E4X: JavaScript on steroids

Use JavaScript E4X for efficient XML processing

Grace Walker

July 15, 2008

Learn to use ECMAScript (aka JavaScript™) for XML (E4X), and discover the capabilities of the E4X enhancement, which was designed to facilitate simple and easy parsing, calculating, editing, and related activities on XML data.

E4X adds support for XML to the JavaScript programming language. XML is widely recognized as the universally accepted means to exchange documents and data across applications and platforms. This recognition is a direct function of XML’s proven track record as an efficient way to resolve interoperability problems associated with sharing documents and data. XML’s versatility and strength are products of the flexibility of its structural components. E4X facilitates the use of XML’s structural components, thereby enhancing that flexibility significantly.

Get the most from this article

This article covers the basics of E4X with the understanding that you are already familiar with XML data structures and the JavaScript language.

E4X is designed to simplify the task of writing JavaScript code for XML. It is an efficient, powerful tool that you can use to interact with XML nodes and attributes. The primary objective of E4X is to give JavaScript developers a straightforward and efficient way to manipulate an XML document without relying on the Document Object Model (DOM).

The JavaScript language uses the E4X utility to impart new properties to global objects. In turn, the XML object has several properties useful in E4X serializing and parsing functions. E4X recycles many existing JavaScript operators and uses them in XML creation, manipulation, and navigation.

E4X consumes less development time and has a very short learning curve. These qualities facilitate efficient and facile Read, Write, and related operations. The economy that these qualities create leads to simplified coding, enhanced code revising, and shorter code deployment cycles. In addition, the flexible and agile E4X technology is tailor-made for those increasingly important mobile applications.
To demonstrate the power and flexibility of E4X, I describe the main features employed to manipulate XML data, using a music catalog as the main input source. Most of the examples in this article are based on the XML document in Listing 1.

**Listing 1. XML music object used for most examples**

```xml
<mp3>
  <music genre="classical">
    <artist>Ludwig van Beethoven</artist>
    <song>Fifth Symphony</song>
  </music>
  <music genre="jazz">
    <artist>Grover Washington, Jr.</artist>
    <song>The Best Is Yet to Come</song>
  </music>
  <music genre="classical">
    <artist>Johann Sebastian Bach</artist>
    <song>Double Concerto in D- for Two Violins</song>
  </music>
  <music genre="jazz">
    <artist>Dave Brubeck</artist>
    <song>Take Five</song>
    <song>Truth Is Fallen</song>
  </music>
  <music genre="classical">
    <artist>Johannes Brahms</artist>
    <song>Piano Sonata No. 1 in C major</song>
  </music>
</mp3>
```

**Creating the XML object**

The first step in the processes is to create the XML object. E4X provides a type called `XML()`, which holds the XML element. You create the new XML object using a format already familiar to JavaScript developers, in Listing 2.

**Listing 2. Create the XML object**

```javascript
var xmlMusic = new XML();
```

After you create the XML object, you can load the XML document. E4X has two preferred E4X methods to create XML objects. In the first method, you create the object, and then load it as in Listing 3.

**Listing 3. Loading the XML object: method 1**

```javascript
<script type="text/javascript; e4x=1">
var xmlMusic = new XML();
xmlMusic = <mp3>
  <music genre="classical">
    <artist>Ludwig van Beethoven</artist>
    <song>Fifth Symphony</song>
  </music>
  <music genre="jazz">
    <artist>Grover Washington, Jr.</artist>
    <song>The Best Is Yet to Come</song>
  </music>
  <music genre="classical">
    <artist>Johann Sebastian Bach</artist>
    <song>Double Concerto in D- for Two Violins</song>
</mp3>
```

E4X: JavaScript on steroids
Alternatively, you can load the XML document as a string into the element when it is created, as shown in **Listing 4**.

### Listing 4. Loading XML object: method 2

```xml
<script type="text/javascript ; e4x=1">
   var xmlMusic=new XML('<mp3>
   <music genre="classical">
      <artist>Ludwig van Beethoven</artist>
      <song>Fifth Symphony</song>
   </music>
   <music genre="jazz">
      <artist>Grover Washington, Jr.</artist>
      <song>The Best Is Yet to Come</song>
   </music>
   <music genre="classical">
      <artist>Johann Sebastian Bach</artist>
      <song>Double Concerto in D- for Two Violins</song>
   </music>
   <music genre="jazz">
      <artist>Dave Brubeck</artist>
      <song>Take Five</song>
      <song>Truth Is Fallen</song>
   </music>
   <music genre="classical">
      <artist>Johannes Brahms</artist>
      <song>Piano Sonata No. 1 in C major</song>
   </music>
</mp3>');</script>
```

Note the expression `e4x=1`, which was added to the `script` statement. E4X is widely used to parse XML into JavaScript objects. However, problems can be associated with the direct use of XML syntax. When embedded in other tag-based environments, it may create some unforeseen difficulties. Recognition of this problem led to the disabling of E4X on HTML pages in Mozilla Firefox version 1.5 by default, because in HTML, it conflicted with the accepted methods of concealing scripts from incompatible browsers. To overcome this problem, you simply can deploy an attribute in the `script` statement of the form `type="text/javascript; e4x=1"` whenever you use E4X.

Although the XML object looks and behaves in a similar way to a regular JavaScript object, the two objects are not the same. Some E4X syntax only works with E4X XML objects. The syntax is familiar to JavaScript developers, but E4X does not facilitate direct mapping from XML to native JavaScript objects; instead, it provides the *illusion* of direct mapping.

### Accessing data

The XML object supports the dot (.) and square bracket ([ ]) notations used in regular JavaScript programming. Rather than access object properties, E4X overloads these operators and uses them to access the children of the element.
The following access examples are based on the xmlMusic object created above. To view the contents of the entire element, use the syntax in Listing 5.

**Listing 5. View the entire element contents**

```
document.write(xmlMusic);
```

The output from this statement is:

<table>
<thead>
<tr>
<th>Artist</th>
<th>Album</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ludwig van Beethoven</td>
<td>Fifth Symphony</td>
</tr>
<tr>
<td>Grover Washington, Jr.</td>
<td>The Best Is Yet to Come</td>
</tr>
<tr>
<td>Johann Sebastian Bach</td>
<td>Double Concerto in D- for Two Violins</td>
</tr>
<tr>
<td>Dave Brubeck</td>
<td>Take Five Truth Is Fallen</td>
</tr>
<tr>
<td>Johannes Brahms</td>
<td>Piano Sonata No. 1 in C major</td>
</tr>
</tbody>
</table>

To find the third artist, enter a statement similar to Listing 6.

**Listing 6. View artist 3**

```
document.write ( xmlMusic.music[2].artist );
```

The output from this statement is:

<table>
<thead>
<tr>
<th>Artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johann Sebastian Bach</td>
</tr>
</tbody>
</table>

**Note:** E4X is zero-based, so the index value used to retrieve the third record is 2.

To retrieve all the contents of the first music node, enter a statement like that in Listing 7.

**Listing 7. Retrieve first music node**

```
document.write (xmlMusic.music[0].*);
```

The output from this statement is:

<table>
<thead>
<tr>
<th>Artist</th>
<th>Album</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ludwig van Beethoven</td>
<td>Fifth Symphony</td>
</tr>
</tbody>
</table>

As you can see from just these few examples, access to XML nodes is extremely simple using E4X.

**Accessing attributes**

You can access the attributes of the XML object using the at sign (@) notation. Attribute manipulation can be one of the more difficult parts of working with XML data. Using E4X simplifies this task tremendously. For example, to retrieve all the data from the classical genre, enter a statement similar to Listing 8.

**Listing 8. Access node attributes**

```
document.write(xmlMusic.music.@genre==classical);`
```
The output from this statement is:

Ludwig van Beethoven Fifth Symphony
Johann Sebastian Bach Double Concerto in D- for Two Violins
Johannes Brahms Piano Sonata No. 1 in C major

To request just the artist's name in the classical genre, enter the statement in Listing 9.

**Listing 9. Request classical artist names**

```javascript
document.write(xmlMusic.music.(@genre=="classical").artist );
```

Which outputs only the names:

Ludwig van Beethoven
Johann Sebastian Bach
Johannes Brahms

The power, flexibility, and simplicity of E4X is forcefully demonstrated here. No longer is it necessary to avoid using attributes simply because they are difficult to manipulate. As you can plainly see here, E4X handles them with ease.

### Adding a child object

E4X allows you to add children to an existing element by using simple JavaScript notation. For example, if you wanted to add a song to an existing artist—say, Johann Sebastian Bach's *Brandenburg Concertos*—use the `+=` operator, as in Listing 10.

**Listing 10. Add a node with the += operator**

```javascript
xmlMusic.music[2].song += "Brandenburg concertos" ;
document.write (xmlMusic.music[2]);
```

The output after the addition would be:

Johann Sebastian Bach
Double Concerto in D- for Two Violins
Brandenburg Concertos

Bach now has two compositions as part of the `xmlMusic` object, where prior to this statement there was only one.

### Changing the content of a child object

To change the value of an element or attribute, assign a new value to it, as in Listing 11.
Listing 11. Modify a node with the = operator

```javascript
xmlMusic.music[3].song = "Pick Up Sticks";
document.write ( xmlMusic.music[3] );
```

The output is:

<table>
<thead>
<tr>
<th>Dave Brubeck</th>
<th>Pick Up Sticks</th>
</tr>
</thead>
</table>

**Note:** Because you used the equal sign (\(=\)) to replace the `xmlMusic.music[3].song` content, both songs by Dave Brubeck were written over, leaving only the new entry.

### Deleting a child node

You can delete a child node with the `delete` statement, as in **Listing 12**.

**Listing 12. Delete a child node**

```javascript
delete xmlMusic.music.song[1];
document.write (xmlMusic.music[1].*);
```

The output is:

<table>
<thead>
<tr>
<th>Grover Washington, Jr.</th>
</tr>
</thead>
</table>

The song node has been removed, but the artist remains.

### Filtering

E4X provides a special filtering operator (parentheses) that you use to select those nodes within a document that match specific criteria. This filtering operator applies a condition to the data that then sifts out the requested elements from a child node. Filtering operations are performed based on an expression that is surrounded by the parentheses. In fact, you have actually seen this operator already. Now, it's time to formally introduce you.

For example, filtering an attribute might take the form shown in **Listing 13**.

**Listing 13. Use an attribute to filter data**

```javascript
document.write (xmlMusic.music.@genre=="jazz" );
```

The filtered output is:

| Grover Washington, Jr. The Best Is Yet to Come |
| Dave Brubeck Truth Is Fallen |

And you can filter by artist, which is not an attribute, as in **Listing 14**.

**Listing 14. Use the artist name to find song**

```javascript
document.write (xmlMusic.music.(artist == "Dave Brubeck").song);
```
The filtered output is:

<table>
<thead>
<tr>
<th>Take Five</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truth Is Fallen</td>
</tr>
</tbody>
</table>

**Working with XMLList**

E4X recognizes two fundamental types: `XML()` and `XMLList()`. The `XMLList` object represents an ordered collection of XML properties—for instance, a list of recipes, continents, phone numbers, or chemical elements. An E4X object that consists of more than a single node is actually composed of a combination of `XML()` and `XMLList()` objects. The `XMLList` type is an invaluable tool in crafting queries—for example, when you search for the children of a given node, an `XMLList` is generated.

The following code is a skeleton outline of what an `XMLList` looks like, with its multiple nodes based on the music example:

```xml
<music></music>
<music></music>
<music></music>
<music></music>
<music></music>
```

An `XMLList` has an *initializer*, which is an expression that describes the initialization of an `XMLList` object. The `XMLList` initializer depicts a structured list of XML properties and uses an anonymous XML element syntax. The initializers start with the `<>` characters and end with the `</>` characters. The code sequence in **Listing 15** illustrates a method that you can use to initialize the `XMLList` object.

**Listing 15. Initialize the XMLList object**

```javascript
var xmlListObject =
    <><artist>Ludwig van Beethoven</artist> <song>Fifth Symphony</song></;```

So, what's the difference between an `XML` object and an `XMLList` object? In essence, the difference is that the `XML` type deals with one specific object that can contain several child nodes, and the `XMLList` deals with a set consisting of one or more XML objects.

E4X is designed to make the distinction between a single `XML` object and an `XMLList` with one item virtually indistinguishable. Consequently, the difference between an `XML` value and an `XMLList` value with a single item is negligible.

**Performing calculations**

With E4X, you can perform calculations on the XML data just as you would in XML. For this example, you'll use a new XML document that contains data that you can manipulate mathematically. **Listing 16** shows the XML document.
Listing 16. Calculation example

```javascript
<script type="text/javascript ; e4x=1">
    var xmlCatalog=new XML() ;
    xmlCatalog=<inventory>
        <item genre="apparel">
            <description>Dress</description>
            <price>50.00</price>
            <quantity>3</quantity>
        </item>
        <item genre="accessory">
            <description>Hat</description>
            <price>25.00</price>
            <quantity>5</quantity>
        </item>
        <item genre="apparel">
            <description>tie</description>
            <price>15.00</price>
            <quantity>7</quantity>
        </item>
        <item genre="accessory">
            <description>Belt</description>
            <price>15.00</price>
            <quantity>1</quantity>
        </item>
        <item genre="apparel">
            <description>Suit</description>
            <price>100.00</price>
            <quantity>2</quantity>
        </item>
    </inventory>
</script>

Here is the script that manipulates the data. Using a `while` loop, the total value of the inventory is calculated along with the total number of items currently in the inventory. **Listing 17** uses the `length()` method to determine the size of the `XML` object and control the loop.

Listing 17. Performing calculations with E4X

```javascript
<script type="text/javascript ; e4x=1">
    var i = 0 ;
    var totItems=0 ;
    var totInventoryValue=0 ;
    while ( i < xmlCatalog.item.length() )
    {
        totInventoryValue += xmlCatalog.item[i].quantity * xmlCatalog.item[i].price ;
        totItems+= xmlCatalog.item[i].quantity * 1 ;
        i = i + 1 ;
    }

    document.write("Total number of items: " + totItems + "\n");
    document.write("Total inventory value: $" + totInventoryValue + "\n");
</script>

The output from script is:
Using the XMLObject methods

E4X provides a set of methods that allows you to manipulate XML objects as well as XMLList objects. You have already seen how the length() method functions when it was used to control the flow in the script shown in Listing 17. The E4X methods can perform numerous tasks, from identifying attributes, namespaces, and elements to adding and identifying children.

As mentioned previously, E4X is designed to obfuscate the distinction between a single XML object and an XMLList with a size value of one. So, in E4X, methods that are available for XML objects are also available for XMLList objects with the appropriate size.

Table 1 lists the methods available to XML and XMLList objects. All the listed methods are available to XML objects. All the methods listed with Yes are available to both XML and XMLList objects.

Table 1. Methods available to XML and XMLList objects

<table>
<thead>
<tr>
<th>XML object methods</th>
<th>Available to XMLList objects</th>
<th>XML object methods</th>
<th>Available to XMLList objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>addNamespace(namespace)</td>
<td>No</td>
<td>name()</td>
<td>No</td>
</tr>
<tr>
<td>appendChild(child)</td>
<td>No</td>
<td>namespace([prefix])</td>
<td>No</td>
</tr>
<tr>
<td>attribute(attributeName)</td>
<td>Yes</td>
<td>namespaceDeclarations()</td>
<td>No</td>
</tr>
<tr>
<td>attributes()</td>
<td>Yes</td>
<td>nodeKind()</td>
<td>No</td>
</tr>
<tr>
<td>child(propertyName)</td>
<td>Yes</td>
<td>normalize()</td>
<td>Yes</td>
</tr>
<tr>
<td>childIndex()</td>
<td>No</td>
<td>parent()</td>
<td>Yes</td>
</tr>
<tr>
<td>children()</td>
<td>Yes</td>
<td>processingInstructions([name])</td>
<td></td>
</tr>
<tr>
<td>comments()</td>
<td>Yes</td>
<td>prependChild(value)</td>
<td>No</td>
</tr>
<tr>
<td>contains(value)</td>
<td>Yes</td>
<td>removeNamespace(namespace)</td>
<td>No</td>
</tr>
<tr>
<td>copy()</td>
<td>Yes</td>
<td>replace(propertyName, value)</td>
<td>No</td>
</tr>
<tr>
<td>descendants([name])</td>
<td>Yes</td>
<td>setChildren(value)</td>
<td>No</td>
</tr>
<tr>
<td>elements([name])</td>
<td>Yes</td>
<td>setLocalName(name)</td>
<td>No</td>
</tr>
<tr>
<td>hasComplexContent()</td>
<td>Yes</td>
<td>setName(name)</td>
<td>No</td>
</tr>
<tr>
<td>hasSimpleContent()</td>
<td>Yes</td>
<td>setNamespace(ns)</td>
<td>No</td>
</tr>
<tr>
<td>inScopeNamespaces()</td>
<td>No</td>
<td>text()</td>
<td>Yes</td>
</tr>
<tr>
<td>insertChildAfter(child1, child2)</td>
<td>No</td>
<td>toString()</td>
<td>Yes</td>
</tr>
<tr>
<td>insertChildBefore(child1, child2)</td>
<td>No</td>
<td>toXMLString()</td>
<td>Yes</td>
</tr>
<tr>
<td>length()</td>
<td>Yes</td>
<td>valueOf()</td>
<td>Yes</td>
</tr>
<tr>
<td>localName()</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In **Listing 10**, you employed the `+=` operator to add a song node to the `xmlMusic` object. You can create the same output if you use the `appendChild()` method. With the `appendChild()` method, you can add a node to the end of an existing object or element. To illustrate this functionality, **Listing 18** repeats the processing performed in Listing 10 but uses the `appendChild()` method rather than the `+=` operator.

**Listing 18. Add a node with the appendChild() method**

```javascript
xmlMusic.music[2].appendChild(<song>Brandenburg Concertos</song>)
document.write (xmlMusic.music[2].*)
```

This code generates the output:

<table>
<thead>
<tr>
<th>Johann Sebastian Bach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Concerto in D- for Two Violins</td>
</tr>
<tr>
<td>Brandenburg Concertos</td>
</tr>
</tbody>
</table>

As you can see, the output shows that the song was added to the third element in the `XMLList` object and that it is the same as the output from the code shown in Listing 10.

You can also use the `prependChild()` method to add nodes. The `prependChild()` method adds a node before a child node of the identified element. For example, using the same Bach concertos, *Brandenburg Concertos*, and the `prependChild()` method, the code in **Listing 19** places the work before the current song listing rather than after.

**Listing 19. Add a node with the prependChild() method**

```javascript
xmlMusic.music[2].song[0].prependChild(<song>Brandenburg Concertos</song>)
document.write (xmlMusic.music[2].*)
```

The output shows that the concertos have been inserted before the song node that was originally there:

<table>
<thead>
<tr>
<th>Johann Sebastian Bach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brandenburg Concertos</td>
</tr>
<tr>
<td>Double Concerto in D- for Two Violins</td>
</tr>
</tbody>
</table>

Using the `insertBefore()` method, you can insert a node between two existing nodes. **Listing 20** manipulates the Dave Brubeck data, which contains two songs. Employing the `insertBefore()` method, the song "Eleven Four," is placed between the current song nodes.

**Listing 20. Add a node with the insertBefore() method**

```javascript
document.write (xmlMusic.music[3].*)
```

This code generates the output:
You can use another method, `attributes()`, to identify attributes contained in an object. For example, to list all the attributes of the `xmlMusic` object, use the code in Listing 21.

**Listing 21. Identify attributes using the attribute() method**

```javascript
document.write (xmlMusic.music.attributes());
```

This method produces a list of all the attributes contained in the `xmlMusic` object:

```
classical
jazz
classical
jazz
classical
```

You can use the `attributes()` method to identify all the elements with the genre `jazz`. With the aid of the `length()` method, all the nodes that have the `jazz` attribute are output by the code in Listing 22.

**Listing 22. Identify the elements with the attribute jazz**

```javascript
var i = 0 ;
while ( i < xmlMusic.music.length()  )
{
  if ( xmlMusic.music[i].attributes() == "jazz" ) {
    document.write (xmlMusic.music[i].* + "<p>");
  }
  i = i + 1 ;
}
```

This code generates the output:

```
Grover Washington, Jr. The Best Is Yet to Come
Dave Brubeck Take Five Truth Is Fallen
```

The E4X methods, like the operators, provide a simple mechanism to handle XML data.

**Browser compatibility**

E4X is one of the simplest and easiest methods to manipulate XML data. However, not all browsers support it. Currently, it is not supported in Windows® Internet Explorer®. However, it
is supported in both Mozilla Firefox and any systems based on open source Rhino JavaScript technology.

**Summary**

As you can see from this brief article, working with E4X simplifies the work required when interacting with XML. The main obstacle for JavaScript developers wanting to work with XML data has been the inability to interact with the data in an easy and efficient manner. The JavaScript scripting language is extremely simple and well organized. And now with the E4X enhancement, JavaScript E4X provides the answer to a critical data manipulation problem. Give it a try.
Related topics

- **ECMA-357 standard** (1.81mb PDF): Read the specification for ECMAScript for XML (E4X).
- **XML in Firefox 1.5, Part 2: Basic XML processing** (Uche Ogbuji, developerWorks, March 2006): Get a general understanding of the processing methods available for XML in Firefox.
- **XML in Firefox 1.5, Part 3: JavaScript meets XML in Firefox** (Uche Ogbuji, developerWorks, August 2006): Learn to manipulate XML in the Firefox browser using JavaScript features.
- **IBM certification**: Find out how you can become an IBM-Certified Developer.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM trial software for product evaluation**: Build your next project with trial software available for download directly from developerWorks, including application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- **developerWorks podcasts**: Listen to interesting interviews and discussions for software developers.

© Copyright IBM Corporation 2008
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)