



XPath 2.0 and XSLT 2.0

Norman Walsh

XML Standards Architect

XML Conference & Exposition 2003

07-12 December 2003

<http://www.sun.com/>



Version 1.1

Table of Contents

Introduction
Speaker Qualifications
Background Material
XPath 2.0
XSLT 2.0
Closing Thoughts

Introduction

- This is a *short* tutorial of XPath 2.0 and XSLT 2.0
- Describe and demonstrate new features
- Aim for breadth over depth, but feel free to ask questions
- Assume some familiarity with XPath 1.0 and XSLT 1.0

Speaker Qualifications

- Elected member of the *W3C Technical Architecture Group*; co-chair of the *XML Core Working Group*; member of the *XSL WG*. Joint editor of several *XSL/XML Query Specs*.
- Chair of the *OASIS DocBook Technical Committee*, member of the *Entity Resolution TC* and the *RELAX NG TC*.
- Co-Spec Lead for *JSR 206: Java API for XML Processing*
- Member of the *TEI Meta Working Group*

Background Material

Specifications

Fitting the Pieces Together

Data Model

Functions and Operators

Language Semantics

Language Semantics (Continued)

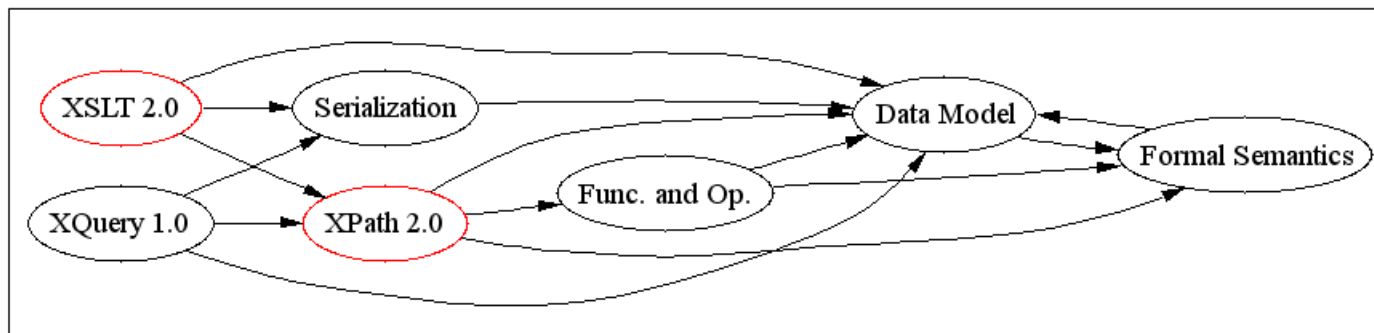
Specifications

Seven core specifications, most in Last Call until 15 Feb 2004.

- XQuery 1.0 and XPath 2.0 Data Model
- XQuery 1.0 and XPath 2.0 Functions and Operators
- XQuery 1.0 and XPath 2.0 Formal Semantics
- XML Path Language (XPath) 2.0
- XQuery 1.0: An XML Query Language
- XSL Transformations (XSLT) Version 2.0
- XSLT 2.0 and XQuery 1.0 Serialization

Fitting the Pieces Together

The family of XSL and XML Query specifications are closely related. Many of the specifications depend on others.



XSL/XML Query Specifications

(This diagram is only illustrative, not complete or exhaustive.)

Data Model

- XPath 2.0 has nodes and typed values. Colloquially, we speak of three kinds of things: *nodes*, *simple* or *atomic* values, and *items*. An item is either a node *or* an atomic value.
- XPath 1.0 has node sets, XPath 2.0 has sequences.
 - Sequences can be in arbitrary order
 - Sequences can contain duplicates
 - Sequences can be heterogenous

Functions and Operators

Functions. Lots of functions.

- String and numeric functions
- Date and time functions
- Sequence manipulation functions
- Casting and type-related functions

Language Semantics

XPath 2.0 has both static and dynamic semantics:

- Static semantics define, informally, what a language means without reference to any particular input.
- Dynamic semantics, again informally, define how a language behaves presented with inputs of various sorts.

Language Semantics (Continued)

- The Formal Semantics specification describes the static semantics of XPath.
- The XPath 2.0 specification describes the dynamic semantics of XPath.
- The XSLT 2.0 specification describes all of the semantics of XSLT.

XPath 2.0

XML Schema Type System

XML Schema Type System (Continued)

Type Names and Type Matching

Atomization

New Types

New Duration Types

New Node Types

Element Tests (1)

Element Tests (2)

Element Tests (3)

Attribute Tests

Type Errors

...

XML Schema Type System

- Probably the most significant semantic change to XPath
- The XPath 1.0 type system is very simple: nodes, strings, numbers, and booleans.
- XPath 2.0 adds W3C XML Schema simple and complex types.
- XPath 2.0 has nodes and *atomic values*.
- Atomic values have simple types: xs:string, xs:integer, xs:dateTime, etc.

XML Schema Type System (Continued)

- Allows matching and selection of elements, attributes, and atomic values by type.
- Supports a set of primitive simple types.
- Implementations may support user-defined simple and complex types.
- Implementations may support additional, non-W3C XML Schema types.

Type Names and Type Matching

- Types are identified by name.
- Available type names are determined by schema import.
- Values are “atomized” before most comparisons.

Atomization

- Atomization transforms a sequence into a sequence of atomic values.
- For each item in the sequence:
 - If the item is an atomic value, use it.
 - Otherwise, use the typed value of the item.
- An error occurs if the item does not have a typed value.

New Types

`xdt:untypedAtomic`

The type name that identifies an atomic value with no known type.

`xdt:untypedAny`

The type name that identifies any value (simple or complex) with no known type.

`xdt:anyAtomicType`

The supertype of all atomic types.

New Duration Types

xdt:yearMonthDuration

A duration that consists of only years and months.

xdt:dayTimeDuration

A duration that consists of only days and times.

The duration types have the feature that they can be totally ordered. **xs:durations** are only partially ordered. (e.g., is one month and five days more or less than five weeks?)

New Node Types

There are several new “node” tests in addition to the familiar `text()`, `comment()`, etc.

- `item()` matches any node *or* any atomic value.
- `document-node()` matches a document node.
- `document-node(ElementTest)` matches a document with a document element that matches *ElementTest*.

Element Tests (1)

An **ElementTest** matches elements.

- `element()` matches any element.

The selection “`select="element()"`” is actually the same as “`select="*"`”. It selects all elements of any name or type.

Element Tests (2)

- `element(ElementName, TypeName)` matches an element named *ElementName* with the type *TypeName*. (Either can be `*`.)

“`element(*, doc:paraType)`” matches elements with any name that have the type `doc:paraType`.

“`element(*:p, doc:paraType)`” matches elements named “`p`” in any namespace that have the type `doc:paraType`.

Element Tests (3)

- `element(ElementName)` matches an element named *ElementName*. If there is a top-level element declaration for *ElementName*, then the type of the element must also match.

“`element(html:p)`” matches (some) HTML “p” elements.
(Compare with “`html:p`” which always matches all of them.)

Attribute Tests

An **AttributeTest** matches attributes.

It has the same forms as an **ElementTest**:

- **attribute()** matches any attribute.
- **attribute(*AttributeName*, *TypeName*)** matches an attribute by name and/or type.
- **attribute(*AttributeName*)** matches an attribute by name and top-level type.

Type Errors

- XPath 1.0 had almost no type errors: if an expression was syntactically valid, it returned a result. “"3" + 1” = 4, “"Tuesday" + 1” = NaN, etc.
- In XPath 2.0 this is not the case. Errors will terminate the evaluation of an expression, stylesheet, or query.
- XPath 2.0 adds new operators that allow you to test if an operation will succeed.

Sequences

- Sequence construction:
 - `(1 to 10)[. mod 2 = 1] =`
`(1, 3, 5, 7, 9)`
 - `($preamble, .//item)`
- Union, Intersection and Exception.
 - `(1, 2, 3, 4) union (1, 3)`
`= (1, 2, 3, 4)`
 - `(1, 2, 3, 4) intersect (1, 3) = (1, 3)`
 - `(1, 2, 3, 4) except (1, 3) = (2, 4)`
- Quantified expressions (**some** and **every**).

“For” Expressions

XPath 2.0 adds a “for” expression:

```
for $bib in blist/bibl
  where contains($bib/orgname, 'W3C')
  return $bib
```

N.B. This is *in XPath*. For example, it might appear in a **select** attribute.

XSLT 2.0 retains the **xsl:for-each** instruction.

“If” Expressions

XPath 2.0 also adds an “if” expression:

```
if ($part/@discounted)
  then $part/wholesale
  else $part/retail
```

Again, this is *in XPath* and might appear in a **select** attribute.

XSLT 2.0 retains the **xsl:if** and **xsl:choose** instructions.

instance of

The **instance of** operator tests if an item is an instance of a given type (or is derived by restriction from it):

`$div instance of eg:Chapter`

returns true if `$div` is a chapter.

`5 instance of xs:decimal`

returns true because 5 is an integer and integers are a restriction of decimals.

treat as

The **treat as** operator fools the static type checker.

Suppose, for example, that you have a function that operates on UK addresses. If the static type of **\$curAddr** isn't a UK address, you can still pass it to the function as follows:

```
$curAddr treat as element(*, eg:UKAddress)
```

A dynamic error will occur if the address isn't a UK address.

cast as

The **cast as** operator coerces the type of an item.

`$x cast as eg:HatSize`

returns the value of **`$x`** as an item of type **`eg:HatSize`**.

castable as

Attempts to make invalid casts, for example string to integer, will raise dynamic errors. The **castable as** operator allows you to test if the cast will succeed.

`$x castable as eg:HatSize`

returns true if the value of **`$x`** can be cast to **`eg:HatSize`**.

General Comparisons

General comparisons: =, !=, <, <=, >, >=

True if any pair of values satisfies the comparison. (XPath 1.0 semantics.)

`$book/author = "Kennedy"`

is true if **`$book`** has one or more authors and at least one of those authors is “Kennedy”.

Value Comparisons

Value comparisons: **eq**, **ne**, **lt**, **le**, **gt**, **ge**

Compare exactly two atomic values.

`$book/author eq "Kennedy"`

is true if and only if **`$book`** has exactly one author and that author is “Kennedy”.

Errors are raised if the comparison is not between two values or if the values cannot be compared.

Node Comparisons

Node comparisons: **is**, **<<**, **>>**

The **is** operator compares two nodes for equality (are they *the same* node?). The operators **<<** and **>>** compare document order of nodes.

```
$book/author is key('authors', 'kennedy')
```

is true if and only if **\$book** has exactly one author and that author element is the same as the one returned by the key expression.

XSLT 2.0

Schema Support

Types

Declaring Types

Declaring Types (Continued)

Type Errors

Implicit Casting

Sorting and Collation

Regular Expression Functions

Regular Expression Instructions

Regular Expression Example

Grouping

Grouping By Key (Data)

...

Schema Support

- Many of the W3C XML Schema simple types are always available.
- In order to refer to additional types, you must import the schema that defines them:

```
<xsl:import-schema  
  namespace="http://example.org/example"  
  schema-location="/path/to/example.xsd"/>
```

You must specify at least the namespace or the schema location, if not both.

Types

XSLT 2.0 allows you to declare:

- The return type of (user-declared) functions.
- The return type of templates.
- Both the type and required type of parameters.
- The type of variables.
- The type of sequences (constructed with **xsl:sequence**)
- The type of keys.

Declaring Types

The “**as**” attribute is used to declare types.

```
<xsl:variable name="i" select="1"/>
```

is the integer value 1.

```
<xsl:variable name="fp" select="1"  
              as="xs:double" />
```

is the double value 1.0.

Declaring Types (Continued)

```
<xsl:variable name="date"  
              select="'2003-11-20'"/>
```

is the *string* “2003-11-20”.

```
<xsl:variable name="date"  
              select="xs:date('2003-11-20')"/>
```

is the date November 20, 2003.

```
<xsl:variable name="date" as="xs:date"  
              select="'2003-11-20'"/>
```

is an error.

Type Errors

If a type cannot be cast to another type, attempting the cast will generate a type error. Some (perhaps many) of the things you're used to doing in XPath 1.0 will generate type errors in XPath 2.0:

- Math operations on strings.
- Invalid lexical representations (e.g., "12/08/2003" is not a valid lexical form for dates, use "2003-12-08" instead).
- Incompatible casts.

Implicit Casting

- From subtypes to supertypes.
- Between numeric types (**xs:decimal** to **xs:double**, etc.)
- Computing effective boolean values
- From “untyped” values

Sorting and Collation

- XPath 2.0 and XSLT 2.0 have *collations*
- Collations determine how sorting and collation work
- Collations are identified by URI
- All processors support “Unicode code-point collation”

Regular Expression Functions

There are three regular-expression functions that operate on strings:

- **matches()** tests if a regular expression matches a string.
- **replace()** uses regular expressions to replace portions of a string.
- **tokenize()** returns a sequence of strings formed by breaking a supplied input string at any separator that matches a given regular expression.

Regular Expression Instructions

The **xsl:analyze-string** instruction uses regular expressions to apply markup to a string.

```
<xsl:analyze-string select="..." regex="...">  
  <xsl:matching-substring>...</xsl:matching-substring>  
  <xsl:non-matching-substring>...</xsl:non-matching-substring>  
</xsl:analyze-string>
```

The **regex-group()** function allows you to look back at matching substrings.

Regular Expression Example

These instructions transform dates of the form “12/8/2003” into ISO 8601 standard form: “2003-12-08”.

```
<xsl:analyze-string select="$date"
  regex="([0-9]+)/([0-9]+)/([0-9]{4})">
  <xsl:matching-substring>
    <xsl:number value="regex-group(3)"
      format="0001"/>
  <xsl:text>-</xsl:text> ...
```

Note that the curly braces are doubled in the regular expression. The **regex** attribute is an attribute value template, so to get a single “{” in the attribute value, you must use “{{” in the stylesheet.

Grouping

Grouping in XSLT 1.0 is *hard*. XSLT 2.0 provides a new, flexible grouping instruction for grouping...

- on a specific key
- by transitions at the start of each group
- by transitions at the end of each group
- by adjacent key values

Groups can also be sorted.

Grouping By Key (Data)

Group the following data by country:

```
<cities>
  <city name="Milano"    country="Italia" />
  <city name="Paris"     country="France" />
  <city name="München"  country="Deutschland" />
  <city name="Lyon"     country="France" />
  <city name="Venezia"  country="Italia" />
</cities>
```

Grouping By Key (Code)

```
<xsl:for-each-group select="cities/city"
                    group-by="@country">
  <tr>
    <td><xsl:value-of select="position()" /></td>
    <td><xsl:value-of select="@country" /></td>
    <td>
      <xsl:value-of select="current-group()/@name"
                    separator=", " />
    </td>
  </tr>
</xsl:for-each-group>
```

Grouping By Key (Results)

```
<tr>
  <td>1</td>
  <td>Italia</td>
  <td>Milano, Venezia</td>
</tr>
<tr>
  <td>2</td>
  <td>France</td>
  <td>Paris, Lyon</td>
</tr>
```

Grouping By Starting Value (Data)

Group the following data so that the implicit divisions created by each **h1** are explicit:

```
<body>
  <h1>Introduction</h1>
  <p>XSLT is used to write stylesheets.</p>
  <p>XQuery is used to query XML databases.</p>
  <h1>What is a stylesheet?</h1>
  <p>A stylesheet is an XML document used to defi
  <p>Stylesheets may be written in XSLT.</p>
  <p>XSLT 2.0 introduces new grouping constructs.
</body>
```

Grouping By Starting Value (Code)

```
<xsl:for-each-group select="*"
                    group-starting-with="h1">
  <div>
    <xsl:apply-templates
      select="current-group()" />
  </div>
</xsl:for-each-group>
```

Grouping By Starting Value (Results)

```
<div>
  <h1>Introduction</h1>
  <p>XSLT is used to write stylesheets.</p>
  <p>XQuery is used to query XML databases.</p>
</div>
<div>
  <h1>What is a stylesheet?</h1>
  <p>A stylesheet is an XML document used to def
  <p>Stylesheets may be written in XSLT.</p>
  <p>XSLT 2.0 introduces new grouping constructs
</div>
```

Grouping By Ending Value (Data)

Group the following data so that continued pages are contained in a **pageset**:

```
<doc>
  <page continued="yes">Some text</page>
  <page continued="yes">More text</page>
  <page>Yet more text</page>
  <page continued="yes">Some words</page>
  <page continued="yes">More words</page>
  <page>Yet more words</page>
</doc>
```

Grouping By Ending Value (Code)

```
<xsl:for-each-group select="*"
    group-ending-with="page[not(@continued
        = 'yes')]">
    <pageset>
        <xsl:for-each select="current-group()">
            <page><xsl:value-of select="."/></page>
        </xsl:for-each>
    </pageset>
</xsl:for-each-group>
```

Grouping By Ending Value (Results)

```
<doc>
  <pageset>
    <page>Some text</page>
    <page>More text</page>
    <page>Yet more text</page>
  </pageset>
  <pageset>
    <page>Some words</page>
    <page>More words</page>
    <page>Yet more words</page>
  </pageset>
</doc>
```

Grouping By Adjacent Key Values (Data)

Group the following data so that lists do not occur *inside* paragraphs:

```
<p>Do <em>not</em> :  
  <ul>  
    <li>talk,</li>  
    <li>eat, or</li>  
    <li>use your mobile telephone</li>  
  </ul>  
while you are in the cinema.</p>
```

Grouping By Adjacent Key Values (Code)

```
<xsl:for-each-group select="node()"
  group-adjacent="self::ul or self::ol">
  <xsl:choose>
    <xsl:when test="current-grouping-key()">
      <xsl:copy-of select="current-group()" />
    </xsl:when>
    <xsl:otherwise>
      <p>
        <xsl:copy-of
          select="current-group()" />
      </p>
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each-group>
```

Grouping By Adjacent Key Values (Results)

```
<p>Do not:
```

```
</p>
```

```
<ul>
```

```
  <li>talk,</li>
```

```
  <li>eat, or</li>
```

```
  <li>use your mobile telephone</li>
```

```
</ul>
```

```
<p>
```

```
  while you are in the cinema.
```

```
</p>
```

Functions

At the instruction level, XSLT 1.0 and 2.0 provide mechanisms for calling named templates. They also provide mechanisms for calling user-defined extension functions. What's new is the ability to declare user-defined functions *in XSLT*.

Function Example

```
<xsl:function name="str:reverse" as="xs:string">
  <xsl:param name="sentence" as="xs:string"/>
  <xsl:sequence select="
if (contains($sentence, ' '))
  then concat(str:reverse(
                substring-after(
                  $sentence, ' '),
                ' '),
                substring-before($sentence, ' '))
  else $sentence"/>
</xsl:function>
```

This can be called *from XPath*: `select="str:reverse('DOG BITES MAN')"`.

Stylesheet Modularity

- Stylesheets can be included or imported:
- **<xsl:include>** provides source code modularity.
- **<xsl:import>** provides logical modularity.
- **<xsl:apply-imports>** allows an importing stylesheet to apply templates from an imported stylesheet.
- What's new is **<xsl:next-match>**

Next Match

- If several templates match, they are sorted by priority
- The highest priority template is executed, the others are not
- `<xsl:next-match>` allows a template to evaluate the *next highest* priority template.
- This is independent of stylesheet import precedence

Access to Result Trees

In XSLT 1.0, result trees are read-only. If you construct a variable that contains some computed elements, you cannot access those elements.

Almost every implementation of XSLT 1.0 provided some sort of extension function to circumvent this limitation.

XSLT 2.0 removes this limitation. It is now possible to perform the same operations on result trees that you can perform on input documents.

Result Documents

- XSLT 1.0 provides only a single result tree.
- Almost all vendors provided an extension mechanism to produce multiple result documents.
- XSLT 2.0 provides a **<xsl:result-document>** instruction to create multiple result documents.
- Provides support for validation.

Sequences

The `<xsl:sequence>` instruction is used to construct sequences of nodes or atomic values (or both).

- `<xsl:sequence select='(1,2,3,4)'/>`
returns a sequence of integers.
- `<xsl:sequence select='(1,2,3,4)'
as="xs:double"/>`
returns a sequence of doubles.

Sequences (Continued)

The following code defines `$prices` to contain a sequence of decimal values computed from the prices of each product.

```
<xsl:variable name="prices">
  <xsl:for-each select="products">
    <xsl:choose>
      <xsl:when test="@price">
        <xsl:sequence select="xs:decimal(@price)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:sequence select="xs:decimal(@cost) * 1.5"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:variable>
```

Sequences (Continued)

So does this:

```
<xsl:value-of  
  select="for $p in products return  
        if ($p/@price)  
        then xs:decimal($p/@price)  
        else (xs:decimal($p/@cost) * 1.5)"
```

Sequences (Continued)

Why is the former better? Because if you run the latter *stylesheet* through a processor, you'll get:

```
<xsl:value-of select="for $p in products
return if ($p/@price) then xs:decimal($p/@price) else (xs:decimal($p/@cost) *
1.5)"/>
```

Which I find a little hard to read. My recommendation: use XSLT whenever you can.

Values, Copies, and Sequences

What's the difference between **xsl:value-of**, **xsl:copy-of**, and **xsl:sequence**?

- **xsl:value-of** always creates a text node.
- **xsl:copy-of** always creates a copy.
- **xsl:sequence** returns the nodes selected, subject possibly to atomization. Sequences can be extended with **xsl:sequence**.

Sequence Separators

You can add a separator when taking the value of a sequence:

```
<xsl:value-of select="(1, 2, 3, 4)"  
              separator="; "/>
```

produces “1; 2; 3; 4” (as a single text node).

Formatting Dates

XPath 2.0 adds date, time, and duration types. XSLT 2.0 provides **format-date** to format them for presentation.

- **xsl:date-format**
- **format-date()**

This is analagous to **xsl:number-format** and **format-number()**.

Character Maps

Character maps give you greater control over serialization. They map a Unicode character to *any* string in the serialized document.

- For XML and HTML output methods, the resulting character stream does not have to be well-formed.
- The mapping occurs *only* at serialization: it is not present in result tree fragments.
- This facility can be used instead of “disabled output escaping” in most cases.

Creating Entities

Suppose you want to construct an XHTML document that uses ` ` for non-breaking spaces, `é` for “é”, etc.

```
<xsl:output method="xml"
            doctype-public="-//W3C//DTD XHTML 1.0
            doctype-system="http://www.w3.org/TR/
            use-character-maps="example-map" />
```

```
<xsl:character-map name="example-map">
  <xsl:output-character character="#233;"
                        string="&eacute;" />
  <xsl:output-character character="#160;"
                        string="&nbsp;" />
</xsl:character-map>
```

Avoiding Disable Output Escaping

Suppose you want to construct a JSP page that contains:

```
<jsp:setProperty name="user" property="id"
                 value='<%= "id" + idValue %>' />
```

Pick some otherwise unused Unicode characters to represent the character sequences that aren't valid XML. For example, “«” for “<%=”, “»” for “%>”, and “.” for the explicit double quotes:

```
<jsp:setProperty name="user" property="id"
                 value='« .id. + idValue »' />
```

Avoiding D-O-E (Continued)

Construct a character map that turns those characters back into the invalid strings:

```
<xsl:character-map name="jsp">
  <xsl:output-character char="«"
                        string="&lt;%" />
  <xsl:output-character char="»"
                        string="%&gt;" />
  <xsl:output-character char="."
                        string="' "' />
</xsl:character-map>
```

Compatibility

- An XSLT 1.0 processor handles 2.0 stylesheets in *forwards compatibility mode*
- An XSLT 2.0 processor handles 1.0 stylesheets:
 - As a 1.0 processor, or
 - in *backwards compatibility mode*
- A mixed-mode stylesheet uses the appropriate mode.

Although the goal is that a 2.0 processor running a 1.0 stylesheet in backwards compatibility mode should produce the same results as a 1.0 processor, this is not guaranteed to be the case for all stylesheets.

Closing Thoughts

Is it Worth It?

Acknowledgements

References

Questions & Comments

Is it Worth It?

XPath 2.0 and XSLT 2.0 are larger and more complex than their predecessors. Schema support and “strong” typing will catch errors, but they will also force more explicit casting.

Does it make sense to start developing for 2.0?

Yes, I think so. Grouping, regular expressions, user-defined functions, character maps, standardized access to result documents and multiple result documents are all going to make stylesheet writing easier.

Acknowledgements

Many of the examples in this tutorial are taken directly from the XSLT 2.0 Specification.

Jeni Tennison's *Typing in Transformations* paper from *Extreme Markup Languages 2003* was instrumental in refreshing my memory about the issues surrounding XPath 2.0 casting rules.

References

XQuery 1.0 and XPath 2.0 Data Model, <http://www.w3.org/TR/xpath-datamodel/>. Mary Fernández, Ashok Malhotra, Jonathan Marsh, *et. al.*, editors. World Wide Web Consortium. 2003.

XQuery 1.0 and XPath 2.0 Functions and Operators, <http://www.w3.org/TR/xpath-functions/>. Ashok Malhotra, Jim Melton, and Norman Walsh, editors. World Wide Web Consortium. 2003.

XQuery 1.0 and XPath 2.0 Formal Semantics, <http://www.w3.org/TR/xquery-semantics/>. Denise Draper, Peter Frankhauser, Mary Fernández, *et. al.*, editors. World Wide Web Consortium. 2003.

References (Continued)

XML Path Language (XPath) 2.0, <http://www.w3.org/TR/xpath20/>. Anders Berglund, Scott Boag, Don Chamberlin, *et. al.*, editors. World Wide Web Consortium. 2003.

XQuery 1.0: An XML Query Language, <http://www.w3.org/TR/xquery/>. Scott Boag, Don Chamberlin, Mary Fernández, *et. al.*, editors. World Wide Web Consortium. 2003.

XSL Transformations (XSLT) Version 2.0, <http://www.w3.org/TR/xslt20/>. Michael Kay, editor. World Wide Web Consortium. 2003.

XSLT 2.0 and XQuery 1.0 Serialization, <http://www.w3.org/TR/xslt-xquery-serialization/>. Michael Kay,



References (Continued)

Norman Walsh, and Henry Zongaro, editors. World Wide Web Consortium. 2003.

Typing in Transformations, <http://www.idealliance.org/papers/extreme03/html/2003/Tennison01/EML2003Tennison01-toc.html>. Jeni Tennison. Extreme Markup Languages. 2003.

Questions & Comments