <image title="Swimming" />
            <image title="Getting eaten by shark" />
        </vacation>
        <girls />
    </mypictures>
</mydocuments>
```

Now assume this is the XML content of the my_xml variable in Flash. The first child of the first element in the document root is family (where mypictures is the first element in the document root). So, in order to reach the family node in my_xml, you would use:

```
my_xml.firstChild.firstChild.firstChild
```

Already you can see the complexity and confusion that's just starting to unravel. If it takes that much just to get to the first pertinent element in a simple XML document such as the one above, imagine the paths needed to get to things in a more complicated XML document. How are you ever to keep track?

Variables. Variables are the key to success and understanding when using XML. Variables and loops. Looping gets you through element children and variables provide you a way to give descriptive names to abnormal paths such as the one given above. What's easier to understand? *my_xml.firstChild.firstChild.firstChild.firstChild* or *family.firstChild*? Saving the original path in a family variable (as it represents the family element) you get yourself a clearer understanding in the reference to the specified image element.

When using loops, you would typically use the childNodes array. Then, just like with any other array, you would loop through each element performing whatever task is needed on each. For example, to loop through all the image elements in vacation, you could use:

```
var family = my_xml.firstChild.firstChild.firstChild;
var images = family.childNodes;
for (var i=0; i<images.length; i++){
    currImage = images[i];
}
```

Using the attributes object, you could then access the titles of each and trace them in the output window

```
var family = my_xml.firstChild.firstChild.firstChild;
var images = family.childNodes;
for (var i=0; i<images.length; i++){
    currImage = images[i];
    trace(currImage.attributes.title); // trace each images' title
}
```

Lets say you then wanted to go about looping through the vacation images. How might you go about that? Well, you could use a similar path to get to vacation as you did with family. Only with vacation, the last child reference would have to be from the childNodes array since vacation is neither the firstChild nor the lastChild of mydocuments.

```
var vacation = my_xml.firstChild.firstChild.childNodes[1];
```

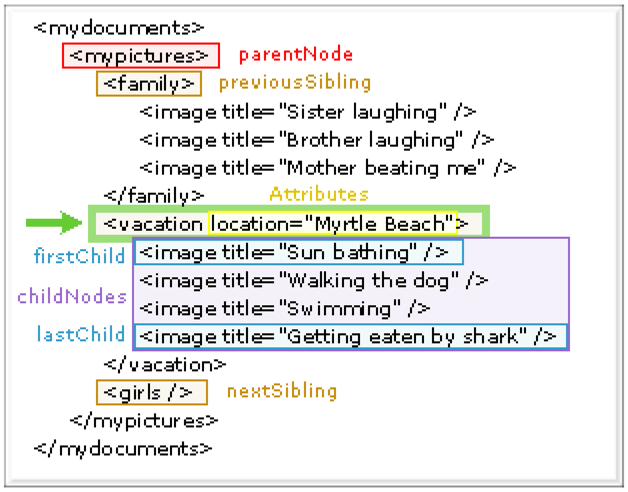
However, since family has already given us much of that path already, nextSibling can be used to get to vacation directly from family.

```
var vacation = family.nextSibling;
```

Unlike the child reference properties, the sibling properties stay within the same element scope being able to reference other nodes which share the same parent (in this case mypictures) as the node from which its being used. So, after going through the family image elements, you can go through vacation images using:

```
var vacation = family.nextSibling;
var images = vacation.childNodes;
for (var i=0; i<images.length; i++){
    currImage = images[i];
    trace(currImage.attributes.title); // trace each images' title
}
```

The following provides a more visual representation of what each property represents in regards to a vacation node within in an XML file.



```
<mydocuments>
   <mypictures>
      <family>
         <image title="Sister laughing" />
         <image title="Brother laughing" />
         <image title="Mother beating me" />
      </family>
      <vacation location="Myrtle Beach">
         <image title="Sun bathing" />
         <image title="Walking the dog" />
         <image title="Swimming" />
         <image title="Getting eaten by shark" />
      </vacation>
      <girls />
   </mypictures>
</mydocuments>
```

[xml references in respect to the vacation element]

You can see that the vacation node has a wide array of direct access when it comes to referencing other nodes within the XML in respect to its location using those properties available to it. The parentNode property references mypictures, the element in which it exists; previousSibling and nextSibling references the child nodes next to vacation within that parent node on either side of itself (a.k.a. "siblings" like brothers and sisters). We saw how family's nextSibling was vacation. Using vacation's previousSibling, you can go back up to family. And then of course you have firstChild, childNodes, and lastChild provide access to vacations own children, its image elements. Parents, siblings, children, XML just *asks* to have a family tree written in it, doesn't it?

There are yet some other properties which you can use to find out more about your XML and its nodes. These don't necessarily help you navigate through your XML so much, but they provide important information nonetheless. They are as follows:

Property	**Represents**
XML Nodes	
nodeName	The node's name. This is the tag name of an element node and `null` for other nodes.
nodeType*	A numerical value representing the node's type: 1 = Element 3 = Text Node (or CDATA Section)
nodeValue	The node's value. This is text for text nodes and CDATA and `null` for elements.
XML Instances	
xmlDecl	XML's declaration example: <?xml version="1.0"?>
docTypeDecl	XML document DOCTYPE declaration example: <!DOCTYPE greeting SYSTEM "hello.dtd">
loaded	True or false depending on whether or not the last load() command has successfully completed for the XML instance.

status	A numeric value representing parsing errors during Flash 's attempts to convert XML text into the ActionScript XML object. They are as follows: 0 No error; parse was completed successfully. -2 A CDATA section was not properly terminated. -3 The XML declaration was not properly terminated. -4 The DOCTYPE declaration was not properly terminated. -5 A comment was not properly terminated. -6 An XML element was malformed. -7 Out of memory. -8 An attribute value was not properly terminated. -9 A start-tag was not matched with an end-tag. -10 An end-tag was encountered without a matching start-tag.

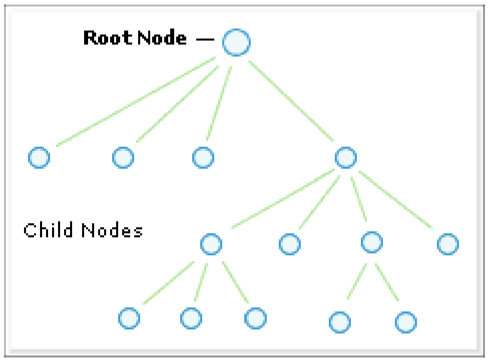
*Read-only and can only be checked, not directly set.

We've already seen nodeName and nodeValue before, but nodeType is a new one. It helps you distinguish elements from text nodes. Other properties listed are for XML objects specifically and not the nodes they contain. The first two, xmlDecl and docTypeDecl let you extract an XML document's declaration and doctype which are not, technically, otherwise included as elements in the XML structure (at least not in Flash). The other two, loaded and status, just help determine information about loading and parsing XML, whether or not they were successful.

With all these properties out of the way, we can start getting into some examples which make practical use of them, hopefully in a comprehensible manner.

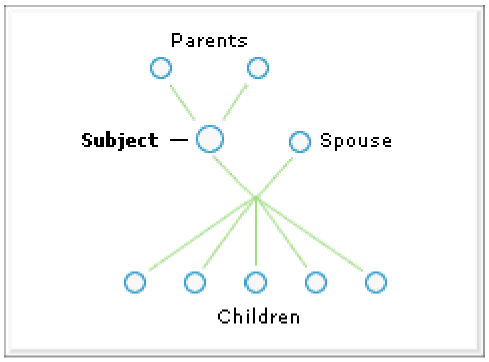
Evaluation: XML And Your Family Tree
I've been ranting on an on throughout that XML seems fit for a family tree. Parents, children, siblings - XML just seems so very family oriented. But is that really the case? Lets take a closer look at how elements in an XML hierarchy are arranged and whether or not they make presenting a family tree easy.

Way back when, the file-folder analogy was used to help describe the structure of XML. You have folders (elements) and in those folders you have files or more folders (elements and other child nodes). Any single folder can have any number of files or folders and any of those folders within a folder can have their own similar collection and so on. A file or folder, however, only exists in a single folder at any given time. You get a relation that looks like this:



[general structure of xml]

Because everyone can relate to their family and know who's who (who's a parent and who's a child), keywords used to describe similar relations are used to make using XML more comprehensible - child nodes, parent node... siblings, etc. However, if you think about it, they aren't quite exactly the same. There's one key difference in the real family structure when compared to that of XML. That difference is having two parents. Look at the following. It represents a portion of the family structure of a single person (subject)



[structure of a family tree]

Here, the subject belongs to two parents, a mother and a father. The children belong not only to the subject, but also to the subject's spouse. So the children, too, have more than one parent. Like with files and folders, this just isn't possible with XML (that's right, XML does not support multiple inheritance). This doesn't, however, make an XML version of a family tree impossible. It just makes it so that an XML representation will not structurally match the content it contains. And there's nothing wrong with that. It's perfectly fine but it also means that interpreting that information will might take a little more effort.

> An example would be... wouldn't you know it, FTML. Yup, Family Tree Markup Language - XML for family trees (FTML is technically based off of SGML like HTML and XML). FTML uses id attributes to manage and relate people to other people as they are linearly listed within a FTML document. Take a look at this quick example:

```
<?xml version='1.0'?>

<!DOCTYPE ftml SYSTEM "ftml.dtd">

<ftml>
    <people>
        <person id="Granddad" sex="male" surname="Example" fornames="Granddad">
            <born date="1915" />
            <died date="1998" />
        </person>
        <person id="Grandma" sex="female" surname="Jones" fornames="Granny" />

        <person id="Dad" sex="male" surname="Example" fornames="Daddy">
            <born date="1940" />
            <mother id="Grandma" />
            <father id="Granddad" />
        </person>
        <person id="Mom" sex="female" surname="Smith" fornames="Mommy" />
        <person id="Uncle" sex="male" surname="Example" fornames="Uncle" />
        <person id="Aunt" sex="female" surname="Trotter" fornames="Aunt" />

        <person id="Me" sex="male" surname="Example" fornames="Me">
            <born date="1973" />
            <mother id="Mom" />
            <father id="Dad" />
        </person>
        <person id="Brother" sex="male" surname="Example" fornames="Brother">
            <born date="1971" />
            <mother id="Mom" />
            <father id="Dad" />
        </person>
    </people>
    <marriages>
        <marriage husband="Dad" wife="Mom" date="1964" />
        <marriage husband="Uncle" wife="Aunt" />
    </marriages>
</ftml>
```

All people within this family structure here are listed linearly in the people element. Each person is given an id which is then referenced in two places: 1) in the person definition where a mother and/or father is specified and 2) in the marriages section where a husband and wife are connected by id (some versions of xml-based family trees keep marriage relations within the person tag under wife or husband).

So, the best use of the XML hierarchy within this document is just maintaining content concerning a single person (and the collection of people). Actual relations are all handled through ids. So despite my hopes in creating an example based on a family tree, given the type of XML design needed above, that's not something we're about to tackle here. Lets continue with another, simpler example. For your own XML projects, however, you may need to consider a setup something similar to FTML. Something to keep in mind.

Example: Squirrel Finder
This example will cover how to generate a list of buttons based on loaded XML. Each button then, when clicked, displays information obtained from the XML associated with that button.

First, the XML document:

squirrel_finder.xml

It should look a little something like the following (the entire file is not shown here):

```
<?xml version="1.0"?>
<menu>
    <menuitems sourceurl="http://spot.colorado.edu/~halloran/sqrl.html">
        <item type="squirrel">
            <species>Abert Squirrels</species>
            <location>
            <![CDATA[Abert squirrels (Sciurus aberti) are only found
            in mountain regions of North America. In the United States
            their range extends from extreme south-central Wyoming to New
            Mexico and Arizona. There are six valid subspecies including
            S.a. ferreus, True, the subspecies that is found along the
            Rocky Mountains in Colorado. Abert squirrels are limited to
            Ponderosa pine (Pinus ponderosa) forests in which they feed
            and build their nests.]]>
            </location>
        </item>
.
.
.
```

Here laid out in this file is a list of items that will populate a simple menu in Flash. Each item has two strings associated with it, a species (of squirrel) and a description of the location of where that particular species can be found in the North America. Ultimately, when brought into Flash, a menu will appear with buttons for each item listed showing on that button the species of squirrel. When clicked, it will bring up a dialog which will display the location text associated with that species. The final result gives you this:

(Doesn't copy and paste well, see http://www.kirupa.com/web/xml/examples/squirrelfinder.htm for the example.

[squirrel finder in action]

Download ZIP

XML Structure
Once you decide what you want/need in terms of your project (in this case, make a squirrel finder), one of the first things to consider is the structure of the XML which is going to deliver your information. I've already decided that, so that kind of takes away much of the thinking that would normally take place here. Still, lets take a look at the XML document and how its set up.

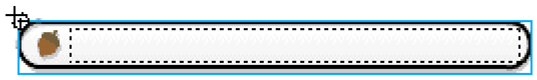
What we're dealing primarily with is a list of an unspecified number of containing two specific pieces of information, species and location description that ends up becoming a menu. In addition to this information, such listed items also have an identifier to specify what kind of item we're dealing with (there can be many species of many kinds of things, though here, we only want squirrels). Since the overall objective is *menu* the root document element reflects that: <menu>. The list of items is itself its own entity so it deserves its own element to reside in, aptly named "menuitems." Menuitems has an added attribute called sourceurl specifying the location this information was obtained. It's not used at all by Flash, but it's good to have nonetheless. Within the menuitems element are the item elements that contain our desired information along with an attribute to specify type of item. Type here is a simple, known keyword used to describe the element. Attributes should not be used for housing excessive text or values of uncertain content. Each item element appropriately represents a single item in the list to appear in Flash. The two bits of information needed about this item are also contained within their own elements within the item element. These are <species> and <location>. Since the species name should be straight-forward and lacking complication, a simple text node should suffice. The location element, however, with the text it contains, may have some unforeseen characters that might conflict with a proper XML document. As such, it is put in a CDATA section.

```
<menu> - root of XML
    <menuitems sourceurl=""> - container of items
        <item type=""> - single item w/ type
            <species></species> - container for species text
            <location> - container for location text
            <![CDATA[ ]]> - keep questionable text away from parser
            </location>
        </item>
        - more items...
    </menuitems>
</menu>
```

Preparing the Flash File
Now that we have or XML defined and more or less out of the way for the time being, its time to begin designing the interface to display this information. Eventually we will need to interpret that XML meaningfully when brought into Flash, but we'll get to that later. First the physical elements of the movie need to be constructed.

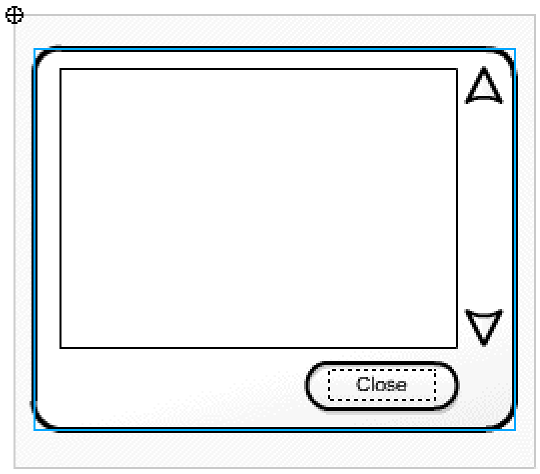
The squirrel finder's interface is pretty simple. We have two basic screen elements: an item button and an information dialog that appears when you click on an item button. That's about it. Items are symbols (MovieClips) that would need to be dynamically attached for each item in the XML while the information dialog only needs to be concerned with one instance of itself which appears only when requested by a clicked menu item.

The buttons for menu items will have to be fairly dynamic. The button itself is hand-drawn, but the label text will be included by the XML when its loaded. This will simply be a dynamic text field. on top of the button. The behavior of this button will also be dynamic, assigned when the XML is loaded at which point the button actually has a purpose. That really requires nothing when making the button other than making sure you give it an instance name. Once the button is complete, it goes in an item movie clip that will reside in the Library awaiting to be attached (since its being attached, remember to make sure it has a linkage ID set).



[item button]

The information dialog is just a movie clip with a text field. that has a scrollbar thrown on to it. A close button is included to close this movie clip (hide it by setting _visible to false) when the viewer is done looking at the information. This movie clip and another empty movie clip will be the only elements on the stage when the Flash movie starts. The empty movie clip is used to attach menu items in (our buttons) which will be performed when the XML file has been loaded and interpreted (in the file, the "empty" clip actually contains a light grey square though not as a necessity).



[dialog on main timeline]

Scripting the Squirrel
With the visual elements complete, you can start scripting and testing. Often it is a good idea, depending on how sure you are of your coding abilities, to build such interaction in Flash little by little in ways that allow you to test an uncompleted version for functionality before continuing. This is a debug as you go approach. It can be time consuming but can also save you frustration in the long run.

There are three basic parts to coding the squirrel finder. They are: defining basic navigation interaction - core behaviors of item buttons and the close button in the information dialog, successfully loading the XML document, and finally interpreting that document to build and assign functionality to the menu that results from it. We'll start with the navigation.

This is pretty simple. The navigation is simple. It would be just wrong to make a simple navigation difficult (and based on that, I haven't). There are two "screens," one containing the menu, "menu_mc", and the other an information dialog, "infobox_mc" (though the information dialog appears to be a floating window above what would be the menu, the menu is hidden for simplification). When one screen is shown, the other is hidden and vise versa. Item buttons hide the menu and show the information dialog while the close button performs respectively for doing the opposite. Each of these actions are defined in functions. Since the close button will actually be present when the function is defined, it can be set immediately to be the close button's onRelease event handler. Item buttons, however, won't exist until after the XML defining the menu has fully loaded and the menu has been created. So in the mean time, it will sit simply under a generic name, "DisplayInfo", until it can be set to individual items' onRelease events dynamically. Each button also contains code to alter the information dialog's contents.

```
function DisplayInfo(){
    menu_mc._visible = false;
    infobox_mc._visible = true;
    infobox_mc.content_txt.text = this.location_text;
}

infobox_mc.close_btn.onRelease = function(){
    menu_mc._visible = true;
    infobox_mc._visible = false;
    infobox_mc.content_txt.text = "";
}
```

You can see that each action pulls the old switcheroo in setting visibility between menu_mc and infobox_mc. DisplayInfo also sets the text field. text within infobox_mc to be a variable called "location_text" assigned to the owner of the function. Right now, that's nothing, but when this function gets assigned to an item button, that variable will reference that button's location_text property, a property assigned to that button based on content grabbed from the XML file. Conversely, the close_btn sets the text to "" or no text clearing the dialog.

Since both of these movieclips exist on the main timeline to start and only one can be visible at a time, that also means we need to set one to start invisible. That would be infobox_mc. That little line can go right below the functions defined above.

```
infobox_mc._visible = false;
```

Next lets tackle the menu creation. This is by far the hardest part of the entire file. It consists of a single function called CreateMenu. Passed into that function is one argument, the XML defining the menu which is to be created - and that is where the fun begins, using that XML data to build the menu of squirrels (items). Luckily this is a simple example so it will be easy to follow.

First we must make sure we understand what is being sent - what is in the XML. We covered the XML earlier, but in brief we have the following to deal with:

- A list of menu items within the root node under the element "menuitems"
- Each item element has a type attribute specifying what it is (each item of type "squirrel" will be represented as a menu item)
- Items have two child elements, "species" and "location" each with text nodes (one using CDATA) to store their related content.

Technically, this list of menu items would be of an undetermined length. We don't really know how many squirrels will be in this menu or if/when new ones will be added. The beauty of using XML is that they can be added (or taken away) very easily without ever touching Flash. All anyone has to do is edit the XML with a text editor and you're set - that is if we're smart enough to make Flash do that. How is with a loop.

Looping is what will get us through the list of menu items. Remember, elements are kept in childNodes arrays that exist in their parent nodes. Each item in the XML exists as an element in the menuitem element's childNodes array. A loop going through those array elements will give you each item no matter how many there are, and this we can tell using childNodes.length.

The loop gets you through each item. From there its just a matter of extracting the needed content and attaching a item movie clip to facilitate it. To help manage the attaching, two variables are used, one, a constant, to decide how far apart to space items as they are added and another to count the number of items attached (also provides a way to assign a unique depth to each one). These are called "item_spacing" and "item_count" respectively.

```
var item_spacing = 28;
var item_count = 0;
```

With that, lets look at what this function looks like:

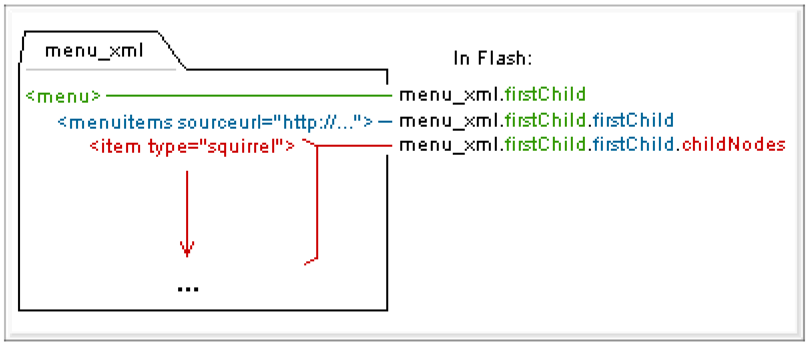
```
function CreateMenu(menu_xml){
    var items = menu_xml.firstChild.firstChild.childNodes;
    for (var i=0; i<items.length; i++) {
        if (items[i].attributes.type == "squirrel") {
            var species = items[i].firstChild;
            var location = items[i].childNodes[1];

            var item_mc = menu_mc.attachMovie("menu_item","item"+item_count, item_count);
            item_mc._y = item_count * item_spacing;
            item_count++;

            item_mc.species_txt.text = species.firstChild.nodeValue;
            item_mc.main_btn.location_text = location.firstChild.nodeValue;
            item_mc.main_btn.onRelease = DisplayInfo;
        }
    }
}
```

You can see a fair number of variables being set here. The first few lines in the function you can see a couple of variable definitions. These aren't necessarily required definitions either. They simply provide shortcuts to paths in the xml. This not only keeps you from typing more, but it also makes the code a lot easier to understand. Its suggested that you do this as frequently as you can in XML code to help improve clarity.

The first variable, items, is the child nodes array of the menuitems element. The menuitems element is identified with:

```
var items = menu_xml.firstChild.firstChild.childNodes;
```

where menu_xml represents the XML object passed into the function. So this is saying get the child nodes of the first child of the first child of xml making up the menu_xml object. The XML object's first child is the root element menu. Menu's first child is menuitems and its childNodes represents the array of item elements we're after.


menu_xml
In Flash:
<menu>
menu_xml.firstChild
<menuitems sourceurl="http://...">
menu_xml.firstChild.firstChild
<item type="squirrel">
menu_xml.firstChild.firstChild.childNodes
...

[referencing items array through child nodes]

We can now work with items more easily as it is a simple representation of the item elements array. There's no point in retyping all those firstChilds again if we don't have to. The items variable will also greatly improve code comprehension.

With the items variable assigned to the array of items, we can no begin cycling through that array creating menu buttons for each item and extracting/assigning content text where needed. This is where the for loop comes in.

```
for (var i=0; i<items.length; i++) {
    ...
}
```

As loops go, this is pretty self-explanatory. This creates a variable i which starts at 0 and increments by 1 while its value remains below the length of the items array. This lets us go through each item as item[i] and have the code within this loop execute for each.

Now we start getting into the accessing the information of the XML object (through the items array). There are three pieces of information we want to extract. First is the type of item to see if it should be included in our menu. Second is the species of our item (of our squirrel item) and finally the text specifying location.

The type of item is probably the easiest to extract. This information is held in an attribute of an item element and is accessible through the attributes object of that node in Flash. Within the for loop, items[i] represents the current (item) node in the loop iteration. The type would come from item[i].attribures.type. You can see that with the following if statement in the for loop:

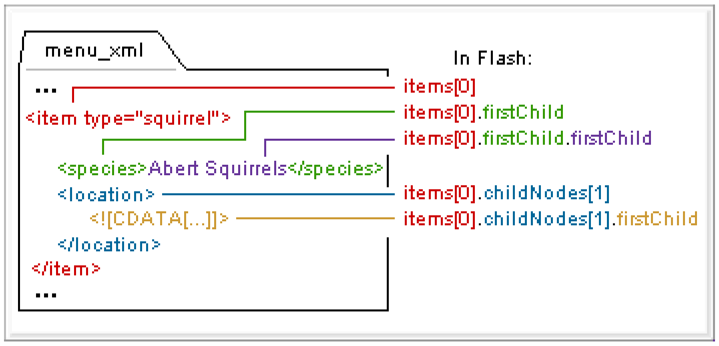
```
if (items[i].attributes.type == "squirrel") {
    ...
}
```

This checks each item's type. If it matches "squirrel" then the code within the block (that which creates the menu items) is run, otherwise, the item is ignored. This means only squirrel items will make it into our menu - just in case some other type of item happened to find its way into the XML file.

Assuming the item is of type squirrel, two more variables are created. These are species and location from where we will extract the text specifying each.

```
var species = items[i].firstChild;
var location = items[i].childNodes[1];
```

The species element is the first child of the item element whereas location is the second child. Now, since there is no "secondChild" property like there is a "firstChild," we're just going to have to use childNodes and an index number to get the second child. Similarly, firstChild could be written as items[i].childNodes[0];. Remember, the text within, text nodes and CDATA, are not a part of the these elements. They exist as their own nodes - as children of those elements - whose text are obtained using the nodeValue property.



[referencing child nodes in first item]

With the species and location variables, it makes it a little more clear where we are and what references what. This will prevent the long stringing of firstChilds and childNodes references as seen in the chart above - or at least reduce them a little. You also get a clearer representation of what non-specific references like items[i].firstChild is - it's species.

Now, before actually using those variables, the menu button will need to be attached. The content we're ultimately extracting with those variables, the text and CDATA nodes, are going to be assigned with that button so it will need to be attached in order for that to happen. That's where the next clump of code comes into play.

```
var item_mc = menu_mc.attachMovie("menu_item","item"+item_count, item_count);
item_mc._y = item_count * item_spacing;
item_count++;
```

If you'll remember, two generic variables were defined earlier in the movie. These were item_count and item_spacing. In attaching a menu item is where these guys come into play. The item_count variable is used to count items as they are added to the menu. You can see that attachMovie is used to attach a "menu_item" symbol from the library into the menu_mc which exists on the main timeline. Item_count is used in attachMovie to not only give the new menu item a unique name, but also provide a unique depth for it to live within the menu_mc.

The attached clip, item_mc, is then positioned based on item_count and its value multiplied by item_spacing. For this example, the item_spacing was set at 28, or about the height of a menu button. When this is combined with an ever-increasing item_count variable, you get vertical positions starting at 0 and counting up by 28 each time item_count increases (item_count starts at 0 so 0 * item_spacing is 0 the first time around). You can see that the attached clip has its _y property set to this combination. Directly below it, item_count is increased so that the next attached button will be properly created and positioned accordingly.

At this point we can now snag the text out from the current item in the loop. This will be both the species text (text node) and the location text (CDATA section). The species text will be assigned as the text in the text field. located within the freshly attached item_mc named species_txt. The location text from the

CDATA section will go into an arbitrary variable assigned within the item button (main_btn). This text is what each item button will send to the location dialog when it is clicked. It does this using the previously defined DisplayInfo function which will will also assign to main_btn here. If you'll remember, DisplayInfo used the location_text variable.

```
item_mc.species_txt.text = species.firstChild.nodeValue;
item_mc.main_btn.location_text = location.firstChild.nodeValue;
item_mc.main_btn.onRelease = DisplayInfo;
```

This is straight-forward assignment. This important part is just seeing how the text is referenced from the XML object - or at least from the variables we have here that each represent a part of that object. As text nodes and CDATA, the text exist as its own node as a child of either species or location. Also, as such, the nodeValue property is what retrieves the text associated with those nodes. So in order to get that text, firstChild.nodeValue is used on each element.

Having set those, there now exists a new fully functional item within our menu. It sports the name of a species of squirrel (assigned to a text field over the button) and when clicked will display that particular squirrel's location information in a separate dialog (sending the text field. of that dialog text defined in the button under location_text). All that remains is loading the XML and putting it into action.

Loading the XML. This should be nothing new. We already did this earlier. The only difference here is that a function is called from the onLoad event which creates the menu desired. Here's the script:

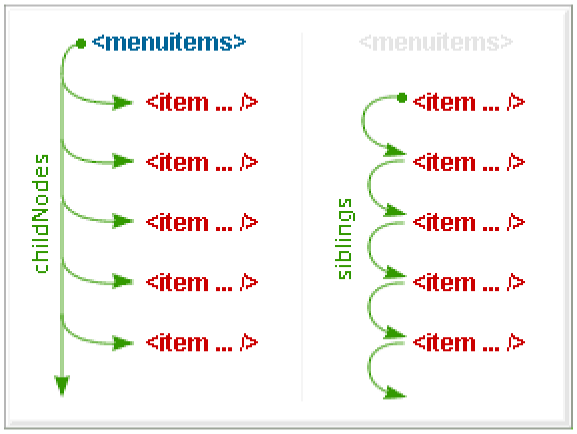
```
var squirrel_xml = new XML();
squirrel_xml.ignoreWhite = true;
squirrel_xml.onLoad = function(success){
    if (success) CreateMenu(this);
    else trace("Error loading XML file");
}
squirrel_xml.load("squirrel_finder.xml");
```

An XML instance is created, the white space is set to be ignored, the onLoad is defined and finally the actual XML document is set to load into the XML instance which will ultimately set off the chain of events that makes what is the squirrel finder. Above an error message is traced if the XML didn't load properly. This will only be seen when working on the movie from within Flash. It will be ignored when viewing the movie on the web (so there, it really serves no purpose).

An Alternate Route
Lets step back a little to the CreateMenu function and the loop that handles each item as it is added to the menu. This particular approach makes sole use of the firstChild and childNodes properties of XML nodes. However, Flash provides more ways to reference XML than just those two. Using other properties, well, namely nextSibling, we can modify the function a little to perform the same action but through a different means.

The current approach cycles through each item in the menuitem element's childNodes array using a for loop. This marks a position in the array and walks through it accessing the next item in the array upon each iteration of the loop. The procedure involves child access through parent.

A different procedure can be implemented - sibling access from another sibling. What this entails is taking the first child of any set of children and continuing through each following sibling until no more remain.



[items via childNodes vs siblings]

This approach uses the nextSibling property to traverse all children and gets rid of the parent reference altogether (aside from using the parent to get a reference to the first child to begin with). But how do you handle this? Surely not from a for loop? That's right; not a for loop but a do..while loop. Here's how that would be set up:

```
var currentSibling = targetParentNode.firstChild;
    do {
        // ... actions on currentSibling
} while (currentSibling = currentSibling.nextSibling);
```

You start off with the first sibling in the set of children of targetParentNode. The do portion of the do..while loop then executes allowing you to operate on that particular sibling immediately. Once that is complete the while condition is checked. This while condition actually does two things. It sets and checks. First currentSibling is assigned to its own nextSibling. This is what will give you the next child in the set. Assuming this currentSibling is valid (i.e. you're not at the last sibling who has no next sibling) then the while loop will loop back through the do block. Otherwise it will terminate and continue with the rest of your script.

The do..while loop, however slick, is not without its shortcomings. Namely, it does not keep an index count of which particular child you're working with. The for loop, for example, gave you an index representing a position in the childNodes array (via the variable "i"). This can be disadvantageous. No one's stopping you from creating your own within the loop. A simple i starting at 0 prior to the loop and and a i++ at the end of a do block will give you this, but is it worth it then? You might as well just use a for loop then. Then again, is it ever worth it? You may never use a do..while to access children this way; it may seem confusing or just unconventional. Maybe, but it is an alternative approach and worth noting.

Here is what the CreateMenu function would like using a do..while and nextSibling to access the squirrel finder menu's items.

```
function CreateMenu(menu_xml){
    var currItem = menu_xml.firstChild.firstChild.firstChild;
    do {
        if (items[i].attributes.type == "squirrel") {
            var species = currItem.firstChild;
            var location = species.nextSibling;

            var item_mc = menu_mc.attachMovie("menu_item","item"+item_count, item_count);
            item_mc._y = item_count * item_spacing;
            item_count++;

            item_mc.species_txt.text = species.firstChild.nodeValue;
            item_mc.main_btn.location_text = location.firstChild.nodeValue;
            item_mc.main_btn.onRelease = DisplayInfo;
        }
    } while (currItem = currItem.nextSibling);
}
```

Example: XML Portfolio
The squirrel finder maintains all of its content within the XML file. It's just text and easy to include. However, the XML file can't always facilitate all of your needed content. Take images for example. Let's say you want to have some jpegs loaded in for each item of interest within your XML document. What then? Well, then you can just include a reference (i.e. URL) to the image within the XML and have Flash include the image from that. Good news. This can also be applied to text. So, basically, your XML can consist almost solely of URLs. This XML portfolio(-esque) example does pretty much just that.

Example doesn't paste. Please see http://www.kirupa.com/web/xml/examples/portfolio.htm for the example.

Download ZIP

Basically the same concepts apply here that did with the squirrel finder in terms of looping through children to add the content listed. This particular example simplifies the interface a little bit by having everything visible on screen at the same time but takes an additional step in content retrieval.

The advantage to doing this, aside from the fact that you can't have jpegs within your XML, is that it keeps your XML smaller and lets you load content on request. This means that the only data you need to initially load when your movie starts is a small collection of URLs. No other text or images are loaded prior to the point at which a viewer requests them. This ultimately saves on bandwidth and load time since it doesn't require everything to preload even when it might not even be seen.

XML Structure
The XML document contains mostly URLs aside from one attribute, title. Here's the document

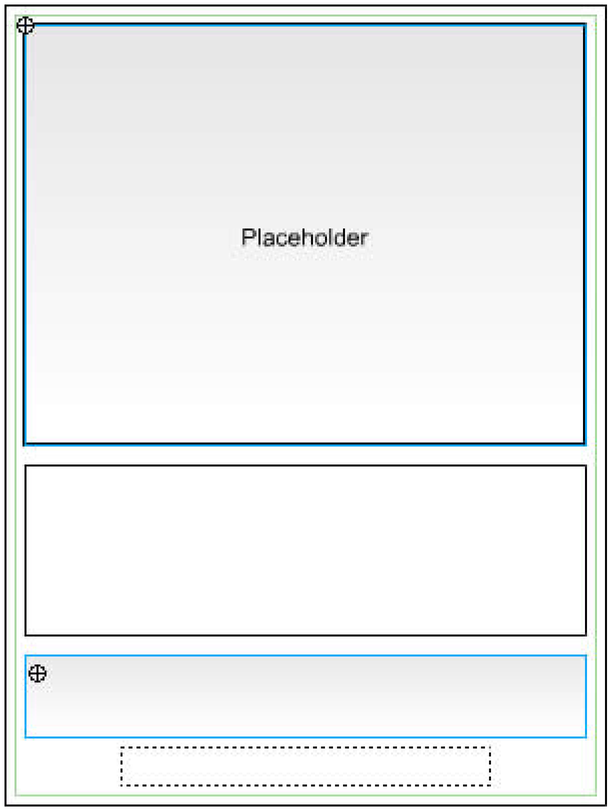
portfolio.xml

Here it is in its entirety:

```
<?xml version="1.0"?>
<portfolio>
    <picture title = "Tiny Disk"
        thumb = "portfolio_images/thumbs/tinydisk.jpg"
        description = "portfolio_text/tinydisk.txt"
        image = "portfolio_images/tinydisk.jpg" />

    <picture title = "Plug"
        thumb = "portfolio_images/thumbs/plug.jpg"
        description = "portfolio_text/plug.txt"
        image = "portfolio_images/plug.jpg" />

    <picture title = "Disk Collection"
        thumb = "portfolio_images/thumbs/diskcollection.jpg"
        description = "portfolio_text/diskcollection.txt"
        image = "portfolio_images/diskcollection.jpg" />
</portfolio>
```

Simple. You got yourself a root element portfolio that contains a list of picture elements. Picture elements then contain a title and 3 urls to that picture's content. You got a thumbnail image, a description and a full sized image. To help with clarity, some indentation was used to present the attributes of the picture tags in a more readable manner. This will not affect the XML negatively in any way - Flash will read it just fine.

Preparing the Flash File
In terms of the setup of the movie, the main screen consists of only four elements. Two are movieclips for loading the images (one for thumbnails and the other for large versions). The other two are text fields (one to display the title and the other for the description text).



[layout of portfolio movie]

Nothing special there. Lets move on to the scripting.

ActionScript

The ActionScript is set up much like the squirrel finder. You load the XML and then have a function called that generates content based on that information. The only difference here is that the interaction added uses functions that call loadMovie and load with a loadVars object to reach out and retrieve the external content as specified in the XML document.

Setting some variables: here a spacing variable for laying out thumbnails is defined along with an instance of LoadVars to handle loading in text from a URL.

```
var thumb_spacing = 40;

var description_lv = new LoadVars();
description_lv.onData = function(raw_text){
	description_txt.text = raw_text;
}
```

Notice that the LoadVars instance doesn't have an onLoad defined, but rather, an onData. The onData event is called with the raw contents of the file make it into Flash. When that happens, the call gets passed the raw data that was loaded loaded - direct text prior to parsing. This onData, by default, as it is defined initially in Flash, then takes that data and parses it as needed and then calls the onLoad event which you should have at this point have defined for the instance. You can, however, as shown above, write your own onData handler to catch that raw data as its loaded directly into Flash. This will prevent an onLoad call (since the internal onData is the method that calls onLoad) but it lets you deal directly with what the data loaded is before it is parsed. This applies to both LoadVars and XML where LoadVars parses the text into Flash variables and

XML parses the text into XMLNode instances. Here, it is being used to load straight text files into a text field. (description_txt). If you wanted, you could also code your own XML parser and use an onData for your XML instance to intercept the raw XML text in order to parse it with your own parser and not Flash's. But why would you ever want to do a crazy thing like that? Ok, some over ambitious people out there probably have their reasons. But, for the most part, Flash's own parser does fine.

Next is the main function for setting up the portfolio based on the loaded content.

```
function GeneratePortfolio(portfolio_xml){
    var portfolioPictures = portfolio_xml.firstChild.childNodes;
    for (var i = 0; i < portfolioPictures.length; i++){
        var currentPicture = portfolioPictures[i];

        var currentThumb_mc = menu_mc.createEmptyMovieClip ("thumbnail_mc"+i,i);
        currentThumb_mc._x = i * thumb_spacing;

        currentThumb_mc.createEmptyMovieClip("thumb_container",0);
        currentThumb_mc.thumb_container.loadMovie (currentPicture.attributes.thumb);

        currentThumb_mc.title = currentPicture.attributes.title;
        currentThumb_mc.image = currentPicture.attributes.image;
        currentThumb_mc.description = currentPicture.attributes.description;

        currentThumb_mc.onRollOver = currentThumb_mc.onDragOver = function(){
            info_txt.text = this.title;
        }
        currentThumb_mc.onRollOut = currentThumb_mc.onDragOut = function(){
            info_txt.text = "";
        }
        currentThumb_mc.onRelease = function(){
            image_mc.loadMovie(this.image);
            description_lv.load(this.description);
        }
    }
}
```

You got your basic loop which goes through the picture child nodes of the portfolio root element creating a movie clip for the thumbnail and adding actions to handle rollovers and releases. Two movieclips are actually created for the thumbnails. One represents the actual thumbnail movie clip and another is created within it used to load the external thumbnail jpeg. This is needed because whenever you use loadMovie on a movie clip, that movie clip loses all variables and functions assigned to it once the content (movie or image) loads. This means that without that inner movie clip to load the jpeg into, all the variables and event handler functions we assign to the movie clip right after the loadMovie call would be cleared once the jpeg has loaded itself in.

Attributes are assigned to each thumbnail movie clip and handlers for mouse interaction are then defined. The rollover functions manage the info_txt text field. which displays the title of the picture when the mouse is over the thumbnail while a release handler uses loadMovie to load the main image into the placeholder movie clip (image_mc) and load on the LoadVars instance to get text into description_txt (as instructed in its onData handler).

Then you setup and load your XML:

```
var portfolio_xml = new XML();
portfolio_xml.ignoreWhite = true;
portfolio_xml.onLoad = function(success){
    if (success) GeneratePortfolio(this);
    else trace("Error loading XML file");
}
portfolio_xml.load("portfolio.xml");
```

Now you got yourself a simple portfolio which is very dynamic and easily editable without the need for even touching Flash (thanks to XML and external content). Change a few text files, swap out a few images and you're set.

Nested Loops

Simple collections of data in XML is fairly easily traversed. So far we've covered to basic ways to do this. One was the more common for loop going through a childNodes array and the other a do..while loop that cycled through siblings. The structure of an XML document is not always as simple as that in the squirrel finder, though. Often you could be dealing with nested collections of elements within other nested collections of other elements. Not only would you have to loop through one collection, but possibly each collection within that collection. For this, you would need to use nested looping.

Just like XML elements can be nested, you too can nest your loops in Flash to facilitate this added complication in an XML document's structure. Depending on how deep your XML goes may decide just how much looping you'll need to do and how many you'll need to nest. For example, take the following:

```
<pirates>
    <pirate name="Black Beard">
        <sayings>
            <saying phrase="Argh!" />
            <saying phrase="Shiver me timbers" />
        </sayings>
    </pirate>
    <pirate name="Francis Drake">
        <sayings>
            <saying phrase="Avast!" />
            <saying phrase="Polly want a cracker?" />
            <saying phrase="Well blow me down" />
        </sayings>
    </pirate>
</pirates>
```

Here we're dealing with 2 element sets that run 2 levels deep; pirates and sayings within individual pirates. Understand that these sayings are not historically accurate. I'm pretty sure Black Beard said "Avast!" just as much as the next guy. Either way, in order to get all the sayings of all the pirates, you would need to first loop through each pirate, then, while on each, loop through their sayings - a loop within a loop. You can potentially have many nested loops, but the fewer the better. Here is a quick code snippet of what you might use to loop through the above pirates XML:

```
var pirates = pirates_xml.firstChild.childNodes;
for (var p=0; p<pirates.length; p++){

    var pirate = pirates[p];
    var sayings = pirate.firstChild.childNodes;

    for (var s=0; s<sayings.length; s++){
        var saying = sayings[s];
        trace(pirate.attributes.name +" says \""+ saying.attributes.phrase +"\"");
    }
}
```

This would output the following in Flash:

```
Black Beard says "Argh!"
Black Beard says "Shiver me timbers"
Francis Drake says "Avast!"
Francis Drake says "Polly want a cracker?"
Francis Drake says "Well blow me down"
```

Of course should you need to implement nested for loops, you probably won't just be tracing the information. More than likely you'll be converting it into a more usable and ActionScript-centric form that works better with the setup of your movie. How that is done depends on your movie.

Searching XML
What about searching? We already know how to get through an XML document. With that knowledge, it can't be hard to find things within it. And really, it's not. You just need to know how to get through your XML (which should be easy as pie now), know what to look for and where.

How, what, and where are often variable and can change based on your preferences or the setup of your XML file. When you search, do you want to check *everything*? Are there only certain elements/text nodes that should be checked? Are you searching only within elements of a certain type or designation? These are all things you'll need to consider when setting up a method to handle your search.

Once you got what you're going to do and how the search will be handled, its then just a matter of setting up a way through your XML and identify items relating to the search terms much in the way you would search any other body of text.

Example: Searching XML
Below is an XML file that contains a few posts from the Best of Kirupa.com forum on KirupaForum.com. These are a bunch of posts (ok, only ten but I didn't feel like adding that many) that no one probably wants to read all at once, especially when they're looking for something in particular. It would be nice to just pull out only the post that relates to whatever it is you are trying to find. For that, a search is required.

Depending on the XML being searched, you will need to decide how you want to approach your search. Either way you look at it, searching XML means going through all elements of that XML (or all that are of interest) and checking its content for whatever is being searched. The XML:

bestofposts.xml

It starts off looking like (the entire file is not shown here):

```
<bestof url="http://www.kirupaforum.com/forums/forumdisplay.php?f=12">
    <post url="http://www.kirupaforum.com/forums/showthread.php?t=43216">
        <title>xml menu DONE :)</title>
        <author>hga77</author>
        <message>
            <![CDATA[Happy New Year kirupa....Hope all you ppl had a great year and didnt stick
            to flash too much

            Anyways I just managed to finish the xml menu i've been working on...

            I've attached it here incase any1 needs something like this. Use it as you like. Let me
            know if you make any additions to it

            UPDATE(04/2004): you can find v2 for this menu on page 3 post #31
            UPDATE(07/2004): you can find v3 for this menu on page 15 post #221]]>
        </message>
    </post>
.
.
.
```

Each post following this one maintains the same format. As you can probably tell, we should be able to get along fine just using loops. Two will be needed; one nested within the other. One will loop through each post while the other will loop through each element within each post checking it for a match with whatever was searched. The final result is:

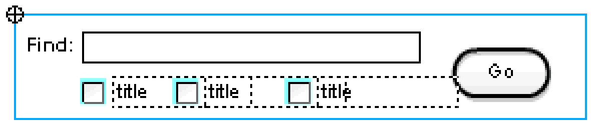
Example did not paste. Please see http://www.kirupa.com/web/xml/examples/searchbestof.htm for working example.

Download ZIP

Preparing the Flash File

The file consists of 2 parts, the display area and the search area. The display area shows the XML (in a text field called results_txt), or at least the XML elements that were found in any given search. The search area shows the search input field, some options and the go button.

Because the XML is external and because it has to be loaded to be searched, the search area is self-contained within a movie clip and hidden from view until the loading of the XML is complete. This prevents errors that might have occurred if someone were to search for something in XML that just isn't there.



[search_fields movie clip symbol]

When the go button is pressed, the value of the find text field (query_txt) will be sent to a function which will find that value in the loaded XML within any of the elements whose checkbox is selected. Note that each checkbox above says title because their names are dynamically created at runtime (each of the three are the same symbol, each relating to a different element as specified in code).

ActionScript
There are a couple of processes working together to complete the search provided within this example. You have a function to determine what is being searched (where). There's the actual search operation. There's a function used to put the nodes found to have the query in the text field, and then there's a function used to highlight those results in the text displayed.

There's also a custom scrollbar for the search results. I won't go into detail about the scrollbar here, the graphics nor the scripting. It's just like any other custom scrollbar used to scroll content. If you're interested in how its made, you can study the code from the Flash source file. Here, the concentration will be on working with the XML aspects of the example. It will pop up in some other examples as well.

As usual, we'll start with the XML instance used for this example since everything starts with and revolves around its existence.

```
var posts_xml = new XML();
posts_xml.ignoreWhite = true;
posts_xml.onLoad = function(success){
    if (success){
        search_fields._visible = true;
    }else results_txt.text = "Error loading XML";
}
search_fields._visible = false;
posts_xml.load("bestofposts.xml");
```

The posts_xml variable here represents the XML instance. It immediately loads the external file "bestofposts.xml" right after the search_fields movie clip has its _visible set to false. Once the loads, in the onLoad event, assuming the loading was successful, search_fields is made visible again and the user can begin to search the data. Otherwise, an error is given in the results_txt text field and you're basically out of luck.

With search_fields visible, the search function is active. This is initiated by the go button on the right. The go button contains the list of commands (functions) that were mentioned earlier. When pressed and released, it:

- checks for a valid search query (the example limits searches to terms at least 3 characters in length).
- determines which parts of the XML is to be searched (as specified with the checkboxes).

- performs the search in the XML with the query provided.
- displays the results in the results text field.
- highlights the search term within the results.

That results in the following for the button's onRelease event:

```
search_fields.search_btn.onRelease = function(){
    if (search_fields.query_txt.text.length < 3){
        results_txt.text = "Please use a search term with 3 or more characters.";
        return (0);
    }

    var searchElements = ElementsToSearch();

    var nodesWithQuery = SearchXML(
        posts_xml.firstChild.childNodes,
        search_fields.query_txt.text,
        searchElements
    );

    if (nodesWithQuery.length){
        DisplayNodes(
            nodesWithQuery,
            results_txt
        );
    }else{
        results_txt.text = "No results for "+search_fields.query_txt.text+".";
        return (0);
    }

    HighlightOccurences(
        search_fields.query_txt.text,
        results_txt,
        search_highlight
    );
}
```

First the check for the valid search query. Since we're only checking for queries with lengths of 3 or greater, this is handled with a simple if condition.

```
if (search_fields.query_txt.text.length < 3){
    results_txt.text = "Please use a search term with 3 or more characters.";
    return (0);
}
```

If the search query (search_fields.query_txt.text) is not at least three characters in length, an error message is displayed and a return statement exits the onRelease preventing anything else from being run.

Next, a function is used to determine which elements within the XML need to be searched.

```
var searchElements = ElementsToSearch();
```

ElementsToSearch is defined as:

```
ElementsToSearch = function(){
    var childElementsToSearch = [];
    if (search_fields.title_check.checked){
        childElementsToSearch.push("title");
    }
    if (search_fields.author_check.checked){
        childElementsToSearch.push("author");
    }
    if (search_fields.message_check.checked){
        childElementsToSearch.push("message");
    }
    return childElementsToSearch;
}
```

What this does is is runs through the checkboxes within the search_fields movie clip, adding the associated element node name to an array if the checkbox is checked. This array is then returned and assigned to searchElements to be used within the search function.

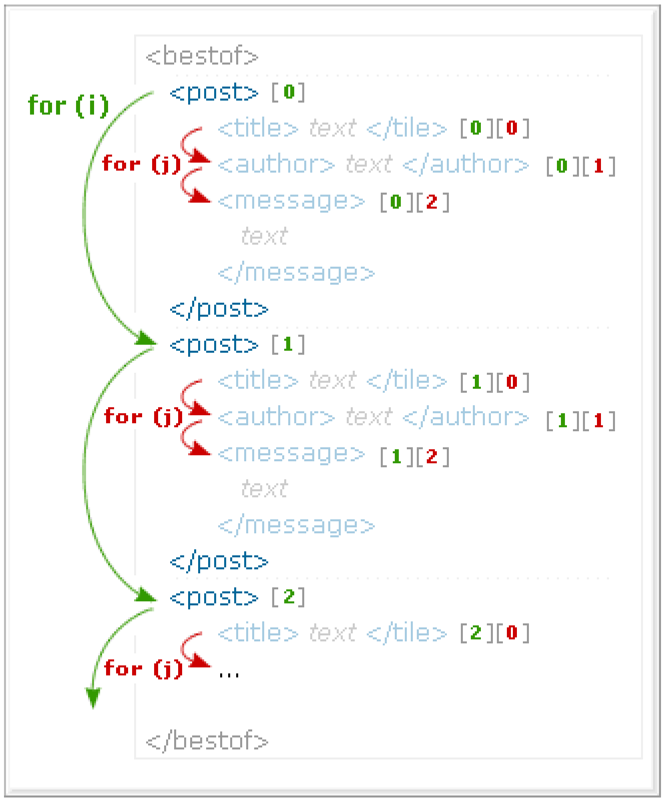
The search is what comes next. The function handling the search is called SearchXML. It takes an array of elements to be searched (a childNodes array), what is to be searched and an array of elements within the elements searched... to be searched.

```
var nodesWithQuery = SearchXML(
    posts_xml.firstChild.childNodes,
    search_fields.query_txt.text,
    searchElements
);
```

posts_xml.firstChild.childNodes represents the list of post elements within posts_xml. Each of those post elements contains title, author and message elements. Which of those to be searched is dictated by the searchElements array originally obtained by the ElementsToSearch function. As for the SearchXML function, it comprises of the following:

```
SearchXML = function(nodes, query, useChildElements){
    var results = [];
    for (var i=0; i<nodes.length; i++){
        for (var j=0; j<nodes[i].childNodes.length; j++){
            currNode = nodes[i].childNodes[j];
            if (useChildElements.contains (currNode.nodeName)){
                if (currNode.firstChild.nodeValue.contains (query)){
                    results.push(nodes[i]);
                    break;
                }
            }
        }
    }
    return results;
}
```

Using 2 for loops, one nested in the other, this takes and places all post elements that contain the query in a returned results variable. The first loop which uses the i variable as an iterator loops through all post elements passed in as the nodes variable (set to posts_xml.firstChild.childNodes when called). The second loop, using j, cycles through all of each of the child nodes within each of those elements; in this case, title, author and message.



[nested for loops for post elements]

Inside those loops, as looping through each post element, the useChildElements array (searchElements) is checked to see if that particular node is to be searched by seeing if its nodeName exists within it. If so, the following if statement does the actual searching through that element's text Node for the query. If that exists, then the post element is added to the results array and the inner for loop (j) is broken out of in order to immediately begin searching the next post element. When all post elements are searched, the results array is returned and assigned to the nodesWithQuery variable back in the go button's onRelease event.

You may not recognize the two functions (though with a similar name) used to check the useChildElements array and the text node for the search query. These are the Array and String contains methods. You probably don't recognize them because they're custom methods defined to work with the String and Array objects. They are defined within the movie as:

```
String.prototype.contains = function(searchString){
    return (this.indexOf(searchString) != -1);
}
Array.prototype.contains = function(searchValue){
    var i = this.length;
    while(i--) if (this[i] == searchValue) return true;
    return false;
}
```

They're simple enough that they could be written directly within the Search XML function, but separating them like this keeps SearchXML cleaner and more understandable. Its much easier to tell what's going on when you read *if string contains word* versus *if string indexof word != -1* when determining when a string (word) exists within another. (If you're unfamiliar with indexof for Strings, it returns the location of one string within another. If the string does not exist within the other string, -1 is returned.)

Getting back to the original go button's onRelease, the next item up for bid is the displaying of the results within the results_txt text field.

```
if (nodesWithQuery.length){
    DisplayNodes(
        nodesWithQuery,
        results_txt
    );
}else{
    results_txt.text = "No results for "+search_fields.query_txt.text+".";
    return (0);
}
```

An if statement is needed here because we need to check to see whether or not there are any results to display. If there are, call DisplayNodes to put the nodesWithQuery elements into results_txt. If not, then display an error message there instead. DisplayNodes is defined as:

```
DisplayNodes = function(nodes, field_txt){
    field_txt.htmlText = "";
    var entry;
    var separator = "<br>________________________<br><br>";
    for (var i=0; i<nodes.length; i++){
        entry = "";
        entry += "<b>"+ nodes[i].childNodes[0].firstChild.nodeValue +"</b>";
        entry += " by: "+ nodes[i].childNodes[1].firstChild.nodeValue;
        entry += "<br>"+ nodes[i].childNodes[2].firstChild.nodeValue;
        if (nodes[i].attributes.url.length){
            entry += "<br><a href='" + nodes[i].attributes.url;
            entry += "'><font color='#0000FF'>Read more...</font></a>";
        }
        field_txt.htmlText += entry + separator;
    }
}
```

DisplayNodes simply formats the XML to be displayed in the text field (using HTML).

At this point the search is complete. All nodes with the query have been found and displayed in the results_txt. To make the results a little more convenient, though, I've added a little function that goes through and highlights the search terms within the results. This is the HighlightOccurences function.

```
HighlightOccurences(
    search_fields.query_txt.text,
    results_txt,
    search_highlight
);
```

It's defined as:

```
search_highlight = new TextFormat();
search_highlight.color = 0xFF0000;
search_highlight.italic = true;

HighlightOccurences = function(str, field_txt, format){
if (!str.length) return (0);
var start = field_txt.text.indexOf(str);
var end = start + str.length;
while (start != -1){
field_txt.setTextFormat(start, end, search_highlight);
start = field_txt.text.indexOf(str, end);
end = start + str.length;
}
}
```

It uses a TextFormat instance to make all the str (query) instances within the field_txt text field (results_txt) turn red - or at least whatever the format passed specifies. And that format you can see defined above the function does in fact set text red.

Recursive Functions
Ok, so looping my not always be the best idea. And yeah, sometimes you're going to have complicated XML that will go levels and levels deep. Sometimes you can't help this - but also you don't want to have to make for loops to handle all that. Have no fear; there are other solutions. One is using recursive function calls to handle like data no matter how nested it is within your XML structure.

A recursive function is a function that calls itself. This is, itself, a form of looping. If you call a function that calls itself, then, as it runs, it will call itself - a function which, since its the same function, also calling itself... which calls itself and so on and so forth until something prevents the function from calling itself again and all the previous function calls resolve. Its just one big function loop. For example, Math.pow, the function that raises a number to a specific power can be written as a recursive function:

```
Math.pow = function(num, power){
    return (power > 0) ? num*Math.pow(num, power-1) : 1;
}
```

Math.pow, in this execution, checks to see if pow is greater than 0. If so, it returns the passed number times itself to the power of the power passed minus 1 (by calling itself). Effectively, what this does is counts power down until its 0 returning that many multiplications of the num argument times itself - one for each call of Math.pow. In the end you have num to the power-th power.

Recursive functions help you deal with problems that are compounding or require like actions to be performed repeatedly during a single operation - much like looping. However, unlike looping, the nesting of recursive functions is automatic since each function call already contains another call to itself. This provides for a theoretically infinite number of nested calls, though, in Flash, you're limited to using only 256. Loops only become as nested when you write them out to be. That flexibility and scalability is what make recursive functions useful.

When dealing with XML recursion can be helpful if you are unsure how deep your XML may go - how many levels of children does each element have? Though often you will (should) know exactly how your XML is structured, sometimes you won't. An example would be a dump of all the files and folders (and the files and folders within each folders) that are within a certain directory of choice. This directory could be completely empty or it could have folders upon folders of files and other folders. You really don't know. Using a recursive function to list out each folder's contents would work really well since once you've reached another folder, you just call the function again to list out it's contents, repeating the process until all subdirectories have been listed.

Example: Flash MX 2004 Classes Directory
This example will list out the contents of the Classes folder in the Flash MX 2004 install directory. Can you guess the format of this information? Do I even need to ask? Yup, XML. The XML contains the full contents of the Classes folder, each file and folder and each file and folder within those folders and so on until there's no more files or folders left to file or fold. A recursive function (which even includes a loop on top of everything else) is then used to list out the contents of each folder and is re-called on if any item is itself a folder thereby listing its contents. And this happens until there is nothing left to file or fold!

As an added bonus to this example, we'll make the folders expandable and collapsible. This way you're not forced to look at all of the files at once; you can be selective. And we all know selection is a good thing. When all done and through, you get the following end result:.

Example did not paste. Please see http://www.kirupa.com/web/xml/examples/mx04ASclasses.htm for working example.

[flash mx 2004 classes directory]

Download ZIP

Note: Class Listing Lag

Given the sheer number of icons attached for this example, there will be a little bit of lag when the XML is loaded and the directory listing is created. As a precaution, should you ever decide to create something similar, you may wish to start with all directories closed loading in the icons only when that directory is opened. For simplicity's sake, here, it was just easier to add them all at once.

XML Structure
There technically isn't much to the XML that makes up this example. You're dealing with files and directories, each with a name and type where type determines icon. That's basically two different kinds of elements with two different kinds of data. The most important aspect of the XML is the structure. Folders contain files and other folders so folder elements will have file and folder element children. Directories within other directories can too have their own files and directories and so on and so forth. So what you get is a lot of nested directories. Here's the XML:

mx04_classes.xml

It should look a little something like the following (the entire file is not shown here):

```
<?xml version="1.0"?>
<directory name="Classes" type="directory">
<file name="Accessibility.as" type="actionscript" />
<file name="Array.as" type="actionscript" />
<file name="AsBroadcaster.as" type="actionscript" />
.
.
.
```

Starting from the classes directory in a Flash MX 2004 install, you get a listing of all the files and directories that are within complete with name and type.

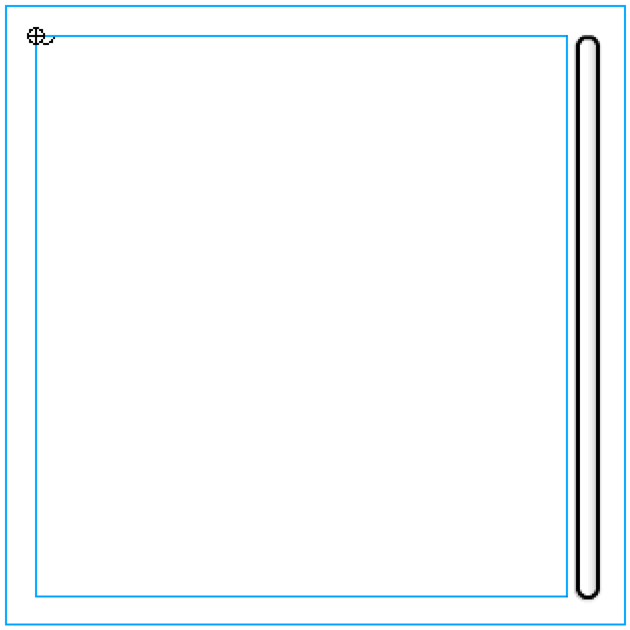
To generate this list, I used this simple PHP script:

```
<?php
function ExportDirectoryXML($base, $indent){
    if ($dir = @opendir($base)){
        while ($file = readdir($dir)){
            if ($file{0} != "."){
                if (is_dir($base . "/" . $file)){
                    echo "$indent&lt;directory name=\"$file\" type=\"directory\"&gt;\n";
                    ExportDirectoryXML($base . "/" . $file, $indent."\t");
                    echo "$indent&lt;/directory&gt;\n";
                }else echo "$indent&lt;file name=\"$file\" type=\"actionscript\" /&gt;\n";
            }
        }
        closedir($dir);
    }
}

$directory = "[DIRECTORY PATH GOES HERE]";

echo "<html><body><pre>\n";
ExportDirectoryXML($directory, "");
echo "\n</pre></body></html>";
?>
```

This generated the directory structure in markup which I then saved it to a XML file. It's that file which is loaded by the Flash movie (the movie loads a files separate from the script though such a script could be used to load XML like this dynamically from a server). If you'll notice, it uses a recursive function as well.

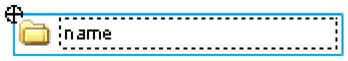
Preparing the Flash File
This Flash file is really no more complicated than the others before it, at least not in setting up. Scripting is a little more involved. Otherwise, in terms of screen elements, you're not dealing with much more than about 3 different elements: a movie clip for containing the full file listing, an attached clip to represent a directory item (a file or directory), and a scrollbar.

The other two parts are the directory items and the movie clip to contain them. The movie clip used to contain them is just an empty clip called listing_mc. Nothing much in terms of preparation goes into that. However, to keep it masked, it is placed under a graphic that hides content scrolling inside. listing_mc itself isn't *masked* masked, it's just being hidden by being beneath this graphic.



[listing_mc beneath movie graphic interface]

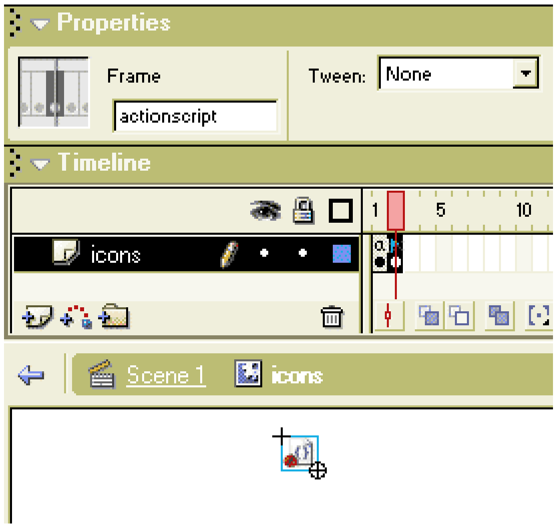
Next it's just a matter of making the movie clip that is to be attached and representative as a directory item. This will be used for both files and directories when added to the listing_mc. This movie clip has three parts: an icon, a text field for a name, and an underlying button used to select it. This example uses the button to open and close directories only, but ideally it would add interaction to the files as well.



[item mc for each directory item]

Take note that the registration point for this movie clip is in the upper left. This will be important later on when positioning each item in the display. This will be linked as "directory_item" in the library so that it can be attached dynamically when needed.

Since this is used for all directory items, files (of any kind) and directories, it will need to facilitate the needs and differences of each. The text field is no problem since, as dynamic text, it can be changed easily at any time. The icon, however, is a little more difficult. The icon is actually another movie clip with its own collection of frames inside. These frames contain each of the icon graphics needed for the file listing, the first frame being a directory. For this example, there's only one other file type so frame 2 is an ActionScript file icon. Each frame also has a frame label named after its icon type. This allows navigation to any particular icon much more straight forward using a name rather than a number. This name, conveniently, should reflect the string used to identify type in the XML document.



[label frames for each icon based on type in xml]

Now, whenever one of these clips is attached as a directory item, it can be given a name and have its icon switched to whatever type is necessary.

ActionScript
For the most part with this example, you get another squirrel finder script-wise. This too revolves around generating a list of movie clips or buttons based on XML. However, here, recursive functions are going to be used to allow directory listings to be created for each other directory within another directory listing. So it's like taking the squirrel finder and change some of its buttons to be other instances of a squirrel finders.

The most difficult aspect of the file listing is arranging the items in the view. Because of the ability to open and close directories, it means that whole groups of items will have to move either up or down in response to more items being displayed from the opening or closing of directories. Instead of attaching items using a variable to control spacing, a new technique is used. This technique positions based on bounding area moving lower items to the bottom of the items above them. This way, should any items be added or removed, basically, you just move the item directly below that change where all items below it can just reposition themselves based on where they should be in respect to the the position of item above them.

We'll get to that script in a second, but first lets look at the definition of the XML instance.

```
var directory_xml = new XML();
directory_xml.ignoreWhite = true;
directory_xml.onLoad = function(success){
    if (success){
        GenerateFileListing(this, listing_mc);
        scrollbar.setTarget(listing_mc, view_mc._y, view_mc._height);
    }else trace("Error loading XML file");
}

directory_xml.load("mx04_classes.xml");
```

The directory_xml variable represents the XML instance. It loads "mx04_classes.xml" using load and has an onLoad which, upon success, runs a function called GenerateFileListing passing in itself and the listing_mc (and of course there's that scrollbar action in there too).

```
GenerateFileListing(this, listing_mc);
```

As you might imagine, GenerateFileListing creates a listing of files based on xml (this) in a movie clip (listing_mc). Here is what GenerateFileListing looks like:

```
function GenerateFileListing(directory_node, target_mc){
    var directory_mc = target_mc.createEmptyMovieClip("directory_mc", 1);
    SetAtBottomOfParent(directory_mc);
    directory_mc._x += item_indent;
    directory_mc.depth = 0;

    var contents = directory_node.childNodes;
    var item_mc, last_item_mc;
    for (var i=0; i<contents.length; i++) {
        item_mc = directory_mc.attachMovie ("directory_item","item"+directory_mc.depth,
            directory_mc.depth);
        directory_mc.depth++;

        item_mc.name_txt.text = contents[i].attributes.name;
        item_mc.icon_mc.gotoAndStop(contents[i].attributes.type);

        if (last_item_mc == undefined) directory_mc.firstChild = item_mc;
        else item_mc.previousSibling = last_item_mc;
            last_item_mc.nextSibling = item_mc;
            last_item_mc = item_mc;

        if (contents[i].attributes.type == "directory"){
            item_mc.select_btn.onPress = Fold;
            GenerateFileListing(contents[i], item_mc);
        }

        ArrangeContents(directory_mc);
    }
}
```

The basic break down is as follows:

i. create and position an empty movie clip to hold the directory's contents
ii. loop through the children of the node passed (representing the directory)
    a. attach an item movie clip for each child in that node
    b. assign name and other properties and change icon based on item type
    c. if type is a directory, call the function again passing in the node representing the item and the clip created for it
    d. position the item

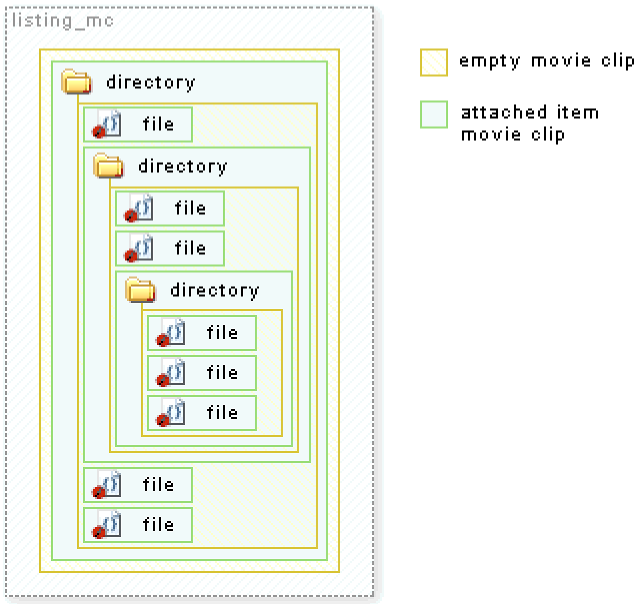
Due to the fact that movie clips are being used as directories (and files) and since movie clips can contain other movie clips within them, it seems perfect sense to maintain a directory's own contents by keeping them within that directory's movie clip as child movie clips. However, given that the option of hiding those child movie clips exists, it would be further beneficial to keep all of those child movie clips within one single movie clip (as a child of the directory). That would allow us an easy way to get rid of every single one of those movie clips at any given time simply by removing hiding or that single movie clip in which they all exist.

This is the initial command of GenerateFileListing, creating that empty movie clip for directory contents to exist.

```
var directory_mc = target_mc.createEmptyMovieClip("directory_mc", 1);
SetAtBottomOfParent(directory_mc);
directory_mc._x += item_indent;
directory_mc.depth = 0;
```

It does this within the passed target_mc which, initially, is the listing_mc on the main timeline. Other commands such as SetAtBottomOfParent and the assignment of _x (based on item_indent, a variable defined earlier on the main timeline) manage its position while a depth variable is set to manage the depths of clips being attached within.

So right away, when the XML first loads, before listing_mc gets any item movie clips attached to it, it gets an empty clip which is to be treated as a content holder for a collection of directories and files within a directory. This means listing_mc itself only gets one movie clip added to it. All contents within listing_mc end up going in that movie clip. Also, every directory item thereafter will too get its own empty movie clip when a recursive call to GenerateFileListing is made to generate its file contents. What you end up getting is something along the lines of this in terms of movie clip structure:



[movie clips in movie clips for directory structure]

Now, if you'll imagine removing or hiding any one of those movie clips holding one of those directory's contents, you can see how the effects would be similar to closing that directory in a directory view which is exactly what we're after.

All that remains is getting the contents in each directory as needed, and that's what the following loop lets us do.

```
var contents = directory_node.childNodes;
var item_mc, last_item_mc;
for (var i=0; i<contents.length; i++) {
    // ...
}
```

First the child nodes are put into a friendly variable named contents. These are the nodes within the current directory node initially passed into GenerateFileListing. A few other variables being used within the loop are also declared here for safe keeping. Then the for loop is used to cycle through each.

In the loop, each item, directory or file, is immediately created within the directory_mc movie clip (the empty one created to hold all contents of a directory).

```
item_mc = directory_mc.attachMovie("directory_item", "item"+directory_mc.depth,
directory_mc.depth);
directory_mc.depth++;
```

Then, the text and icon are set for that item.

```
item_mc.name_txt.text = contents[i].attributes.name;
item_mc.icon_mc.gotoAndStop(contents[i].attributes.type);
```

This is pretty straight-forward. Simply take the text from attributes the current node of the loop. The text comes from the name attribute and is placed into the name_txt text field while the icon_mc is told to go to and stop on the frame label specified by the type attribute. Directly after this, however, are some assignments that are less obvious.

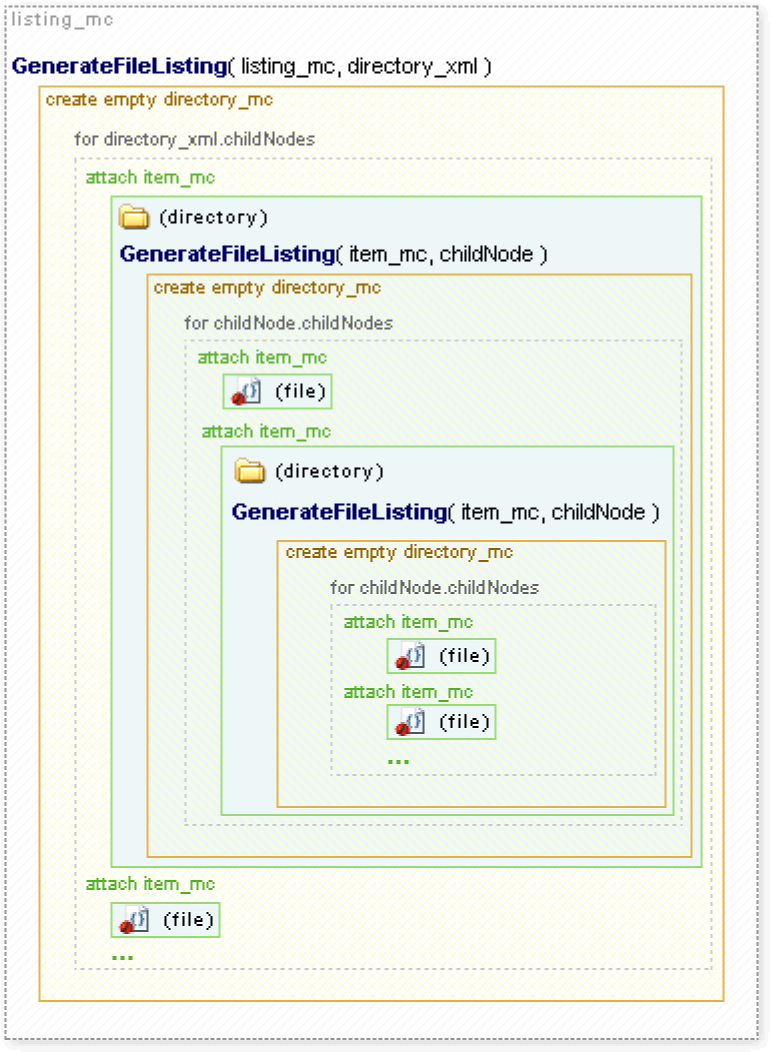
```
if (last_item_mc == undefined) directory_mc.firstChild = item_mc;
else item_mc.previousSibling = last_item_mc;
last_item_mc.nextSibling = item_mc;
last_item_mc = item_mc;
```

What we have here is actually the borrowing of naming conventions used by XML to help handle the movie clip positioning of child movie clips within a directory. Since item movie clips have to be aware of their preceding item in order to position themselves to it, a reference to that item is added within the movie clip. This is defined as previousSibling and handled through the last_item_mc variable defined prior to the loop which following this assignment becomes the current item. Similarly, nextSibling is used to be able to start at the top of a list of directory items and work our way down in order to perform the positioning of each in the correct order. This will work much in the same way the alternative approach to the squirrel finder example did. The directory_mc too gets a reference to a movie clip, the first in the list (when last_item_mc is undefined, it means the current item is the first) in order to be able to start a chain of positioning beginning with it's first item listed. Putting this into motion is handled by four other functions that are used for directory opening and closing.

Next in the loop is the recursive call, or at least the check to see if a recursive call to GenerateFileListing is needed. When it is needed is when the type of the current item is of type "directory."

```
if (contents[i].attributes.type == "directory"){
    item_mc.select_btn.onPress = Fold;
    GenerateFileListing(contents[i], item_mc);
}
```

If so, a button action is added (using Fold, covered shortly) and the call is made. This particular call to GenerateFileListing uses the current node in the loop and the current item movie clip. Each exists within the previous arguments passed in to GenerateFileListing during the current call. So here, and in many cases, the nesting of the function calls reflects the nesting of the objects in which it affects.


listing_mc
GenerateFileListing(listing_mc, directory_xml)
create empty directory_mc
for directory_xml.childNodes
attach item_mc
(directory)
GenerateFileListing(item_mc, childNode)
create empty directory_mc
for childNode.childNodes
attach item_mc
(file)
attach item_mc
(directory)
GenerateFileListing(item_mc, childNode)
create empty directory_mc
for childNode.childNodes
attach item_mc
(file)
attach item_mc
(file)
...
attach item_mc
(file)
...

[generatefilelisting called for each directory]

After each child is created, and if a directory all its children or contents are created, then it is positioned with a final

```
ArrangeContents(directory_mc);
```

And that brings us to those functions used to handle item position and directory opening and closing. They are as follows:

```
function SetAtBottomOfParent(below_mc){
    if (below_mc._parent._height) {
        below_mc._y = below_mc._parent.getBounds(below_mc._parent).yMax;
    }
}
function SetBelow(below_mc, top_mc){
    if (top_mc != undefined) {
        below_mc._y = top_mc.getBounds(top_mc._parent).yMax;
    }
}
function ArrangeContents(directory_mc){
    var item = directory_mc.firstChild;
    if (item != undefined){
        while (item = item.nextSibling){
            SetBelow(item, item.previousSibling);
        }
    var parent_container = directory_mc._parent._parent;
    if (parent_container != undefined)
        ArrangeContents(parent_container);
    }
}
function Fold(){
    if (this._parent.directory_mc._visible){
        this._parent.directory_mc._visible = false;
        this._parent.directory_mc._yscale = 0;
    }else{
        this._parent.directory_mc._visible = true;
        this._parent.directory_mc._yscale = 100;
    }
    ArrangeContents(this._parent._parent);
    scrollbar.contentChanged();
}
```

The first two functions, SetAtBottomOfParent and SetBelow are used to position the individual empty directory movie clips and attached item clips (respectively).

```
function SetAtBottomOfParent(below_mc){
    if (below_mc._parent._height) {
        below_mc._y = below_mc._parent.getBounds(below_mc._parent).yMax;
        }
    }
function SetBelow(below_mc, top_mc){
    if (top_mc != undefined) {
        below_mc._y = top_mc.getBounds(top_mc._parent).yMax;
    }
}
```

Each function vertically positions a passed clip, below_mc, to be either below its _parent or another movie clip through the use of the getBounds method. As you might be able to imagine, SetBelow will be useful with a directory item and its previousSibling property. And what do you know, the next function, ArrangeContents, does exactly that:

```
function ArrangeContents(directory_mc){
    var item = directory_mc.firstChild;
    if (item != undefined){
        while (item = item.nextSibling){
            SetBelow(item, item.previousSibling);
        }
    var parent_container = directory_mc._parent._parent;
    if (parent_container != undefined)
        ArrangeContents(parent_container);
    }
}
```

ArrangeContents gets passed to it a directory container movie clip. If it has a firstChild, it would mean there are movie clips within it. This would have been assigned in the for loop attaching the clips. Then positioning would be performed, otherwise the function just ends.

To position items, a while loop is used to cycle through each sibling of the movie clips starting with the first in the directory_mc. SetBelow is then used to position that sibling below the sibling before it.

Now, the thing to remember is that if you reposition the contents of a directory, other directories, such as the one containing the directory in question, will have to react and reposition their content as to prevent gaps in the directory layout. ArrangeContents handles this by calling itself (recursively) on the directory_mc's _parent._parent if it exists, ultimately correcting all movie clip positions that the initial change could have effected. (_parent._parent is used since _parent reflects the attached item clip used for the directory and we want the directory clip in which it exists or its _parent clip).

That brings us to the remaining Fold function which is assigned as an onPress event for directories.

```
function Fold(){
    if (this._parent.directory_mc._visible){
        this._parent.directory_mc._visible = false;
        this._parent.directory_mc._yscale = 0;
    }else{
        this._parent.directory_mc._visible = true;
        this._parent.directory_mc._yscale = 100;
    }
    ArrangeContents(this._parent._parent);
    scrollbar.contentChanged();
}
```

This shows or hides directory contents. Because this is assigned to a button within an attached item movie clip, _parent is used to access the directory_mc within that item. Then the directory is hidden *and* set to have a _yscale of 0. The inclusion of the _yscale is necessary because, despite the fact that this movie clip may be invisible, its height would still register as part of its parent's height. This means that hiding the clip alone will not be enough for ArrangeContents to recognize there being a gap. The clip would actually have to be literally folded up. When that happens, ArrangeContents is called and all movie clips get to react to the new change in position, whether it was movie clips being shown or being hidden.

Managing XML Data In Flash
In the squirrel finder and file listing examples, the XML loaded was only accessed one time, at the point directly after loading was complete. After that, anything dealing with the XML instance directly was, well, nonexistent. The information needed was immediately extracted and the XML instance was thereafter abandoned.

When dealing with loaded XML, this is, I think, the general approach (that should be) taken using XML in Flash. First obtain (load) the XML, then get what you need from it, converting its content into whatever internal Flash representation you need it to be in and then forget about it. In the squirrel finder, this representation was that of a list of buttons and a few variables which get used to display a text field; none of which screams "I'm XML!" In doing this, you're simplifying the process of dealing with the complications of XML as you only have to deal with it once.

Now this ideology changes a little bit when you need to take loaded XML and send it back out of Flash as XML. If you need that XML to change while in Flash, you will want to continuously work from the XML instance to assure its contents reflect what is seen in Flash.

A Better Understanding Of The ChildNodes Array
When it comes to editing XML, if there's one thing to be wary of, it's the childNodes array. It seems simple enough but it can cause a whole lot of confusion when you are trying to manage and alter your XML data. The reason is because a childNodes array is not the actual construct of the internal workings of an XML object used to hold XML nodes. It is merely a representation. It contains valid node references but altering a childNodes array will not effect the the XML from which it came. For example, putting:

```
node.childNodes[0] = node.childNodes[1];
```

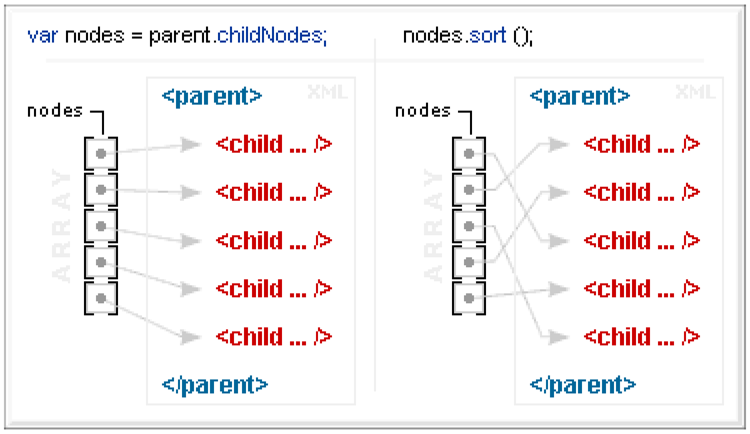
will do absolutely nothing despite your thoughts that you may have just copied the contents of childNodes[1] into childNodes[0]. Why is this so you ask? It has to do with what the childNodes property really is.

As you know, the childNodes property for any given XMLNode instance (from within an XML instance) provides you with an array of the nodes contained within that node. However, this array is a copy - a copy of whatever internal array or structure is used to truly represent the list of children within any element. So rather than thinking of childNodes as a direct property of an element, think of it more as an instruction or a function call that returns a new array with XMLNode references of that element's children. Because of this any change to an XML array will not be reflected in the actual XML. A loose internal representation of what the childNodes property does as a function may look like this:

```
function childNodes(){
    var children = new Array();
    for (child_elements in this_element) {
        children.push( this_element[child_element] );
    }
    return children;
}
```

As a function, childNodes here creates a new array and adds child elements of the current element into the array one by one for each contained within. That array is then returned and given to the childNodes property.

Now, that having been said, it's important to understand that each XMLNode *within* a childNodes array *is* directly related to the XML. So adding a or changing an attribute within an element of a child nodes array will actually change the internal XML itself since you're dealing with the real XML node. It's just when you're talking about the array elements directly, their order and their direct (non referential) values. For instance, sorting an array assigned to equal a childNodes array will sort the array but have no affect on the order of the elements within the XML as the following diagram shows.


```
var nodes = parent.childNodes;
```
```
nodes.sort ();
```
```
<parent>
<child ... />
<child ... />
<child ... />
<child ... />
<child ... />
</parent>
```

[sorting a childNodes array wont affect the xml]

Though the array itself has rearranged *its* order, the organization of the actual XML remains unchanged. This actually can be a good thing since sorting an array is a lot easier than sorting XML. If you are keeping a sortable list of XML data, using an array to order that data is a lot easier than sorting the XML nodes individually. The Array object in Flash provides a sort function to ease this process. No such method exists for XML.

Example: Sorted Grades List
Editing XML can be a pain. If you have data that needs to be sorted, you wouldn't want to do that directly from the XML instance. Instead, you could be better off handling it through an array as arrays have sort functions built right in. Because of how a childNodes array works, as an array copy representative of the XML node structure, we can keep XML node references in an array from a childNodes call and just use that to sort and handle our data. If things go horribly wrong, the original data in the XML instance will remain in its original order without disruption. Using a childNodes array still means that we'd be working from XML nodes from the XML instance, but the original *order* will remain unchanged for the XML instance. This example shows sorting performed on a childNodes array copy that has no effect on the internal XML instance order.

So, starting with this list of students in no particular order:

```
<?xml version="1.0" ?>
<grades year="1994">
<student first="James" last="Johnson" gpa="3.3" />
<student first="John" last="Smith" gpa="4.0" />
<student first="Serena" last="Williams" gpa="3.8" />
<student first="Indiana" last="Jones" gpa="3.2" />
<student first="Linda" last="Brown" gpa="3.9" />
<student first="Robert" last="Davis" gpa="3.2" />
<student first="Seno" last="Cular" gpa="4.0" />
<student first="Mike" last="Wilson" gpa="3.0" />
<student first="Mary" last="Moore" gpa="3.0" />
<student first="Will" last="Taylor" gpa="2.5" />
<student first="Thomas" last="Anderson" gpa="2.1" />
<student first="David" last="Thomas" gpa="3.7" />
</grades>
```

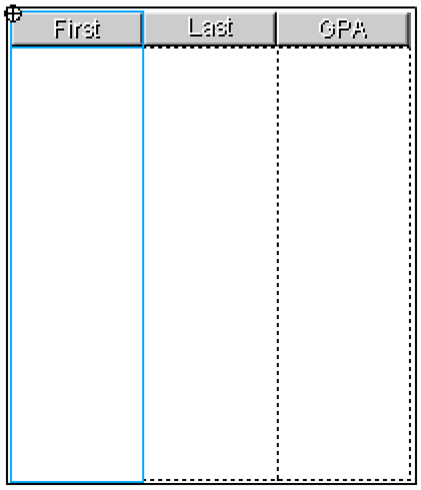
(Which can be found here: grades_1994.xml)

You can use the power of Array.sort() on a childNodes array to rearrange them and display them at will:

Example did not paste. Go to
http://www.kirupa.com/web/xml/examples/sortedgradeslist.htm
to see the full example

Download ZIP

Preparing the Flash File
The XML document containing students to be sorted uses 3 attributes: first names, last names and gpa. Without using any fancy-schmancy components, we can handle this using three separate, multi-lined (non-wrapping) text fields, one for each attribute. Above each will go the section headings which will also pose as buttons for sorting. Again, one for each attribute labeled accordingly.



[three columns with text fields and a buttons]

In addition to those buttons, an extra is added to restore order back to that which was specified in the original XML document.

[restore button]

This is just a copy of all the others with a different name so its really no big deal. And, in fact, the script that is associated with this is far less complicated as well.

ActionScript
The bulk of this example revolves around array sorting. Like many of the simpler examples from before, the XML itself is loaded once and for the most part discarded. Everything after the content is loaded revolves around an array that is assigned as a childNodes array of one of the elements of the XML, more specifically the document root or the grades element.

So lets check out some ActionScript. First, the XML instance:

```
var students_array;
var grades_xml = new XML();
grades_xml.ignoreWhite = true;
grades_xml.onLoad = function (success) {
    if (success) {
        var grades = this.firstChild;
        students_array = grades.childNodes;
        PopulateLists(students_array);
    } else {
        trace('Error loading XML file.');
    }
}
grades_xml.load('grades_1994.xml');
```

For the most part typical. The difference here is the addition of a students_array variable and the fact that the function being called from the onLoad to handle the loaded content doesn't specifically pass the XML instance or one of its nodes. Rather, it gets the students_array which had been just assigned to a childNodes array from the grade element (which is the XML instance's first child or the XML document's root). This array initially has the order of the XML but can be altered without altering the XML order. The contents of the array are references to XML nodes. If they were altered, as references, they would actually affect the XML.

Here is the PopulateLists function:

```
PopulateLists = function(xml_array){
first_txt.text = last_txt.text = gpa_txt.text = "";
for (var i=0; i<xml_array.length; i++){
var student = xml_array[i].attributes;
first_txt.text += student.first + "\n";
last_txt.text += student.last + "\n";
gpa_txt.text += student.gpa + "\n";
}
}
```

First the text in the text fields that make up the columns is cleared by setting them each to "".

```
first_txt.text = last_txt.text = gpa_txt.text = "";
```

Then, a for loop is created to cycle through all the elements within the passed array. Since these elements are references to nodes, they will give us access to the XML. Conveniently, a properly named variable is immediately created to reflect that XML information (it could be used to represent the element but since we're dealing with attributes only here, it can be assigned to reference the attributes object).

```
for (var i=0; i<xml_array.length; i++){
var student = xml_array[i].attributes;
// ...
}
```

Now all that remains is adding the information from each node within the loop to their respective columns.

```
first_txt.text += student.first + "\n";
last_txt.text += student.last + "\n";
gpa_txt.text += student.gpa + "\n";
```

Because each text field is getting a new line of text at the same time, the information will match horizontally. Once the loop is complete, the columns will be populated with all the students in the XML; first and last names and gpa.

All that remains is being able to sort these guys. That poses no considerate problem considering that an array is used to maintain order. However, what that doesn't mean there is *no* problem. There is still the issue of what to sort *by*. Considering we have three different categories to sort by, first, last and gpa, it means that every array element within students_array would have to be able to be sorted by any one of those 3 properties. To handle this, we need to create custom sort functions to use with Array.sort(). Otherwise, sort will just sort the XML nodes alphabetically based on their string representations. Custom sort functions would allow us to specifically pin point the part of the node we wish to sort by. Here's what we get in terms of sort functions:

```
FIRST_SORT = function(a,b){
    return a.attributes.first > b.attributes.first;
}
LAST_SORT = function(a,b){
    return a.attributes.last > b.attributes.last;
}
GPA_SORT = function(a,b){
    return parseFloat(a.attributes.gpa) < parseFloat(b.attributes.gpa);
}
```

Each is designed to specifically pin point a particular property of the XML nodes from which to base the array sort from. Now we can sort the students_array and pass it in to the PopulateLists function to create a sorted version of the student listing. To handle that we can create the following function:

```
var current_sort;
SortListBy = function(sortType){
    if (current_sort == sortType){
        students_array.reverse();
    }else{
        students_array.sort(sortType);
    }
    current_sort = sortType;
    PopulateLists(students_array);
}
```

SortListBy takes a sortType, which is actually a sort function, and sorts students_array based on that. You can see a variable current_sort being used to keep track of which sort was used last. This will allow us to reverse the listing if a user attempts to use the same sort more than one time in a row. Following the sorting (or reversing) PopulateLists is called to update the text fields with the newly sorted array.

With that defined, simply add button actions to the column heading buttons specifying the correct sort type for each and your set.

```
first_btn.onRelease = function(){
    SortListBy(FIRST_SORT);
}
last_btn.onRelease = function(){
    SortListBy(LAST_SORT);
}
gpa_btn.onRelease = function(){
    SortListBy(GPA_SORT);
}
```

In an effort show that the XML has not been altered, the restore button was added to reset the students_array back to the grade element's current childNodes array. When PopulateLists is called with the new definition of students_array, you can see the sorting has returned to its original unsorted order.

```
restore_btn.onRelease = function(){
    var grades = grades_xml.firstChild;
    students_array = grades.childNodes;
    PopulateLists(students_array);
}
```

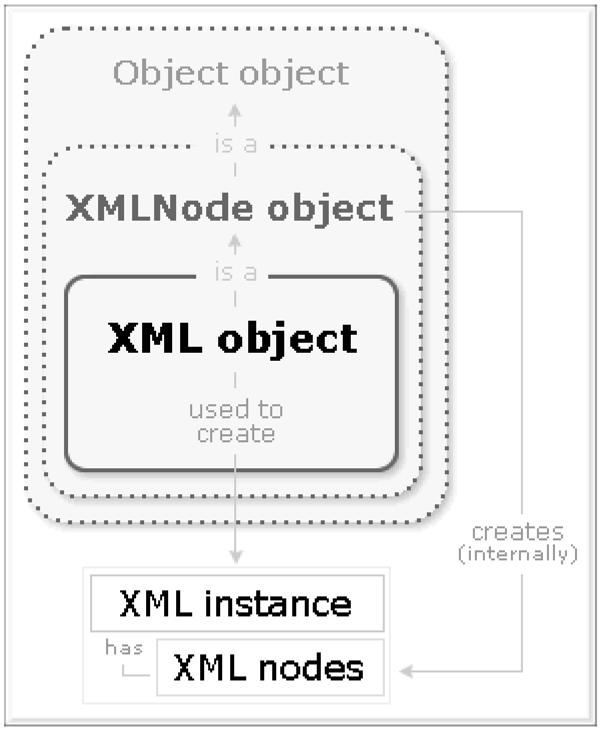
Use this button and you can see the original order unchanged.

Creating and Editing XML With ActionScript
Extracting information from an XML document or from an XML object from Flash is one thing. Editing that information and its structure after it's loaded into Flash is another. It's great that Flash can *receive* content via external documents but sometimes it also has to *send* it. And for information to be properly sent, it needs to be properly formatted, changed, altered, computed... whatever. Luckily, ActionScript provides methods for doing just that, all courtesy of the XML object. I suppose even if you don't need to send anything anywhere, you still *may* need to alter XML in Flash just to facilitate the ever changing needs of your Flash movie, though, as I said before, it may be easier to extract it once and keep it in a format that's more usable for you.

Straight out, here are the methods used to make/edit XML in ActionScript. We'll go through each one individually and see exactly how they work. Also understand that these are available for reference in Flash help (Flash help is your friend!).

Method	**Actions**
Constructor	
new XML("xml text")	Creates a new XML instance.
XML Instances	
parseXML("xml text")	Sets the XML of an instance to the passed text.
createElement("node name")	Creates an element node.
createTextNode("text")	creates a text node.
XML Nodes	
appendChild(childNode)	Inserts a child node at the end of the current element's child nodes.
insertBefore(childNode, beforeNode)	Inserts a child node before any child specified within the current element's child nodes.
cloneNode(deep);	Duplicates a node.
removeNode();	Removes a node.
hasChildNodes();	Determines if node has child nodes.

It is important to remember that when working with XML, you're dealing with two different object types, an XML instance and XMLNodes that define the XML within that instance. Since the XML object extends or inherits from XMLNode, that means that all XML instances can also be treated as XMLNodes (if you'll recall, an XML instance itself acts as a node to hold all other XML within the instance). So anything above specific to XMLNodes also works with XML instances, however those for XML instances are not meant to be used on XML nodes.



[xml instances are also xml node instances (are also objects)]

Remembering this is especially important on quirky methods like createElement and createTextNode which work off of XML instances and not nodes within that instance. So lets begin with the methods for creating and editing XML.

XML Methods
The following XML methods are used to create and edit XML.

new XML("xml text");
Number 1 is the XML object itself and the call to the constructor that lets you create XML instances. If you're working with XML in Flash, you'll simply need to have one of these.

There should be nothing new here. If you've been following along up to this point, you've noticed this used a few times. The optional parameter that lets you generate XML within the instance created right off the bat. Passing an XML string, written just as XML would be written in an external XML document, into the constructor call will let you define XML directly within the instance without needing to load in an external document. Example:

```
var my_xml = new XML("<letter><to>senocular</to><body>Get a life</body></letter>");
trace(my_xml.status); // traces "0" (No error)
trace(my_xml.firstChild.nodeName); // traces "letter"
trace(my_xml.firstChild.firstChild.nodeName); // traces "to"
trace(my_xml.firstChild.firstChild.firstChild.nodeValue); // traces "senocular"
```

Easy! ...next

XMLInstance.parseXML("xml text");
This method basically condenses the constructor's action into a method letting you create XML for your instance by passing in XML text. Any XML created this way will replace any current XML existing in the

object. So, you may be thinking why not just redefine the variable with a new XML object via a constructor call? Well, using parseXML lets you retain any custom properties assigned to the XML object or settings such as ignoreWhite.

```
var my_xml = new XML();
my_xml.parseXML("<letter><to>senocular</to><body>send money</body></letter>");
trace(my_xml.firstChild.firstChild.firstChild.nodeValue); // traces "senocular"
trace(my_xml.firstChild.childNodes[1].firstChild.nodeValue); // traces "send money"
```

Again, easy. Notice, though, that parseXML is a method of an XML instance and not for use on an XMLNode. This means it can't be used selectively on parts of internal XML. Other methods, however, do allow for that.

XMLInstance.createElement("node name");
This one is kind of self explanatory. It creates an element. Now, the catch is that the createElement method *does not* create an element *within* the XML instance that it's used on. Instead, a homeless, "free floating" element node is returned from the createElement call. The instance calling it is just a host for spitting this guy out. This free floating element can then be inserted into whatever XML instance you want within any node using other XML methods like appendChild and insertBefore.

The text passed into createElement, "node name," is the desired name for the element created. This is not a tag (using < and >) like text used in parseXML, just a name. The tag aspect of the created node is automatic. So, if it is your intent to create the element <bob />, then you would pass "bob" not "<bob />" into createElement.

```
var my_xml = new XML();
trace(my_xml); // traces ""
var my_element = my_xml.createElement("bob");
trace(my_element); // traces "<bob />"
trace(my_xml); // still traces ""
```

XMLInstance.createTextNode("text value");
This is just like createElement but it creates a text node instead, using string passed in to represent its nodeValue (or its text). It too, like createElement, returns a "free floating" node which exists no where until you decide to put it somewhere.

```
var my_xml = new XML();
trace(my_xml); // traces ""
var my_textnode = my_xml.createTextNode("What about");
trace(my_textnode.nodeValue); // traces "What about"
trace(my_xml); // still traces ""
```

Note: there is no direct method to create CDATA sections using any particular current ActionScript method as Flash just treats them like text nodes anyway. The good news is that the createTextNode method will actually transform unacceptable characters (<, >, &, ', and ") passed to it into their respective character entity references if you're worried about having invalid characters going into your text nodes. This makes it acceptable for storing markup like HTML too so long as you don't mind the character entity references being used. Using nodeValue within Flash will give you the markup with without character entity references while the toString method (what you normally see with a trace) will show you the actual XML text as it really exists with character entity references.

```
var my_xml = new XML("<text/>");
var my_textnode = my_xml.createTextNode("<P><B>Bold</B></P>");
trace(my_textnode.toString()); // traces "&lt;P&gt;&lt;B&gt;Bold&lt;/B&gt;&lt;/P&gt;"
trace(my_textnode.nodeValue); // traces "<P><B>Bold</B></P>"

my_xml.firstChild.appendChild(my_textnode);
trace(my_xml); // traces "<text>&lt;P&gt;&lt;B&gt;
Bold&lt;/B&gt;&lt;/P&gt;</text>"
```

XMLNodeInstance.appendChild(childNode);
Pretty much everything called directly off of XML instances was facilitating creation. Now that we're in the XMLNode instance methods, we're dealing with manipulation. The appendChild method takes a node and throws it to the set of elements making that XML node's children putting it at the end of the list. Using this and insertNode (to be covered next) is where a good bulk of the action is when fiddling with the structure of your XML internally in Flash. Using one of these two methods lets you find a home for those free floaters created with createElement and createTextNode. When used on nodes already existing within an XML structure, it will move them to the specified XMLNode instance - this so long as the node is not moved into the same set of children in which it already exists. In other words, appendChild cannot reorder, just move nodes to *other* parents.

```
var my_xml = new XML();
var node = my_xml.createElement("pickup");
my_xml.appendChild(node);
node = my_xml.createElement("truck");
my_xml.firstChild.appendChild(node);
trace(my_xml); // traces "<pickup><truck /></pickup>"

node = my_xml.createTextNode("Drive Me!");
my_xml.firstChild.firstChild.appendChild( node);
trace(my_xml); // traces "<pickup><truck>Drive Me!</truck></pickup>"

node = my_xml.createElement("car")
my_xml.firstChild.appendChild(node);
trace(my_xml); // traces "<pickup><truck>Drive Me!</truck><car /></pickup>"

my_xml.firstChild.appendChild( my_xml.firstChild.firstChild.firstChild );
trace(my_xml); // traces "<pickup><truck /><car />Drive Me!</pickup>"
```

Here createElement and createTextNode create nodes which are then added to the my_xml XML instance (or a node within) using appendChild. Each time the node is added as a child at the end of other children of the parent element. You can see this more specifically with the car element which, when added, was added after the previously appended truck element.

The last part shows how appendChild can be used to move nodes. This, however, would not have worked if you tried to append truck to pickup. Though common sense tells you it would move to the end, the appendChild method would actually fail since you're trying to append to the same parent in which the node passed already exists - and Flash just doesn't like the idea of that. Moving nodes within their parents will be addressed later in using custom functions.

XMLNodeInstance.insertBefore(childNode, beforeNode);
The insertBefore method is similar to that of appendChild. The main difference is that insertBefore lets you reach all the positions within an element's child set that appendChild can't reach. While appendChild throws nodes at the end of a collection of children, insertBefore puts them everywhere else, and actually, includes the end as well.

When using insertBefore, you use a reference to the node you wish to insert (childNode) and a reference to the node in which its supposed to be inserted before (beforeNode). Take note that this is *not* an index or position number, its the actual child node that exists within the XMLNode instance insertBefore is being used on. If you pass an illegal *node* reference, something that is a node but a node that is not a child of the node instance this insertBefore was used on, the method will fail. So really, the function looks more like this:

beforeNode.parentNode.insertBefore(childNode, **beforeNode**);

or

XMLNodeInstance.insertBefore(childNode, **XMLNodeInstance.childNodes[*n*]**);

If you pass a non-node value into the beforeNode parameter, the childNode will be, by default, inserted at the end of the child nodes list as if using appendChild. Note that not using any beforeNode parameter at all or passing *null* will cause the method to fail.

```
var my_xml = new XML("<baseballfield />");
var node = my_xml.createElement("firstbase");
my_xml.firstChild.appendChild(node);
node = my_xml.createElement("thirdbase");
my_xml.firstChild.appendChild(node);
node = my_xml.createElement("secondbase");
var thirdbase = my_xml.firstChild.childNodes[1];
my_xml.firstChild.insertBefore(node, thirdbase);
trace(my_xml); // traces "<baseballfield><firstbase /><secondbase /><thirdbase /></baseballfield>"
```

The secondbase node was a little late in being created (either that or the thirdbase element cut in line!) so he needed to jump back in his rightful spot. The insertBefore method allowed this in passing thirdbase as the node in which it needed to be inserted in front of.

XMLNodeInstance.cloneNode(deep);
This method takes an existing node, copies it, and returns it as a new node. The deep parameter (why they chose "deep" to describe this parameter is beyond me) determines whether or not all of the child nodes of the cloned node are also copied and cloned along with it. If deep is true, child nodes are included in the copy. If false, the element is copied as an empty element with no children but retains any attributes it may have. It may be better to think of this as the "include children?" parameter.

```
var my_xml = new XML("<aliens><invader>SuperKiller</invader></aliens>");
var clone_with_children = my_xml.firstChild.firstChild.cloneNode(true);
var empty_clone = my_xml.firstChild.firstChild.cloneNode(false);
my_xml.firstChild.appendChild(clone_with_children);
my_xml.firstChild.appendChild(empty_clone);
trace(my_xml); // traces
<aliens><invader>SuperKiller</invader><invader>SuperKiller</invader><invader /></aliens>
```

Notice how the last invader, the second one appended, doesn't have a "SuperKiller" text node in it. This is because the deep (or "include children?") argument was false meaning no children were copied over into the returned clone node.

XMLNodeInstance.removeNode();
Last but not least, removeNode. It should be pretty apparent what this method does. Its the equivalent to MovieClip.removeMovieClip, only it removes nodes instead of movie clips. This is how you get rid of unwanted nodes in your XML object or other nodes within. Example:

```
var my_xml = new XML("<life><fun /><money /><friends /><taxes /></life>");
trace(my_xml); // traces "<life><fun /><money /><friends /><taxes /></life>"
my_xml.firstChild.childNodes[3].removeNode();
trace(my_xml); // traces "<life><fun /><money /><friends /></life>"
```

Understand that all attributes and child nodes of removed nodes are removed right along with them. If you want to retain child nodes but ditch the parent (how often have we wanted to ditch our parents as a kid?) then later we can discuss a way to do through a custom function.

What about Attributes?
Attributes are not forgotten. They are just more "manual". The attributes object can be manually edited just as any other object in Flash and does not require methods to alter it. This makes them easy to work with as you don't necessarily have to deal with potentially confusing XML methods. Example:

```
var my_xml = new XML("<friends><friend /></friends>");
my_xml.firstChild.firstChild.attributes.girlfriend = "Julie"
trace(my_xml); // traces "<friends><friend girlfriend="Julie" /></friends>"

delete my_xml.firstChild.firstChild.attributes.girlfriend;
trace(my_xml); // traces "<friends><friend /></friends>"
```

Working with attributes is probably more easy than anything else. Because of this, people like to stick everything into attributes, but, as I said before, you should keep it simple.

Editing Text Nodes and Element Names
If you'll remember, the nodeValue (text nodes) and nodeName (elements) properties for XML nodes were not read-only as many of the other properties like firstChild and nextSibling were. This means that, like attribute values, they too can be edited directly. Example:

```
var my_xml = new XML("<loss>We cannot perform well.</loss>");
trace(my_xml); // traces "<loss>We cannot perform well.</loss>"

my_xml.firstChild.nodeName = "victory";
my_xml.firstChild.firstChild.nodeValue = "We dominate!";
trace(my_xml); // traces "<victory>We dominate!</victory>"
```

And much like createTextNode, setting nodeValue will transform undesirables into character entity references.

Additional Helpful Methods
Let's not stop with what Macromedia has provided for us to edit XML in Flash. We can take that framework and build more useful and straight-forward methods to further help in editing XML content while in Flash. In fact, maybe we can take what's been given to us and make them better. The following represents a few examples that add methods to the XMLNode object and/or the XML object in Flash.*

Download AS

```
// formats XML into a multi-lined, indented format
// a good toString replacement for XML with ignoreWhite = true
XMLNode.prototype.format = function(indent){
    if (indent == undefined) indent = "";
    var str = "";
    var currNode = this.firstChild;
    do{
        if (currNode.hasChildNodes()){
            str += indent + currNode.clodeNode(0).toString().slice(0,-2) + ">\n";
            str += currNode.format(indent+"\t");
            str += indent + "</" + currNode.nodeName + ">\n";
        }else{
            str += indent + currNode.toString() + "\n";
        }
    }while (currNode = currNode.nextSibling);
    return str;
}

// moves a child within its set of children (to before a beforeNode)
XMLNode.prototype.moveBefore = function(beforeNode){
    beforeNode.parentNode.insertBefore( this.clodeNode(true), beforeNode);
    this.removeNode();
}
```

```
// adds a root property that references the document root
// element of an XML instance through any of its nodes
XMLNode.prototype.root = function(){
    var r = this;
    while(r.parentNode) r = r.parentNode;
    return r.firstChild;

};
XMLNode.prototype.addProperty("root",
    function(){
        return this.root();
    },
    function(n){
        r = this.root();
        r.parentNode.appendChild(n);
        r.removeNode();
    }
);

// adds a firstNodes property (read-only) which returns
// an array of all the nested first child elements within an
// xml node. ex: firstChild.firstChild.firstChild == firstNodes[2]
XMLNode.prototype.addProperty("firstNodes",
    function(){
        var c = [];
        var n = this;
        while(n = n.firstChild) c.push(n);
        return c;
    },null
);

// adds a text property that represents the text (nodeValue)
// of a text node. This can be called from a text node *or* an
// element which contains a text node (retrieves the last child)
// making it easy to access text from an element which only
// contains one text node
XMLNode.prototype.text = function(){
    var n = this;
    if (n.nodeType == 1) n = this.lastChild;
        return n;
}
XMLNode.prototype.addProperty("text",
    function(){
        var n = this.text();
        if (n.nodeType == 3) return n.nodeValue;
        return "";
    },
    function(txt){
        var n = this.text();
        if (n.nodeType == 3) n.nodeValue = txt;
        else this.appendChild(new XML().createTextNode(txt));
    }
);

// Find a child element by name and index (nth element of that name)
XMLNode.prototype.findChild = function(name, index){
    var count = 0;
    var currNode = this.firstChild;
    do{
        if (currNode.nodeName == name){
            count++;
            if (count == index) return currNode;
        }
    }while (currNode = currNode.nextSibling);
    return false;
}

// Removes and returns an XML node based on index
XMLNode.prototype.extractChild = function(i){
    var n = this.childNodes[i].clodeNode(true);
    this.childNodes[i].removeNode();
    return n;
}
// Swap the positions of two child nodes within their parent's childNodes array
XMLNode.prototype.swapChildren = function(i, i2){
    if (i==i2) return(0);
    if (i > i2){
        var temp = i;
        i = i2; i2 = temp;
    }
    var n = this.extractChild(i);
    var n2 = this.extractChild(i2-1);
    this.insertBefore(n2, this.childNodes[i]);
    this.insertBefore(n, this.childNodes[i2]);
}
// Sort child nodes based on a compare function (uses a simple quicksort)
XMLNode.prototype.sortChildren = function(cmp, low,high) {
    var a = this.childNodes;
    if (cmp == undefined) cmp = this.sortChildren.defaultCompare;
    if (low == undefined) low = 0;
    if (high == undefined) high = a.length;
    var p = a[low], w = low+1, h = high;
    while(w < h) {
        if (cmp(a[w], p)) w++;
        else this.swapChildren(w, (--h));
    }
    this.swapChildren(low, (--w));
    if (w-low > 1) this.sortChildren(cmp, low, w);
    if (high-h > 1) this.sortChildren(cmp, h, high);
}
```

```
// A default sort function for use in sortChildren
XMLNode.prototype.sortChildren.defaultCompare = function(a,b){
    return a.nodeName <= b.nodeName;
}

// removes a node but leaves all child nodes it contained in its place
XMLNode.prototype.explodeIntoParent = function(){
    if (!this.parentNode) return false;
    while (this.hasChildNodes()){
        this.parentNode.insertBefore(this.firstChild, this);
    }
    this.removeNode();
    return true;
}

// gets the index of the current node in its parent's childNodes array
XMLNode.prototype.getIndexInParent = function(){
    var i = 0, node = this;
    while (node = node.previousSibling) i++;
    return i;
}

// trims white space around text
// can be used to remove white space around formatted text node text
_global.TrimWhiteSpace = function(str){
    var beg = 0, end = str.length-1;
    while (str.charAt(beg).isWhite()) beg++;
    while (str.charAt(end).isWhite()) end--;
    return str.slice(beg, end+1);
}
// determines if a string is made up of white space
// used by TrimWhiteSpace
String.prototype.isWhite = function(){
    return !isNaN(" "+this+" 0");
}
```

* All methods provided are in ActionScript 1.0 format. They will work, for the most part, with AS 2.0 but would require that you not strictly type your XML instances (or, rather, add the methods used into the intrinsic class definition). Extending XMLNode in ActionScript 2.0 is not really an option as it's the XML class that instantiates XMLNodes internally, not you. So you have no way of controlling how they are made and with using which constructor. It's this kind of situation where the flexibility of AS 1.0 really pays off.

You can find more such examples at Layer51 Prototypes. There's also XPath implementation for Flash (AS1, AS2) provided by XFactor Studio. XPath provides an alternate, more user friendly means of addressing XML content.

Example: MP3 Playlist
This example will demonstrate editing XML from within Flash using an XML based playlist which technically could be sent back to a server and saved for later use. This particular example will not actually be doing any saving, but you will be able to see the XML as its edited in an output window.

The XML used is a simple, generic playlist example containing a title, artist and file path within attribute values of song elements.

mp3_playlist.xml

Here's what you get:

```
<?xml version="1.0" ?>
<playlist>
<song title="Chocobo Piano Loop" artist="DJ_xBrav" src="mp3s/NG5402.mp3" />
<song title="House Of Kalm" artist="ProfitZ" src="mp3s/NG7890.mp3" />
<song title="Strobe Riding" artist="Dreamscaper" src="mp3s/NG3221.mp3" />
<song title="Awakening" artist="Evil Dog" src="mp3s/NG3413.mp3" />
<song title="blue eyes - limp RMX" artist="czer323" src="mp3s/NG4348.mp3" />
<song title="Little Organ Diddy" artist="Dustball" src="mp3s/NG10179.mp3" />
</playlist>
```

This is loaded into the player giving us a list of songs by name which can be selected and played.

Example did not paste. See http://www.kirupa.com/web/xml/examples/MP3playlist.htm for working example.

[mp3 player with playlist]

Download ZIP

Editing XML With ActionScript
I won't get into the details of how everything works for this example. That might get a little too involved. I'll just cover that which deals with the manipulation of the XML. So starting with your basic XML instance definition, you get:

```
var playlist_xml = new XML();
playlist_xml.ignoreWhite = true;
playlist_xml.onLoad = function(success){
    if (success){
        GenerateSongListing(this, playlist_mc);
    }else trace("Error loading XML file");
}
playlist_xml.load("mp3_playlist.xml");
```

This then calls GenerateSongListing passing in the XML object and the movie clip in which the playlist is created. GenerateSongListing like most other functions of its type goes through and creates a list of songs by attaching movie clips for each one, each time adding to the movie clip attached, a name, some other properties and button actions etc.

The button actions allow both the selection of a song and the reordering with in the list (note: some actions. like onPress, are assigned to a button within a playlist item movie clip while others, like onMouseMove, are assigned to the movie clip itself). Song selection is performed on a simple click when there was no dragging of the item, whereas reordering is performed when dragging has been detected. A simple moved variable within each item is set to record this.

```
this.moved = true;
```

When the mouse is released from an item, this moved variable is checked. If there was no movement, the song is selected and sent through to the MP3 player controller (this is from a button event so to reference the playlist item movie clip, _parent is used).

```
if (!this._parent.moved){
    SetSong(this._parent.node);
}
```

Otherwise, the playlist is reordered taking the item being moved (this.node) and switching it with the item that the mouse has dragged over (over_node) and the list is regenerated.

```
if (this.moved){
    this.node.moveBefore(over_node);
    GenerateSongListing(playlist_xml, playlist_mc);
}
```

Its the moveBefore method here that is doing all the work in this example. This is one of the XMLNode methods posted earlier. As a reminder, this is what it looks like:

```
XMLNode.prototype.moveBefore = function(beforeNode){
if (this == beforeNode) return (0);
beforeNode.parentNode.insertBefore( this.cloneNode(true), beforeNode);
this.removeNode();
}
```

This acts much like insertBefore, but it moves an existing node to before any other node even if that node exists within the same element (a case where insertBefore would fail) by using a clone of itself rather than the actual node its being used on. Because the clone has no home, it insertBefore won't complain about it being placed in the same parent as it would normally. Of course, having used the clone, the original would then have to be removed.

This lets us rearrange the order of the nodes within the playlist XML directly as opposed to using an array as was done with the grades sorting example; the advantage being that this XML could then be re-saved and new order maintained.

Sending XML Out From Flash

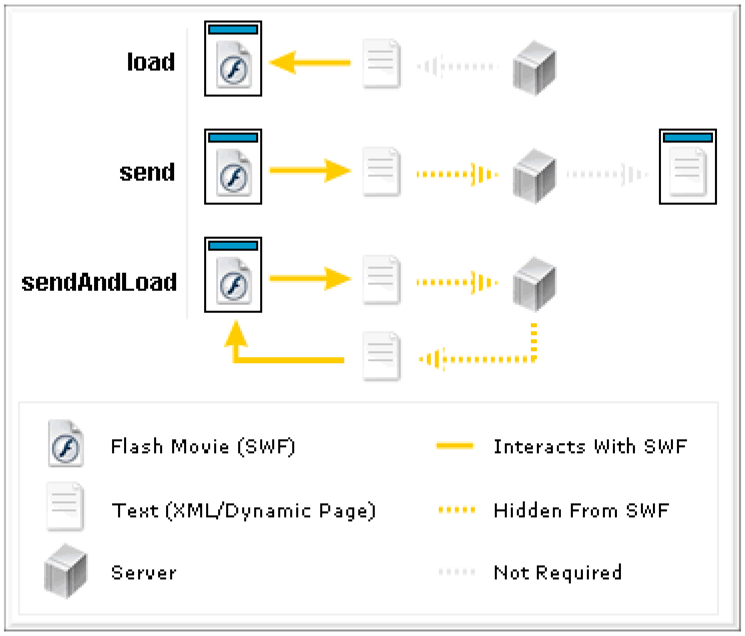
We've already covered loading XML in Flash and editing it while its there, but haven't really touched getting it back out again. This is something you may need to do once you spend all that time editing an XML document for some anxiously awaiting server. Hurry hurry hurry!

Unlike editing XML, sending it out of Flash is not that hard. In fact, it may be easier than loading it (maybe). It involves using one of two XML methods:

Property	Represents
XML Instances	
send("URL", "target")	sends the XML to the passed URL.
sendAndLoad("URL", receiver)	sends the XML to the passed URL and performs a load command on the XMLInstance passed loading the result.

Both send and sendAndLoad send your XML to a URL, namely some dynamic page waiting to handle your XML for the server. The sendAndLoad method takes the extra step of loading back in that URL to a target receiver object (typically an XML or LoadVars instance). With the send method you just pretty much have to assume all went to plan, though it does offer a optional target HTML window parameter ("target") to display the results. Valid targets include "_self", "_blank", and HTML frames etc.

Using sendAndLoad is much like calling both the send and load method back to back only here, it gives the URL time to react to the XML sent. That result then ends up into the receiver object and its onData is called (which would then call onLoad if you haven't rewritten it yourself).



[interaction of external content with a flash movie]

You can see from the above diagram that Flash really doesn't interact directly with a server here. When using load, send, and sendAndLoad you're interacting with dynamic pages that interact with the server for you. Flash has ActionScript; your dynamic pages use PHP or ASP (or other). The interaction of Flash and the server revolves around text documents passed back and forth between these dynamic pages. Flash really only cares about the resulting text of its request. When you load something like XML into Flash, you're loading text. This text can be generated by a server but there are no direct exchanges of commands between that server or Flash. Even with sendAndLoad, Flash sends its text to a dynamic page which has its own commands to interact with the server that then allows it to generate a new, dynamic page of text that can then be brought back into Flash.

When XML data is sent to a server from Flash using one of these methods, it arrives in the form of raw POST data. This means that it comes as POST data that has no variable associated with it as often is the case POST data. Once your XML reaches your server you'll have to know how to handle that using server-side scripting (the two examples that follow will provide a simple method of doing so for PHP).

One thing to be careful of when using send and sendAndLoad is that both send XML to a URL with a default content type of application/x-www-form-urlencoded. This is the format of urlencoded variables that you deal with in using a LoadVars instance. XML, however, is *not* urlencoded - its just XML text. If you're working with your server-side script and find that your server isn't liking your XML it may be because of this. Fixing this means changing the content type of your XML instance, and for that you use the contentType property.

```
var my_xml = new XML();
my_xml.ignoreWhite = true;
my_xml.contentType = "text/xml";
...
```

The text/xml content type will help your server recognize that it is, in fact, receiving XML and not urlencoded variables and may help solve any problems you've been having. Like with ignoreWhite, if you plan on using send or sendAndLoad, setting the contentType to "text/xml" is pretty standard practice.

But wait! There's more! Another precaution! Wha... Yeah, I know, I made it sound like it could have been something good, but it's not. It's just something else to worry about. This precaution has to deal with working in the Flash authoring environment when using methods like send and sendAndLoad. When working *in* Flash, when you test your movie, Flash will send sent XML data through GET and *not* POST as it does through a web page. This can screw you over just as much as the contentType. Your best bet for testing from Flash when dealing with server interaction like this is to perform a publish preview in HTML so you can see the movie working directly in the browser.

Note: XML Declaration Bug

There exists a XML declaration bug where when saving (and reloading) an XML file to the server repeatedly from Flash, Flash can inadvertently create duplicate XML declaration tags to be stacked on to the XML document. To prevent this behavior, you'll just have to do something such as start with a fresh XML instance of clear the XML declaration from the XML instance in use. You will see this dealt with first hand in the examples to come.

Example: Simple Editor

Back to the basics: simple loading and simple sending (and reloading) of XML. Here, you have a simple, two-paned editor. The top pane is the input pane (a text field named input_txt) and the bottom pane is the output pane (a text field named output_txt). What you do is load existing XML into the input pane, edit it (or just start anew), then send or "save" it and the changes will be sent out to the server using sendAndLoad. The results (what you just sent) is displayed in the output pane thanks to the load aspect of sendAndLoad.

Example did not paste. Please see
http://www.kirupa.com/web/xml/examples/simpleeditor.htm
For working example.

[save and load using input and output panes]

Download ZIP

The XML
The XML is about as simple as its going to get. It starts out as:

```
<?xml version="1.0"?>
<edit>This is what is to be edited.</edit>
```

And then changes based on whoever was last to save what. All you have is a simple edit element for the document root node and a text node as a child. The text within this text node is what represents what you see in the input and output panes of the example.

ActionScript
Much of the Flash-based scripting is old hat. The big difference in this example is that two XML instances are created to deal with the same XML document. One handles the loading and sendAndLoading for the input pane and the other receives data from the return result of the sendAndLoad call from the first. You could actually use the same XML instance to send and load into. However, since the first XML instance handles loading into the input pane, it would be slightly more difficult to also have it handle content loading into the output pane. Lets take a look at their definitions:

```
var input_xml = new XML();
input_xml.ignoreWhite = true;
input_xml.contentType = "text/xml";
input_xml.onLoad = function(success){
    if (success) input_txt.text = this.firstChild.firstChild.nodeValue;
    else input_txt.text = "Error loading input XML";
}

var output_xml = new XML();
output_xml.ignoreWhite = true;
output_xml.onLoad = function(success){
    if (success) output_txt.text = this.firstChild.firstChild.nodeValue;
    else output_txt.text = "Error loading output XML";
}
```

Each input_xml (for the input pane) and output_xml (for the output pane) have their own separate onLoad events to handle each of the text fields' text assignments from the XML loaded where this.firstChild.firstChild.nodeValue represents the text within the text node child of the edit element. Since input_xml is sending data to a server, contentType is also specified for it, set to "text/xml."

```
input_xml.contentType= "text/xml";
```

Button actions make up the rest of the code. They control when XML is loaded and sent and also provide ways to clear the two panes.

```
var xml_file = "simple_edit.xml";
var server_file = "simple_edit.php";

load_btn.onRelease = function(){
input_xml.load(xml_file + "?uniq=" + new Date().getTime());
input_txt.text = "Loading...";
}
send_btn.onRelease = function(){
input_xml.firstChild.firstChild.nodeValue = input_txt.text;
input_xml.sendAndLoad(server_file, output_xml);
output_txt.text = "Loading...";
}

clearin_btn.onRelease = function(){
input_txt.text = "";
}
clearout_btn.onRelease = function(){
output_txt.text = "";
}
```

Some variables are first defined for keeping track of what files are being used. These aren't necessary but keep some organization going when dealing with multiple files.

```
var xml_file = "simple_edit.xml";
var server_file = "simple_edit.php";
```

Next are the button actions that interact with the server. The first is the load button. It simply loads the xml file into the input_xml instance. To make sure that a fresh version of the file is loaded and not a cached version, an addition query string is added to file url. This adds a (mostly) unique number to the end of the url so that the same url will not be loaded for each instance the XML is loaded into the Flash Player - hence, no cached version.

```
input_xml.load(xml_file + "?uniq=" + new Date().getTime());
```

The send button is much like load except it first updates the input_xml with the input_txt text field's text and uses sendAndLoad to send the XML to a server-side PHP script.

```
input_xml.sendAndLoad(server_file, output_xml);
```

What this does is takes the XML within input_xml, sends it to the server_file url as raw POST data and then loads the result of that url into output_xml. Once output_xml receives the result, its onLoad will be called and the output_txt text field can be updated.

The remaining buttons simply clear the fields. They set the text of either of the two panes to "" which removes any other text that might have been there before.

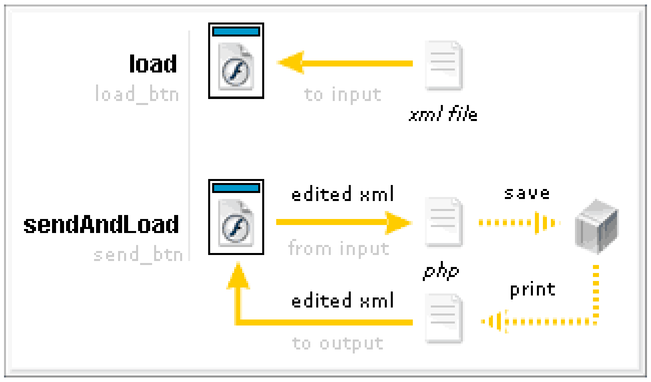
```
input_txt.text = "";
```

Server-side Scripting
The server-side script used in sendAndLoad does half the work. For this particular example, I used PHP though feel free to use whatever you wish. For it to do its job, the script simply needs to print what its been sent and save it on the server. A very basic version of that in PHP is the following:

```
<?php
$filename = "simple_edit.xml";
$raw_xml = file_get_contents("php://input");

print $raw_xml;

$fp = fopen($filename, "w");
fwrite($fp, $raw_xml);
fclose($fp);
?>
```

This takes input from raw post data (sent by the swf) and saves it as a variable called $raw_xml. This is printed as output and then saved to simple_edit.xml.

Normally, the print command would display to a web page. And, for the most part, it's still doing that. However, you never see this web page yourself. Instead, it's sent back to Flash and loaded in as part of the load aspect of sendAndLoad. Send throws the XML at the PHP script as raw data and load brings the results back in.



[load and sendandload interaction with xml and the server]

A few modifications and this can be doing something a little more useful. That brings us to our next example.

Example: News Editor
This example takes XML data editing to the next level. It does not simply display and let you change one text node. This example will let you add, edit and delete up to five *entries* each of which consist of two editable text fields (nodes). Together they create a small and simple news feed application. Take a gander:

Example did not paste. Please see http://www.kirupa.com/web/xml/examples/newseditor.htm for working example.

[show, add, edit, and/or delete news items]

Download ZIP

Initially you see the news display where all news is loaded and formatted for viewing. Using the refresh button, you can reload the content reflecting any changes that might have occurred since it was initially loaded when the movie began. The edit button brings you to the edit screen.

The edit screen, which lies on a different frame, allows you to edit the XML. It provides arrows for navigating the entries currently loaded within the player from the initial news view and provides text fields for changing their values. The first entry is labeled as [New] and is not an entry at all but a way to create new entries. This was to keep the file as simple as possible while still maintaining a good deal of functionality. Likewise, there is no delete button but you can delete entries by clearing their text fields and saving.

Whenever you save from the edit screen, the XML will be edited, sent to the server and the result will be reloaded and displayed back in the original news view.

The XML
Since basically all nodes within the XML are dynamically created, the base XML file simply consists of one node:

```
<news />
```

(If you're too lazy to create your own, you can get this here.)

Every entry within the news element is created from within the Flash movie. The structure of a complete entry is as follows:

```
<entry date="Thu Aug 26 10:35:20 GMT-0400 2004">
    <title>title text</title>
    <body>body text</body>
</entry>
```

Where up to five may exist within the news element at any given time (this is a limit created within the code that can be altered if desired). The text within the title and body elements are both editable whereas the date attribute value is defined automatically.

ActionScript
Since this example is far more complex than some of the others, I'm only going to cover the important parts and let you work your through the source file to figure out all of the other details. Those important parts are creating new entries and editing existing ones. The methods involved in doing that are AddNewsEntry and EditNewsEntry.

AddNewsEntry generates a a new, full entry node assigning title and body to respective values passed into the function call.

```
AddNewsEntry = function(news_xml, title, body){
    var entryNode = news_xml.createElement("entry");
    entryNode.attributes.date = new Date().toString();

    var titleNode = news_xml.createElement("title");
    var titleText = news_xml.createTextNode(title);
    titleNode.appendChild(titleText);
    entryNode.appendChild(titleNode);

    if (body == "") body = "(none)";
    var bodyNode = news_xml.createElement("body");
    var bodyText = news_xml.createTextNode(body);
    bodyNode.appendChild(bodyText);
    entryNode.appendChild(bodyNode);

    if (news_xml.firstChild.hasChildNodes()){
        news_xml.firstChild.insertBefore(entryNode, news_xml.firstChild.firstChild);
    }else news_xml.firstChild.appendChild(entryNode);

    while (GetNewsCount(news_xml) > max_news_items){
        news_xml.firstChild.lastChild.removeNode();
    }
}
```

You can see that AddNewsEntry starts off creating an entry element ("free floating") using createElement. Added to its attributes object is a variable called date which is assigned to be the current data (as determined by Flash using the Date object).

```
var entryNode = news_xml.createElement("entry");
entryNode.attributes.date = new Date().toString();
```

Next the title element is created.

```
if (title == "") title = "(none)";
var titleNode = news_xml.createElement("title");
var titleText = news_xml.createTextNode(title);
titleNode.appendChild(titleText);
entryNode.appendChild(titleNode);
```

The title element also contains a text node child element that contains the text for the title of the entry. This text is passed in to AddNewsEntry as the variable title and is used in creating a text node called titleText (assuming its not empty, in which case "(none)" is used to describe its contents). After both nodes are created (the title element and the text node to contain the title text), which are both homeless at the time of creation, they are placed in their respective final locations with appendChild; the text node within the title element and the title element within the entry node created earlier.

The body element is created next in the same manner as title.

```
if (body == "") body = "(none)";
var bodyNode = news_xml.createElement("body");
var bodyText = news_xml.createTextNode(body);
bodyNode.appendChild(bodyText);
entryNode.appendChild(bodyNode);
```

This gives the entry node two children, title and body, both of whom have a single text node to contain the title and body text. The entry node itself, however, is still without a home. Giving it one is just a matter of putting it in the main news node of the XML.

Before doing that, however, think about how XML is typically read and how these news entries are to be ordered. Typically, XML is read from top to bottom, where you look at the first element of a set of child nodes and work your way through to the bottom. When the news is accessed, displayed and read, you would want the newest news to be at the top with older news below it. Since AddNewsEntry creates new news, this means that the new entry being created should be inserted at the top of the list of all the other news. Of course, if there are no news entries, it would just be added normally. That's what the next section of code does.

```
if (news_xml.firstChild.hasChildNodes()){
    news_xml.firstChild.insertBefore(entryNode, news_xml.firstChild.firstChild);
}else news_xml.firstChild.appendChild(entryNode);
```

This checks to see whether or not the news element (news_xml.firstChild) has any child nodes. If so, insertBefore is used to put the new entry at the top of the list within those nodes. Otherwise, appendChild is used to simply add it.

That brings us to the end of the function where one final action takes place to make sure that the news entries don't go beyond the specified maximum number of entries.

```
while (GetNewsCount(news_xml) > max_news_items){
    news_xml.firstChild.lastChild.removeNode();
}
```

Where GetNewsCount is a custom function used to count the number of news items or entries within the news element of the passed XML.

The EditNewsEntry function works a little differently, obviously. After all, its editing news, not adding it. Here's what it looks like:

```
EditNewsEntry = function(news_xml, node_index, title, body){
    var entry = GetEntry(news_xml, node_index);
    if (title == "" && body == ""){
        entry.removeNode();
        return (0);
    }else{
        if (title == "") title = "(none)";
        if (body == "") body = "(none)";
    }
    entry.attributes.date = new Date().toString();
    var titleTextNode = entry.firstChild.firstChild;
    var bodyTextNode = entry.firstChild.nextSibling.firstChild;
    titleTextNode.nodeValue = title;
    bodyTextNode.nodeValue = body;
}
```

Since this function can edit any one of the multiple news entries, in addition to the arguments received by AddNewsEntry, EditNewsEntry also gets a node_index argument that specifies which node exactly is to be edited. The reference to that particular node is then obtained using the custom function GetEntry.

```
var entry = GetEntry(news_xml, node_index);
```

Next some checks are made for both the title and news arguments. If both are empty, the entry is removed and the function is exited using return - this being the little added delete feature (which could have just as well been used separately). Otherwise if any one of them is empty, "(none)" is used to represent their contents.

```
if (title == "" && body == ""){
    entry.removeNode();
    return (0);
}else{
    if (title == "") title = "(none)";
    if (body == "") body = "(none)";
}
```

All that remains is the simple re-assignment of text values. As with many other cases, variables are used to help show where this is happening.

```
entry.attributes.date = new Date().toString();
var titleTextNode = entry.firstChild.firstChild;
var bodyTextNode = entry.firstChild.nextSibling.firstChild;
titleTextNode.nodeValue = title;
bodyTextNode.nodeValue = body;
```

Once a new entry is added with AddNewsEntry or an existing one edited or deleted with EditNewsEntry, The XML can then be saved to the server. This is accomplished using the SaveNews function.

```
SaveNews = function(news_xml){
    content_txt.htmlText = "<i>Saving and Loading...</i>";
    news_xml.xmlDecl = ""; // fixes duplication bug
    news_xml.sendAndLoad(save_script, news_xml);
}
```

Using sendAndLoad the new version is sent to the server to be saved and the result which is simply the new XML is sent back to the news_xml instance whose onLoad displays the entries in the XML received.

```
news_xml.onLoad = function(success){
    if (success) ShowNews(this);
    else content_txt.text = "Error loading XML file";
}
```

That combined with the other details of the file give you a working news editor! Granted, these aren't RSS feeds we're dealing with here, but the simple format of the XML used here still gets the job done (and an RSS feed would actually not be too terribly far off from this).

Note: If you look at the ShowNews function in the source file, you'll notice that the title and news body is set to the text field's htmlText.

```
content_txt.htmlText += "<b>" + title +"</b><br>"
content_txt.htmlText += entries[i].attributes.date + "<br>"
content_txt.htmlText += "                                        <br>"
content_txt.htmlText += body +"<br><br>";
```

This means that when you make or edit a news entry, you would need to use text that is valid HTML (i.e. using character entity references where needed. This does NOT mean that the XML contains invalid characters. The XML correctly replaces those characters as needed when stored. However, when they reloaded and placed in the textfield, they are restored to their respective characters. For example, &lt; becomes <. This can be desirable or not. If you don't want this behavior, use the text nodes toString() value as opposed to their nodeValue to be placed within the htmlText of the text field.

Specific Issues

Even knowing all you need to know or all you should know doesn't mean you won't find yourself caught up in a jam without knowing why things aren't working the way they're supposed to. This section will point out some of those specific issues (that haven't been addressed earlier) and what can be done to resolve them.

XML Node Reference Not Returning Value

Though its possible the reference is not correct (something you should tripple check) this problem is usually associated with XML loaded into an XML instance whose ignoreWhite property was not set to true. When ignoreWhite is... ignored, its default is used with is false. This means all white space between adjacent elements are treated as text nodes throwing off your nodes count and disrupting the order of where you think everything should be. Set ignoreWhite to true and you should be golden.

```
my_xml.ignoreWhite = true;
```

Trouble With XML onLoad In AS2 Classes
This is probably the biggest issue out there for people using XML in ActionScript 2.0 classes. This doesn't necessarily revolve around XML so much, but rather about dealing with callback event handlers like onLoad from within classes.

The problem is that if you define an onLoad event from within a class method, the onLoad script cannot access class properties. Example

```
class XMLContentLoader {
    public var target_txt:TextField;
    private var _xml:XML;

    function XMLContentLoader(url:String, target:TextField){
        target_txt = target;
        _xml = new XML();
        _xml.ignoreWhite = true;
        _xml.onLoad = function(success){
            if (success) target_txt.text = this.firstChild.toString();
        }
        _xml.load(url);
    }
}
```

Here, the XMLContentLoader class creates instances that loads XML from a URL string and displays its firstChild (the full XML document) into a textfield saved under the target_txt property. Problem is, the onLoad can't correctly reference the target_txt property. WHY, sweet child of mine WHY!?!

The reason for this is because once you've entered that onLoad method, you are no longer in the scope of the class instance. You have now entered the scope of the XML instance. Within this scope, when attempting to access target_txt, you are attempting to access target_txt within the XML instance and not the XMLContentLoader instance. Thankfully, there are a couple of ways around this.

1. Use a local variable. If you created the event handler (onLoad) function within a class method, then that function has access to all local variables declared within that method in which its defined. If you assign the needed property reference to a local variable in that host method, it will carry into the scope of the scope of the handler giving you a valid reference.

```
var txt = target_txt;
_xml.onLoad = function(success){
    if (success) txt.text = this.firstChild.toString();
}
```

For variable reference only; not safe for calling class methods.

2. Define the variable within the object receiving the handler. Since when in the onLoad the _xml instance is looking for its own target_txt and not the class instance's, what you could do is just copy that variable, keeping the same name, within the _xml instance. Then, the onLoad would correctly resolve that variable when it is referenced. Now, the flip side to this in AS 2.0 is that, at least with the XML object, that you're dealing with instances of a non-dynamic class. This means that you cannot, technically, add or access

properties or methods that are not defined within its class definition. You can, however, trick Flash's compiler to ignore this by using associative array syntax to define your property, in this case, target_txt.

```
_xml["target_txt"] = target_txt;
_xml.onLoad = function(success){
    if (success) target_txt.text = this.firstChild.toString();
}
```

Using _xml["target_txt"] tricks the compiler as when using [] to access variables. The compiler can't be certain of what value is within the brackets (it could be a variable that could be valid or invalid) so it ignores the reference altogether letting you get by with compiler deceiving heathenry. When in the onLoad, target_txt is correctly found because it is defined within the _xml instance. To add injury to insult, the compiler isn't smart enough to understand this change in scope either (actual running ActionScript knows the change, just not the compiler creating the SWF) so as you attempt to access target_txt in the onLoad function without [], the compiler won't complain as it still assumes that when you're using you're in the scope of the class where target_txt is a valid property.

For variable reference only; not safe for calling class methods.

3. Define a reference to the class instance within the object receiving the handler. This approach is a little like the previous only this one adds a reference to the class instance itself and not just one property. This is helpful if you're dealing with many variables or you need to pass the class instance itself in to function calls within the handler function's scope etc. Here, again, the compiler gets in the way when dealing with non-dynamic XML instances. There's an added complication, though, as when using a reference in this manner, you're not duplicating an existing class property so the compiler will know it doesn't exist where ever you use it. Thankfully, the compiler did learn one thing in its limited schooling, and that was that this-referenced properties within non-class function definitions should go ignored. So, within the onLoad, a class reference variable can be left alone by the compiler if this is used to specify that the property does in fact belong to the object in which the handler is being defined (of course this is a little contradictory to some of its other behavior, but what are you going to do). The initial class reference when defined within the XML instance will still need [] to be ignored by the compiler, however.

```
_xml["hostInstance"] = this;
_xml.onLoad = function(success){
if (success) this.hostInstance.target_txt.text = this.firstChild.toString();
}
```

Note: you could also create an undefined Object property in the class called hostInstance to prevent the need to use this. Also, if you're confident in your coding, you could also just drop the typing altogether for the XML instance preventing any of the compiler complications mentioned above.

```
private var _xml;
```

3.1. Combine 3 and 1. Use a local variable to reference the class instance. This takes away the need to a) define any extra variables within the object getting the event handler b) trick the compiler and c) define multiple variables for what ever properties you wish to reference

```
var host = this;
_xml.onLoad = function(success){
    if (success) host.target_txt.text = this.firstChild.toString(); }
```

4. Use the Delegate Utility. Available as of Flash MX 2004 7.2, this is probably the most "official", though possibly also the most confusing (and less apparent) method. Delegate is a class in mx.utils available to ActionScript 2.0 that allows you to change the scope of a function call from one object to another. Basically its a wrapper for the functionality that function.call() and function.apply() provide. Delegate just makes it a little easier to intercept function definitions in one object which are to be executed within the scope of another. Here's a simple example of its use.

```
import mx.utils.Delegate;
var objectA:Object = {name: "object A"};
var objectB:Object = {name: "object B"};

function getName():String {
    return this.name;
}
objectB.getName = Delegate.create(objectA, getName);
trace( objectB.getName() ); // traces "object A";
```

Here, the static method create is used off of the Delegate class to create a function (getName) that is assigned to object B but, when run, is run in the scope of object A. This means that any instance of 'this' in that function will refer to object A instead of object B even though it is being called from object B.

This can then be applied to the XML example were we could have the onLoad be run in the scope of the class thereby making all references to class properties valid as the call would be within the scope of the class instance and not the XML object. For the sake of simplicity, we'll make the function to be used in the onLoad a method of the class. This is not uncommon to do anyway. Here, it makes the assignment using Delegate.create() a lot easier. Here is the full modified class:

```
import mx.utils.Delegate;
class XMLContentLoader {
    public var target_txt:TextField;
    private var _xml:XML;

    function XMLContentLoader(url:String, target:TextField){
        target_txt = target;
        _xml = new XML();
        _xml.ignoreWhite = true;
        _xml.onLoad = Delegate.create(this, onLoadEvent);
        _xml.load(url);
    }

    function onLoadEvent(success:Boolean):Void {
        if (success) target_txt.text = _xml.firstChild.toString();
    }
}
```

First, in order to use Delegate, at least by its short name, import is used to bring it in from mx.utils.Delegate (otherwise, you'd have to run it as mx.utils.Delegate.create()). That happens above the class definition before anything else. Then, in the constructor you can see it being used in defining the onLoad event for the _xml XML instance. It takes two arguments, an object and a function. The object is the object in which the function is going to be run. Since we want the onLoad to run within the scope of the class, this is passed in (representing the class instance) as the object. The function is the class's own onLoadEvent function defined below. It now, when run, will correctly reference target_txt as a property of the class without any trouble. Of course, you can also see that because the scope of the onLoad method is no longer within the XML instance and this represents the class, _xml has to be used in order to access content of the XML instance and what had been loaded.

Note: If you use a method for your onLoad event that is defined as a method within the class like the onLoadEvent method above, then options 1 and 3.1 which use local variables, would not be a valid solution for the scoping problem.

XML URL Is Correct, But Still Won't Load
If you're trying to load XML across domains, i.e. trying to have a SWF on your site load an XML document from another site, you're XML may not make it because of security restrictions first implemented in Flash Player 6. To see how to get around this, read a technote available from Colin Moock's web site.

Sending Formatted XML And/Or CDATA To The Server
In using ignoreWhite, Flash physically removes the extraneous white space between nodes from the XML document as its brought into Flash often ruining your formatting. If that XML is then sent back to the server, the white space remains removed and what you get is a long line of elements and text nodes that seem all to blend together.

Similarly, Flash converts CDATA sections in XML to simple text nodes within an XML instance. Flash reads CDATA fine when loaded but, like with the white space, in sending off the XML to the server, that CDATA is no longer CDATA and is instead a converted text node.

When Flash removes all your white space and converts CDATA to text nodes, you've just lost that readability. This is especially annoying when using other markup like HTML and more so when you want to look at that XML in some other context other than within Flash. After all, part of the advantage of having XML is that it's readable. What's more readable to you in the following examples?

```
<?xml version="1.0"?>
<gallery>
    <title>Image Gallery A</title>
    <author>senocular</author>
    <page date="01/04/2005" name="photo05.html">
        <![CDATA[<html>
            <head>
                <title>Photo #05</title>
            </head>
            <body>
                <div align="center">
                <p><b>Photograph 05 of 50</b></p>
                <img src="images/photo05.jpg" alt="Photo #05" />
                </div>
            </body>
        </html>]]>
    </page>
    <comments>Please add the date of posting.</comments>
</gallery>
```

or

```
<?xml version="1.0"?>
<gallery><title>Image Gallery
    A</title><author>senocular</author><page date="01/04/2005" name="photo05.html">
    <![CDATA[&lt;html&gt;
        &lt;head&gt;
            &lt;title&gt;Photo #05&lt;/title&gt;
        &lt;/head&gt;
        &lt;body&gt;
            &lt;div align="center"&gt;
            &lt;p&gt;&lt;b&gt;Photograph 05 of 50&lt;/b&gt;&lt;/p&gt;
            &lt;img src="images/photo05.jpg" alt="Photo #05" /&gt;
            &lt;/div&gt;
        &lt;/body&gt;
    &lt;/html&gt;]]>
</page><comments>Please add the date of posting.</comments></gallery>
```

You can get around this using the previously mentioned format function. The format function, in its original design, is used to create a multi-lined, indented string representation of XML despite it having been formatted otherwise (say, as a result of using ignoreWhite). Using this to generate XML that is to be sent to the server pretty much solves the ignoreWhite problem right there. You would most likely have to use a loadVars object to send the string, but that's not really a problem.

The CDATA still remains. But, similarly, this too can be solved using format. What format does is re-writes XML node for node so that it can be relayed in an alternative format, namely, a readable one. A small modification to this, and some manual editing of your XML instance, can make it so that format correctly converts text nodes back into CDATA sections.

Because all CDATA sections are initially converted into text nodes, there's no real way to identify a real text node from a previously defined-as-CDATA node. This is why the manual editing previously mentioned is required. Basically you'd just have to go through the XML and manually mark specific text nodes as being CDATA so that when format runs through all the nodes, it can check for this mark and appropriately change all the appropriate text nodes into CDATA. Here is the rewrite of format with the alteration in bold:

```
XMLNode.prototype.format = function(indent){
    if (indent == undefined) indent = "";
    var str = "";
    var currNode = this.firstChild;
    do{
        if (currNode.hasChildNodes()){
            str += indent + currNode.cloneNode(0).toString().slice(0,-2) + ">\n";
            str += currNode.format(indent+"\t");
            str += indent + "</" + currNode.nodeName + ">\n";
        }else{
            if (currNode.isCDATA) str += indent + "<![CDATA[" + currNode.nodeValue + "]]>\n";
            else str += indent + currNode.toString() + "\n";
        }
    }while (currNode = currNode.nextSibling);
    return str;  }
```

An additional if statement is added that checks for an *isCDATA* property within the current node in the iteration. If the value is true then a CDATA section is created using the node's nodeValue instead of it's toString() representation which gets rid of all the character entity references and uses the actual node's value which, being within the CDATA tag is acceptable.

Now its just a matter of going through and defining a true isCDATA property to all the text nodes you wish to formatted as CDATA sections.

```
my_xml.firstChild.firstChild.isCDATA = true;
```

Double-spacing Occurs With Text Nodes And/Or CDATA
When text is brought into Flash and displayed in a text field, whitespace from that text is retained allowing you to mainttain tabbing and line breaks. Depending on your operating system, line breaks may consist of one of the following:

Character	Represents	OS
\r	carriage return	Mac OS <= 9
\n	newline	Unix (OSX)
\r\n	(both)	Windows

Flash understands both \r and \n as line breaks. The problem comes with Windows line breaks which actually uses both \r and \n to represent one single line break. When this is brought into Flash, that single line break is interpreted as two.

To solve this problem, you can either replace all instances of either \r or \n from within Flash before the text is added to a text field or you can simply remove it beforehand (prefered) using a more advanced text editor - something other than notepad - which will let you distinguish between characters used for line breaks.

Trouble With Raw Post Data In PHP
So the Flash XML methods send and sendAndLoad send your XML to a URL as raw Post data. The variable commonly used to access this post data in PHP is:

```
$HTTP_RAW_POST_DATA
```

Sometimes this doesn't always work (or your version of PHP may not support it). If you're having problems using $HTTP_RAW_POST_DATA, try instead using

```
file_get_contents("php://input");
```

php://input represents the standard input stream of the PHP file. Not only does the above solution use less memory than $HTTP_RAW_POST_DATA, but it also does not require any special php.ini directives to function properly (which may be causing you problems with $HTTP_RAW_POST_DATA in the first place). For versions of PHP >= 4.3.0.

Special Characters in XML
If you're having trouble getting special characters to load into Flash, there are a couple of things you can do.

One thing you could do is try saving your XML with UTF-8 encoding (UTF-8 stands for Unicode Transformation Format-8, an octet lossless encoding of Unicode characters). Windows Notepad offers this through an encoding pull-down in the save dialog. The default is ASCII. Using UTF-8 will make characters like æ, ø, and å recognizable by Flash*.

If for some reason you can't save your XML as UTF-8, you can tell Flash to interpret loaded text based on the traditional code page of the OS. Since, normally, for Flash to interpret a text file as Unicode, it has to *be* Unicode, you can instead use System.useCodepage to tell Flash to treat them as otherwise. By default, System.useCodepage is false. Setting it to true will tell Flash to interpret externally loaded text based on the code page of the OS.

```
System.useCodepage = true;
```

Then, your characters should come through as expected.

Another alternative is using Unicode escape characters. A Unicode escape sequence consists of \u and four characters to represent a character in Unicode. For example, \u00A5 represents the character for Yen (¥)*. Placing these escape sequences within your text can provide you with the Unicode characters needed much in the same way as \r and \n create line breaks. You can find more character sequences from unicode.org.

* Assuming the font you're using supports the characters being used.

In using UTF-8 encoding, you may want to specify this in your XML declaration.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Conclusion
It wasn't my intent to cover XML and database interaction here (nor XML Sockets); I just wanted to hit the bases of normal XML usage. So if you were expecting more out of this *introductory* tutorial, I apologize. I will say a few things about databases before wrapping up, however.

First off, yes, you can store XML in a database like a MySQL database. Nothing surprising there. But you don't *need* a database to use XML. You can just as easily save it as a simply text file on your server with something like PHP or ASP... or even manually your own darn self using a FTP client (if you have no plans of letting users edit it dynamically). After all, XML is just text. You just can't be using the FTP client via Flash as you can with PHP and server-side scripting of the likes.

Now there are XML-based databases in existence. For the most part, however, XML is not a formidable replacement for other forms of databases (like MySQL). Often is the case that the XML content, especially when dealing with a lot of it, would be preserved much more efficiently in another database system rather than an XML "database" or as an XML file. This doesn't mean XML is out the window at this point. It just means that there are more efficient means of handling (a lot of) content compared to that provided by XML.

For more XML's roll on the server-side/database side of things, see the following discussion from xml.org (not directly Flash related).

You can also find a little more about XML and server-side interaction from the Flash XML Faq.

There are a lot of possibilities with XML in Flash as we've seen with the many examples presented throughout this tutorial. But still, it really all boils down to the interaction of external data with a running Flash movie. Even the more complicated examples deal more with data management and simply dealing with the complex nature of the XML object. Once mastered, however, you'll be able to take full advantage whatever content you need and do with that content anything your imagination and creativity can unleash.

Hopefully this tutorial and the examples within have provided you with a better understanding of it all. Some parts may have been long-winded while others may have been a little brief in the ways of description, but all in all I hope you've come out of it learning at least a little something.

cheers

senocular
senocular.com

External Links Summary	
XML on xml.org	origin
XML on w3.org	origin
SGML	origin
Entity references	origin
XHTML	origin
FTML	origin
Layer51 Prototypes	origin
XPath	origin
XFactor Studio XPath for AS1	origin
XFactor Studio XPath for AS2	origin
Flash MX 2004 7.2 Updater	origin
Colin Moock: Loading Cross Domain XML	origin
Unicode Charts	origin
XML vs Database Discussion	origin
Flash XML FAQ	origin

Source Files Summary	
Load Security Text (zip)	origin
Squirrel Finder (zip)	origin
Portfolio (zip)	origin
Searching Best of Kirupa.com (zip)	origin
MX 2004 ActionScript Classes (zip)	origin
Sorted Grades List (zip)	origin
Additional Helpful Methods (as)	origin
MP3 Playlist (zip)	origin
Simple Editor (zip)	origin
News Editor (zip)	origin