**Marknote Developer's Guide**

**Parsing XML**

**Reading XML From a String**

*Marknote* includes a marknote.Parser class to parse XML from a string or from a URL. Its output is a marknote.Document object.

Here is an example of parsing XML from a string into a marknote.Document object:

var str =  
 '<books>' +  
 '<book title="A Tale of Two Cities"/>' +  
 '<book title="1984"/>' +  
 '</books>';  
  
var parser = new marknote.Parser();  
var doc = parser.parse(str);  
  
alert(doc.toString()); // show the formatted XML

The code above produces the following output:

<books>  
 <book title="A Tale of Two Cities" />  
 <book title="1984" />  
</books>

**Reading XML Directly From a URL**

*Marknote* can easily parse XML documents from a URL within its own HTTP domain (if you want to access an XML at a URL external to an application's HTTP domain, one way to do that is by pointing to a local URL which streams back the external XML).

Here is an example of code that reads XML directly from a specified URL:

var url = "myXmlFiles/movies.xml";  
var parser = new marknote.Parser();  
  
// optional HTTP request parameters  
// (used to construct the HTTP query string, if any)  
var urlParams = {  
 param1: "zzz",  
 param2: "abc"  
};  
  
// parse the file  
// (for POST requests, use "POST" instead of "GET")  
var doc = parser.parseURL(url, urlParams, "GET");  
  
// alert out the parsed document  
alert(doc.toString());

while this code checks the status of the request:

// check the status  
alert("Request status is: " + parser.getXHRStatus() +  
 " and the status text is: " + parser.getXHRStatusText());

which will produce output such as the following, if the request fails:

Request status is: 404 and the status text is: Not Found

or, like this, if the request succeeds:

Request status is: 200 and the status text is: OK

An HTTP status code of 200 indicates success. For more information on HTTP status codes, see <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

Note that, unlike an asynchronous AJAX call, when you read XML directly from a URL this way, you're reading the XML *synchronously*, meaning that Javascript will wait until the parseURL command completes before processing its next instruction. Since the command is synchronous, you don't need to code a callback function (as you would with a standard AJAX request) to process the XML that comes back.

In those situations where it makes makes more sense to read XML *asynchronously* using AJAX, refer to the next section.

**Reading XML Asynchronously Using AJAX**

**Creating the AJAX Callback Function**

To parse XML from the responseText object resulting from an AJAX call, first define a callback function to handle the response received back from the AJAX request.

Here's an example:

function myCallback(doc, params) {  
   
 var pretty = doc.toString();  
   
 alert(pretty); // alert out the formatted XML  
   
 var callbackParamValue = params.foo;  
   
 if (callbackParamValue === 'foo') {  
 // do something  
 } else if (callbackParamValue === 'bar') {  
 // do something else   
 } else {  
 // do something else   
 }  
  
}

The first parameter of the callback function, doc, is a marknote.Document object holding the XML parsed from the URL. The second parameter of the callback function, params, is optional, and holds an object containing any additional information we want to pass to the callback function.

**Making the AJAX Call**

Once you've got the callback function set up, then all that's left to do is to fire off the AJAX request and use the callback function to process what comes back in the AJAX response.

Here is an example of code that reads XML asynchronously from a specified URL, and then fires the callback function shown above to process the resulting XML:

var url = "movies.xml";  
var ajax = new marknote.AJAX();  
  
// optional request parameters  
var urlParams = {  
 foo: "zzz",  
 bar: "abc"  
};  
  
// optional callback parameters  
var callbackParams = {  
 foo: "aaa",  
 bar: "bbb"  
};   
  
// read the URL  
// (the callback handles the response)  
ajax.read(  
 url,  
 urlParams,   
 myCallback,  
 callbackParams,  
 "GET"  
);

The code above produces output such as:

<movies>  
 <movie title="The Seventh Seal" />  
 <movie title="The Terminator" />  
</movies>

**Checking the Success of the AJAX Response**

You may want to include AJAX response status checking in your AJAX callback function.

Here's an example of callback code that includes a check of the status of the AJAX response:

//  
// the callback function  
//  
function betterCallback(doc, params) {  
  
 // "this" is the marknote.AJAX object  
 var ajaxObj = this;  
  
 alert("HTTP status code is: " + ajaxObj.getStatus() + " and the HTTP status text is: " + ajaxObj.getStatusText());  
  
 // branch code here depending on the status, as needed  
  
 var pretty = doc.toString();  
  
 alert("xml is:\n" + pretty);  
  
}  
  
//  
// read the URL  
//  
var url = "movies.xml";  
var ajax = new marknote.AJAX();  
ajax.read(url, null, betterCallback);

Assuming this code parses the XML successfully, it will produce this output:

HTTP status code is: 200 and the HTTP status text is: OK

and then the formatted XML, such as:

<movies>  
 <movie title="The Seventh Seal" />  
 <movie title="The Terminator" />  
</movies>

The marknote.AJAX object also includes a getRequest function to get the underlying XML HTTP Request (*XHR*) object, and a getResponseText function to get the underlying XHR response text.

**Using Marknote and AJAX within a Javascript Framework**

Many of the excellent Javascript frameworks that are now available, such as jQuery (<http://jquery.com>) or Ext JS (<http://sencha.com>), and a number of others, provide strong AJAX handling. If you use such a framework, you can then of course use that framework's AJAX classes to make your AJAX call instead of using a marknote.AJAX object to fire off the request.

The bottom line is that all AJAX calls work the same way: you make the AJAX call, and you provide a callback function to handle the response text that comes back from the AJAX call. You may use *Marknote* code in any AJAX callback handler.

Frameworks such as Ext JS also include a proxy class to facilitate making AJAX calls to external URL's.

**Working with XML Documents**

**Creating an XML Document**

**Ways to Create an XML Document**

There are several ways to create a new marknote.Document object:

* Parse the XML from a string or URL.
* Create a new marknote.Document object from scratch.
* Clone an existing marknote.Document object.

**Reading XML into a Document**

*Marknote* can parse XML into a marknote.Document object from a string or from a URL. See [Parsing XML](http://code.google.com/p/marknote/wiki/DevelopersGuide#Parsing_XML).

**Creating an XML Document from Scratch**

To create a new marknote.Document object from scratch, simply instantiate it, and then add content to it.

Here is an example of code that creates a new marknote.Document object, and then adds elements and attributes to the document:

// create the document, attach the root element  
var doc = new marknote.Document();  
var rootElement = new marknote.Element("songs");  
doc.setRootElement(rootElement);  
  
// add Truckin'  
var truckingElement = new marknote.Element("song");  
truckingElement.setAttribute("artist", "Grateful Dead");  
truckingElement.setAttribute("title", "Truckin'");  
rootElement.addChildElement(truckingElement);  
  
// add Smoke on the Water  
var smokeElement = new marknote.Element("song");  
smokeElement.setAttribute("artist", "Deep Purple");  
smokeElement.setAttribute("title", "Smoke on the Water");  
rootElement.addChildElement(smokeElement);  
  
// add Hair of the Dog  
var dogElement = new marknote.Element("song");  
dogElement.setAttribute("artist", "Nazareth");  
dogElement.setAttribute("title", "Hair of the Dog");  
rootElement.addChildElement(dogElement);

The code above produces this XML document:

<songs>  
 <song artist="Grateful Dead" title="Truckin'" />  
 <song artist="Deep Purple" title="Smoke on the Water" />  
 <song artist="Nazareth" title="Hair of the Dog" />  
</songs>

**Cloning an Existing XML Document**

Cloning an existing XML document creates a new deep copy, identical to the original document.

Here is an example of cloning an XML document:

var parser = new marknote.Parser();  
var xml = '<songs><song title="Smoke on the Water"/></songs>';  
var doc = parser.parse(xml);  
  
// clone the first document  
var clonedDoc = doc.clone();  
  
alert(clonedDoc.toString());

which produces this output:

<songs>  
 <song title="Smoke on the Water" />  
</songs>

**Setting the DOCTYPE**

An XML document type declaration (DOCTYPE) ties an XML document to a document type definition (DTD). A DOCTYPE, in turn, consists of several components, including the XML document root (top) element name, an availability indicator, a full public identifier (FPI), and a DTD URL.

The simplest way to create a marknote.DOCTYPE object is to pass in a string representation of the full DOCTYPE element (including the <! and > tag characters) into its object constructor. For example, this DOCTYPE:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

... this sets a marknote.DOCTYPE to a marknote.Document object:

var parser = new marknote.Parser();  
var xml = '<songs><song title="Smoke on the Water"/></songs>';  
var doc = parser.parse(xml);  
  
var doctype =   
 new marknote.DOCTYPE(  
 '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">'  
 );  
  
doc.setDOCTYPE(doctype);  
  
alert(doc.toString());

The code above produces this output:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">  
<songs>  
 <song title="Smoke on the Water" />  
</songs>

You can also set a DOCTYPE from its individual parts. For example, given the DOCTYPE:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

the following comprise the DOCTYPE's parts:

|  |  |
| --- | --- |
| **DOCTYPE Part** | **Value** |
| Top Element | html |
| Availability | PUBLIC |
| Formal Public Identifier (FPI) | "-//W3C//DTD XHTML 1.0 Strict//EN" |
| DTD URL | "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>" |

The formal public identifier (FPI), in turn, also consists of several parts.

For example, given the FPI:

"-//W3C//DTD XHTML 1.0 Strict//EN"

the following comprise the FPI's parts:

|  |  |
| --- | --- |
| **FPI Part** | **Value** |
| Registration | - (- means not registered, whereas + means registered) |
| Organization | W3C |
| Public Text Class | DTD |
| Public Text Description | XHTML 1.0 Strict |
| Public Text Language | EN |

The following example sets the DOCTYPE for an XML document from its parts:

var parser = new marknote.Parser();  
var xml = '<songs><song title="Smoke on the Water"/></songs>';  
var doc = parser.parse(xml);  
  
// top element, availability  
var topElement = "html";  
var availability = "PUBLIC";  
  
// create the formal public identifier  
var registration = "-";  
var organization = "W3C";  
var publicTextClass = "DTD";  
var publicTextDescription = "XHTML 1.0 Strict";  
var publicTextLanguage = "EN";  
var fpi = new marknote.FPI(  
 registration,  
 organization,  
 publicTextClass,  
 publicTextDescription,  
 publicTextLanguage  
);  
  
// the DTD URL  
var url = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd";  
  
// create the DOCTYPE  
var doctype = new marknote.DOCTYPE(topElement, availability, fpi, url);  
  
// set the DOCTYPE to the XML document  
doc.setDOCTYPE(doctype);  
  
alert(doc.toString());

which again produces:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">  
<songs>  
 <song title="Smoke on the Water" />  
</songs>

**Adding Processing Instructions**

A processing instruction consists of a target, as well as a set of attributes which comprise the processing instruction data.

The default processing instruction for an XML document is:

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

Taking the above example, xml is the processing instruction target, while encoding="UTF-8" is the processing instruction data.

To create a marknote.ProcesssingInstruction object with the default target of xml and the default data of encoding="UTF-8", simply instantiate the object; for example:

var parser = new marknote.Parser();  
var xml = '<songs><song title="Smoke on the Water"/></songs>';  
var doc = parser.parse(xml);  
  
//create a default processing instruction of  
var pi = new marknote.ProcessingInstruction();  
  
//add the processing instruction to the document  
doc.addProcessingInstruction(pi);  
  
alert(doc.toString());

The code above produces this output:

<?xml version="1.0" encoding="UTF-8" ?>  
<songs>  
 <song title="Smoke on the Water" />  
</songs>

To add a processing instruction other than the default xml processing instruction, specify the target and data (attributes) for the processing instruction. For example, to add the processing instruction:

<?xml-stylesheet href="mystyles.css" type="text/css">

use code such as:

// create a processing instruction  
var pi = new marknote.ProcessingInstruction("xml-stylesheet");  
pi.setAttributeValue("href", "mystyles.css");  
pi.setAttributeValue("type", "text/css");  
  
// add the processing instruction to a document  
var doc = new marknote.Document();  
doc.addProcessingInstruction(pi);

To remove a processing instruction from a marknote.Document object, use the removeProcessingInstruction method, specifying the processing instruction to remove by its target name. For example, to remove a processing instruction with a target name of xml-stylesheet:

// doc is a marknote.Document object  
doc.removeProcessingInstruction("xml-stylesheet");

**Writing XML Output**

**Formatting XML Output**

To generate a formatted XML serialization of a marknote.Document object, use its toString method. For example:

// "doc" is a marknote.Document object  
var xmlString = doc.toString();

**Controlling Output Identation**

When using the toString method of a marknote.Document object, the output formatter uses the tab character for indentation by default. To instead provide an alternate indentation, pass the indentation as a parameter to the toString method. For example:

var parser = new marknote.Parser();  
var xml = '<songs><song title="Smoke on the Water"/></songs>';  
var doc = parser.parse(xml);  
  
var indent = " ";  
var xmlString = doc.toString(indent);  
  
alert(xmlString);  
  
// now use two tabs as the indent instead  
var anotherIndent = "\t\t";  
var xmlString2 = doc.toString(anotherIndent);  
  
alert(xmlString2);

The code above produces:

<songs>  
 <song title="Smoke on the Water" />  
</songs>

and:

<songs>  
 <song title="Smoke on the Water" />  
</songs>

**Working with Elements**

**Creating an Element**

**Ways to Create an Element**

A marknote.Element object represents an XML element. To create a new `marknote.Element object, you can either create one from scratch, or clone an existing one.

**Creating an Element from Scratch**

To create a new marknote.Element object from scratch, simply instantiate it, and add any applicable attributes and content to it.

Here is an example of code that creates a new marknote.Element object, and then adds attributes and text content to it:

var truckingElement = new marknote.Element("song");  
  
truckingElement.setAttribute("artist", "Grateful Dead");  
truckingElement.setAttribute("title", "Truckin'");  
  
truckingElement.setText("A story about a guy being set up like a bowling pin.");  
  
alert(truckingElement);

The code above produces this output:

<song artist="Grateful Dead" title="Truckin'">A story about a guy being set up like a bowling pin.</song>

**Creating an Element Using a Qualified Name**

A "marknote.Element` object can use a qualified name, derived from an XML namespace. For example, the following element:

<myNamespace:myElementName>here is some content</myNamespace:MyElementName>

has the name myElementName, within the myNamespace namespace.

To create a qualified name, use a marknote.QName object, passing in the prefix and local part as parameters to its constructor. For example, the following code creates a marknote.Element object for the element shown above:

var qname = new marknote.QName("myNamespace", "myElementName");  
var elem = new marknote.Element(qname);  
  
elem.setText("here is some content");

**Cloning an Existing Element**

Cloning an existing XML element creates a new deep copy, identical to the original element.

For example, this code:

var elem = new marknote.Element("lyrics");  
elem.setText("On a dark desert highway, cool wind in my hair");  
  
var clonedElem = elem.clone();  
clonedElem.setAttribute("color", "orange");  
  
alert(clonedElem);

produces this output:

<lyrics color="orange">On a dark desert highway, cool wind in my hair</lyrics>

**Setting and Reading Attributes**

**Setting Attributes**

An XML element may contain any number of attributes. For example, given the following XML element:

<element1 attribute1="foo" attribute2="bar" />

attribute1 and attribute2 are attributes of element1, with the values "foo" and "bar", respectively.

This code shows two ways of creating a marknote.Attribute object:

//quick way to create an attribute  
var attrib1 = new marknote.Attribute("attribute1", "foo");  
  
// longer way to create an attribute  
var attrib2 = new marknote.Attribute();  
attrib2.setName("attribute2");  
attrib2.setValue("bar");  
  
var elem = new marknote.Element("element1");  
elem.setAttribute(attrib1);  
elem.setAttribute(attrib2);  
  
alert(elem.toString());

and produces this output:

<element1 attribute1="foo" attribute2="bar" />

This code first sets an element's attributes using an array of marknote.Attribute objects, and then adds one more additional attribute:

var attributes = [  
 new marknote.Attribute("attribute1", "foo"),  
 new marknote.Attribute("attribute2", "bar")  
];  
  
var elem = new marknote.Element("element1");  
  
// set an array of attributes  
elem.setAttributes(attributes);  
  
// set one additional attribute  
elem.setAttribute("color", "orange");  
  
alert(elem.toString());

and produces this output:

<element1 attribute1="foo" attribute2="bar" color="orange" />

**Reading Attributes**

The following code reads the value of an attribute from an element:

var attributes = [  
 new marknote.Attribute("bass", "medium"),  
 new marknote.Attribute("treble", "low")  
];  
  
var elem = new marknote.Element("element1");  
  
elem.setAttributes(attributes);  
  
var attr = elem.getAttribute("treble");  
var value = attr.getValue();  
  
alert(value);

and produces this output:

low

This code reads an array holding the marknote.Attribute objects for an element:

var attributes = [  
 new marknote.Attribute("bass", "medium"),  
 new marknote.Attribute("treble", "low")  
];  
  
var elem = new marknote.Element("element1");  
  
elem.setAttributes(attributes);  
  
// get the attributes of the element as an array of marknote.Attribute objects  
var attributes = elem.getAttributes();  
  
// loop through the array of attributes, and read each attribute name & value  
var output = 'attributes:';  
for (i in attributes) {  
 output += '\n name: ' + attributes[i].getName() + ' value: ' + attributes[i].getValue();  
}  
  
alert(output);

and produces this output:

attributes:  
 name: bass value: medium  
 name: treble value: low

**Removing Attributes**

The following code removes an attribute from an element:

var artist = new marknote.Element("artist");  
  
artist.setAttribute("lastName", "Mozart");  
artist.setAttribute("firstName", "Wolfgang");  
artist.setAttribute("middleName", "Amadeus");  
  
artist.removeAttribute("firstName");  
  
alert(artist.toString());

and produces this output:

<artist lastName="Mozart" middleName="Amadeus" />

While this code removes all of the attributes from an element:

var artist = new marknote.Element("artist");  
  
artist.setAttribute("lastName", "Mozart");  
artist.setAttribute("firstName", "Wolfgang");  
artist.setAttribute("middleName", "Amadeus");  
  
artist.removeAllAttributes();  
  
alert(artist.toString());

and produces this output:

<artist />

**Cloning Attributes**

To create a deep clone of a marknote.Attribute object, use the clone method. For example, this code:

var artist = new marknote.Element("artist");  
artist.setAttribute("firstName", "Wolfgang");  
  
var anotherFirstName = artist.getAttribute("firstName").clone();  
alert(anotherFirstName.toString());

` produces this output:

firstName="Wolfgang"

**Converting an Attribute to a String Representation**

To serialize a marknote.Attribute object, use the toString method; for example:

// attrib is a marknote.Attribute object  
var str = attrib.toString();

The resulting string will be of the form:

attributeName="attributeValue"

**Setting and Reading Content**

**Adding Content**

An element may contain various types of content, including text content, a computer data (CDATA) section, child elements, or comments.

The following example illustrates management of content for an element:

// create the parent element, and add a comment to it  
var worksOfArt = new marknote.Element("worksOfArt");  
var comment = new marknote.Comment("Works of art includes songs, paintings, etc.");  
worksOfArt.addContent(comment);  
  
// create the first child element, and add text to it  
var song1 = new marknote.Element("song");  
text = new marknote.Text("Set up like a bowling pin, Knocked down, it gets to wearin' thin");  
song1.addContent(text);  
  
// create the second child element, and add a CDATA section to it  
var song2 = new marknote.Element("song");  
var cdata = new marknote.CDATA("We all came out to Montreux, On the Lake Geneva shoreline");  
song2.addContent(cdata);  
  
// add the child elements to the parent element  
worksOfArt.addContent(song1);  
worksOfArt.addContent(song2);

The result of the above code will be a marknote.Element object representing the following XML:

<worksOfArt>  
 <!--  
 Works of art includes songs, paintings, etc.  
 -->  
 <song>Set up like a bowling pin, Knocked down, it gets to wearin&apos; thin</song>  
 <song>  
 <![CDATA[  
 We all came out to Montreux, On the Lake Geneva shoreline  
 ]]>  
 </song>  
</worksOfArt>

**Removing Content**

The following example removes content at a specific position within an element's contents:

//create the parent element, and add a comment to it  
var worksOfArt = new marknote.Element("worksOfArt");  
var comment = new marknote.Comment("Works of art includes songs, paintings, etc.");  
worksOfArt.addContent(comment);  
  
// create the first child element, and add text to it  
var song1 = new marknote.Element("song");  
text = new marknote.Text("Set up like a bowling pin, Knocked down, it gets to wearin' thin");  
song1.addContent(text);  
  
// create the second child element, and add a CDATA section to it  
var song2 = new marknote.Element("song");  
var cdata = new marknote.CDATA("We all came out to Montreux, On the Lake Geneva shoreline");  
song2.addContent(cdata);  
  
// add the child elements to the parent element  
worksOfArt.addContent(song1);  
worksOfArt.addContent(song2);  
  
// remove child content at position 1  
worksOfArt.removeContent(1);  
  
alert(worksOfArt.toString());

The above code produces this output:

<worksOfArt>  
 <!--  
 Works of art includes songs, paintings, etc.  
 -->  
 <song>  
 <![CDATA[  
 We all came out to Montreux, On the Lake Geneva shoreline  
 ]]>  
 </song>  
</worksOfArt>

**Reading Content**

The following code retrieves an array of all content for an element:

// worksOfArt is an existing marknote.Element object  
var contents = worksOfArt.getContents();  
  
// iterate through the content and alert out the data type of each one  
for (var i=0; i<contents.length; i++) {  
 alert(marknote.Util.dataType(contents[i]));  
}

To retrieve content at a specific position within the an element's contents, use code such as:

// worksOfArt is an existing marknote.Element object  
// read the object at content index 1 for this element  
var content = worksOfArt.getContentAt(1);

**Setting and Reading Text and CDATA Text**

**Element Text Overview**

In addition to the content-handling methods outlined above, the marknote.Element class includes additional methods specific to managing text and CDATA text.

Elements often, but don't necessarily, include text content. For example, given the element:

<lyrics>We all came out to Montreux, On the Lake Geneva shoreline</lyrics>

"We all came out to Montreux, On the Lake Geneva shoreline" is the text content of the element. On the other hand, the following element contains no text content:

<house type="frame" stories="2" rooms="7" />

**Setting Text Content**

The following code sets text content to an element:

var lyrics = new marknote.Element("lyrics");  
lyrics.setText("We all came out to Montreux, On the Lake Geneva shoreline");

Setting the text of an element using setText results in all content first getting removed from the element (including child elements, except comments), before the text gets set.

**Setting CDATA Content**

The following code sets CDATA text content to an element:

var lyrics = new marknote.Element("lyrics");  
  
lyrics.setCDATAText("We all came out to Montreux, On the Lake Geneva shoreline");  
  
alert(lyrics);

and produces this output:

<lyrics>  
 <![CDATA[  
 We all came out to Montreux, On the Lake Geneva shoreline  
 ]]>  
</lyrics>}}}

Setting the CDATA text of an element using setCDATAText results in all content first getting removed from the element (including child elements), except comments, before the text gets set.

**Removing Text**

The following code removes all text (including all text and CDATA sections) attached directly to an element:

// lyrics is an existing marknote.Element object  
lyrics.removeText();

**Reading Text Content**

The following code reads the text content, or the CDATA text (if applicable), of an element:

// lyrics is an existing marknote.Element object  
var text = lyrics.getText();

**Character Entity Reference Conversion**

*Marknote* automatically performs character entity reference encoding and decoding when setting or reading text content for an element (this does not apply to CDATA text, since CDATA text is read and written literally). For example, given the string 4<5, *Marknote* will encode this value to 4&lt;5 when setting it as the element's text content, and will automatically decode it back to 4<5 when reading the text content back out.

For example, this code sets and then reads text content containing a character entity reference:

// create an element, and set its text  
elem = new marknote.Element("mathExpression");  
elem.setText("4<5");  
  
// alerts the string: <mathExpression>4&lt;5</mathExpression>  
alert(elem.toString());  
  
// alerts the string: 4<5  
alert(elem.getText());

**Setting and Reading Comment Text**

**Setting Comment Text**

In addition to using the content-handling methods outlined previously, the marknote.Element class includes additional methods specific to managing comment text.

The following code first removes all comments attached directly to an element, and then adds a new comment with the given text to the element:

var elem = new marknote.Element("lyrics");  
elem.setCommentText("This is a comment");  
  
alert(elem);

and produces this output:

<lyrics>  
 <!--  
 This is a comment  
 -->  
</lyrics>

**Removing Comments**

The following code removes all comments attached directly to an element:

// elem is an existing marknote.Element object  
elem.removeComments();

**Reading Comment Text**

The following code gets comment text for an element (if the element contains more than one marknote.Comment object, it returns the concatenation of all the comment text):

// elem is an existing marknote.Element object  
var commentText = elem.getCommentText();

**Working with Child Elements**

**Child Element Overview**

In addition to using the content-handling methods outlined previously, the marknote.Element class includes additional methods specific to managing child elements.

An element may contain child elements, which in turn may contain other child elements. The XML document itself contains a root element which acts as the top parent to all child elements in the document tree.

**Setting the Root Element**

To assign an element as the root element of a document, use code such as the following:

// create a document, attach the root element  
var doc = new marknote.Document();  
var songs = new marknote.Element("songs");  
doc.setRootElement(songs);

**Adding Child Elements**

To add child elements to a parent element, use code such as the following:

// create a document, attach the root element  
var doc = new marknote.Document();  
var songs = new marknote.Element("songs");  
doc.setRootElement(songs);  
  
// the first child element  
var firstSong = new marknote.Element("song");  
firstSong.setAttribute("title", "Truckin'");  
  
// the second child element  
var secondSong = new marknote.Element("song");  
secondSong.setAttribute("title", "Smoke on the Water");  
  
// attach the child elements to the parent element  
songs.addChildElement(firstSong);  
songs.addChildElement(secondSong);  
  
alert(doc.toString());

which produces the following output:

<songs>  
 <song title="Truckin'" />  
 <song title="Smoke on the Water" />  
</songs>

**Removing Child Elements**

The following code removes all the child elements from a parent element:

// remove all child elements  
parent.removeChildElements();

While this code removes only child elements of a given name from a parent element:

// remove all child elements named "art:painting"  
parent.removeChildElements("art:painting");  
  
// remove all child elements named "art:song", this time using a qname  
qname = marknote.QName("art", "song");  
parent.removeChildElements(qname);

This example removes only the first child element of a given name from a parent element:

// remove the first child element having the name "art:painting"  
parent.removeChildElement("art:painting");

Finally, this example removes a child element at the given position among the child elements:

// remove the child element at index 1  
// (i.e., the second child element)  
parent.removeChildElementAt(1);

**Reading Child Elements**

To read an array of all child elements of a given parent element, use the getChildElements method; for example:

// parent is an existing marknote.Element object  
var children = parent.getChildElements();  
  
for (var i=0; i<children.length; i++) {  
  
 var child = children[i];  
  
 var message =   
 "Element name is " +   
 child.getName() +   
 " and title attribute value is " +   
 child.getAttributeValue("title");  
 alert(message);  
  
}

To read an array of all child elements matching a given element name, also use the getChildElements method, but this time passing in the targeted element name as a parameter; for example:

// create the parent element  
var worksOfArt = new marknote.Element("worksOfArt");  
  
// create some song elements  
var song1 = new marknote.Element("song");  
song1.setAttribute("title", "Truckin'");  
var song2 = new marknote.Element("song");  
song2.setAttribute("title", "Smoke on the Water");  
  
// attach the song elements to the parent element  
worksOfArt.addChildElement(song1);  
worksOfArt.addChildElement(song2);  
  
// create some painting elements  
var painting1 = new marknote.Element("painting");  
painting1.setAttribute("title", "Mona Lisa");  
var painting2 = new marknote.Element("painting");  
painting2.setAttribute("title", "Starry Night");  
  
// attach the painting elements to the parent element  
worksOfArt.addChildElement(painting1);  
worksOfArt.addChildElement(painting2);  
  
// now, read out an array of \*only\* the painting elements  
var paintings = worksOfArt.getChildElements("painting");  
for (var i=0; i<paintings.length; i++) {  
 var child = paintings[i];  
 alert(  
 "Element name is " +   
 child.getName() +   
 " and title is " +   
 child.getAttribute("title").getValue()  
 );  
}

Use a marknote.QName (qualified name) object to identify child elements by qualified name. For example:

// the parent element  
var worksOfArtQName = new marknote.QName("art", "worksOfArt");  
var worksOfArt = new marknote.Element(worksOfArtQName);  
  
// the child elements  
var paintingQName = new marknote.QName("art", "painting");  
var firstPainting = new marknote.Element(paintingQName);  
var secondPainting = new marknote.Element(paintingQName);  
  
// set attributes for the child elements  
firstPainting.setAttribute("title", "Mona Lisa");  
secondPainting.setAttribute("title", "Starry Night");  
  
// attach the child elements to the parent element  
worksOfArt.addChildElement(firstPainting);  
worksOfArt.addChildElement(secondPainting);  
  
// alert out the parent art:worksOfArt element along with its child elements  
alert(worksOfArt);  
  
// find the art:painting child elements using the QName  
var paintings = worksOfArt.getChildElements(paintingQName);  
  
// alert out each of the art:painting child elements  
for (var i in paintings){  
 alert(paintings[i].toString());  
}

A string using the appropriate prefix syntax can also identify child elements having a qualified name. For example:

// worksOfArt is an existing marknote.Element object  
var paintings = worksOfArt.getChildElements("art:painting");

The following example retrieves the first occurrence only of an element having a given name:

// worksOfArt is an existing marknote.Element object  
var firstPainting = worksOfArt.getChildElement("art:painting");

While this code retrieves a child element by its specific position within the array of parent elements (using zero-based indexing):

// worksOfArt is an existing marknote.Element object  
var thirdElement = worksOfArt.getChildElementAt(2); // 0 is the first element, 1 is the second, etc.

**Converting XML to JSON and JSON to XML**

**Introduction**

Converting from XML to JSON and back using *Marknote* requires a working knowledge of JSON syntax, so this is a slightly more advanced topic than the preceding sections. However, once you have a command of JSON, converting back and forth is very straightforward.

You have complete control over how your conversion works. Drill as deep as you want to into the JSON or XML, or filter what you're converting however you please.

**An Example**

The following example steps through an example of first creating XML, then converting the XML to JSON, and finally converting the JSON back to XML again.

In the first step of our example, we create the XML:

// create the document, attach the root element  
var doc = new marknote.Document();  
var rootElement = new marknote.Element("songs");  
doc.setRootElement(rootElement);  
  
// add Truckin'  
var truckingElement = new marknote.Element("song");  
truckingElement.setAttribute("artist", "Grateful Dead");  
truckingElement.setAttribute("title", "Truckin'");  
rootElement.addChildElement(truckingElement);  
  
// add Smoke on the Water  
var smokeElement = new marknote.Element("song");  
smokeElement.setAttribute("artist", "Deep Purple");  
smokeElement.setAttribute("title", "Smoke on the Water");  
rootElement.addChildElement(smokeElement);  
  
// add Hair of the Dog  
var dogElement = new marknote.Element("song");  
dogElement.setAttribute("artist", "Nazareth");  
dogElement.setAttribute("title", "Hair of the Dog");  
rootElement.addChildElement(dogElement);  
  
alert(doc);

The result of the above is an XML document representing this XML:

<songs>  
 <song artist="Grateful Dead" title="Truckin'" />  
 <song artist="Deep Purple" title="Smoke on the Water" />  
 <song artist="Nazareth" title="Hair of the Dog" />  
</songs>

In the next step of our example, we convert the above XML to a JSON object:

// first, grab the root element and its child elements from the XML doc  
// (for this simple example, we'll only drill in one child level deep)  
var rootElem = doc.getRootElement();  
var childElements = rootElem.getChildElements();  
  
var json = {};  
var rootName = rootElem.getName(); // the name of the root element is "songs"  
  
// now, create the root property of our JSON object, and set its value to a new array,  
// which will result in JSON that looks like this:  
// { songs: [ ] }  
json[rootName] = [ ];   
  
// loop through the child XML elements and create a JSON object for each one;  
// within each of those child JSON objects, we'll also convert the child element's  
// attributes to JSON properties  
for (var i=0; i<childElements.length; i++) {  
  
 var elem = childElements[i];  
 var attributes = elem.getAttributes();  
   
 var child = {}; // a song  
  
 for (var j=0; j<attributes.length; j++) { // add the attributes of the song as properties  
 var attribute = attributes[j];  
 child[attribute.getName()] = attribute.getValue(); // { attName: attValue }   
 }  
  
 // attach the new child JSON object to the parent array  
 json[rootName].push(child); // { songs: [ { attName: attValue, attName: attValue, ... }, ... ] }   
   
}

This results in a JSON object of the form:

{ songs: [{   
 artist: "Grateful Dead",   
 title: "Truckin'"  
 }, {  
 artist: "Deep Purple",   
 title: "Smoke on the Water"  
 }, {  
 artist: "Nazareth",   
 title: "Hair of the Dog"  
 }  
 ]  
}

In the final step of our example, we take the above JSON object and convert it back into XML:

// docFromJson is a new XML doc that will hold the converted JSON  
var docFromJson = new marknote.Document();  
  
// after this, we have:  
// <songs />   
docFromJson.setRootElement(new marknote.Element(rootName));   
  
// items is the JSON array that the JSON object's root property points to  
// (in this case, each item is a song object)  
var items = json[rootName];  
  
// loop through the items  
for (var i=0; i<items.length; i++) {  
  
 // get the next song object  
 var item = items[i];  
  
 // in XML we have to give our element a name, unlike our JSON item;  
 // unfortunately, the JSON item doesn't give us any information   
 /// about how to name the element  
 var elem = new marknote.Element("song"); // <song />  
  
 // after this, we have:  
 // <songs> ... <song /></songs>  
 docFromJson.getRootElement().addChildElement(elem);   
  
 for (var propName in item) {  
 // after this, we have:  
 // <songs> ... <song attrib="..." ... /></songs>  
 elem.setAttribute(propName, item[propName]);   
 }   
  
}  
  
alert(docFromJson);

This results in this XML (the same as our original XML):

<songs>  
 <song artist="Grateful Dead" title="Truckin'" />  
 <song artist="Deep Purple" title="Smoke on the Water" />  
 <song artist="Nazareth" title="Hair of the Dog" />  
</songs>