Processing XML with E4X

Introduced in JavaScript 1.6

Table of contents

First introduced in JavaScript 1.6, **E4X** introduces a native XML object to the JavaScript language, and adds syntax for embedding literal XML documents in JavaScript code.

A full definition of E4X can be found in the Ecma-357 specification. This chapter provides a practical overview of the language; it is not a complete reference.

Compatibility issues

Prior to widespread browser support for the `<script>` element, it was common for JavaScript embedded in a page to be surrounded by HTML comment tags to prevent `<script>` unaware browsers from displaying JavaScript code to the user. This practice is no longer necessary, but remains in some legacy code. For backwards compatibility, E4X defaults to ignoring comments and CDATA sections. You can add an `e4x=1` argument to your `<script>` tag to disable this restriction:

```
view plain  print  ?
01. <script type="text/javascript;e4x=1">
02. ...
03. </script>
```

Creating an XML object

E4X offers two principle ways of creating an XML object. The first is to pass a string to the `XML` constructor:

```
view plain  print  ?
01. var languages = new XML('<languages type="dynamic"><lang>JavaScript</lang><lang>Python</lang>
    </languages>');
```

The second is to embed the XML directly in your script, as an XML literal:

```
view plain  print  ?
01. var languages = <languages type="dynamic">
02.    <lang>JavaScript</lang>
03.    <lang>Python</lang>
04.  </languages>;
```


In both cases, the resulting object will be an E4X XML object, which provides convenient syntax for both accessing and updating the encapsulated data.

While the XML object looks and behaves in a similar way to a regular JavaScript object, the two are not the same thing. E4X introduces new syntax that only works with E4X XML objects. The syntax is designed to be familiar to JavaScript programmers, but E4X does not provide a direct mapping from XML to native JavaScript objects; just the illusion of one.

It is possible to interpolate variables into an XML literal to create an element name (or to create content).

```
view plain  print  ?
01. var h = 'html';
02. var text = "Here's some text";
03. var doc = <{h}><body>{text}</body></{h}>;
04. alert(doc.toXMLString());
05. // Gives
06. <html>
07.   <body>Here's some text</body>
08. </html>
```

Working with attributes

XML literal syntax has a significant advantage over the XML constructor when you need to create markup dynamically. With E4X it is easy to embed dynamic values in markup. Variables and expressions can be used to create attribute values by simply wrapping them with braces ({}) and omitting quotation marks that would normally go around an attribute value, as the following example illustrates:

```
view plain  print  ?
01. var a = 2;
02.  var b = <foo bar={a}>"hi"</foo>;
```

Upon execution the variable is evaluated and quotes are automatically added where appropriate. The preceding example would result in an XML object which looks like this: `<foo bar="2">"hi"</foo>`.

In attribute substitution, quotation marks are escaped as " while apostrophes are handled normally.

```
view plain  print  ?
01. var b = 'He said "Don\'t go there."';
02. var el = <foo a={b}/>;
03. alert(el.toXMLString());
04. // Gives: <foo a="He said &quot;Don't go there.&quot;"/>
```

Less than and ampersand signs are escaped into their entity equivalents. Since a greater than sign is not escaped, it is possible to get an XML error if the CDATA closing sequence (]]>) is included.

It is not possible to directly interpolate variables amidst other literal (or variable) attribute content, however (e.g., `bar="a{var1}{var2}"`). One must instead either calculate the variable with a JavaScript expression (e.g.,

bar={'a'+var1+var2}), define a new variable before the element literal which includes the full interpolation and then include that variable or retrieve the attribute after the literal to alter it (see below).

While one can interpolate attribute names as well as attribute values:

```
view plain  print  ?
01. var a = 'att';
02. var b = <b {a}='value'/>;
03. alert(b);
04. // Gives:
05. <b att="value"/>
```

...one cannot interpolate a whole expression at once (e.g., <b {a}>.)

After executing the above example, the variable languages references an XML object corresponding to the <languages> node in the XML document. This node has one attribute, type, which can be accessed and updated in a number of ways:

```
view plain  print  ?
01. alert(languages.@type); // Alerts "dynamic"
02.  languages.@type = "agile";
03.  alert(languages.@type); // Alerts "agile"
```

```
view plain  print  ?
01. alert(languages.toString());
02.  /* Alerts:
03.    <languages type="agile"><lang>JavaScript</lang><lang>Python</lang></languages>
04.  */
```

Note that if one wishes to make comparisons of retrieved attributes with other strings, it is necessary to convert the attribute first, even though the attribute may be converted to a string when used in other contexts (such as insertion into a textbox).

```
view plain  print  ?
01. if (languages.@type.toString() === 'agile') {
02. ...
03. }
```

or, simply:

```
view plain  print  ?
01. if (languages.@type == 'agile') {
02. ...
03. }
```

Working with XML objects

XML objects provide a number of methods for inspecting and updating their contents. They support JavaScript's regular dot and [] notation, but instead of accessing object properties E4X overloads these operators to access the element's children:

```
view plain print ?
01. var person = <person>
02.   <name>Bob Smith</name>
03.   <likes>
04.     <os>Linux</os>
05.     <browser>Firefox</browser>
06.     <language>JavaScript</language>
07.     <language>Python</language>
08.   </likes>
09. </person>;
10.
11. alert(person.name); // Bob Smith
12. alert(person['name']); // Bob Smith
13. alert(person.likes.browser); // Firefox
14. alert(person['likes'].browser); // Firefox
```

If you access something with more than one matching element, you get back an `XMLList`:

```
view plain print ?
01. alert(person.likes.language.length()); // 2
```

As with the DOM, * can be used to access all child nodes:

```
view plain print ?
01. alert(person.likes.*.length()); // 4
```

While the `.` operator accesses direct children of the given node, the `..` operator accesses all children no matter how deeply nested:

```
view plain print ?
01. alert(person..*.length()); // 11
```

The `length()` method here returns 11 because both elements and text nodes are included in the resulting `XMLList`.

Objects representing XML elements provide a number of useful methods, some of which are illustrated below:

```
view plain print ?
01. alert(person.name.text()) // Bob Smith
02.
03. var xml = person.name.toXMLString(); // A string containing XML
04.
05. var personCopy = person.copy(); // A deep copy of the XML object
06.
07. var child = person.child(1); // The second child node; in this case the <likes> element
```

Working with XMLLists

In addition to the XML object, E4X introduces an `XMLList` object. `XMLList` is used to represent an ordered collection of XML objects; for example, a list of elements. Continuing the above example, we can access an `XMLList` of the `<lang>` elements in the page as follows:

```
view plain print ?
01. var langs = languages.lang;
```

`XMLList` provides a `length()` method which can be used to find the number of contained elements:

```
view plain  print  ?
01.  alert(languages.lang.length());
```


Note that unlike JavaScript arrays length is a method, not a property, and must be called using `length()`.

We can iterate through the matching elements like so:

```
view plain  print  ?
01.  for (var i = 0; i < languages.lang.length(); i++) {
02.       alert(languages.lang[i].toString());
03.   }
```

Here we are using identical syntax to that used to access numbered items in an array. Despite these similarities to regular arrays, `XMLList` does not support `Array` methods such as `forEach`, and Array generics such as `Array.forEach()` are not compatible with `XMLList` objects.

We can also use the `for each...in` statement introduced in JavaScript 1.6 as part of JavaScript's E4X support:

```
view plain  print  ?
01.  for each (var lang in languages.lang) {
02.       alert(lang);
03.   }
```

`for each...in` can also be used with regular JavaScript objects to iterate over the values (as opposed to the keys) contained in the object. As with `for...in`, using it with arrays is strongly discouraged.

It is possible to create an `XMLList` using XML literal syntax without needing to create a well-formed XML document, using the following syntax:

```
view plain  print  ?
01.  var xmllist = <>
02.     <lang>JavaScript</lang>
03.     <lang>Python</lang>
04.   </>;
```

The += operator can be used to append new elements to an `XMLList` within a document:

```
view plain  print  ?
01.  languages.lang += <lang>Ruby</lang>;
```

Note that unlike node lists returned by regular DOM methods, `XMLLists` are static and are not automatically updated to reflect changes in the DOM. If you create an `XMLList` as a subset of an existing `XML` object and then modify the original `XML` object, the `XMLList` will not reflect those changes; you will need to re-create it to get the most recent updates:

```
view plain  print  ?
01.  var languages = <languages>
02.     <lang>JavaScript</lang>
03.     <lang>Python</lang>
04.   </languages>;
05.
06.   var lang = languages.lang;
07.   alert(lang.length()); // Alerts 2
08.
09.   languages.lang += <lang>Ruby</lang>;
10.   alert(lang.length()); // Still alerts 2
11.
12.   lang = languages.lang; // Re-create the XMLList
13.   alert(lang.length()); // Alerts 3
```

Searching and filtering

E4X provides special operators for selecting nodes within a document that match specific criteria. These filter operations are specified using an expression contained in parentheses:

```
view plain  print  ?
01.  var html = <html>
02.    <p id="p1">First paragraph</p>
03.    <p id="p2">Second paragraph</p>
04.  </html>;
05.
06.  alert(html.p.(@id == "p1")); // Alerts "First paragraph"
```

Nodes matching the path before the expression (in this case the paragraph elements) are added to the scope chain before the expression is evaluated, as if they had been specified using the with statement.

Consequently, filters can also run against the value of a single node contained within the current element:

```
view plain  print  ?
01.  var people = <people>
02.    <person>
03.      <name>Bob</name>
04.      <age>32</age>
05.    </person>
06.    <person>
07.      <name>Joe</name>
08.      <age>46</age>
09.    </person>
10.  </people>;
11.
12.  alert(people.person.(name == "Joe").age); // Alerts 46
```

Filter expressions can even use JavaScript functions:

```
view plain  print  ?
01.  function over40(i) {
02.      return i > 40;
03.  }
04.
05.  alert(people.person.(over40(parseInt(age))).name); // Alerts Joe
```

Handling namespaces

E4X is fully namespace aware. Any XML object that represents a node or attribute provides a `name()` method which returns a `QName` object, allowing easy inspection of namespaced elements.

Default

```
view plain  print  ?
01. default xml namespace = "http://www.w3.org/1999/xhtml";
02. // No need now to specify a namespace in the html tag
03. var xhtml = <html><head><title></title></head><body>
04.             <p>text</p></body></html>;
05. alert(xhtml.head); // No need to specify a namespace on subelements here either
```

Non-default

```
view plain  print  ?
01. var xhtml = <html xmlns="http://www.w3.org/1999/xhtml">
02.     <head>
03.         <title>Embedded SVG demo</title>
04.     </head>
05.     <body>
06.         <h1>Embedded SVG demo</h1>
07.         <svg xmlns="http://www.w3.org/2000/svg"
08.             viewBox="0 0 100 100">
09.             <circle cx="50"
10.                 cy="50"
11.                 r="20"
12.                 stroke="orange"
13.                 stroke-width="2px"
14.                 fill="yellow" />
15.         </svg>
16.     </body>
17. </html>;
18.
19. alert(xhtml.name().localName); // Alerts "html"
20. alert(xhtml.name().uri); // Alerts "http://www.w3.org/1999/xhtml"
```

To access elements that are within a non-default namespace, first create a `Namespace` object encapsulating the URI for that namespace:

```
view plain  print  ?
01. var svgns = new Namespace('http://www.w3.org/2000/svg');
```

This can now be used in E4X queries by using `namespace::localName` in place of a normal element specifier:

```
view plain  print  ?
01. var svg = xhtml..svgns::svg;
02. alert(svg); // Shows the <svg> portion of the document
```

Using Generators/Iterators with E4X

As of JavaScript 1.7, it is possible to use generators and iterators, giving more options for traversing E4X.

In a manner akin to DOM tree walkers, we can define our own walkers for E4X. While the following is already achievable by iterating an E4X object with `for each...in`, it demonstrates how a more customized one could be created.

view plain print ?

```
function xmlChildWalker (xml) {
    var i = 0;
    var child = xml.*[0];
    while (child != undefined) {
        yield child;
        child = xml.*[++i];
    }
    yield false;
}

var a = <a><b/><c/></a>;
var xcw = xmlChildWalker(a);

var child;
while ((child = xcw.next()) !== false) {
    alert(child.toXMLString()); // "<b/>" then "<c/>"
}
```

See also

- E4X
- E4X Tutorial

Page last modified 16:48, 13 Aug 2010 by user01