ECMA/TC39-TG1/2003/12 ECMA/TC39/2003/22

Draft Standard ECMA-XXX
September 2002

ECMA International

Standardizing Information and Communication Systems

ECMAScript for XML (E4X) Specification

Draft 06 – May 2003

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

Draft Standard ECMA-999
September 2002

ECMA International

Standardizing Information and Communication Systems

ECMAScript for XML (E4X) Specification

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

MBo 2003tg1-012.doc 26-05-03 11,50

Brief History

TBD.

This ECMA Standard has been adopted by the ECMA General Assembly of ....

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.

- **Security [Edition 4]**: E4X for ECMAScript Edition 4 should support verification of the integrity of XML data types using XML Schemas. New E4X features should not introduce unacceptable security vulnerabilities.

- **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.

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

5 Lexical Conventions

This section introduces the lexical conventions E4X adds to ECMAScript.

5.1 Identifiers

E4X extends the identifier definition in ECMAScript with the following production:

Identifier ::

AttributeIdentifier

WildcardIdentifier

QualifiedIdentifier

This specification defines the semantics of *AttributeIdentifiers, QualifiedIdentifiers* and *WildcardIdentifiers* in the context of member lookup for XML values. The semantics of these identifiers in other contexts is outside the scope of this specification.

Issue #14: We may want to express this in a more restrictive way that allows us to catch errors in grammar analysis

Waldemar will look for grammar ambiguities. In any case, he would like to see these defined in the context of the dot operator instead of here. This is also related to the syntax for filtering predicates.

5.1.1 Attribute Identifiers

E4X extends ECMAScript by adding attribute identifiers. The syntax of an attribute identifier is specified by the following production:

AttributeIdentifier ::

@ *Identifier*

An *AttributeIdentifier* is used to identify an XML attributes within an XML value. The preceding "@" character distinguishes a XML attribute from a XML property with the same name. This *AttributeIdentifier* syntax was chosen for consistency with the familiar XPath syntax.

5.1.2 Wildcard Identifiers

E4X extends ECMAScript by adding a wildcard identifier. The syntax of the wildcard identifier is specified by the following production:

WildcardIdentifier ::

*

The *WildcardIdentifier* is used to identify all XML properties of an XML value or all XML attributes of an XML value if used in an attribute identifier.

5.1.3 Qualified Identifiers

E4X extends ECMAScript by adding qualified identifiers. The syntax for qualified identifiers was chosen for consistency with future versions of ECMAScript and is specified by the following productions:

QualfiedIdentifier ::

Qualifier :: *Identifier*

Qualifier ::

Identifier

QualifiedIdentifiers are used to identify values defined within a specific namespace. They may be used to access, manipulate and create namespace qualified XML elements and attributes. The *Qualifier* must identify a value of type Namespace (see section <TBD>). For example,

```
var message = <soap:Envelope
        xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <soap:Body>
                <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
                        <symbol>DIS</symbol>
                </m:GetLastTradePrice>
        </soap:Body>
</soap:Envelope>

// declare the SOAP and stocks namespaces
var soap = new Namespace("http://schemas.xmlsoap.org/soap/envelope/");
var stock = new Namespace("http://mycompany.com/stocks");

// extract the soap encoding style and body from the soap message
var encodingStyle = message.@soap::encodingStyle;
var body = message.soap::Body;

// change the stock symbol
message.soap::Body.stock::GetTradePrice.symbol = "MYCO";
```

5.2 Punctuators

E4X adds the following punctuators to the existing list of ECMAScript punctuators:

. .

5.3 Literals

E4X adds an XML literal and an XMLList literal to ECMAScript. See section 9.1.1 for details.

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 or String. Each XML attribute is an instance of the XML type. The parent may reference a value of type XML or Null.

Each value of type XML represents an XML element, attribute, comment, processing-instruction or text node. The internal [[*Class*]] property is set to XML, XMLAttribute, XMLComment, XMLPI or XMLText as appropriate. Each XML value representing an XML attribute, comment, processing-instruction (PI) or text node has a single property with the name "0" (zero) and a String value representing the value of the associated attribute, comment, PI or text node.

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 #1: Modeling XML attributes as objects or named, string valued properties

Use case: We may need to model XML attributes as objects with a name, value and parent so we have a sensible, consistent return type for the "element.@*" expression for retrieving all the attributes of an element (e.g., if we want to iterate over each one). We need an XMLAttribute object so we can determine the name and value of each attribute in the returned list. The parent pointer is also useful for operations such as order.item.@* where item is a collection so we can determine which attributes go with with items.

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.
[[Descendants]]	(*PropertyName*)	Returns an XMLList containing the XML valued descendants of this XML object with names tha match *propertyName*.
[[Insert]]	(*PropertyName*, *Value*)	Inserts one or more new properties before the property with name *PropertyName* (a numeric index).
[[Replace]]	(*PropertyName*, *Value*)	Replaces the value of the property with name *PropertyName* (a numeric index) with one or more new properties

Note: With the current data model, we do not need an internal [[methods]] property to achieve separate storage for methods and public properties as previously thought. Clashes between public property names and public method names are not possible because all public property names are numeric and public method names are alphabetic. Instead, we just need to specify the semantics of member lookup and call operations to look in the appropriate place for their target.

The value of the [[Name]] property must be a String containing a legal XML element name, attribute name, PI name or the empty string. The value of the [[Name]] property is the empty string if and only if the XML value represents an XML document, comment or text node.

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 #2: Necessity of parent pointer.

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 XML objects. The value of the [[Attributes]] property is always an Array of size zero if the XML value represents an XML document, attribute, comment, PI or text node.

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

6.1.1.1 [[Get]] (P)

Overview

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 elements by their name. The input variable *P* may be a numeric property name, an XML attribute name (distinguished from the name of XML elements by a leading "@" symbol), an XML element name, the properties wildcard "*" or the attribute wildcard "@*".

Semantics

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. If *x*.[[*Class*]] ∈ {XMLText, XMLComment, XMLPI, XMLAttribute}, Return undefined
    b. Call the Object.[[Get]] method with *x* as the **this** object and parameter *P* and return the result
2. Let *l* be an empty XMLList
3. If *x*.[[*Class*]] ∈ {XMLText, XMLComment, XMLPI, XMLAttribute}, Return *l*
4. If *P*[0] == '@'
    a. Let *n* = *P*.substring(1, *P*.length)
    b. for each *a* in *x*.[[*Attributes*]]
        i. if (*n* == "*") or (*n* == *a*.[[*Name*]]),
            1. Call [[Append]] with *l* as the **this** object and parameter *a*
    c. Return *l*.
5. for each property *q* in *x*
    a. if (*P* == "*") or ((*x*[*q*].[[*Class*]] == XML) and (*x*[*q*].[[*Name*]] == *P*))
        i. call [[Append]] with *l* as the **this** object and parameter *x*[*q*]
6. Return *l*.

Decision: Barring a counter example, we will represent Attributes as Strings and the absence of an attribute as the undefined value. (Offline thought required – may be reversed).

Counter example: see issue #1.

Issue #4: Unexpected property names may cause unexpected results.

It's possible unintended consequences could result if property names are read dynamically from a source not controlled by the program (e.g., name of attribute, numeric property name instead of element name, etc.)

6.1.1.2 [[Put]] (P, V)

Overview

The XML type overrides the internal [[Put]] method defined by the Object type. The XML [[Put]] method may be used to modify, replace, and insert properties or XML attributes 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, an XML element name or the properties wildcard "*". The input variable *V* may be an XML value, an XMLList value or any value that may be converted to a String with ToString()

Semantics

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

1. *x*.[[*Class*]] ∈ {XMLText, XMLComment, XMLPI, XMLAttribute}

a. Return

2. Create a deep copy *c* of *V*
3. If *P*[0] == "@"
    a. Let *n* = *P*.substring(1, *P*.length)
    b. if *P* is not a valid XML attribute name (or "*"), throw an XML exception
    c. if *c*.[[*Class*]] == XMLList
        i. if *c*.[[*Length*]] == 0, *c* = ""
        ii. else
            1. Let *s* = *c*[0]
            2. for *i* = 1 to *c*.[[*Length*]], s += " " + ToString(*c*[*i*])
        iii. *c* = *s*
    d. else
        i. *c* = ToString(*c*)
    e. Let *a* = null
    f. For *j* = 0 to *x*.[[*Attributes*]].length
        i. If *x*.[[*Attributes*]][*j*].[[*Name*]] == *n*, *a* = *x*.[[*Attributes*]][*j*]
    g. if *a* == null
        i. *a* = a new XML value with *a*.[[*Name*]] = *n*, *a*.[[*Class*]] == XMLAttribute and *a*.[[*Parent*]] = *x*
        ii. append *a* to *x*.[[*XMLAttributes*]]
    h. *a*[0] = *c*
    i. Return
4. If ToString(ToUint32(*P*)) == *P*
    a. call [[Replace]] with *x* as the **this** object and parameters *P* and *c* and return
5. Set *i* = undefined
6. Let *primitiveAssign* = ((*c*.[[*Class*]] ∉ {XML, XMLList, XMLComment, XMLPI}) and (*P* != "*"))
7. for (*k* = 0 to *x*.[[*Length*]])
    a. if (*P* == "*") or ((*x*[*k*].[[*Class*]] == XML) and (*x*[*k*].[[*Name*]] == *P*))
        i. if (*i* == undefined), *i* = *k*
        ii. else Call [[Delete]] with *x* as the **this** object and parameter ToString(*k*)
8. if *i* == undefined
    a. *i* = *x*.[[*Length*]]
    b. if (*primitiveAssign*)
        i. if *P* is not a valid XML element name, throw an XML exception
        ii. Create a new XML object *y* with *y*.[[*Name*]] = *P*, *y*.[[*Class*]] = XML and *y*.[[*Parent*]] = *x*
        iii. call [[Replace]] with *x* as the **this** object and parameters ToString(*i*) and *y*
9. if (*primitiveAssign*)
    a. delete all the properties of the XML value *x*[*i*]
    b. call [[Replace]] with *x*[*i*] as the **this** object and parameters "0" and *c*

10. else
    a. call [[Replace]] with *x* as the **this** object and parameters ToString(*i*) and *c*
11. Return

Note: Need to explicitly handle special XML characters like "<" and ">" embedded in string values. Also need to handle double-quotes included in attribute values

To do: Verify that x.y = "" results in an empty element <y/> inside x. Send e-mail to list.

Issue #21: As currently defined, appending to an empty child list does not impact the original document

6.1.1.3 [[Delete]] (P)

Overview

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), an XML value name, the properties wildcard "*" or the attributes wildcard "@*".

Semantics

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

1. *x*.[[*Class*]] ∈ {XMLText, XMLComment, XMLPI, XMLAttribute}
    a. Return true
2. Let *i* = ToUint32(*P*)
3. 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
4. If *P*[0] == '@'
    a. Let *n* = *P*.substring(1, *P*.length)
    b. for each *a* in *x*.[[*Attributes*]]
        i. if (*n* == "*") or (*n* == *a.name*), remove *a* from *x*.[[*Attributes*]]
    c. Return true.

5. let *dp* = 0
6. for each property *q* in *x*
    a. if (*P* == "*") or (*x*[*q*].[[*Class*]] == XML and *x*[*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*)
7. *x*.[[*Length*]] = *x*.[[*Length*]] - *dp*
8. Return true.

6.1.1.4 [[Descendants]] (P)

Overview

The XML type adds the internal [[Descendants]] method to the internal properties defined by the Object type. The XML [[Descendants]] methed may be used to retrieve all the XML valued descendants of this XML object (i.e., children, grandchildren, great-grandchildren, etc.) with names matching the input variable *P*. The input variable *P* may be an XML attribute name (distinguished from the name of XML elements by a leading "@" symbol), an XML element name, the properties wildcard "*" or the attribute wildcard "@*".

Semantics

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

1. Let *l* be an empty XMLList
2. If (*x*.[[*Class*]] ∈ {XMLText, XMLAttribute, XMLComment, XMLPI}, return *l*
3. If (*P*[0] == '@')
    a. Let *n* = *P*.substring(1, *P*.length)
    b. For each XMLAttribute *a* in *x*.[[*Attribute*]]
        i. If (*n* == "*") or (*n* == *a.name*)
            1. Call [[Append]] with *l* as the **this** object and parameter *a*
4. for each property *q* in *x*
    a. if (*P* == "*") or ((*x*[*q*].[[*Class*]] == XML) and (*x*[*q*].[[*Name*]] == *P*))
        i. call [[Append]] with *l* as the **this** object and parameter *x*[*q*]
    b. Call [[Descendants]] with parameters *x*[*q*] and *P*
    c. Call [[Append]] with *l* as the **this** object and parameter Result(b)
2. return *l*

6.1.1.5 [[Insert]] (P, V)

Overview

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 a value of type XML, XMLList, XMLComment, XMLPI, XMLText or any value that can be converted to a String with ToString().

Semantics

When the [[Insert]] method of an XML 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*) throw IllegalArgument exception
3. Let *n* = 1
4. If *V*.[[*Class*]] == XMLList, *n* = *V*.[[*Length*]]
5. if *n* == 0, Return
6. for *j* = *x*.[[*Length*]]-1 downto *i*, rename property *j* of *x* to ToString(*j* + *n*)
7. *x*.[[*Length*]] += *n*
8. if *V*.[[*Class*]] == XMLList
    a. for *j* = 0 to *V*.[[*Length*-1]] , call [[Replace]] with *x* as the **this** object and parameters ToString(*i* + *j*) and *V*[*j*]
9. else
    a. call [[Replace]] with *x* as the **this** object and parameters *i* and *V*
10. Return

6.1.1.6 [[Replace]] (P, V)

Overview

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

Semantics

When the [[Replace]] method of an XML 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*) throw IllegalArgument exception
3. If *i* >= *x*.[[*Length*]],
    a. *P* = ToString(*x*.[[*Length*]])
    b. *x*.[[*Length*]]++

4. If *V*.[[*Class*]] ∈ {XML, XMLComment, XMLPI, XMLText}
    a. set *V*.[[*Parent*]] to *x*
    b. set the value of property *P* of *x* to *V*
5. else if *V*.[[*Class*]] == XMLList
    a. Call [[Delete]] with *x* as the **this** object and parameter *P*
    b. Call [[Insert]] with *x* as the **this** object and parameters *P* and *V*
6. else
    a. Let *s* = ToString(*V*)
    b. Let *t* = ToXML(*s*)
    c. Set *t*.[[*Parent*]] = *x*
    d. set the value of property *P* of *x* to *t*
7. 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 has a value of type XML, XMLAttribute, XMLComment, XMLPI, XMLText, String, Boolean or Number. A value of type XMLList may represent an XML Document, XML Fragment or an arbitrary collection of properties (e.g., a query result).

Note: Unlike the XML type, the XMLList type allows properties of type Boolean and Number. This capability allows XML operators that return non-XML types to be applied to XMLLists. For example, order.item.price.childIndex() will return a List containing the numeric child index of each the price element within each item element.

Issue #5: XMLLists store lvalues or rvalues.

There is an open discussion regarding the types of values XMLLists should store. It is important to preserve the ability to use XMLLists to update fragments of XML.

Issue #6: XMLList vs. generalized list type.

Can we introduce a general List type with implicit iteration and toString() which concatentates results of content.toString() instead of the specialized XMLList type?

Issue #7: Immutable list of methods for type XMLList (or List)

Should the list of XMLList (or List) methods be immutable?

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.
[[Append]]	(*Value*)	Appends a new property to the end of the list.
[[Descendants]]	(*PropertyName*)	Returns an XMLList containing the XML valued descendants of this XML values in this XMLList with names tha match *propertyName*.

Todo: describe method lookup for XML and XMLList. See section 11.2.3 & 8.7.1 of E3

Issue #3: Distinguishing built-in XML methods from XML valued properties with the same name.

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

Proposed Resolution: This can be achieved by defining appropriate call and member lookup semantics for XML types. Built-in methods may be represented in the data model as properties without name clashes since all XML valued properties have numeric property names that will not clash with method names.

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

6.2.1.1 [[Get]] (P)

Overview

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), an XML value name, the properties wildcard "*" or the attributes wildcard "@*".

Semantics

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 property *q* in *x*,
    a. if *x*[*q*].[[*Class*]] == XML,
        i. call [[Get]] with *x*[*q*] as the **this** object and parameter *P*
        ii. call [[Append]] with *l* as the **this** object and parameter Result(i)
4. Return *l*.

Issue #8: Using "." to operate on and return lists.

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)

Overview

The XMLList type overrides the internal [[Put]] method defined by the Object type. The XMLList [[Put]] method may be used to modify or replace an XML value within the XMLList and the context of its parent. In addition, when the XMLList contains a single property with an XML value, the [[Put]] method may be used to modify, replace, 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 numeric property name, XML attribute name (distinguished from XML valued property names by a leading "@" symbol), an XML name, the properties wildcard "*" or the attributes wildcard "@*". The input variable *V* may be a value of type XML, XMLList, XMLComment, XMLPI or any value that can be converted to a String with ToString().

Semantics

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. Create a new XML value *y*, with *y*.[[*Parent*]] = Null and *y*.[[*Class*]] = XMLText
        ii. *i* = *x*.[[*Length*]]
        iii. if (*i* > 0) and (*x*[ToString(*i*-1)].[[*Class*]] != XMLAttribute)
            1. Let *parent* = *x*[ToString(*i*-1)].[[*Parent*]]
            2. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*[*i*-1]
            3. call [[Insert]] with *parent* as the **this** object and parameters ToString(ToUint32(*q*)+1) and *y*
        iv. call [[Append]] with *x* as the **this** object and parameter *y*
    b. if *x*[*i*].[[*Class*]] = XMLAttribute
        i. Let *attname* = "@" + *x*[*i*].[[*Name*]]
        ii. Call [[Put]] with *x*[*i*].[[*Parent*]] as the **this** object and parameters *attname* and *V*
    c. else if *V*.[[*Class*]] == XMLList

i. Create a shallow copy *c* of *V*
ii. Let *parent* = *x*[*i*].[[*Parent*]]
iii. if *parent* != Null
1. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*[*i*]
2. Call [[Put]] with *parent* as the **this** object and parameters *q* and *c*
3. for *j* = 0 to *c*.[[*Length*]]
a. *c*[ToString(*j*)] = *parent*[ToString(ToUint32(*q*)+*j*)]
iv. for *j* = *x*.[[*Length*]]-1 downto *i*, rename property *j* of *x* to ToString(*j* + *c*.[[*Length*]])
v. for *j* = 0 to *c*.[[*Length*]], *x*[*i* + *j*] = *c*[*j*]
vi. *x*.[[*Length*]] += *c*.[[*Length*]]

d. else if (*V*.[[*Class*]] ∈ {XML, XMLComment, XMLPI}) or (*x*[*i*] ∈ {XMLText, XMLComment, XMLPI})
i. Let *parent* = *x*[*i*].[[*Parent*]]
ii. if *parent* != Null
1. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*[*i*]
2. Call [[Put]] with *parent* as the **this** object and parameters *q* and *V*
3. *V* = *parent*[*q*]
iii. *x*[*i*] = *V*

e. else
i. Call [[Put]] with *x*[*i*] as the **this** object and parameters "*" and *V*

3. else if *x*.[[*Length*]] == 1 and *x*["0"].[[*Class*]] == XML
a. Call [[Put]] with *x*["0"] as the **this** object and parameters *P* and *V*
4. else
a. throw TypeException
5. Return

Note: When a non-numeric property name is given, we could have just as well applied the [[Put]] operation to every XML valued 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 exception.

6.2.1.3 [[Delete]] (P)

Overview

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), an XML value name, the properties wildcard "*" or the attributes wildcard "@*".

Semantics

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. Let *parent* = *x*[*i*].[[*Parent*]]
        ii. if *parent* != Null
            1. if *x*[*i*].[[*Class*]] == XMLAttribute
                a. Let *attname* = "@" + *x*[*i*].[[*Name*]]
                b. Call [[Delete]] with *parent* as the **this** object and parameter *attname*
            2. else
                a. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*[*i*]
                b. Call [[Delete]] with *parent* as the **this** object and parameter *q*
        iii. remove the property with the name *P* from *x*
        iv. for each property *q* of *x* such that ToUint32(*q*) > *i*, rename *q* to ToString(ToUint32(*q*) – 1)
        v. *x*.[[*Length*]] = *x*.[[*Length*]] – 1
    c. Return true
3. for each *q* in *x*,
    a. if *x*[*q*].[[*Class*]] == XML, call [[Delete]] with *x*[*q*] as the **this** object and parameter *P*
4. Return true

6.2.1.4 [[Append]] (V)

Overview

The XMLList type adds the internal [[Append]] method to the internal properties defined by the Object type. The XMLList [[Append]] method may be used to append a value *V* to the end of the XMLList. The input variable *V* may be an XML value or any value that can be converted to a String with ToString().

Semantics

When the [[Append]] method of an XMLList object *x* is called with value *V*, the following steps are taken:

1. Let *i* = *x*.[[*Length*]]
2. Let *n* = 1
3. if *V*.[[*Class*]] == XMLList,
    a. *n* = *V*.[[*Length*]]

b. if *n* == 0, Return

c. for *j* = 0 to *V*.[[*Length*-1]], *x*[*i* + *j*] = *V*[*j*]

4. else

a. Set the value of property *i* of *x* to *V*

5. *x*.[[*Length*]] += *n*

6. Return

6.2.1.5 [[Descendants]] (P)

Overview

The XMLList type adds the internal [[Descendants]] method to the internal properties defined by the Object type. The XML [[Descendants]] methed may be used to retrieve all the XML valued descendants of the properties in this XMLList (i.e., children, grandchildren, great-grandchildren, etc.) with names matching the input variable *P*. The input variable *P* may be an XML attribute name (distinguished from the name of XML elements by a leading "@" symbol), an XML element name, the properties wildcard "*" or the attribute wildcard "@*".

Semantics

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

1. Let *l* be an empty XMLList
2. for each property *q* in *x*
   a. if (*x*[*q*].[[*Class*]] == XML)
      i. Call [[Descendants]] with parameters *x*[*q*] and *P*
      ii. Call [[Append]] with *l* as the **this** object and parameter Result(b)
3. return *l*

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 methods 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 7.1.1.
XMLList	Return the XMLList value as a string as defined in section 7.1.2.

7.1.1 ToString Applied to the XML Type

Overview

Given an XML value *x*, the operator ToString converts *x* to a string *s*. If a value of type XML contains only properties of type XMLText (i.e., contains a primitive value), ToString returns the String contents of the XML value, omitting the start tag, attributes and end tag. 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 the example above, the String valued 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.

For values of type XMLAttribute and XMLText, ToString simply returns their value as a string.

Semantics

Given an XML value *x*, ToString takes the following steps:

1. Let *s* be an empty string.

2. Let *primitive* = *true*.
3. For each property *p* in *x*
    a. if *x*[*p*].[[*Class*]] != XMLText, *primitive* = false.
4. If not *primitive*, return ToXMLString(*x*)
5. Otherwise, for each property *p* in *x*, append ToString(*x*[*p*]) to *s* in order.
6. Return *s*.

Issue #9: Comparing XML strings.
Should we support 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?

To do: Make sure we handle escaped characters properly here and in ToXMLString()

7.1.2 ToString Applied to the XMLList Type

Overview

The operator ToString converts an XMLList value *l* to a string *s*. The return value is a comma separated list of the string representation for each item in the XMLList.

Note that the result of calling ToString on a list of size one is identical to the result of calling ToString on the single item contained in the XMLList. 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.

Semantics

Given an XMLList value *l*, ToString performs the following steps:

1. Let *s* be an empty string.
2. For i = 0 to *l*.[[Length]],
    a. append ToString(*l*[*i*]) to *s*.
    b. if *i* < (*l*.[[Length]] – 1), append "," to *s*
3. Return *s*.

Issue #11: Methods defined on XMLList and XML may hide methods added to String
For example, if we have an XML object called "order" and the user adds a toXMLString() method to the String object, the expression "order.customer.name.toXMLString()" will execute the XMLList toXMLString() method, not the String toXMLString() method.

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 tag, attributes and end tag associated with an XML element, 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 7.2.1.
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

Semantics

Given an XML value *x*, ToXMLString converts it to an XML encoded string *s* by taking the following steps:

1. Let *s* be an empty string
2. If *x*.[[*Class*]] == XMLText or *x*.[[*Class*]] == XMLAttribute, Return *x*[0]
3. if *x*.[[*Class*]] = XMLComment, Return "<!--" + *x*[0] + "-->"
4. if *x*.[[*Class*]] = XMLPI, Return "<?" + *x*.[[*Name*]] + " " + *x*[0] + "?>"
5. Append "<" to *s*.
6. Append *x*.[[*Name*]] to *s*.
7. 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*.

8. If *x* contains no properties, append "/>" to *s* and return *s*.
9. Otherwise, append ">" to *s*.
10. For each property *p* in *x*, append ToXMLString(*x*[*p*]) to *s* in order.
11. Append "</" to *s*.
12. Append *x*.[[*Name*]] to *s*.
13. Append ">" to *s*.
14. Return *s*.

Issue #15: Behavior of ToString on XML elements with complex content

There is not yet agreement on the behavior of ToString for XML elements with complex content. Some believe ToString should generate the String equivalent of the XML value, complete with start tag, attributes, children and end tag. Others believe the start tag, end tag and attributes should be omitted emitting only the children of the XML value.

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	Convert the input argument to a string using ToString then convert the result to XML as specified in section 7.3.1.
Number	Convert the input argument to a string using ToString then convert the result to XML as specified in section 7.3.1.
String	Create an XML from the String as specified below in section 7.3.1.
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, throw a **TypeError** exception.
W3C DOM Element	Create an XML value from the W3C DOM Element as specified below in section 7.3.2.
Object	Throw a **TypeError** exception.

Would we also like to have ToXML defined for type Object? (Rok is checking for MSFT)

7.3.1 ToXML Applied to the String Type

Overview

When ToXML is applied to a string type, it converts it to XML by parsing the string as XML. 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 +”>”);
```

Semantics

Given a String value *s*, ToXML converts it to XML using the following steps:

1. Parse *s* as a W3C DOM Element *e*.
2. If the parse succeeds, return ToXML(*e*)
3. Create a new XML value *x* with *x*.[[*Class*]] = XMLText, *x*.[[*Parent*]] = Null, and *x*[0] = *s*
4. 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.

7.3.2 ToXML Applied to a W3C DOM Element

Semantics

A W3C DOM Element *e* is converted to type XML as follows:

1. Create a new value *x* of type XML.
2. Set *x*.[[*Name*]] to *e.tagName*.
3. For each attribute *a* in *e.attributes*,
    a. Let *name* = “@” + *a.nodeName*
    b. Call [[Put]] with *x* as the **this** object and parameters *name* and *e.nodeValue*
4. For each node *n* in *e.childNodes*
    a. Let *child* = undefined
    b. If *n.nodeType* == ELEMENT_NODE, *child* = ToXML(*n*)
    c. If *n.nodeType* == TEXT_NODE or *n.nodeType* == CDATA_SECTION_NODE,
        i. *child* = new String(*n.nodeValue*).
    d. If *n.nodeType* == COMMENT_NODE,
        i. *child* = new XML value with *c*.[[*Class*]] = XMLComment and *c*[0] = *n.nodeValue*
    e. if *n.nodeType* == PROCESSING_INSTRUCTION_NODE,
        i. *child* = new XML value with *c*.[[*Class*]] = XMLPI, *c*.[[*Name*]]=*n.nodeName* and *c*[0]=*n.NodeValue*
    f. Call [[Replace]] with *x* as the **this** object and parameters ToString(*x*.[[*Length*]]) and *child*.
5. Return *x*.

8 Execution Contexts

9 Expressions

9.1 Primary Expressions

E4X extends the primary expressions defined by ECMAScript with the following production

PrimaryExpression :

XMLLiteral

XMLListLiteral

9.1.1 XML Initializer

An XML initializer is an expression describing the initialization of an XML value written in a form resembling a literal. It provides the name, XML attributes and XML properties of an XML value using ordinary XML element syntax.

XML initializers begin with the character "<". Upon encountering this character, the parser scans to the end of the literal and passes the string of characters comprising the XML literal to the XML constructor uninterpreted. The XML constructor interprets the string according to its own more stringent grammar. The productions below describe the syntax for an XML literal and are used by the parser to find the end of the XML literal.

XMLLiteral ::

< *XMLChars* />

< *XMLChars* > *XMLChars* </ *XMLChars* >

< *XMLChars* > *XMLLiterals* </ *XMLChars* >

<? *SourceCharacters* ?>

<!-- *SourceCharacters* -->

<![CDATA[*SourceCharacters*]]>

XMLLiterals ::

[empty]

XMLLiterals XMLLiteral

XMLChars ::

[empty]

XMLChars XMLChar

XMLChar ::

SourceCharacter **but not < or / or >**

Todo: 1) Separate grammar from inside of tag and XML content (also inside of PI and comment)

2) CDATA content cannot contain "]]>"

Todo: check to see if it is legal to have a space between "--" and ">" in a comment terminator

To do: resolve ambiguity in grammar per previous meeting using regex solution (see notes)

To do: adjust grammar to correctly identify terminating characters for comments, PIs

To do: adjust grammar to accept mixed content

Below are two examples of XML initializers.

```
// an XML value representing a person with a name and age
var person = <person><name>John</name><age>25</age></person>;

// a variable containing an XML value representing two employees
var e = <employees>
            <employee id="1"><name>Joe</name><age>20</age></employee>
            <employee id="2"><name>Sue</name><age>30</age></employee>
        </employees>;
```

Portions of an XML literal may be computed dynamically using expressions embedded in curly braces. For example,

```
for (i = 0; i < 10; i++)
        e[i] = <employee id={i}>                                // compute id value
                <name>{names[i].toUpperCase()}</name>  // compute name value
                <age>{ages[i]}</age>                                    // compute age value
              </employee>;
```

Each expression embedded in curly braces is evaluated and replaced by its value prior to parsing the literal XML value. Therefore, any portion of the XML literal can be determined dynamically. For example the following expression,

```
var tagname = "name";
var attributename = "id";
var attributevalue = 5;
var content = "Fred";

var x = <{tagname} {attributename}={attributevalue}>{content}</{tagname}>;
```

would assign the following XML value to the variable x.

```
<name id="5">Fred</name>
```

Issue # 13: String length for multi-line strings may differ on different platforms

Because different platforms use different character sequences to represent an end-of-line (e.g., "\n" or "\r\n"), the string length of multi-line string values within XML literals may differ on different platforms.

Issue #16: Rok proposes that we remove XML literal and change delimiter for XMLList literal to "#" (use for both lists and degenerate case of XML element)

There are concerns about the need to escape all "#" characters in XML pasted into the code. There are also concerns about complicating the common case of an XML element literal. Also interaction with IDREFs

9.1.2 XMLList Initialiser

An XMLList initializer is an expression describing the initialization of an XMLList value written in a form resembling a literal. It describes an ordered list of XML properties using an anonymous XML element syntax.

Like XML initializers, XMLList initializers begin with the character "<". Upon encountering this character, the parser scans to the end of the literal and passes the string of characters comprising the XMLList literal to the XMLList constructor uninterpreted. The XMLList constructor interprets the string according to its own more stringent grammar. The production below describes the syntax for an XMLList literal and is used by the parser to find the end of the XMLList literal.

XMLList ::

<> *XMLLiterals* </>

Below are some examples of XMLList Initializers,

```
var docfrag = <><name>Phil</name><age>35</age><hobby>skiing</hobby></>;

var emplist =  <>
               <employee id="0" ><name>Jim</name><age>25</age></employee>
               <employee id="1" ><name>Joe</name><age>20</age></employee>
               <employee id="2" ><name>Sue</name><age>30</age></employee>
               </>;
```

9.2 Left-Hand-Side Expressions

E4X extends the left-hand-side expressions defined in ECMAScript with the following productions:

MemberExpression :

MemberExpression .. *Identifier*

MemberExpression . (*Expression*)

CallExpression :

CallExpression .. *Identifier*

CallExpression . (*Expression*)

In addition, E4X defines new semantics for existing left-hand-side expressions applied to values of type XML and XMLList.

To do: modify call expression to include these operators similarly

9.2.1 XML Property Accessor

Syntax

E4X reuses ECMAScript's property accessor syntax for accessing properties and XML attributes within values of type XML and XMLList. XML properties may be accessed by name, using either the dot notation:

MemberExpression . *Identifier*

CallExpression . *Identifier*

or the bracket notation:

MemberExpression [*Expression*]

CallExpression [*Expression*]

such that

MemberExpression . *Identifier*

is identical in behavior to

MemberExpression [<*identifier-string*>]

and similarly

CallExpression . *Identifier*

Is identical in behaviour to

CallExpression [<*identifier-string*>]

where <*identifier-string*> is a string literal containing the same sequence of characters as the *Identifier*.

Issue #17: We need to support the descendant operator with a run-time property name argument.

Overview

When *MemberExpression* or *CallExpression* evaluate to a XML value, the property accessor uses the XML [[Get]] method to determine the result. If the bracket notation is used with a numeric identifier, the XML [[Get]] method simply returns the property of the left operand with a property-name matching the numeric identifier. Otherwise, the XML [[Get]] method examines the XML properties and XML attributes of the left operand and returns an XMLList containing the ones with names that match its right operand in order. For example,

```
var order = <order id = "123456" timestamp="Mon Mar 10 2003 16:03:25 GMT-0800 (PST)">
                <customer>
                        <firstname>John</firstname>
                        <lastname>Doe</lastname>
                </customer>
                <item>
                        <description>Big Screen Television</description>
                        <price>1299.99</price>
                        <quantity>1</quantity>
                </item>
            </order>;

var customer = order.customer;          // get the customer element from the order
var id = order.@id;                     // get the id attribute from the order
var secondChild = order[1];             // get the second child element from the order by numeric index
var orderChildren = order.*;            // get all the child elements from the order element
var orderAttributes = order.@*;         // get all the attributes from the order element
```

When *MemberExpression* or *CallExpression* evaluate to an XMLList, the property accessor uses the XMLList [[Get]] method to determine the result. If the bracket notation is used with a numeric identifier, the XMLList [[Get]] method simply returns the property of the left operand with a property-name matching the numeric identifier. Otherwise, the XMLList [[Get]] method applies the property accessor operation to each XML value in the list and returns a new XMLList containing the results in order. For example,

```
var order = <order>
                <customer>
                        <firstname>John</firstname>
                        <lastname>Doe</lastname>
                </customer>
                <item id = ”3456”>
                        <description>Big Screen Television</description>
                        <price>1299.99</price>
                        <quantity>1</quantity>
                </item>
                <item id = “56789”>
                        <description>DVD Player</description>
                        <price>399.99</price>
                        <quantity>1</quantity>
                </item>
            </order>;

var descriptions = order.item.description;     // get the list of all item descriptions
var itemIds = order.item.@id;                  // get the list of all item id attributes
var secondItem = order.item[1];                // get second item by numeric index
var itemChildren = order.item.*;               // get the list of all child elements in all item elements
```

In the first property accessor statement above, the expression “order.item” examines the XML properties of the XML value bound to “order” and returns an XMLList containing the two named “item”. The expression “order.item.description” then examines the XML properties of each item in the resulting XMLList and returns an XMLList containing the two XML values named “description”.

Semantics

E4X extends the semantics of the property accessor by providing more elaborate [[Get]] methods used when *MemberExpression* or *CallExpression* evaluate to a value of type XML or XMLList (see sections 6.1.1.1 and 6.2.1.1 respectively).

9.2.2 XML Descendant Accessor

Syntax

E4X extends ECMAScript by adding a descendant accessor. The following productions describe the syntax of the descendant accessor:

MemberExpression :

MemberExpression .. Identifier

CallExpression :

CallExpression .. Identifier

Overview

When the *MemberExpression* or *CallExpression* evaluate to an XML value or an XMLList, the descendant accessor examines all of the descendant XML properties (i.e., children, grand children, great-grandchildren, etc) of its left operand and returns an XMLList containing those with names that match its right operand in order.

The descendant operator provides the expressive power of XPath's descendant operator (i.e., "//") within the context of a modern programming language. For example,

```
var e = <employees>
          <employee id="1"><name>Joe</name><age>20</age></employee>
          <employee id="2"><name>Sue</name><age>30</age></employee>
        </employees>;

var names = e..name;              // get all the names in e
```

Semantics

Given a *MemberExpression* and an *Identifier*, the XML Descendant Accessor performs the following steps:

1. Let *x* = evaluate(*MemberExpression*)
2. Let *P* = evaluate(*Identifier*)
3. Call [[Descendants]] with *x* as the **this** object and parameters *P*
4. Return Result(3)

The production *CallExpression* : *CallExpression* .. *Identifier* is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

9.2.3 XML Filtering Predicate Operator

Syntax

E4X extends ECMAScript by adding a filtering predicate operator. The following productions describe the syntax of the filtering predicate operator:

MemberExpression :

MemberExpression . (*Expression*)

CallExpression :

CallExpression . (*Expression*)

Overview

When the left operand is an XML value, the filtering predicate adds the left operand to the front of the scope chain of the current execution context, evaluates the *Expression* with the augmented scope chain, converts the result to a Boolean value, then restores the scope chain. If the result is true, the filtering predicate returns an XMLList containing the left operand. Otherwise it returns an empty XMLList.

When the left operand is an XMLList, the filtering predicate is applied to each XML property in the XMLList in order using the XML value as the left operand and the *Expression* as the right operand. It concatenates the results and returns them as a single XMLList containing all the XML properties for which the result was true. For example,

```
var john = e.employee.(name == “John”);                    // employees with name John
var twoemployees = e.employee.(@id == 0 || @id == 1);      // employees with id’s 0 & 1
var emp = e.employee.(@id == 1).name;                      // name of employee with id 1
```

The effect of the filtering predicate is similar to SQL’s WHERE clause or XPath’s filering predicates.

In essence, the statement:

```
// get the two employees with ids 0 and 1 using a predicate
var twoEmployees = e..employee.(@id == 0 || @id == 1);
```

is semantically equivalent to the following:

```
// get the two employees with the ids 0 and 1 using a for loop
var i = 0;
var twoEmployees = new XMLList();
for (var p in e..employee) {
        if (p.@id == 0 || p.@id == 1) {
                twoEmployees[i++] = p;
        }
}
```

Semantics

Given a value *x* = evaluate(*MemberExpression*) of type XML and an *Expression*, the filtering predicate performs the following steps:

1. Add *x* to the front of the scope chain
2. Evaluate *Expression* using the augmented scope chain from step 1
3. Let *match* = ToBoolean(Result(2))
4. Remove *x* from the front of the scope chain
5. Let *l* = an empty XMLList
6. if (*match* == *true*) Call [[Append]] with *l* as the **this** object and parameter *x*
7. Return *l*

The production *CallExpression* : *CallExpression* . (*Expression*) is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

Given a value *x* = evaluate(*MemberExpression*) of type XMLList and an *Expression*, the filtering predicate performs the following steps:

1. Let *l* = an empty XMLList
2. for each property *p* of *x*
    a. if *x*.[[*Class*]] ∈ {XML, XMLAttribute, XMLComment, XMLPI, XMLText}
        i. Let *m* = *p*.(*Expression*)
        ii. Call [[Append]] with *l* as the **this** object and parameter *m*
3. Return *l*

The production *CallExpression* : *CallExpression* . (*Expression*) is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

Issue #18: Same problem as with statement.

Possible alternative is using "." To reference the context node. Waldemar will check grammar to see if this is possible.

Issue #19: Need some way to test against computed property names ala square brackets

Possible option is child() method that calls [[Get]]

9.3 Unary Operators

9.3.1 The Delete Operator

Syntax

E4X reuses the ECMAScript delete operator for deleting XML properties and XML attributes from XML values and XMLLists. The syntax of the delete operator is described by the following production:

UnaryExpression :

delete *UnaryExpression*

Overview

When *UnaryExpression* evaluates to an XML value or an XML attribute, the delete operator gets the parent of the XML value or XML attribute, then removes the XML value or XML Attribute from the list of XML properties or XML attributes

associated with the parent. When *UnaryExpression* evaluates to an XMLList, the delete operator is applied to each XML property in the XMLList. For example,

```
delete order.customer.address;      // delete the customer address
delete order.customer.@id           // delete the customer ID
delete order.item.price[0];         // delete the first item price
delete order.item;                  // delete all the items
```

Semantics

E4X extends the semantics of the delete operator by providing more elaborate [[Delete]] methods used when *UnaryExpression* evaluates to a value of type XML or XMLList (see sections 6.1.1.3 and 6.2.1.3 respectively).

9.4 Additive Operators

Syntax

E4X reuses the ECMAScript addition operator to concatenate two values of type XML or XMLList. The ECMAScript syntax for the addition operator is described by the following production:

AdditiveExpression :

AdditiveExpression + *MultiplicativeExpression*

Overview

When both *AdditiveExpression* and *MultiplicativeExpression* evaluate to either an XML value or an XMLList, the addition operator starts by creating a new, empty XMLList as the return value. If the left operand is an XML value, it is added to the return value. If the left operand is an XMLList, each XML property of the XMLList is added to the return value in order. Likewise, if the right operand is an XML value, it is added to the return value. Otherwise, if it is an XMLList each XML property of the XMLList is added to the return value in order.

For example,

```
// create an XMLList containing the elements <name>, <age> and <hobby>
var employeedata = <name>Fred</name> + <age>28</age> + <hobby>skiing</hobby>;

// create an XMLList containing three item elements extracted from the order element
var myitems = order.item[0] + order.item[2] + order.item[3];

// create a new XMLList containing all the items in the order plus one new one
var newitems = order.item + <item><description>new item</description></item>;
```

Note: Using the addition operator with operands of type XML and XMLList always results in an XMLList. When numeric addition of XML values is desired, the operands must be explicity coerced to Numbers. Than may be accomplished by using the unary “+” operator or the Number conversion function. For example,

```
// add the prices of the first and third items in the order (coersion with unary +)
var totalPrice = +order.item[0].price + +order.item[2].price

// add the prices of the second and fourth items in the order (coersion using Number conversion function)
var totalPrice = Number(order.item[1].price) + Number(order.item[3].price)
```

Likewise, when string concatenation of XML values is desired, the operands must be explicitly coerces to Strings. This may be accomplished by concatenating them to the empty string ("") or using the String conversion function. For example,

```
// concatenate the street and the city of the customer's address (coersion with the empty string)
var streetcity = "" + order.customer.address.street + order.customer.address.city;

// concatenate the state and the zip of the customer's address (coersion using String conversion function)
var statezip = String(order.customer.address.state) + order.customer.address.zip;
```

Semantics

Given a left operand *x* = evaluate(*AdditiveExpression*) and a right operand *y* = evaluate(*MultiplicativeExpression*) that both evaluate to type XML or XMLList, the addition operator performs the following steps:

1. Let *l* = an empty XMLList
2. Call [[Append]] with *l* as the **this** object and parameter *x*
3. Call [[Append]] with *l* as the **this** object and parameter *y*
4. Return *l*

Issue #10: "+" operator for concatenating lists

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

Issue #20: Potential issue extracting several text nodes and concatenating them expecting a string (given decision to represent text nodes as XML instead of String)

Todo: change all Result(#) notation instances to local variable references (Let)

9.5 Assignment Operators

9.5.1 XML Assignment Operator

Syntax

Todo: restructure into another section and make non-normative.

E4X reuses the ECMAScript assignment operator to modify, replace and insert properties and XMLAttributes in an XML value. The ECMAScript syntax for the assignment operator is described by the following production:

AssignmentExpression :

LeftHandSideExpression = *AssignmentExpression*

Overview

The assignment operator begins by evaluating the *LeftHandSideExpression*, which resolves to a reference *r* consisting of a base object *parent* and a *property-name*. If *parent* is an XML value, the assignment operator performs the steps described in section (see section 9.5.2 for the steps performed if *parent* is an XMLList).

If the *property-name* begins with the character "@", the XML assignment operator creates or modifies an XMLAttribute in the *parent*. If the named XMLAttribute already exists, the assignment operator modifies its value, otherwise it creates a new XMLAttribute with the given name and value. If *AssignmentExpression* evaluates to an XMLList, the value of the named attribute will be a space separated list of values (i.e., an XML attribute list) constructed by converting each value in the XMLList to a string and concatenating the results separated by spaces. If the *AssignmentExpression* does not evaluate to an XMLList, the value of the named attribute will be derived by evaluating the *AssignmentExpression* and calling ToString on the result. For example,

```
order.item[1].@id = 123;                  // change the value of the id attribute on the second item
order.item[1].@newattr = "new value";     // add a new attribute to the second item
order.@allids = order.item.@id;           // construct an attribute list containing all the ids in this order
```

If the *property-name* is the string representation of a number (i.e., an array index), the XML assignment operator replaces an existing property or appends a new property to an XML value according to the property's ordinal position within the XML value (i.e., its numeric property name). If a property already exists at the given location, the assignment operator replaces it, otherwise it appends a new property to the end of the *parent*. If the *AssignmentExpression* evalutes to an XML value, the assignment operator replaces the value of the property at the given position with a deep copy of the given XML value. If the *AssignmentExpression* evalutes to an XMLList, the assignment operator replaces the value of the property at the given position with a deep copy of each item in the XMLList in order, effectively deleting the original property and inserting the contents of the XMLList in its place. If the *AssignmentExpression* does not evaluate to a value of type XML or XMLList, the assignment operator calls ToString on the given value and replaces the property at the given position with the result. For example,

```
// replace the first child of the order element with an XML value
order[0] = <customer>
        <name>Fred</name>
        <address> … </address>
</customer>;

// replace the second child of the order element with a list of items
order[1] = <item> item one </item>
        +  <item> item two </item>
        +  <item> item three </item>;

// replace the third child or the order with a text node
order[2] = "A text node";

// append a new item to the end of the order
order[order.length] = <item> new item </item>;
```

If the *property-name* does not begin with "@" and is not the string representation of a number, the XML assignment operator replaces, modifies or appends one or more XML values in the *parent* by XML name. If only one XML valued property exists with the given name and the *AssignmentExpression* evalutes to an XML value or XMLList, the assignment operator replaces the identified XML value with the given value. If there are no XML properties with the given name, a new XML property with the given name and value is appended to the end of the *parent*. If more than one XML valued property exists with the given name and the *AssignmentExpression* evaluates to an XML value or XMLList, the assignment operator replaces the first XML property with a matching name with the given value and deletes the remaining XML properties with the given name, essentially replacing all the XML valued properties with the given name with the given value. If the *AssignmentExpression* does not evaluate to a XML value or XMLList, the assignment operator calls ToString on the given value and replaces the properties (i.e., the content) of the appropriate XML value (as opposed to replacing the XML value itself). This provides a simple, intuitive syntax for setting the value of a named XML property to a primitive value. For example,

Todo: These are XMLList examples and shouldn't be presented as XML.

```
order.customer.name = "Fred Jones";          // change the customer's name
order.item[1].price = 99.95;                 // change the price of the second item

// replace the employee with id=1, with a new employee with id=3
emps.employee.(@id == 1) = <employee id="3"><name>Fred</name></employee>;

// append some hobbies to the new employee using an XMLList
emps.employee.(@id == 3).hobby = <hobby>skiing</hobby>
                               +     <hobby>kayaking</hobby>
                               +     <hobby>piano</hobby>;

// replace all the employee's hobbies with their new favorite pastime
emps.employee.(@id == 3).hobby = "working";

// replace the employee with an open requisition
emps.employee.(@id == 3) = <requisition id="23" status="open"/>
```

Semantics

E4X extends the semantics of the assignment operator by providing more elaborate [[Put]] methods used when *MemberExpression* evaluates to a value of type XML or XMLList (see sections 6.1.1.2 and 6.2.1.2 respectively).

9.5.2 XMLList Assignment Operator

Syntax

Todo: restructure into another section and make non-normative

E4X reuses the ECMAScript assignment operator to replace or append values to XMLLists and their associated XML values. The ECMAScript syntax for the assignment operator is described by the following production:

AssignmentExpression :

LeftHandSideExpression = *AssignmentExpression*

Overview

The assignment operator begins by evaluating the *LeftHandSideExpression*, which resolves to a reference *r* consisting of a *base-object* and a *property-name*. If *parent* is an XMLList, the assignment operator performs the steps described in section (see section 9.5.1 for the steps performed with *parent* is an XML value).

Todo: stick with one terminology for base-object / parent objects.

Todo: Make sure prose and algorithm descriptions are consistent regarding ToXML vs. delegation of assignment to XML item.

Todo: Change "string representation of a number" to "an array index" throughout document. Defined in 15.4 of E3. Also look for more precise definition.

If the *property-name* is not the string representation of a number (i.e., not an array index), the assignment operator attempts to implicitly convert the XMLList to an XML value by calling the ToXML operator on this XMLList object. If the conversion succeeds (i.e., does not return **undefined**), the XMLList assignment operator invokes the XML assignment operator (see section 9.5.1) to assign the right hand *AssignmentExpression* to the resulting XML value. Otherwise, it throws an XMLException. This treatment intentionally blurs the distinction between a single XML value and an XMLList containing only one XML value. For example,

```
// set the name of the only customer in the order to Fred Jones
order.customer.name = "Fred Jones";

// attempt to set the sale date of the item. Throw an exception if more than 1 item exists.
order.item.saledate = "05-07-2002";

// replace the list of hobbies for the only customer in the order
order.customer.hobby = "shopping";
```

In the first statement above, the expression “order.customer” returns an XMLList containing only one XML item. The expression “order.customer.name” implicitly converts this XMLList to an XML value and assigns the value “Fred Jones” to that value.

If the *property-name* is the string representation of a number (i.e., an array index), the assignment operator replaces the property identified by *property-name* in the XMLList or appends a new property if none exists with that *property-name*. In addition, if the property identified is an XML value with a non-null *parent*, the XML value is also replaced in the context of its *parent*. If the *AssignmentExpression* evalutes to an XML value, the assignment operator replaces the value of the property identified by *property-name* with a deep copy of the given XML value. If the *AssignmentExpression* evalutes to an XMLList, the assignment operator replaces the value of the property identified by *property-name* with a deep copy of each item in the XMLList in order, effectively deleting the original property and inserting the contents of the XMLList in its place. If the *AssignmentExpression* does not evaluate to a value of type XML or XMLList, the assignment operator calls ToString on the given value and replaces the property at the given position with the result. For example,

```
// replace the first employee with George
e.employee[0] = <employee><name>George</name><age>27</age></employee>;

// add a new employee to the end of the employee list
e.employee[e.employee.length] = <employee><name>Frank</name></employee>;
```

Semantics

E4X extends the semantics of the assignment operator by providing more elaborate [[Put]] methods used when *MemberExpression* evaluates to a value of type XML or XMLList (see sections 6.1.1.2 and 6.2.1.2 respectively).

9.5.3 Compound Assignment (op=)

Syntax

Todo: restructure into another section and make non-normative.

E4X benefits from the compound assignment operator “+=” without requiring additional ECMAScript extensions. The syntax of the compound assignment “+=” is described by the following production:

AssignmentExpression :

LeftHandSideExpression += *AssignmentExpression*

Overview

When the left operand is an XML value, the “+=” operator has the effect of inserting one or more XML elements specified by the right operand just after the ordinal position of the left operand within its parent. For example,

```
var e = <employees>
          <employee id=”1”><name>Joe</name><age>20</age></employee>
          <employee id=”2”><name>Sue</name><age>30</age></employee>
      </employees>;

// insert employee 3 and 4 after the first employee
e.employee[0] += <employee id=”3”><name>Fred</name></employee> +
                        <employee id=”4”><name>Carol</name></employee>;
```

Following the expressions above, the variable “e” would contain the XML value:

```
<employees>
          <employee id=”1”><name>Joe</name><age>20</age></employee>
          <employee id=”3”><name>Fred</name></employee>
          <employee id=”4”><name>Carol</name></employee>
          <employee id=”2”><name>Sue</name><age>30</age></employee>
</employees>;
```

When the left operand is an XMLList, the “+=” operator has the effect of appending one or more values specified by the right operand to the XMLList. If the last last item in the XMLList is an XML value with a non-null *parent*, the “+=” operator also appends the items to the XML value refered to by *parent* just after the position of the last item in the XMLList. For example,

```
var e = <employees>
          <employee id=”1”><name>Joe</name><age>20</age></employee>
          <employee id=”2”><name>Sue</name><age>30</age></employee>
      </employees>;

// append employees 3 and 4 to the end of the employee list
e.employee += <employee id=”3”><name>Fred</name></employee> +
                      <employee id=”4”><name>Carol</name></employee>;
```

Following the expressions above, the variable “e” would contain the XML value:

```
<employees>
          <employee id=”1”><name>Joe</name><age>20</age></employee>
          <employee id=”2”><name>Sue</name><age>30</age></employee>
          <employee id=”3”><name>Fred</name></employee>
          <employee id=”4”><name>Carol</name></employee>
</employees>;
```

Semantics

E4X extends the semantics of the compound assignment operator by providing more elaborate [[Get]] and [[Put]] methods used when *MemberExpression* evaluates to a value of type XML or XMLList (see sections 6.1.1.1, 6.1.1.2, 6.2.1.1 and 6.2.1.2 respectively).

To do: add description and example of how to accomplish numeric addition with leaf nodes returned as strings.

10 Statements

E4X extends the statements provided in ECMAScript with the following production:

Statement :

NamespaceStatement

NamespaceUseStatement

10.1 Use Namespace Statement

Syntax

E4X reuses the use namespace statement defined in the ECMAScript Edition 4 draft for bringing specific XML namespaces in scope. The syntax of the use namespace statement is specified by the following production:

UseNamespace :

use namespace (*Expression*)

Overview

The use namespace statement adds a namespace specified by *Expression* to the current list of default namespaces. Expressions may reference any name in the default namespaces using the unqualified name syntax as long as there is no more than one entry in the default namespaces with the given name.

Semantics

Given an *Expression*, the namespace statement executes the following steps:

1. Let *e* = evaluate(*Expression*)
2. if *e*.[[*Class*]] != Namespace, throw TypeException
3. Let *defaultNS* = get the [[defaultNamespaces]] property from the top of the scope chain
4. *defaultNS* = *defaultNS* ∪ *e*

todo: replace the "scope chain" verbage with some prose description of context from E4

todo: local vars are treated as copies by value. Rewrite "let" semantics above.

10.2 The for-in Statement

Syntax

E4X reuses the ECMAScript for-in statement for iterating through the XML properties of an XMLList. The syntax of the for-in statement is specified by the following production:

IterationStatement :

for (*LeftHandSideExpression* in *Expression*) *Statement*

Overview

When the value of *Expression* evaluates to an XMLList, the for-in statement iterates through each XML property in the list. For each XML property, the for-in operator assigns the XML property to the variable identified by *LeftHandSideExpression* and evaluates the *Statement*. For example:

```
// print all the employee names
for (var n in e..name) {
        print (“Employee name: “ + n);
}
```

In this example, the expression “e..name” returns an XMLList containing all of the descendant XML properties of the XML value “e” with the name “name”. The for-in statement iterates through the list in order. For each XML property in the list, it assigns the variable “n” to the XML property and executes the code nested in curly braces.

Note: The for-in operation behaves differently for XMLLists than it does for native ECMAScript arrays. With native ECMAScript arrays, for-in assigns the loop variable over the domain of the array. However with XMLLists, for-in assigns the loop variable over the range of the array.

Semantics

Todo: describe semantics for mutation. See E3, but semantics are different because operations can insert. Waldemar recommends making semantics undefined when inserting items into a list.

Todo: modify E3 for in definition to check type and dispatch accordingly

The production *IterationStatement* : for (*LeftHandSideExpression* in *Expression*) *Statement* is evaluated as follows:

1. Let *e* = Evalute(*Expression*)
2. Let *l* = Call GetValue(*e*)
3. Let *V* = **empty**
4. for j = 0 to *l*.length
    a. Let *i* = Evaluate(*LeftHandSideExpression*)
    b. Call PutValue(*i*, *l*[*j*].[[*Value*]])
    c. Let *s* = Evaluate(*Statement*)
    d. If *s*.value is not **empty**, let *V* = *s*.value

e. If *s*.type is **break** and *s*.target is in the current label set, return (**normal**, *V*, **empty**)

f. If (*s*.type != **continue**) or (*s*.target is not in the current label set)

i. If *s* is an abrupt completion, return *s*

5. Return (**normal**, *V*, **empty**)

11 Native E4X Objects

E4X adds two native objects to ECMAScript, the native XML object and the native XMLList object. In addition, E4X adds new properties to the global object.

11.1 The Global Object

11.1.1 Function Properties of the Global Object

E4X extends ECMAScript by adding the following function properties to the global object.

11.1.1.1 isXMLName (string)

Overview

The isXMLName function examines the given *string* and determines whether it is a valid XML name that can be used as an XML element or attribute name. If so, it returns true, otherwise it returns false.

Semantics

When the isXMLName function is called with one parameter *string*, the following steps are taken:

1. If the first character of *string* ∉ {*Letter*[1], '_', ':'}, return false;
2. for *i* = 1 to *string*.length
   a. if the *i*th character of *string* ∉ {*Letter*[1] , *Digit*[1], '.', '-', '_', ':', *CombiningChar*[1], *Extender*[1]}, return false
3. return true

[1] The productions *Letter*, *Digit*, *CombiningChar* and *Extender* are defined in the XML 1.0 specification.

11.1.2 Constructor Properties of the Global Object

E4X extends ECMAScript by adding the following constructor properties to ECMAScript.

11.1.2.1 XML (. . .)

See sections 11.2.1 and 11.2.2.

11.1.2.2 XMLList (. . .)

See section 11.3.1 and 11.3.2.

11.2 XML Objects

11.2.1 The XML Contructor Called as a Function

Syntax

XML (*value*)

Overview

When XML is called as a function rather than as a constructor, it performs a type conversion.

Semantics

When the XML function is called with one argument *value*, the following step is taken.

1. Return ToXML(*value*)

11.2.2 The XML Constructor

Syntax

new XML (*value*)

Overview

When XML is called as part of a new expression, it is a constructor and may create a new XML object

Semantics

When the XML constructor is called with a single parameter *value*, the following steps are taken:

1. Let *x* = ToXML(*value*)
2. If *value*.[[*Class*]] = String, return *x*
3. If *value* is a W3C DOM Element, return *x*
4. Return a deep copy of *x*

11.2.3 Properties of the XML Object

In addition to the internal properties of the XML object, the XML object has the following property

11.2.3.1 settings

The global XML object has a built-in property of type XML named “settings” used to customize the behavior of all XML objects. The initial value of the global XML settings property is:

```
<settings>
        <ignoreComments> true </ignoreComments>
        <ignorePIs> true </ignorePIs>
        <ignoreWhitespace> true </ignoreWhitespace>
        <prettyPrint> true </prettyPrint>
        <prettyIndent> 2 </prettyIndent>
</settings>
```

The settings property of the global XML object can be modified like any other XML value. For example,

```
// setup XML representation and serialization preferences
XML.settings.ignoreComments = false;
XML.settings.ignorePIs = false;
XML.settings.prettyIndent = 5;
```

11.2.4 XML Built-in Methods

Each value of type XML has a set of built-in methods available for performing common operations. These built-in methods are described in the following sections.

11.2.4.1 appendChild (child)

Overview

The appendChild method appends a deep copy of the given *child* to the end of this XML object's properties and returns a reference to this XML object. For example,

```
var e = <employees>
        <employee id="0" ><name>Jim</name><age>25</age></employee>
        <employee id="1" ><name>Joe</name><age>20</age></employee>
    </employees>;

// Add a new child element to the end of Jim's employee element
e.employee.(name == "Jim").appendChild(<hobby>snorkeling</hobby>);
```

Semantics

When the appendChild method of an XML object *x* is called with one parameter *child*, the following steps are taken:

1. Call [[Put]] with *x* as the **this** object and parameters *x*.[[*Length*]] and *child*
2. Return *x*

11.2.4.2 attribute (attributeName)

Overview

The attribute method returns an XMLList containing zero or one XML attributes associated this XML object that have the given *attributeName*. For example,

```
// get the id of the employee named Jim
e.employee.(name == “Jim”).attribute(“id”);
```

Semantics

When the attribute method of an XML object *x* is called with a parameter *attributeName*, the following steps are taken:

1. Let *p* = “@” + ToString(*attributeName*)
2. Call [[Get]] with *x* as the **this** object and parameter *p*
3. Return Result(2).

11.2.4.3 attributes ( )

The attributes method returns an XMLList containing the XML attributes of this object. For example,

```
// print the attributes of an XML value
function printAttributes(x) {
        for (var a in x.attributes()) {
                print(“The attribute named “ + a.name() + “ has the value “ + a);
        }
}
```

Semantics

When the attributes method of an XML object *x* is called, the following steps are taken:

1. Call [[Get]] on this object with parameter “@*”
2. return Result(1)

11.2.4.4 child (propertyName)

Overview

The child method calls the internal XML [[Get]] method on this object passing the parameter *propertyName*, then returns the results.

11.2.4.5 childIndex ( )

Overview

The childIndex method returns a Number representing the ordinal position of this XML object within the context of its parent. For example,

```
// Get the ordinal index of the employee named Joe.
var joeindex = e.employee.(name == “Joe”).childIndex();
```

Semantics

When the childIndex method of an XML object *x* is called, it performs the following steps:

1. Let *parent* = *x*.[[*Parent*]]
2. If *parent* == Null, return **NaN**
3. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*
4. Return ToNumber(*q*.[[*Name*]])

11.2.4.6 children ( )

Overview

The children method returns an XMLList containing all the properties of this XML object. For example,

```
// Get child elements of first employee: returns an XMLList containing:
// <name>Jim</name>, <age>25</age> and <hobby>Snorkeling</hobby>
var emps = e.employee[0].children();
```

Semantics

When the children method of an XML object *x* is called, it performs the following steps:

1. Call [[Get]] on this object with parameter “*”
2. Return Result(1)

11.2.4.7 comment ( )

Overview

The comment method returns an XMLList containing the properties of this XML object that represent XML comments.

Semantics

When the comment method of an XML object *x* is called, it performs the following steps:

1. Let *l* = a new XMLList
2. for *i* = 0 to *x*.[[*Length*]]

a. if *x*[*i*].[[*Class*]] == XMLComment, Call [[*Append*]] with *l* as the **this** object and parameter *x*[*i*]

3. Return *l*

11.2.4.8 copy ( )

Overview

The copy method returns a deep copy of this XML object with the internal [[Parent]] property set to Null.

11.2.4.9 descendants ([name])

Overview

The descendants method returns all the XML valued descendants (children, grandchildren, great-grandchildren, etc.) of this XML object with the given *name*. If the *name* parameter is omitted, it returns all descendants of this XML object.

Semantics

When the descendants method on an XML object *x* with the optional parameter *name*, the following steps are taken:

1. If *name* == undefined, *name* = "*"
2. Call [[Descendants]] with *x* as the **this** object and parameter *name*
3. Return Result(2)

11.2.4.10 domNode( )

Overview

The domNode method returns a W3C DOM Node representation of this XML Object.

11.2.4.11 domNodeList( )

Overview

The domNodeList method returns a W3C DOM NodeList containing a single W3C DOM Node representation of this XML Object.

11.2.4.12 insertAfter (child)

Overview

The insertAfter method inserts the given *child* after this XML object in the context of this XML object's parent. If the parent of this XML object is Null, insertAfter performs no action.

Semantics

When the insertAfter method is called on an XML object *x* with parameter *child*, the following steps are taken:

1. Let *parent* = *x*.[[*Parent*]]
2. if *parent* == Null, return
3. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*
4. *q* = ToString(ToUint32(*q*) + 1)
5. call [[Insert]] with *parent* as the **this** object and parameters *q* and *child*

11.2.4.13 insertBefore (child)

Overview

The insertBefore method inserts the given *child* before this XML object in the context of this XML object's parent. If the parent of this XML object is Null, insertBefore performs no action.

Semantics

When the insertBefore method is called on an XML object *x* with parameter *child*, the following steps are taken:

1. Let *parent* = *x*.[[*Parent*]]
2. if *parent* == Null, return
3. Let *q* = the property of *parent*, where *parent*[*q*] is the same object as *x*
4. call [[Insert]] with *parent* as the **this** object and parameters *q* and *child*

11.2.4.14 isComment ( )

Overview

The isComment method returns true if this XML object represents an XML comment. Otherwise, it returns false.

Semantics

When the isComment method is called on an XML Object *x*, the following step is taken:

1. Return *x*.[[*Class*]] == XMLComment

11.2.4.15 isProcessingInstruction ([name])

Overview

When the isProcessingInstruction method is called with a single *name* parameter, it returns true if this XML object represents an XML processing instruction with the given *name* and false otherwise. When the isProcessingInstruction method is called with no parameters, it returns true if this XML object represents an XML processing instruction regardless of its name.

Semantics

When the isProcessingInstruction method is called on an XML object *x* with optional parameter *name*, the following steps are taken:

1. if (*x*.[[*Class*]] != XMLPI) return false
2. if *name* == undefined or *name* == *x*.[[*Name*]], return true

11.2.4.16 isText ( )

Overview

The isText method returns true if this XML object represents an XML text node. Otherwise, it returns false.

Semantics

When the isText method is called on an XML object *x*, the following step is taken:

1. Return *x*.[[*Class*]] == XMLText

11.2.4.17 length ( )

Overview

The length method returns the number of properties in this XML object. For example,

```
// print each child element of the first employee element stored in e
for (var i = 0; i < e.employee[0].length(); i++) {
        print("Child element:" + e.employee[0][i]);
}
```

Semantics

When the length method is called on an XML object *x*, the following step is taken:

1. Return *x*.[[*Length*]]

11.2.4.18 name ( )

Overview

The name method returns the name associated with this XML object.

Semantics

When the name method is called on an XML object *x*, the following step is taken:

1. Return *x*[[*Name*]]

11.2.4.19 normalize ( )

Overview

The normalize method puts all text nodes in this and all descendant XML objects into a normal form by merging adjacent text nodes and eliminating empty text nodes.

Semantics

When the normalize method is called on an XML object *x*, the following steps are taken:

1. Let *i* = 0
2. while *i* < *x*.[[*Length*]]
    a. if *x*[*i*].[[*Class*]] == XML
        i. call normalize with *x*[*i*] as the **this** object
        ii. *i*++
    b. else if *x*[*i*].[[*Class*]] == XMLText
        i. while ((*i*+1) < *x*.[[*Length*]]) and (*x*[*i* + 1].[[*Class*]] == XMLText)
            1. *x*[*i*][0] += *x*[*i* + 1][0]
            2. call remove with *x*[*i* + 1] as the **this** object
        ii. if *x*[*i*][0].length == 0
            1. call remove with *x*[*i*] as the **this** object
        iii. else
            1. *i*++
    c. else
        i. *i*++

11.2.4.20 parent ( )

Overview

The parent method returns the parent of this XML object. For example,

```
// Get the parent element of the second name in “e”. Returns <employee id=”1” …
var firstNameParent = e..name[1].parent()
```

Semantics

When the parent method is called on an XML object *x*, the following step is taken:

1. Return *x*.[[*Parent*]]

11.2.4.21 processingInstruction ([name])

Overview

When the processingInstruction method is called with one parameter *name*, it returns an XMLList containing all the children of this XML object that are processing-instructions with the given *name*. When the processingInstruction method is called with no parameters, it returns an XMLList containing all the children of this XML object that are processing-instructions regardless of their name.

Semantics

When the processingInstruction method is called on an XML object *x* with optional parameter *name*, the following steps are taken:

1. Let *l* = a new XMLList
2. for *i* = 0 to *x*.[[*Length*]]
    a. if *x*[*i*].[[*Class*]] == XMLPI
        i. if *name* == undefined or *name* == *x*[*i*].[[*Name*]]
            1. Call [[*Append*]] with *l* as the **this** object and parameter *x*[*i*]
3. Return *l*

11.2.4.22 prependChild (child)

Overview

The prependChild method inserts a deep copy of the given *child* into this object prior to its existing XML properties. It returns a reference to this XML object. For example,

```
// Add a new child element to the front of John's employee element
e.employee.(name == "John").prependChild(<prefix>Mr.</prefix>);
```

Semantics

When the prependChild method is called on an XML object *x* with parameter *child*, the following steps are taken:

1. Call [[Insert]] on this object with the parameters "0" and *child*
2. return *x*

11.2.4.23 setInnerXML (childList)

Overview

The setInnerXML method replaces the XML properties of this XML object with a new set of XML properties from *childList*. *childList* may be a single XML value or an XMLList. setInnerXML returns a reference to this XML value. For example,

```
// Replace the entire contents of Jim's employee element
e.employee.(name == "Jim").setInnerXML(<name>John</name> + <age>35</age>);
```

Semantics

When the setInnerXML method is called on an XML object *x* with parameter *childList*, the following steps are taken:

1. Call [[Put]] with *x* as the **this** object and parameters "*" and *childList*
2. return *x*

11.2.4.24 text ( )

Overview

The text method returns an XMLList containing all XML properties of this XML object that represent XML text nodes.

Semantics

When the text method of an XML object *x* is called, the following steps are taken:

4. Let *l* = a new XMLList
5. for *i* = 0 to *x*.[[*Length*]]
    a. if *x*[*i*].[[*Class*]] == XMLText, Call [[*Append*]] with *l* as the **this** object and parameter *x*[*i*]
6. Return *l*

11.2.4.25 toString ( )

Overview

The toString method returns a string representation of this XML object per the ToString conversion operator described in section 7.1.

Semantics

When the toString method of an XML object *x* is called, the following step is taken:

1. Return ToString(*x*)

11.2.4.26 toXMLString ( )

Overview

The toXMLString() method returns an XML encoded string representation of this XML object per the ToXMLString conversion operator described in section 7.2. Unlike the toString method, toXMLString provides no special treatment for XML values that contain only XML text nodes (i.e., primitive values). The toXMLString method always includes the start tag, attributes and end tag of the XML value regardless of its content. It is provided for cases when the default XML to string conversion rules are not desired. For example,

```
var item = <item>
        <description>Laptop Computer</description>
        <price>2799.95</price>
        <quantity>1</quantity>
</item>;

// returns “Description stored as <description>Laptop Computer</description”
var logmsg = “Description stored as “ + item.description.toXMLString();

// returns “Thank you for purchasing a Laptop Computer!” (with tags removed)
var message = “Thank you for purchasing a “ + item.description + “!”;
```

Semantics

When the toXMLString method of an XML object *x* is called, the following step is taken:

1. Return ToXMLString(*x*)

11.2.4.27 validate (schemaURI, [typeName])

<TBD>

11.2.4.28 xpath (XPathExpression)

Overview

The xpath method evaluates the XPathExpression using this XML object as the context node in accordance with the W3C XPath standard. It returns the results of the XPathExpression as an XMList. For example,

```
// Use an xpath expression to get the employee named John
var jim = e.xpath("//employee[name='John']")
```

11.3 XMLList Objects

11.3.1 The XMLList Constructor Called as a Function

When XMLList is called as a function rather than as a constructor, it creates and initializes a new XMLList object. Thus, the function call **XMLList (…)** is equivalent to the object creation expression **new XMLList(…)** with the same arguments.

11.3.2 The XMLList Constructor

Syntax

new XMLList ([value])

Overview

When XMLList is called as part of a new expressions, it is a constructor and creates a new XMLList object. When the XMLList constructor is called with no arguments, it returns an empty XMLList. When the XMLList is called with a *value* of type XMLList, the XMLList constructor returns a shallow copy of the *value*. When the XMLList constructor is called with a non-XMLList value, it returns a new XMLList containing the given *value* as property "0".

Semantics

When the XMLList constructor is called with an optional parameter *value*, the following steps are taken:

1. Let *l* = a new XMLList object
2. if *value*.[[*Class*]] == XMLList
    a. for *i* = 0 to *value*.[[*Length*]]
        i. Call [[Append]] with *l* as the **this** object and parameter *value*[*i*]
3. else if *value* != undefined
    a. Call [[Append]] with *l* as the **this** object and parameter *value*
4. Return *l*

11.3.3 XMLList Built-in Methods

Each value of type XMLList has a set of built-in methods available for performing common operations. These built-in methods are described in the following sections.

11.3.3.1 appendChild (child)

Overview

The appendChild method calls the ToXML operator on this XMLList object. If the conversion succeeds, it calls appendChild on the resulting value passing *child* as a parameter and returns the result. Otherwise, the ToXML operator throws an XMLException.

Semantics

When the appendChild method of an XMLList *l* is called with a parameter *child*, the following steps are taken:

1. Let *x* = ToXML(*l*)
2. Call appendChild with *x* as the **this** object and parameter *child*.
3. Return Result(2)

11.3.3.2 attribute (attributeName)

Overview

The attribute method calls the attribute() method on each XML valued property in this XMLList object passing attributeName as a parameter and returns an XMLList containing the results in order.

Semantics

When the attribute method is called on an XMLList object *l* with parameter *attributeName*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    a. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call attribute with *l*[*i*] as the **this** object and parameter *attributeName*
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.3 attributes ( )

Overview

The attributes method calls the attributes() method on each XML valued property in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the attributes method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    a. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call attributes with *l*[*i*] as the **this** object

ii. Call [[Append]] with *m* as the **this** object and parameter *r*

3. return *m*

11.3.3.4 child (propertyName)

Overview

The child method calls the child() method on each XML valued property in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the child method is called on an XMLList object *l* with parameter *propertyName*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    b. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call child with *l*[*i*] as the **this** object and parameter *propertyName*
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.5 childIndex ( )

Overview

The childIndex method calls the childIndex() method on each XML valued property in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the childIndex method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    c. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call childIndex with *l*[*i*] as the **this** object
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.6 children ( )

Overview

The children method calls the children() method on each XML valued property in this XMLList object and returns an XMLList containing the results concatenated in order. For example,

```
// get all the children of all the items in the order
var allitemchildren = order.item.children();

// get all grandchildren of the order that have the name price
var  grandChildren = order.children().price;
```

Semantics

When the children method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    d. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call children with *l*[*i*] as the **this** object
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.7 comment ( )

Overview

The comment method calls the comment method on each XML valued propery in this XMLList object and returns an XMLList containing the results concatenated in order.

Semantics

When the comment method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    e. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call comment with *l*[*i*] as the **this** object
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.8 copy ( )

Overview

The copy method returns a deep copy of this XMLList object.

11.3.3.9 descendants ([name])

The descendants method calls the descendants method on each XML valued property in this XMLList object with the optional parameter *name* (or undefined if *name* is omitted) and returns an XMLList containing the results concatenated in order.

Semantics

When the descendants method is called on an XMLList object *l* with optional parameter *name*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    f. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call descendants with *l*[*i*] as the **this** object and parameter *name*
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.10 domNode ( )

Overview

The domNode method calls the ToXML operator on this XMLList object. If the conversion succeeds, it calls domNode on the resulting value and returns the result. Otherwise, the ToXML operator throws an XMLException.

Semantics

When the domNode method of an XMLList *l* is called, the following steps are taken:

1. Let *x* = ToXML(*l*)
2. Call domNode with *x* as the **this** object
3. Return Result(2)

11.3.3.11 domNodeList( )

Overview

The domNodeList method returns a W3C DOM NodeList representation of this XMLList Object.

11.3.3.12 insertAfter (child)

Overview

The insertAfter method calls the ToXML operator on this XMLList object. If the conversion succeeds, it calls insertAfter on the resulting value passing *child* as a parameter. Otherwise, the ToXML operator throws an XMLException.

Semantics

When the insertAfter method of an XMLList *l* is called with a parameter *child*, the following steps are taken:

1. Let *x* = ToXML(*l*)
2. Call insertAfter with *x* as the **this** object and parameter *child*.
3. Return

11.3.3.13 insertBefore (child)

Overview

The insertBefore method calls the ToXML operator on this XMLList object. If the conversion succeeds, it calls insertBefore on the resulting value passing *child* as a parameter. Otherwise, the ToXML operator throws an XMLException.

Semantics

When the insertBefore method of an XMLList *l* is called with a parameter *child*, the following steps are taken:

1. Let *x* = ToXML(*l*)
2. Call insertBefore with *x* as the **this** object and parameter *child*.
3. Return

11.3.3.14 isComment ( )

Overview

If this XMLList object contains only a single XML valued property, the isComment method calls isComment on the single XML valued property and returns the result. Otherwise, the isComment method returns false.

Semantics

When the isComment method of an XMLList *l* is called, the following steps are taken:

1. if *l*.[[*Length*]] = 1 and *l*[0].[[*Class*]] == XML
    a. Call isComment with *l*[0] as the **this** object and return the result

2. return false

11.3.3.15 isProcessingInstruction ([name])

Overview

If this XMLList object contains only a single XML valued property, the isProcessingInstruction method calls isProcessingInstruction on the single XML valued property passing the parameter *name* (or undefined if *name* is omitted) and returns the result. Otherwise, the isPI method returns false.

Semantics

When the isProcessingInstruction method of an XMLList *l* is called with an optional parameter *name*, the following steps are taken:

1. if *l*.[[*Length*]] = 1 and *l*[0].[[*Class*]] == XML
    a. Call isProcessingInstruction with *l*[0] as the **this** object and parameter *name* return the result
2. return false

11.3.3.16 isText ( )

Overview

If this XMLList object contains only a single XML valued property, the isText method calls isText on the single XML valued property and returns the result. Otherwise, the isText method returns false.

Semantics

When the isText method of an XMLList *l* is called, the following steps are taken:

3. if *l*.[[*Length*]] = 1 and *l*[0].[[*Class*]] == XML
    a. Call isText with *l*[0] as the **this** object and return the result
4. return false

11.3.3.17 length ( )

Overview

The length method returns the number of properties in this XMLList object. For example,

```
for (var i = 0; i < e..name.length(); i++) {
        print("Employee name:" + e..name[i]);
}
```

Semantics

When the length method of an XMLList object *l* is called, the following step is taken:

1. Return *l*.[[*Length*]]

11.3.3.18 name ( )

Overview

The name method calls the name method on each XML valued property in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the name method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    g. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call name with *l*[*i*] as the **this** object
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.19 normalize ( )

Overview

The normalize method calls the normalize method on each XML valued property in this XMLList.

Semantics

When the normalize method is called on an XMLList object *l*, the following steps are taken:

1. for *i* = 0 to *l*.[[*Length*]]
    h. If *l*[*i*].[[*Class*]] == XML, Call normalize with *l*[*i*] as the **this** object

11.3.3.20 parent ( )

Overview

The parent method calls the parent method on each XML valued property in this XMLList object and returns an XMLList containing the results in order.

Semantics

When the parent method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    i. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call parent with *l*[*i*] as the **this** object
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.21 processingInstruction ([name])

Overview

The processingInstruction method calls the processingInstruction method on each XML valued property in this XMLList object passing the optional parameter *name* (or undefined if it is omitted) and returns an XMList containing the results in order.

Semantics

When the processingInstruction method is called on an XMLList object *l* with optional parameter *name*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    j. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call processingInstruction with *l*[*i*] as the **this** object and parameter *name*
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

11.3.3.22 prependChild (child)

Overview

The prependChild method calls the ToXML operator on this XMLList object. If the conversion succeeds, it calls prependChild on the resulting value passing *child* as a parameter and returns the result. Otherwise, the ToXML operator throws an XMLException.

Semantics

When the prependChild method of an XMLList *l* is called with a parameter *child*, the following steps are taken:

1. Let *x* = ToXML(*l*)
2. Call prependChild with *x* as the **this** object and parameter *child*.
3. Return Result(2)

11.3.3.23 setInnerXML (childList)

Overview

The setInnerXML method calls the ToXML operator on this XMLList object. If the conversion succeeds, it calls setInnerXML on the resulting XML value passing *childList* as a parameter and returns the result. Otherwise, the ToXML operator throws an XMLException. *childList* may be a single XML value or an XMLList.

Semantics

When the setInnerXML method of an XMLList *l* is called with a parameter *childList*, the following steps are taken:

1. Let *x* = ToXML(*l*)
2. Call setInnerXML with *x* as the **this** object and parameter *childList*.
3. Return Result(2)

11.3.3.24 text ( )

Overview

The text method calls the text method on each XML valued property contained in this XMLList object and returns an XMLList containing the results concatenated in order.

Semantics

When the text method is called on an XMLList object *l*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    k. If *l*[*i*].[[*Class*]] == XML

i. Let *r* = Call text with *l*[*i*] as the **this** object

ii. Call [[Append]] with *m* as the **this** object and parameter *r*

3. return *m*

11.3.3.25 toString ( )

Overview

The toString method returns a string representation of this XMLList object per the ToString conversion operator described in section 7.1.

Semantics

When the toString() method of an XMLList object *l* is called, the following step is taken:

1. Return ToString(*l*)

11.3.3.26 toXMLString ( )

Overview

The toXMLString() method returns an XML encoded string representation of this XMLList object per the ToXMLString conversion operator described in section 7.2. Unlike the toString method, toXMLString provides no special treatment for XML values that contain only XML text nodes (i.e., primitive values). The toXMLString method always calls toXMLString on each property contained within this XMLList object, concatenates the results in order and returns a single string.

Semantics

When the toXMLString() method of an XMLList object *l* is called, the following step is taken

1. Return toXMLString(*l*)

11.3.3.27 validate (schemaURI, [typeName])

<TBD>

11.3.3.28 xpath (XPathExpression)

Overview

The xpath method evaluates the XPathExpression for each XML property contained in this XMLList object and concatenates the results an XMLList containing the results concatenated in order.

Semantics

When the xpath method is called on an XMLList object *l* with parameter *XPathExpression*, the following steps are taken:

1. Let *m* = a new XMLList
2. for *i* = 0 to *l*.[[*Length*]]
    1. If *l*[*i*].[[*Class*]] == XML
        i. Let *r* = Call xpath with *l*[*i*] as the **this** object and parameter *XPathExpression*
        ii. Call [[Append]] with *m* as the **this** object and parameter *r*
3. return *m*

Pri 2: consider adding ECMAScript array methods to XMLLists

To do: validate the need for a method to check for well-formedness no longer exists

12 Resolved Issues

Issue #12: Should an empty element be serialized as an empty string or an empty element?

Resolution: The group agreed that empty elements should be serialized as an empty string. The appropriate semantics are reflected in ToString().

Free printed copies can be ordered from:
ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Email: documents@ecma.ch

Files of this Standard can be freely downloaded from the ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.