ECMA/TC39/2002/00

Draft Standard ECMA-999
September 2002

ECMA International

Standardizing Information and Communication Systems

ECMAScript for XML (E4X) Specification

Draft 02 – November 2002

Draft Standard ECMA-999
September 2002

ECMA International

Standardizing Information and Communication Systems

ECMAScript for XML (E4X) Specification

Brief History

TBD.

Table of contents

1 Scope

This standard defines the syntax and semantics of ECMAScript for XML (E4X), a set programming language extensions adding native XML support to ECMAScript.

2 Status of this Document

This is a working draft produced to motivate and facilitate discussions related to E4X with the goal of creating a general purpose, cross platform, vendor neutral E4X standard. Comments and suggestions are solicited and encouraged.

3 Motivation

3.1 The Rise of XML Processing

Developing software to create, navigate and manipule XML data is a significant part of every Internet developer's job. Developers are inundated with data encoded in the eXtensible Markup Language (XML). Web pages are increasingly encoded using XML vocabularies, including XHTML and Scalable Vector Graphics (SVG). On mobile devices, data is encoded using the Wireless Markup Language (WML). Web services interact using the Simple Object Access Protocol (SOAP) and are described using the Web Service Description Language (WSDL). Deployment descriptors, project make files and configuration files and now encoded in XML, not to mention an endless list of custom XML vocabularies designed for vertical industries. XML data itself is even described and processed using XML in the form of XML Schemas and XSL Stylesheets.

3.2 Current XML Processing Approaches

Current XML processing techniques require ECMAScript programmers to learn and master a complex array of new concepts and programming techniques. The XML programming models often seem heavyweight, complex and unfamiliar for ECMAScript programmers. This section provides a brief overview of the more popular XML processing techniques.

3.2.1 The Document Object Model (DOM)

One of the most common approaches to processing XML is to use a software package that implements the interfaces defined by the XML DOM (Document Object Model). The XML DOM represents XML data using a general purpose tree abstraction and provides a tree-based API for navigating and manipulating the data (e.g., getParentNode(), getChildNodes(), removeChild(), etc.).

This method of accessing and manipulating data structures is very different from the methods used to access and manipulate native ECMAScript data structures. ECMAScript programmers must learn to write tree navigation algorithms instead of object navigation algorithms. In addition, they have to learn a relatively complex interface hierarchy for interacting with the XML DOM. The resulting XML DOM code is generally harder to read, write, and maintain than code that manipulates native ECMAScript data structures. It is more verbose and often obscures the developer's intent with lengthy tree navigation logic. Consequently, XML DOM programs require more time, knowledge and resources to develop.

3.2.2 The eXtensible Stylesheet Language (XSLT)

XSLT is a language for transforming XML documents into other XML documents. Like the XML DOM, it represents XML data using a tree-based abstraction, but also provides an expression language called XPath designed for navigating trees. On top of this, it adds a declarative, rule-based language for matching portions of the input document and generating the output document accordingly.

From this description, it is clear that XSLT's methods for accessing and manipulating data structures are completely different from those used to access and manipulate ECMAScript data structures. Consequently, the XSLT learning curve for ECMAScript programmers is quite steep. In addition to learning a new data model, ECMAScript programmers have to learn a declarative programming model, recursive decent processing model, new expression language, new XML language syntax, and a variety of new programming concepts (templates, patterns, priority rules, etc.). These differences also make XSLT code harder to read, write and maintain for the ECMAScript programmer. In addition, it is not possible to use familiar development environments, debuggers and testing tools with XSLT.

3.2.3 Object Mapping

Several have also tried to navigate and manipulate XML data by mapping it to and from native ECMAScript objects. The idea is to map XML data onto a set of ECMAScript objects, manipulate those objects directly, then map them back to XML. This would allow ECMAScript programmers to reuse their knowledge of ECMAScript objects to manipulate XML data.

This is a great idea, but unfortunately it does not work. Native ECMAScript objects do not preserve the order of the original XML data and order is significant for XML. Not only do XML developers need to preserve the order of XML data, but they also need to control and manipulate the order of XML data. In addition, XML data contains artifacts that are not easily represented by the ECMAScript object model, such as attributes, comments and mixed element content.

3.3 The E4X Approach

ECMAScript for XML was envisioned to address these problems. E4X extends the ECMAScript object model with native support for XML data. It reuses familiar ECMAScript operators for creating, navigating and manipulating XML, such that anyone who has used ECMAScript is able to start using XML with little or no additional knowledge. The extensions include a native XML data type, XML literals and a small set of new operators useful for common XML operations, such as searching and filtering.

E4X applications are smaller and more intuitive to ECMAScript developers than comparable XSLT or DOM applications. They are easier to read, write and maintain requiring less developer time, skill and specialized knowledge. The net result is reduced code complexity, tighter revision cycles and shorter time to market for Internet applications. In addition, E4X is a lighter weight technology enabling a wide range of mobile applications.

4 Design Principles

The following design principles are used to guide the development of E4X and encourage consistent design decisions. They are listed here to provide insight into the design rational and to anchor discussions on desirable E4X traits

- **Simple:** One of the most important objectives of E4X is to simplify common programming tasks. Simplicity should not be compromised for interesting or unique features that do not address common programming problems.

- **Consistent:** The design of E4X should be internally consistent such that developers can anticipate its behaviour.

- **Familiar:** Common operators available for manipulating ECMAScript objects should also be available for manipulating XML data. The semantics of the operators should not be surprising to those familiar with ECMAScript objects. Developers already familiar with ECMAScript objects should be able to begin using XML objects with minimal surprises.

- **Minimal:** Where appropriate, E4X defines new operators for manipulating XML that are not currently available for manipulating ECMAScript objects. This set of operators should be kept to a minimum to avoid unnecessary complexity. It is a non-goal of E4X to provide, for example, the full functionality of XPath.

- **Loose Coupling:** To the degree practical, E4X operators will enable applications to minimize their dependencies on external data formats. For example, E4X applications should be able to extract a value deeply nested within an XML structure, without specifying the full path to the data. Thus, changes in the containment hierarchy of the data will not require changes to the application.

- **Maintains Data Type Integrity [Edition 4]**: E4X for ECMAScript Edition 4 should support verification of the integrity of XML data types using XML Schemas.

- **Complementary:** E4X should integrate well with other languages designed for manipulating XML, such as XPath, XSLT and XML Query. Specifically, E4X should be able to invoke complementary languages when additional expressive power is needed without compromising the simplicity of the E4X language itself. In addition, invoking E4X code from other host languages should be possible.

This list is evolving and may be revised as the specification progresses.

5 Lexical Conventions

<tbd>

6 Types

E4X extends ECMAScript by adding two new fundamental data types for representing XML data. Future versions will also provide the capability to derive user-defined types for specific XML vocabularies using XML Schemas.

Note: I've used an ad-hoc pseudo code syntax to express the logic of operators in this version of the document. The algorithms have not yet been rigorously checked. As we refine the spec, I will adjust the semantics and syntax of the algorithms as the group determines appropriate.

6.1 XML

The XML type is an *ordered* collection of properties with a name, a set of XML attributes and a parent. Each property has a unique numeric property name *P*, such that ToString(ToUint32(*P*)) is equal to *P*. Each property has a value of type XML, String or Object. Values of type Object must be instances of the built-in XMLComment or XMLPI objects. Each XML attribute is an instance of the built-in XMLAttribute object. The parent may reference a value of type XML or Null. Each value of type XML represents an XML element.

The E4X compiler or interpreter may use this type information to determine the semantics of operations performed on values of type XML and to decide when to implicitly coerce values to or from the XML type.

Issue: It is not clear there is a strong case for modeling each XML attribute as an object as opposed to a string value associated with a named property.

6.1.1 Internal Properties and Methods

The XML type is logically derived from the Object type and inherits its internal properties. The following table summarises the internal properties the XML type adds to those defined by the Object type.

Property	***Parameters***	***Description***
[[Name]]	none	The name of this XML object.
[[Parent]]	none	The parent of this XML object.
[[Attributes]]	none	The attributes associated with this XML object.
[[Length]]	none	The number of properties in this XML object.
[[Insert]]	(*PropertyName*, *Value*)	Inserts one or more new properties before the property with name *PropertyName*.

The value of the [[Name]] property must be a String containing a legal XML element name. It holds the name of the XML element represented by this XML object.

The value of the [[Parent]] property must be either an XML object or Null. When an XML object occurs as a property (i.e., a child) of another XML object, the [[Parent]] property provides easy access to the containing XML object (i.e., the parent).

Issue: There has been some discussion about the necessity of the parent pointer and its role in the requirement for making a deep copy of any XML value assigned as a child of another XML value.

The value of the [[Attributes]] property is an Array of zero or more XMLAttribute objects.

The value of the [[Length]] property is a non-negative Number.

Issue: We will also need a way to distinguish built-in XML methods from other XML properties to avoid name clashes (e.g., an internal [[methods]] property and a separate Call mechanism)

6.1.1.1 [[Get]] (P)

The XML type overrides the internal [[Get]] method defined by the Object type. The XML [[Get]] method may be used to retrieve the value of a property by its numeric property name, an XML attribute by its name or a set of XML valued properties by their name. The input variable *P* may be a numeric property name, an XML attribute name (distinguished from the name of XML valued properties by a leading "@" symbol) or the name of zero or more XML valued properties.

When the [[Get]] method of an XML object *x* is called with property name *P*, the following steps are taken:

```
1. If ToString(ToUint32(P)) == P
    a. call the Object.[[Get]] method with x as the this object and parameter P
    b. return Result (a)
2. Let l be an empty XMLList
3. If P[0] == '@'
    a. Let n = P.substring(1, P.length)
    b. for each XMLAttribute a in x.[[Attributes]]
        i. if n == a.name, append a to l.
    c. Return l.
4. for each property q in x
    a. if Type(q) == XML and q.[[Name]] == P, append q to l
5. Return l.
```

6.1.1.2 [[Put]] (P, V)

The XML type overrides the internal [[Put]] method defined by the Object type. The XML [[Put]] method may be used to replace, append and insert properties or XMLAttributes in an XML value. The input variable *P* identifies which portion of the XML value will be affected and may be an XML attribute name (distinguished from XML valued property names by a leading "@" symbol), numeric property name, or the name of zero or more XML valued properties. The input variable *V* may be an XML value, an XMLList value or any value that can be converted to a String with ToString()

When the [[Put]] method of an XML object *x* is called with property name *P* and value *V*, the following steps are taken:

```
1. If P[0] == "@"
    a. Let n = P.substring(1, P.length)
    b. if x doesn't have an XMLAttribute with name n
        i. create a new XMLAttribute a with name n and value ToString(V)
        ii. append a to x.[[XMLAttributes]]
    c. else
        i. set the value of the XMLAttribute of x with name n to ToString(V)
    d. Return
2. Let i = ToUint32(P)
3. If ToString(i) == P
    a. If i >= x.[[Length]],
        i. P = ToString(x.[[Length]])
        ii. Append a new property P to x
    b. If Type(V) == XML,
        i. create a deep copy c of V,
        ii. set c.[[Parent]] to x
        iii. set the value of property P of x to c
    c. else if Type(V) == XMLList
        i. Call [[Delete]] with x as the this object and parameter P
        ii. Call [[Insert]] with x as the this object and parameters P and V
    d. else
        i. set the value of property P of x to ToString(V)
    e. Return
4. Set i = -1
5. for (k = 0 to x.[[Length]])
    a. if (Type(x[k]) == XML and x[k].[[Name]] == P)
        i. if (i == -1), i = k
        ii. else Call [[Delete]] with x as the this object and parameter ToString(k)
6. if i == -1
    a. Create a new XML object y with y.name = P and y.[[Parent]] = x
    b. Append a new property i to x with value y
7. if Type(V) == XML
    a. create a deep copy c of V,
    b. set c.[[Parent]] to x,
    c. set the value of property i of x to c
8. else if Type(V) == XMLList
```

```
        a.  Call [[Delete]] with x as the this object and parameter ToString(i)
        b.  Call [[Insert]] with x as the this object and parameters ToString(i) and V
9.  else
        a.  remove all the properties of the XML value x[i]
        b.  append a new property "0" to the XML value x[i] with value ToString(V)
10. Return
```

6.1.2 [[Delete]] (P)

The XML type overrides the internal [[Delete]] method defined by the Object type. The XML [[Delete]] method may be used to remove a property by its numeric property name, an XML attribute by its name or a set of XML valued properties by their name. Unlike, the [[Delete]] method defined by Object, the XML [[Delete]] method shifts all the properties after deleted properites up to fill in the gaps created by the delete. The input variable *P* may be a numeric property name, an XML attribute name (distinguished from the name of XML valued properties by a leading "@" symbol) or the name of zero or more XML valued properties.

When the [[Delete]] method of an XML object *x* is called with property name *P*, the following steps are taken:

```
1.  Let i = ToUint32(P)
2.  If ToString(i) == P
        a.  if i >= x.[[Length]], return true
        b.  else
                i.   remove the property with the name P from x
                ii.  for each property q of x such that ToUint32(q) > i, rename q to ToString(ToUint32(q) – 1)
                iii. x.[[Length]] = x.[[Length]] – 1
        c.  Return true
3.  If P[0] == '@'
        a.  Let n = P.substring(1, P.length)
        b.  for each XMLAttribute a in x.[[Attributes]]
                i.   if n == a.name, remove a
        c.  Return true.
4.  let dp = 0
5.  for each property q in x
        a.  if Type(q) == XML and q.[[Name]] == P,
                i.   remove the property with name q from x
                ii.  dp = dp + 1
        b.  else
                i.   if dp > 0, rename q to ToString(ToUint32(q) – dp)
6.  x.[[Length]] = x.[[Length]] - dp
7.  Return l.
```

6.1.2.1 [[Insert]] (P, V)

The XML type adds the internal [[Insert]] method to the internal properties defined by the Object type. The XML [[Insert]] method may be used to insert a value *V* at a specific position *P*. The input variable *P* must be a numeric property name. The input variable *V* may be an XML value, an XMLList value or any value that can be converted to a String with ToString().

When the [[Insert]] method of an XML object *x* is called with property name *P*, the following steps are taken:

```
1.  Let n = 1
2.  If Type(V) == XMLList, n = V.[[Length]]
3.  Let i = ToUint32(P)
4.  for j = i to x.[[Length]]-1, rename property j of x to ToString(j + n)
5.  if Type(V) == XML
        a.  create a deep copy c of V,
        b.  set c.[[Parent]] to x,
        c.  set the value of property i of x to c
6.  else if Type(V) == XMLList
        a.  for j = 0 to V.[[Length-1]]
                i.   create a deep copy c of V[j]
```

```
            ii.  set c.[[Parent]] to x
            iii. set the property ToString(i + j) of x to c
7. else
    a. set the property i of x to the value ToString(V)
8. Return
```

6.2 XMLList

The XMLList type is an *ordered* collection of properties. Each property has a unique numeric property name *P*, such that ToString(ToUint32(*P*)) is equal to *P*. Each property references a value of type XML, String, Boolean, Number or Object. Values of type Object must be instances of the built-in XMLAttribute, XMLComment or XMLPI objects. A value of type XMLList may represent an XML Document, XML Fragment or an arbitrary collection of properties (e.g., a query result).

Issue: there is an open discussion regarding whether lists should store lvalues or rvalues, which is important when using lists to update fragments of XML.

Note: Unlike the XML type, the XMLList type allows XML properties of type Boolean and Number. This capability allows XML operators that return non-XML types to be applied to XMLLists. For an example, see the childIndex() function described in section <tbd>.

The E4X compiler or interpreter may use this type information to determine the semantics of operations performed on values of type XMLList and to decide when to implicitly coerce values to or from the XMLList type.

6.2.1 Internal Properties and Methods

The XMLList type is logically derived from the Object type and inherits its internal properties. The following table summarises the internal properties the XMLList type adds to those defined by the Object type.

Property	***Parameters***	***Description***
[[Length]]	none	The number of properties contained in this XMLList object.
[[Insert]]	(*PropertyName*, *Value*)	Inserts one or more new properties before the property with name *PropertyName*.

The value of the [[Length]] property is a non-negative Number.

Note: Don't spend any time reviewing the XMLList algorithms. They are not yet completed and will change significantly.

6.2.1.1 [[Get]] (P)

The XMLList type overrides the internal [[Get]] method defined by the Object type. The XMLList [[Get]] method may be used to retrieve a specific property of the XMLList by its numeric property name or to iterate over the XML valued properties of the XMLList retrieving their XML attributes by name or their XML values by name. The input variable *P* may be a numeric property name, an XML attribute name (distinguished from the name of XML valued properties by a leading "@" symbol) or an XML value name.

When the [[Get]] method of an XMLList object *x* is called with property name *P*, the following steps are taken:

```
1. If ToString(ToUint32(P)) == P
    a. call Object.[[Get]] with x as the this object and parameter P
    b. return Result (a)
2. Let l be an empty XMLList
3. for each q in x,
    a. if Type(x[q]) == XML,
        i.  call [[Get]] with x[q] as the this object and parameter P
        ii. Append Result(i) to l
4. Return l.
```

Issue: There is an open discussion about having the dot operator (and thus the [[Get]] method) operate on and returns lists since this is inconsistent with the current use of the dot operator in ECMAScript.

6.2.1.2 [[Put]] (P, V)

The XMLList type overrides the internal [[Put]] method defined by the Object type. The XMLList [[Put]] method may be used to replace, append and insert properties at a specific location in the XMLList. In addition, when the XMLList contains a single property with an XML value, the [[Put]] method may be used to replace, append and insert properties or XML attributes of that value by name. The input variable *P* identifies which portion of the XMLList or XML value will be affected and may be a numberic property name, XML attribute name (distinguished from XML valued property names by a leading "@" symbol), or the name of zero or more XML valued properties. The input variable *V* may be an XML value, an XMLList value or any value that can be converted to a String with ToString()

When the [[Put]] method of an XMLList object *x* is called with property name *P* and value *V*, the following steps are taken:

```
1.  Let i = ToUint32(P)
2.  If ToString(i) == P
    a.  If i >= x.[[Length]],
        i.   P  = ToString(x.[[Length]])
        ii.  Append a new property P to x
    b.  If Type(V) == XML,
        i.   create a deep copy c of V,
        ii.  set the value of property P of x to c
    c.  else if Type(V) == XMLList
        i.   call [[Delete]] with x as the this object and parameter P
        ii.  call [[Insert]] with x as the this object and parameters P and V
    d.  else
        i.   set the value of property P of x to ToString(V)
    e.  Return
3.  if x.[[Length]] == 1 and Type(x[0]) == XML
        a.  call [[Put]] on x[0] with parameters P and V
4.  else
        a.  throw a TypeException
5.  Return
```

Note: When a non-numberic property name is given, we could have just as well applied the [[Put]] operation to every XML property in the XMLList; however, changing several values as the result of a single operation will likely be confusing to the scriptor. Therefore, all "update" operations over multiple XMLList properties currently throw an error.

6.2.2 [[Delete]] (P)

The XMLList type overrides the internal [[Delete]] method defined by the Object type. The XMLList [[Delete]] method may be used to remove a specific property of the XMLList by its numeric property name or to iterate over the XML valued properties of the XMLList removing their XML attributes by name or their XML values by name. The input variable *P* may be a numeric property name, an XML attribute name (distinguished from the name of XML valued properties by a leading "@" symbol) or an XML value name.

When the [[Delete]] method of an XML object *x* is called with property name *P*, the following steps are taken:

```
1.  Let i = ToUint32(P)
2.  If ToString(i) == P
        a.  if i >= x.[[Length]], return true
        b.  else
                i.   remove the property with the name P from x
                ii.  for each property q of x such that ToUint32(q) > i, rename q to ToString(ToUint32(q) – 1)
                iii. x.[[Length]] = x.[[Length]] – 1
        c.  Return true
3.  for each q in x,
        a.  if Type(x[q]) == XML, call [[Delete]] with x[q] as the this object and parameter P
```

```
4. Return true
```

6.2.2.1 [[Insert]] (P, V)

The XMLList type adds the internal [[Insert]] method to the internal properties defined by the Object type. The XMLList [[Insert]] method may be used to insert a value *V* at a specific position *P*. The input variable *P* must be a numeric property name. The input variable *V* may be an XML value, an XMLList value or any value that can be converted to a String with ToString().

When the [[Insert]] method of an XMLList object *x* is called with property name *P*, the following steps are taken:

```
1. Let $n = 1$
2. If Type($V$) == XMLList, $n = V.[[Length]]$
3. Let $i$ = ToUint32($P$)
4. for $j = i$ to $x.[[Length]]-1$, rename property $j$ of $x$ to ToString($j + n$)
5. if Type($V$) == XML
     a. create a deep copy $c$ of $V$,
     b. set the value of property $i$ of $x$ to $c$
6. else if Type($V$) == XMLList
     a. for $j = 0$ to $V.[[Length-1]]$
          i. create a deep copy $c$ of $V[j]$
          ii. set the property ToString($i + j$) of $x$ to $c$
7. else
     a. set the property $i$ of $x$ to the value ToString($V$)
8. Return
```

7 Type Conversion

E4X extends the automatic type conversion operators defined in ECMAScript. Note: as in ECMAScript, these type conversion functions occur implicitly as needed in E4X and are described here to aid specification of type conversion semantics. In addition, ToString and ToXMLString are exposed indirectly to the E4X user via the built-in member functions toString() and toXMLString() defined in sections <tbd>.

7.1 ToString

E4X extends the behavior of the ToString operator by specifying its behavior for the following types.

Input Type	***Result***
XML	Return the XML value as a string as defined in section 0.
XMLList	Return the XMLList value as a string as defined in section 7.1.2.

7.1.1 ToString Applied to the XML Type

Given an XML value *x*, the operator ToString converts *x* to a string *s*. If the XML value contains at least one property, contains only properties of type String and contains no XML attributes (i.e., contains a primitive value), ToString returns the String contents of the XML value, omitting the start and end tags. Otherwise, ToString returns an XML encoded string representing the entire XML value, including the start tag, attributes and the end tag.

Combined with ToString's treatment of XMLLists (see section 7.1.2), this behavior allows E4X programmers to access the values of XML leaf nodes in much the same way they access the values of object properties. For example, given a variable named *order* assigned to the following XML value:

```
<order>
	<customer>
		<firstname>John</firstname>
		<lastname>Doe</lastname>
	</customer>
	<item>
		<description>Big Screen Television</description>
		<price>1299.99</price>
		<quantity>1</quantity>
	</item>
</order>
```

the E4X programmer can access individual values of the XML value like this:

```
// Construct the full customer name
var name = order.customer.firstname + " " + order.customer.lastname;

// Calculate the total price
var total = order.item.price * order.item.quantity;
```

Unlike the W3C DOM, E4X does not require the programmer to explicitly select the text nodes associated with each leaf element or explicitly select the first element of each XMLList return value. For cases where this is not the desired behavior, the ToXMLString operator is provided (see section 7.2). Note: in this example, the String valued XML properties associated with the XML values order.item.price and order.item.quantity are implicitly converted to type Number prior to performing the multiply operation.

The semantics of the ToString conversion applied to an XML value are specified as follows:

```
1. Let s be an empty string.
2. Let primitive = true.
3. If x contains any XML attributes, primitive = false.
4. If x contains no properties, primitive = false.
5. For each property p in x
    a. if Type(p) != String, primitive = false.
6. If not primitive, return ToXMLString(x)
7. Otherwise, for each property p in x, append ToString(p) to s in order.
8. Return s.
```

Issue: Some are interested in a canonical string representation of an XML value (with a predefined attribute ordering, etc.) so that the string representations of two equivalent XML values would always be identical.

7.1.2 ToString Applied to the XMLList Type

The operator ToString converts an XMLList value *l* to a string *s*. If the XMLList value contains only one property, ToString returns the result of calling ToString on that property, which may omit the start and end tags depending on the content of the property. Otherwise, ToString returns an XML encoded string representing the entire XMLList value, including start tags, attributes and the end tags.

This treatment intentionally blurs the distinction between a single XML value and an XMLList containing only one value to simplify the programmer's task. It allows E4X programmers to access the value of an XMLList containing only a single primitive value in much the same way they access object properties. For an example, see section 0 above.

The semantics of the ToString operation applied to an XMLList value are specified as follows:

1. If *x* contains exactly one property, return ToString(*l*[0]);.
2. Otherwise, let *s* be an empty string.
3. For each property *p* in *l*, append ToXMLString(*p*) to *s* in order.
4. Return *s*.

7.2 ToXMLString

E4X adds the conversion operator ToXMLString to ECMAScript. ToXMLString is a variant of ToString used to convert its argument to an XML encoded string. Unlike ToString, it always includes the start and end tags, regardless of content. This is useful in cases where the default ToString behavior is not desired. The semantics of ToXMLString are specified by the following table.

Input Type	***Result***
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return ToString(input argument)
Number	Return ToString(input argument)
String	Returns the input argument (no conversion).
XML	Create an XML encoded string value based on the content of the XML value as specified in section 0.
XMLList	Create an XML encoded string value by calling ToXMLString on each property of the XMLList in order and concatenating the results to form a single string.
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint String) 2. Call ToString(Result(1)) 3. Return Result(2)

7.2.1 ToXMLString Applied to the XML Type

Given an XML value *x*, ToXMLString converts it to an XML encoded string *s*. The semantics of the ToXMLString operator applied to an XML value are specified as follows:

```
1.  Let s be an empty string.
2.  Append "<" to s.
3.  Append x.name to s.
4.  For each XML attribute a in x
        b.  Append " " to s.
        c.  Append a.name to s.
        d.  Append "=" to s.
        e.  Append a double-quote character (i.e. ") to s.
        f.  Append a.value to s.
        g.  Append a double-quote character (i.e. ") to s.
5.  If x contains no properties, append "/>" to s and return s.
6.  Otherwise, append ">" to s.
7.  For each property p in x, append ToXMLString(p) to s in order.
8.  Append "</" to s.
9.  Append x.name to s.
10. Append ">" to s.
11. Return s.
```

7.3 ToXML

E4X adds the operator ToXML to ECMAScript. ToXML converts its argument to a value of type XML according to the following table:

Input Type	***Result***
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.

Boolean	Throw a **TypeError** exception.
Number	Throw a **TypeError** exception.
String	Parse the contents of the string as XML and create an XML value as specified below in section 0.
XML	Return the input argument (no conversion).
XMLList	If the XMLList contains only one property and the type of that property is XML, return that property. Otherwise, return undefined.
W3C DOM Element	Create an XML adaptor (a.k.a. wrapper) object for accessing and manipulating the DOM node using XML operators.
Object	Throw a **TypeError** exception.

7.3.1 ToXML Applied to the String Type

Any value *s* of type String can be converted to type XML as follows:

```
1. Parse s as a W3C DOM Element e.
2. If the parse fails, throw an XMLException.
3. Create a new value x of type XML.
4. Set x.name to e.tagName.
5. For each attribute a in e.attributes, add an XML attribute to x with name a.nodeName and value a.nodeValue.
6. For each node n in e.childNodes
    a. If n.nodeType == ELEMENT_NODE, create a new XML value c = ToXML(n), set c.[[Parent]] to x and
       append c to the list of properties in x.
    b. If n.nodeType == TEXT_NODE or n.nodeType == CDATA_SECTION_NODE, create a new String s = new
       String(n.nodeValue) and append a property to x with value s.
    c. If n.nodeType == COMMENT_NODE, create a new instance of the built-in XMLComment object c = new
       XMLComment(n.nodeValue), set c.[[Parent]] to x and append a property to x with the value c.
    d. if n.nodeType == PROCESSING_INSTRUCTION_NODE, create a new instance of the built-in XMLPI
       object p = new XMLPI(n.nodeName, n.nodeValue), set p.[[Parent]] to x and append a property to x with the
       value p.
7. Return x.
```

Note, the use of the XML DOM Element is purely illustrative. The XML DOM is not required to perform this type conversion and implementations may use any mechanism that provides the same semantics.

Prior to conversion, string arithmetic can be used to construct portions of the XML value without regard for XML constraints such as well-formedness. For example, consider the following.

```
var John = "<employee><name>John</name><age>25</age></employee>";
var Sue ="<employee><name>Sue</name><age>32</age></employee>";
var tagName = "employees";
var employees = new XML("<" + tagName +">" + John + Sue + "</" + tagName +">");
```

8 Execution Contexts

9 Expressions

Issue: There are some concerened about using the "+" operator to concatenate lists.

10 Statements

11 Native E4X Objects