W3C®

I18n Sensitive Processing with XQuery and XSLT

Felix Sasaki
World Wide Web Consortium

W3C®

Purpose

Enable the audience to use XQuery and XSLT for i18n sensitive processing and make them aware of i18n aspects of XQuery and XSLT which have to be handled carefully.

The purpose of this presentation is to enable the audience to use XQuery and XSLT for i18n sensitive processing and make them aware of i18n aspects of XQuery and XSLT which have to be handled carefully.

Topics

- Introduction
- The common underpinning: XPath 2.0
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

The tutorial will give an overview of the general purposes of XQuery and XSLT and XPath 2.0. XPath 2.0 is the common underpinning of both languages. A part of XQuery and XSLT which is of specific interest for i18n sensitive processing is the generation of output documents, the so-called serialization. Further topics to be covered encompass string and IRI processing, dates and processing of language information. Throughout the tutorial, the benefits of XQuery and XSLT will be introduced, but also its aspects which have to be handled carefully when processing multilingual data.

Introduction

- 17 (!) specifications about "XQuery" and "XSLT", abbreviated as "QT"
- QT encompasses a bunch of i18n related features
- A complex architecture
- QT describes input, processing and output of XML data

In recent years, the W3C has worked on 17 (!) documents which deal with the XML query language "Query 1.0" and the transformation language for XML documents "XSLT 2.0". Both are henceforth noted as "QT". QT has a lot of i18n related features. But due to its complexity, many parts of the design of QT have to be taken into account. This tutorial will give an overview of the QT architecture, the QT processing model, and describe i18n specific features for the input, processing and output of XML data.

The different pieces of the cake

1. The common underpinning of XQuery and XSLT: XPath 2.0 data model & formal semantics
2. How to select information in XML documents: XPath 2.0
3. Manipulating information: XPath functions and operators
4. Generating output: Serialization
5. The XQuery 1.0 and XSLT 2.0 specifications, which deploy 1-4

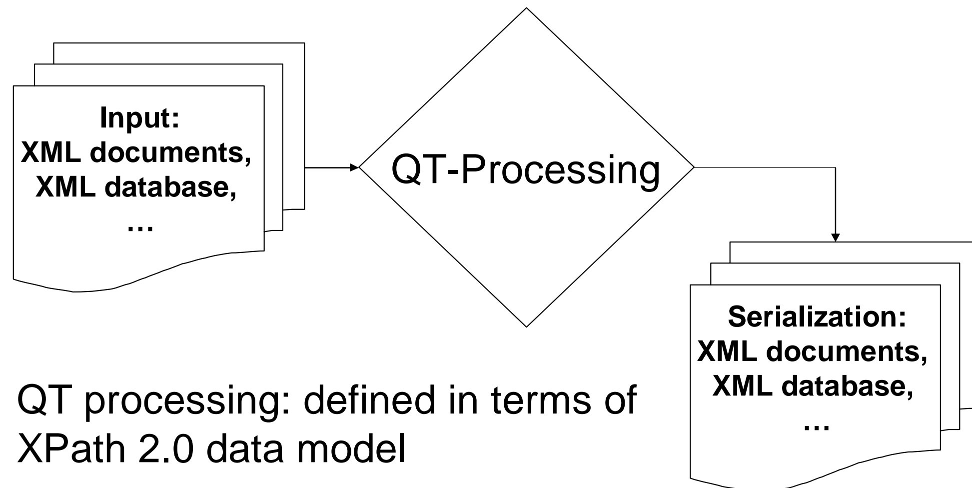
The main specifications are listed on this slide. The common basis for XQuery and XSLT is the XPath 2.0 data model and its formal semantics. The data model describes what information is part of XML documents, e.g. element nodes, attribute nodes or namespace nodes. XPath 2.0 is a means to select information from XML documents. The XPath functions and operators are a tool to manipulate the selected information. Finally a specification describes how the result of the XQuery or XSLT processing can be serialized, i.e. as XML, HTML, XHTML or text. For the languages XQuery 1.0 and XSLT 2.0 themselves, there are two specifications which deploy the specifications 1 – 4 and add some extensions.

Attention!

Basis of this presentation: A set of WORKING DRAFTS!
Things might still change!

Topics

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization



What are the main purposes of XQuery and XSLT? XQuery is a query language for XML. It takes as an input zero or more source documents. The output are zero or more result documents. XSLT is a transformation language for XML. It takes as an input zero or more source documents. The output are zero or more result documents. In the center is the QT processing, which is defined in terms of the XPath 2.0 data model.

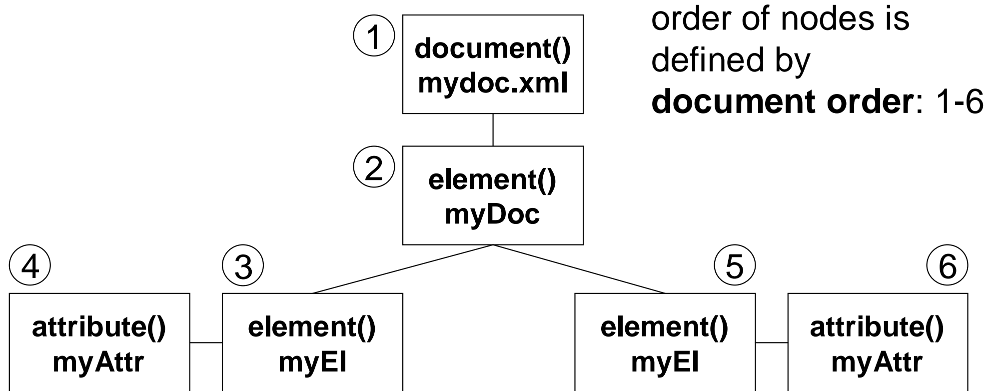
XPath 2.0 data model:

- sequences of items, i.e. **nodes** …
  - document node
  - element nodes: <myDoc>…</myDoc>
  - attribute nodes: <myEl myAttr="myVal1"/>
  - namespace nodes: <myns:myEl>…</myns:myEl>
  - text nodes: <p>My yellow (and small) flower.</p>
  - comment node: <!-- my comment -->
  - processing instruction: <?my-pi … ?>
- and / or **atomic values** (see below)

The XPath 2.0 data model defines the information in an XML document as sequences of items. Items can be nodes or atomic values. There are seven kinds of nodes: the document node, elements, attributes, namespaces, text nodes, comment nodes and nodes for processing instructions. Atomic values will be discussed below.

Visualization of nodes

```
<myDoc>
 <myEl myAttr="myVal1"/>
 <myEl myAttr="myVal2"/>
</myDoc>
```



This slides visualizes some of the nodes which are contained in the document "mydoc.xml". There is the document node and three element nodes. To each <myEl> element, an attribute node is attatched. The concept of "document order" assures that there is a definite sequence of the nodes. In "mydoc.xml", there are six nodes. The first node is the document node, followed by the root element <myDoc>. The following nodes are its child elements with their attributes respectively.

Atomic values

- Nodes in XPath 2.0 have string values and typed values, i.e. a sequence of atomic values
- "string" function: returns a string value, e.g.
  - string(doc("mydoc.xml"))

Nodes in XPath 2.0 have string values and typed values, i.e. a sequence of atomic values. The "string" function returns the string value of nodes. For example, string(doc("mydoc.xml") returns the string value of the document "mydoc.xml". Since there is no textual content in the elements, this is an empty value.

i18n related typed values

- From XML Schema: built in primitive data types like anyURI, dateTime, gYearMonth, gYear, …
- specially for XPath 2.0: xdt:dayTimeDuration, …
- Good for: URI processing, time related processing

For i18n related data, there are various types which will be discussed in this presentation. XPath 2.0 deploys the built-in datatypes from XML Schema. Interesting are the URI type anyURI or the time related types like dateTime or gYearMonth. XPath 2.0 adds some types like xdt:dayTimeDuration.

Not in the data model

- ... is:
  - Character encoding schema
  - CDATA section boundaries
  - entity references
  - DOCTYPE declaration and internal DTD subset
- All this information might get lost during XQuery / XSLT processing
- Mainly XSLT allows the user to parameterize the output, i.e. the serialization of the data model

One has to be careful about some information which is in an XML document, but which is not represented in the data model. Among these is the character encoding scheme. On the level of the data model, it does not exist. It comes into play as the data model is serialized into an output format. We will see later how the serialization works.

CDATA section boundaries, entity references, the DOCTYPE declaration and the internal DTD subset are also not part of the data model. What does it mean that something is not in the data model? This information might get lost during XPath 2.0 based processing. What is lost or not, depends on the language which deploys XPath 2.0, i.e. XQuery or XSLT. As we will see later, especially in the case of XSLT the user can specify what information she wants to retain or create for the serialization.

Topics

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

General processing of XQuery / XSLT

- XQuery:
  - Input: zero or more source documents
  - Output: zero or more result documents
- XSLT:
  - Input: zero or more source documents
  - Output: zero or more result documents
- What is the difference?

As had been said before, the general processing of XQuery and XSLT is very similar. Both take as an input zero or more source documents. The output are zero or more result documents. Naturally the question arises what the difference between the two is. We will discuss some examples to provide an answer.

An example

- Processing input "mydoc.xml":

```
<myDoc>
 <myEl myAttr="myVal1"/>
 <myEl myAttr="myVal2"/>
</myDoc>
```

- Desired processing output "yourdoc.xml":

```
<yourDoc>
 <yourEl yourAttr="myVal1"/>
 <yourEl yourAttr="myVal2"/>
</yourDoc>
```


As a possible input document we have again "mydoc.xml". It consists of an <mydoc> element which contains two <myEl> elements. These have two attributes @myAttr. The task is to create an output document "yourdoc.xml". In "yourdoc.xml", the names of the elements and attributes are renamed, to <yourDoc>, <yourEl> and @yourAttr respectively.

XSLT

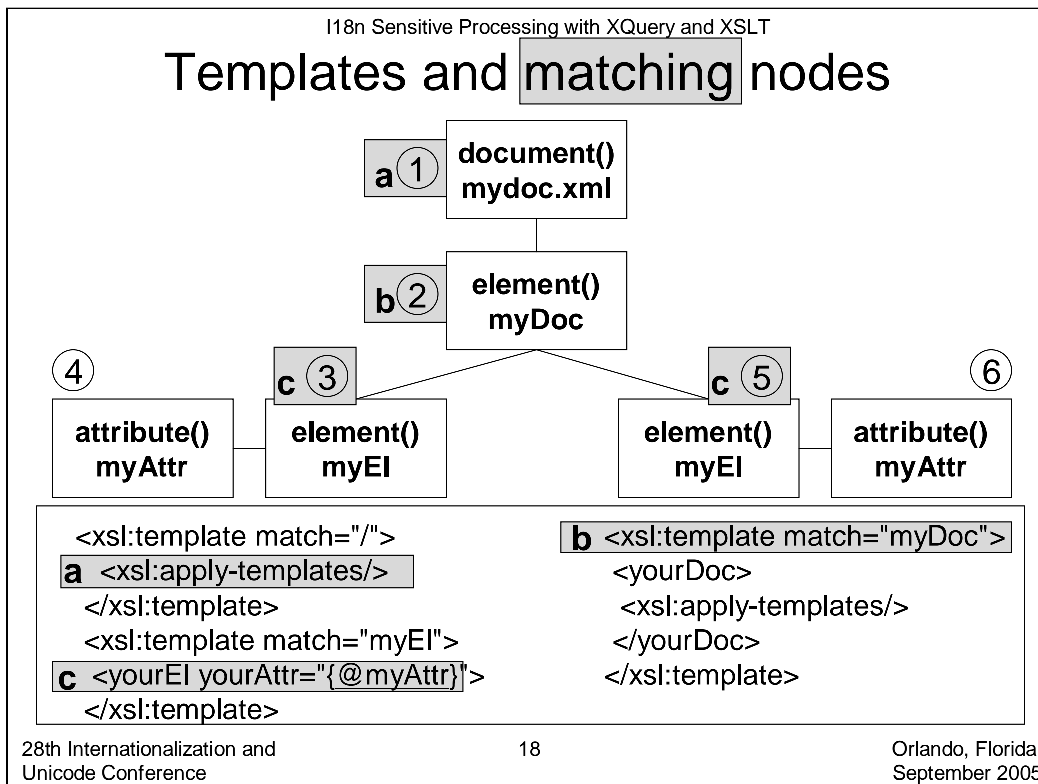
```
<xsl:stylesheet …>
 <xsl:template match="/">
   <xsl:apply-templates/>...
 </xsl:template>

 <xsl:template match="myEl">
  <yourEl yourAttr="{@myAttr}">
 </xsl:template>

 <xsl:template match="myDoc">
  <yourDoc>
   <xsl:apply-templates/>
  </yourDoc>
 </xsl:template>
</xsl:stylesheet>
```

- **Template** based processing
- Traversal of input document, ***match*** of templates
- "Push processing": Nodes from the input are pushed to matching templates

How can this task be accomplished by XSLT 2.0? XSLT processes input documents in terms of templates. The input document is traversed in "document order" until a so called "initial template" matches a node. Then the content of the templates is processed. This process can encompass the creation of nodes for the result document or the application of further templates. This kind of processing is called "push processing", because the processed nodes are pushed to the stylesheet in a way "let's see, which template matches the current node!".



This slides visualizes how the document is traversed in document order and which nodes are matched by which template. In the sample XSLT stylesheet, the template "a" has the matching rule match="/". This matches the document node, so this template is the initial template. Via <xsl:apply-templates/>, further templates are applied for the child of the document node. This is the element <myDoc>. The template "b" with the rule match="myDoc" matches the <myDoc> element. In this template, the <yourDoc> element is created. With <xsl:apply-templates/> as the content of <yourDoc>, again further templates are being applied for the child elements. There are two <myEl> child elements. The template "c" with the rule match="myEl" matches these two elements. In this template, the <yourEl> element is created with an attribute @yourAttr. Its value is the value of the attribute @myAttr from the source document.

XQuery

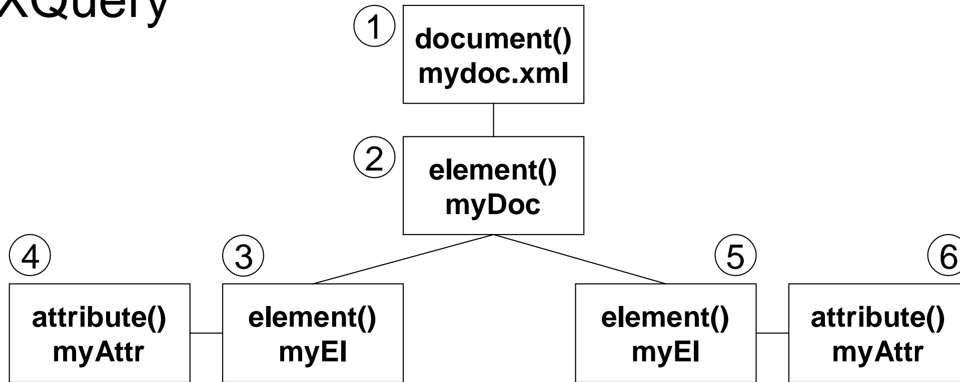
```
xquery version "1.0";
<yourDoc>
{
let $input := doc("mydoc.xml")
for $elements in $input//myEl
return
<yourEl
yourAttr="{$elements/@myAttr}"/>
}
</yourDoc>
```

- "Pull processing": XPath expressions **pull information** out of document(s)

This sample XQuery document creates the same result document as the XSLT stylesheet. The difference is that there is no template based processing. XQuery applies "pull processing": The XPath expressions pull information out of source documents.

I18n Sensitive Processing with XQuery and XSLT

XQuery




```
xquery version "1.0";
<yourDoc>
{
let $input := doc("mydoc.xml")
for $elements in $input//myEl
return
<yourEl
yourAttr="{$elements/@myAttr}"/>
}
</yourDoc>
```

28th Internationalization and Unicode Conference — 20 — Orlando, Florida, September 2005

This slides visualizes how this pulling works. In the sample query, first the <yourDoc> element is created for the result document. Inside the <yourDoc> element, the variable $input is bound to the document node of the document "mydoc.xml" via the "let" expression. Via the "for" expression, all <myEl> elements are bound to the variable $elements. For each <myEl> element, a <yourEl> element is being created. Via the expression yourAttr="{$elements/@myAttr}", an attribute @yourAttr is being attached to this element. Like in the XSLT stylesheet, its value is the value of the attribute @myAttr from the source document.

XPath 2.0 expressions

```
<xsl:template match="myDoc">
 <yourDoc>
  <xsl:apply-templates/>
 </yourDoc>
</xsl:template>
…
<xsl:template match="myEl">
 <yourEl yourAttr="{@myAttr}">
</xsl:template> ...
```

```
xquery version "1.0";
<yourDoc>
{
let $input :=
 doc("mydoc.xml")
for $elements in
$input//myEl
return
<yourEl yourAttr=
"$elements/@myAttr"/>
}</yourDoc>
```

In both languages: **selection of nodes** in single or multiple documents. In XSLT: "***patterns***" as subset of XPath for matching rules

The main task of XPath expressions is the selection of information in single or multiple source documents. We will not go into detail here, but explain the examples briefly. In the sample query, the "let" expression selects the document node of the document "mydoc.xml". The variable $input is bound by this node. The "for" expression selects all <myEl> elements which are under the document node. The variable $elements is bound by these elements. The expression $elements/@myAttr selects the attribute @myAttr.

XSLT uses as subset of XPath 2.0 for the description of matching rules for templates, so-called "patterns". In the XSLT stylesheet, the patterns match the <myDoc> element and the <myEL> element respectively.

I18n Sensitive Processing with XQuery and XSLT

XPath 2.0 data model	**General processing**	Strings, numbers	IRI processing	Dates, language	Output: serialization

When to use XSLT

- Good for processing of mixed content, e.g. text with markup. Example task:

```
<para>My <emph>yellow</emph> <note>and
  small</note> flower.</para>
```

should become

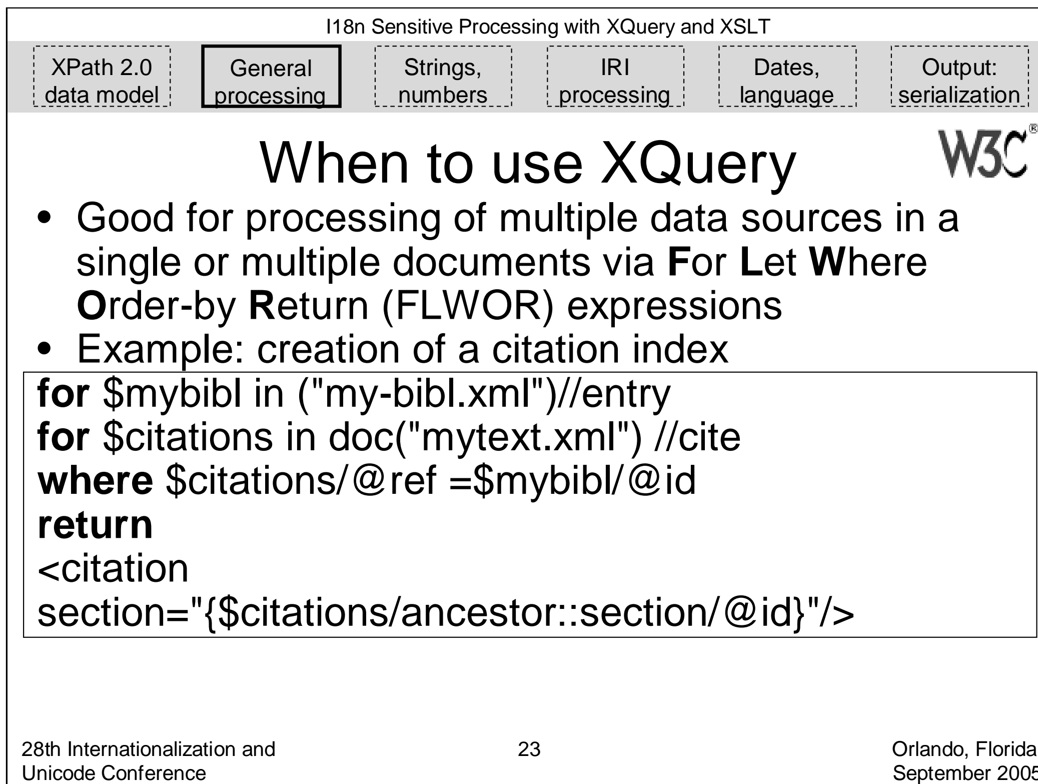
```
<p>My <em>yellow</em> (and small) flower.</p>
```

Solution: push processing of the <para> content

```
<xsl:template match="para">
<p><xsl:apply-templates/></p> </xsl:template>
<xsl:template match="emph">…</xsl:template> …
```

28th Internationalization and Unicode Conference — 22 — Orlando, Florida, September 2005

The terminology "pull processing" versus "push processing" has been created by the ISO Working Group which developed SGML or DSSSL. A rule of thumb when to use XSLT is for push processing. (This is only a rule of thumb; of course, XSLT can also be used for pull processing.) Especially if the source XML document contains many elements with mixed content, e.g. text with markup, XSLT is very convenient. The example shows a <para> element with mixed content. The content of the <para> element can be processed simply by creating a template for each element, e.g. a template with the matching rule match="emph". The text nodes and the element nodes are pushed to these templates. For each node the appropriate output is created.



XQuery has no facilities for such push processing. The rule of thumb here is: Use XQuery for processing of multiple data sources in a single or multiple source documents. The mechanism for this task is called "FLWOR" expression. The name is derived from the first letters of the parts of such expressions: (f)or, (l)et, (w)here, (o)rder-by and (r)eturn.

The sample query creates a citation index, using information from a document with bibliographic entries "my-bibl.xml". The first "for" expression iterates other each <entry> element in that document. The second "for" expression iterates other each <cite> element in the document "mytext.xml". The "where" expression filters the <cite> elements whose @ref attribute has the same value as the @id attribute of the <entry> element. For these <cite> elements, an element <citation> is created in the result document. Its @section attribute contains the value of the @id attribute from the <section> element respectively. The "order-by" expression is not used in this exampe. It allows the user to specify an order of the returned sequence.

These two rules of thumb must be seen as a general guideline. Since both XQuery and XSLT have the same underpinning, i.e. the XPath 2.0 and XPath 2.0 expressions, many processing tasks can be accomplished with both languages.

Topics

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

Aspects of string processing

- What is the scope: characters (code points)
- String counting
- Codepoint conversion
- String comparison: collations
- String comparison: regular expressions
- Normalization
- The role of schemas e.g. in the case of white space handling

There are various aspects which have to be taken into account for string processing with QT. These will be discussed in the following slides.

Scope of string processing

- Basic operation: Counting 'characters'
- Good message: QT counts code points, not bytes or code units
- Attention: All string processing uses string values, not typed values!

String processing in QT takes "characters" in the sense of Unicode code points as the basic unit. It is a good message that QT deals not with bytes or code units. Nevertheless, there is one aspect of QT which the user has to take care of: All string processing uses string values, not typed values!

I18n Sensitive Processing with XQuery and XSLT

XPath 2.0 data model | General processing | **Strings, numbers** | IRI processing | Dates, language | Output: serialization

String values versus typed values

string-length($myDoc/myEl/revision-date@)

string-length(**xs:string**($myDoc/myEl/revision-date@))

- With a schema: type of @revision-date = xs:date
- Works not works

28th Internationalization and Unicode Conference — 27 — Orlando, Florida, September 2005

The difference between string values and typed values can be seen in the two examples. Both examples deploy the XPath 2.0 function string-length. The length of the @revision-date attribute should be calculated. It is assumed that there is a schema which defines @revision-date with the XML Schema datatype xs:date. With such a schema, the first example would not work.

String values versus typed values

- Difference: second example uses adequate **type casting**
- Type casting is not always possible: http://www.w3.org/TR/xpath-functions/#casting-from-primitive-to-primitive

The reason is that string-length expects a string value as the input. To be able to apply xs:string to @revision-date, one has to use type casting. In the second example, the type xs:date is casted to xs:string via the XPath 2.0 function xs:string.

Type casting is not always possible. The link on the slide provides information about what types can be casted to what other types.

Codepoints versus strings: XQuery

```
<text>{"string to code points:  su&#xE7;on
becomes ",
string-to-codepoints("su&#xE7;on"),
"code points to string:  115 117  231 111 110
becomes ",
codepoints-to-string((115, 117, 231, 111, 110))
}</text>
```

```
<text>
string to code points:  suçon becomes 115 117 231 111 110.
code points to string:  115 117  231 111 110 becomes suçon
</text>
```


Two functions provide access to codepoints to string conversion and vice versa. In the example, the codepoints of the string "suçon" are generated via string-to-codepoints, and the string for the codepoints is generated via codepoints-to-string. The output of this sample query is shown below.

Codepoints versus strings: XSLT

```
<text>
 <xsl:text>string to code points:  su&#xE7;on
becomes </xsl:text>
<xsl:value-of select="
string-to-codepoints('su&#xE7;on')"/>
<xsl:text>. code points to string:  115 117  231 111
110 becomes </xsl:text>
<xsl:value-of select="
codepoints-to-string((115, 117, 231, 111,
110))"/>
</text>
```


With XSLT, the same output can be generated with the same XPath 2.0 functions. The difference to the XQuery example is that XSLT uses XSLT elements to evoke the same processes as within XQuery. The text "string to code points: suçon" is generated via the <xsl:text> element, and the two XPath 2.0 functions are evoked via the @select attribute at the <xsl:value-of> element.

Collation functions: compare()

- Returns "0":

```
<xsl:value-of select="compare('abc', 'abc')"/>
```

```
compare("abc", "abc")
```

- Returns "-1":

```
<xsl:value-of select="compare('abc', 'bbc')"/>
```

- Returns "1":

```
<xsl:value-of select="compare('bbc', 'abc')"/>
```


As for collations, QT deploys a codepoint-based collation. The first example, given both in XSLT and XQuery, shows a compare function which returns "0". This is the case if the two arguments are equal. Compare returns "-1" if the first argument is less than the second (second example), and "1" if the first argument is greater than the second (third example).

Collation based function compare()

- Identification of collation via an URI.
- Example: returns "1" if 'myCollation' describes the order respectively:

```
<xsl:value-of select"compare('Strasse', 'Straße',
'myCollation')"/>
```

```
compare("Strasse", "Straße", "myCollation")
```


Other collations can be evoked via an absolute or relative URI. In the example it is assumed that there is a collation "myCollation" which defines that "ß" is lower than "ss". For this collation, the result will be "1".

Collation identification

- Identification via an URI. Codepoint-based collation:

http://www.w3.org/2005/04/xpath-functions/collation/codepoint

- Parameterization via an URI:

http://myQtProcessor.com/collation?lang=de;strength=primary

QT refers to a codepoint-based collation with the URI http://www.w3.org/2005/04/xpath-functions/collation/codepoint. It is also possible to describe a parameterization with an URI, as exemplified on the slide.

String comparison: regular expressions

- Based on regular expressions for XML Schema datatypes, with some additions
- **Flags** for case mapping based on Unicode case mapping tables:

```
<xsl:value-of select="
matches('myLove', 'mylove','i')"/>
```


A lower level string comparison in QT is provided by the regular expressions. They are based on the regular expression syntax for the XML Schema datatypes, with some minimal additions. Interesting here is that although the regular expressions do not allow for the application of collations, they do deploy information which goes beyond code point order. The flag "i" is used to describe case mapping. In the example, the matches function will return "true", since lower and upper case are folded.

Normalization

- XML documents: not always with early unicode normalization
- Unicode collation algorithm ensures equivalent results
- Normalization can be ensured for **NCF**, NFD, NFKC, NFKD:

```
<xsl:value-of select="
unicode-normalize('suc&#x0327;on','NFC')"/>
```

- Output:

```
su&#xE7;on
```


Not all XML documents provide early Unicode normalization. For collation sensitive operations like with the compare function, the Unicode collation algorithm ensures equivalent results for both normalized and not normalized data. In addition, the function unicode-normalize allows the user to create a specified normalization form. In the example, the input string "suçon" contains the COMBINING CEDILLA. It is part of the combining sequence "c¸". This is not Unicode-normalized since "c¸" should appear instead as the precomposed "ç". The output of the function unicode-normalize, with the normalization form "NFC", is this desired precomposed version.

White space and typed values

- Assuming a type for @lastname:

```
<person lastname="Dr.&#x20;&#x20;No"/>
```

- Comparison of typed values via **eq**

```
<xsl:value-of select="
string($myDoc/person/@lastname) eq 'Dr.&#x20;No'
"/>
```

- Collation might also affect white space handling

As has been stated before, string processing uses string values, not typed values. If typed values come into play, one has to be careful about the underlying schema definitions. In the example, it is assumed that the @lastname attribute is defined with a type which collapses whitespace.

The choice of collation also affects the way whitespace is handled. Different collations can and do handle whitespace (and other "less-significant" characters such as hyphens) in different ways.

White space and typed values

- Result: "false" or "true":
  - "false" if type of @lastname collapses whitespace
  - "true" if type of @lastname does not collapse whitespace

If a document contains the attribute lastname="Dr. No", this would be collapsed to "Dr.No". The comparison of "Dr.No" to "Dr. No" then would result in "false". If the type of @lastname does not collapse whitespace, the result would be "true".

Number processing: rounding

- number / currency formatting:

round(2.5) returns 3.
round(2.4999) returns 2.
round(-2.5) returns -2

- does not deploy culture specific rounding conventions, e.g.
  - round 3rd digit less than 3 to 0 or drop it (Argentina)

As for number processing, QT provides for example a rounding function which is exemplified here. Nevertheless, the rounding conventions are fixed and cannot be adapted. E.g. it is not possible to specify an Argentina rounding, where a 3rd digit less than 3 is rounded to 0 or dropped. QT has no specific data type for "currency", therefore it cannot adopt rounding conventions for currency that are different from those applying to other numeric quantities. Such functionality might be implemented by user-defined functions to QT.

XSLT-specific: Numbering

- Conversion of numbers into a string, controlled by various **attributes**:

<xsl:number value="position()" **format**="Ww" **lang**="de" **ordinal**="-e" />
<xsl:number value="position()" **format**="ア"/> <!-- ア is ア-->
<xsl:number value="position()" format="๑"/> <!–
๑ is ア -->

XSLT provides optional attributes at the <xsl:number> element to control the conversion of numbers into a string. This process is not number processing, which we discussed before, but the creation of formatted numbers and strings respectively. The @format attribute at the <xsl:number> element provides a sequence of format tokens. E.g. the format token "Ww" generates title-case words like "First" or "Second". With the @lang attribute, a language can be specified, e.g. "de" for German numbering. The @ordinal attribute specifies ordinal numbering. The value of that attribute can be used to describe language-specific conventions for ordinal numbering. In the example, gender and correspondence to noun declination is specified via "-e".

If in the @format attribute a Unicode character with a decimal digit value of 1 is given, this Unicode character is the starting point for numbering.

XSLT-specific: Numbering

- Output for a sequence of three items:

Ersteア๑Zweiteイ๒Dritteウ๓

In the example, the Unicode characters for Japanese Katakana numbering and for Thai numbering are given. For a sequence of three items, e.g. three element nodes, the result is as displayed.

XSLT-specific: Numbering

- format-number(): designed for numeric quantities (not necessarily whole numbers)

xsl:number is designed primarily for e.g. section numbers. The XSLT function format-number() is designed for numeric quantities (not necessarily whole numbers).

Topics

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

Status of IRI in QT

- In the data model: Support for IRI will be normative.
- data type xs:anyURI: relies on xml schema anyURI, still defined in terms of URI

The underlying data model of QT, XPath 2.0, currently does not reference IRI. Nevertheless, for the next version of the QT working drafts, IRI will be a normative reference. The XML Schema data type xs:anyURI, which is also deployed in QT, is still defined in terms of URI. Since the developers of the QT specifications do not want create contradictions to XML Schema, we have to wait until xs:anyURI will be redefined in terms of IRI.

Functions for IRI / URI processing

- casting to xs:anyURI: from untyped values or string:

```
xs:anyURI("http://example.m&#xfc;ller.com")
```


For URI / IRI processing, QT provides various functions. Casting to xs:anyURI is possible from untyped values or string values. The example shows the type casting for the URI "http://example.müller.com".

Functions for IRI / URI processing

- escaping URI via escape-uri, escaped-reserved="false"

```
escape-uri
("http://example.d&#xfc;rst.com",false())
```

- output:

```
http://example.d%C3%BCrst.com
```


The function escape-uri escapes URI values. It has a parameter "escape-reserved" which can be set to "true" or "false". If it is true, all characters are escaped other than the lower and upper case letters a-z, digits 0-9, the PERCENT SIGN "%", the NUMBER SIGN "#" characters and "marks". If escape-reserved is set to "false", additional characters are not escaped. In terms of RFC 3986, the URI specification, these are reserved characters like SEMICOLON ";" or QUESTION MARK "?". This function always generates hexadecimal values using the upper-case letters A-F. If a user wants to escape the PERCENT SIGN "%", they should do that manually by replacing it with "%25".

Functions for IRI / URI processing

- output with escaped-reserved="true":

http%3A%2F%2Fexample.d%C3%BCrst.com

Topics

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

Dates and time types

- Basis:
  - date and time types from XML Schema
  - QT specific extensions: xdt:yearMonthDuration, xdt:dayTimeDuration
- Operations: time comparison, time adjustment, timezone sensitive operations

The basis of date and time data types in QT are again data types from XML Schema. QT makes some extensions for duration based types, e.g. xdt:yearMonthDuration or xdt:dayTimeDuration. Operations on the date and time data types encompass time comparison, time adjustment or timezone sensitive operations.

Comparison of date types

- Comparison of date types:
  xdt:yearMonthDuration("P1Y6M") eq
  xdt:yearMonthDuration("P1Y7M")
- output:

```
false
```


Comparison of date types is exemplified here with the function yearMontDuration-equal. Its input are two values of the type xdt:yearMonthDuration. The output is "true" or "false", depending whether the values are identical or not.

Component extraction

- Extracting the timezone from a date value:

```
timezone-from-date
(xs:date("2005-07-12+07:00"))
```

- output:

```
PT7H
```


Date and time values work with timezones. The user can give the timezone explicitly, as in the example xs:date("2005-07-12+07:00"). QT then provides functions which allow the user to extract the timezone from the data. The function timezone-from-date executes the extraction. Other components of a date or time value like the hours can be extracted with other functions respectively.

Arithmetic functions on dates and times

- Subtract dayTimeDurations:

```
xdt:dayTimeDuration("P2DT12H") -
xdt:dayTimeDuration("P2DT12H30M")
```

- output:

```
-PT30M
```


It is also possible to subtract, add, multiply or divide date or time values. In the example a value of the type xdt:dayTimeDuration is subtracted from another value. The result is a value of the same type.

XSLT: Formatting Dates / Times

- Some parameters for formatting conventions: picture string with [components]; **presentation modifier**; **language**

```
<xsl:value-of select="format-date(xs:date('2005-09-07'),'[MNn] [D1o] [Y]', 'en', (), ())"/>
<xsl:value-of select="format-date(xs:date('2005-09-07'),'[D1o] [MNn] [Y]', 'de', (), ())"/>
```


In XSLT, functions are provided to format dates and times as a string. The slides shows the function format-date which is used to format values of the type xs:date. The function takes as an argument a date, e.g. "2005-09-07". Another argument called "picture string" indicates the order of the components, e.g. of the year "Y", month "M" and day "D" component. For each component, a presentation modifier can be added, e.g. "Nn" for title-case words or "1" for decimal output. Ordinal numbering of decimal output can be specified by "o". In addition to the picture string, the language can be specified. In the example there are the two languages "en" and "de". In two arguments which are not given in the examples, it is also possible to specify the calendar (e.g. a Japanese calendar) and the country.

XSLT: Formatting Dates / Times

- Output:

September 7th 2005
7. September 2005

Processing of language information

- function lang:

```
/myRoot/myEl/text()[lang("de")]
```

- returns the content of <myEl>, assuming the document:

```
<myRoot xml:lang="de">
<myEl>Some german text.</myEl>
</myRoot>}
```


Language information, provided by the attribute xml:lang, can be processed via the lang function. It works as follows: as evoked, the function retrieves the value of the attribute xml:lang on the current node or an ancestor node. If there are several xml:lang attributes, the xml:lang attribute matches which is closest to the current node. It is then tested whether the input value to the lang function is identical with the value of that xml:lang attribute. If the values are the same, the result is "true".

Processing of language information

- no value for xml:lang: lang("de") returns "false"

One must be careful with xml:lang attributes which have no values. They do not denote something like "any language", but an empty string. Hence, comparing for example lang("de") with xml:lang="" returns "false".

Topics

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

Serialization – basic concept

- XQuery / XSLT: process XML in terms of the XPath 2.0 data model
- Output: described in terms of serialization parameters

Both XQuery and XSLT do not define a serialization. There is a separate specification of serialization parameters. In this specification, various output parameters are described. XSLT and XQuery differ from each other with respect to the deployment of these parameters.

Some serialization parameters

- byte-order-mark
- cdata-section-elements
- encoding
- escape-uri-attributes
- media-type
- normalization-form
- use-character-maps

Some parameters which are important for i18n sensitive processing are listed on this slide. The parameter "byte-order-mark" allows for specifying the output of a byte order mark. The parameter cdata-section-elements describes in which elements CDATA sections from the input should be preserved. The encoding parameter describes the character encoding. escpape-uri-attributes lists the attributes whose values should be escaped according to the rules for URI escaping described before. media-type defines a media-type for the output document, and normalization-form specifies one of "NFC", "NFD", "NFKC", "NFKD". use-character-maps is only applicable for XSLT and will be described below.

Output methods

- Pre-configuration of various serialization parameters for:
  - XML
  - XHTML
  - HTML
  - Text
- XQuery:
  - Mandatory output method: XML, version="1.0"
  - No need for implementations to support further serialization parameters

The serialization specification describes four output methods which deploy these parameters in various ways: XML, XHTML, HTML and text. As for XQuery, only the output method XML in the XML version "1.0" is mandatory. Implementation do not need to support further serialization parameters.

Output methods in XSLT

- Provides support for serialization parameters and output methods via
  - xsl:output
- Support also not mandatory

XSLT provides the element <xsl:output> to specifiy output parameters. Unfortunately, XSLT implementations are also not forced to support the parameters.

XSLT character maps

- Mapping characters to other characters
- Desired output:

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


Character maps are a convenient way in XSLT to replace characters in a document with other characters. The <xsl:character-map> contains one ore more <xsl:output-character> elements which define the mapping between characters. This eases the task of creating not well-formed output. Suppose you want to create a JSP page like <jsp:setProperty name="user" property="id" value='<%= "id" + idValue %>'/>.

XSLT character maps

- Character map:

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


This can be achieved by a character map which maps the problematic characters ">", "<" and '"' to other characters like "«", "»" and " § " which are not used in the document.

Regular expressions with XSLT

```
<xsl:template match="text()">
 <xsl:analyze-string select="." regex="&#xE001;">
  <xsl:matching-substring>
   <myChar type="E001"/>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
  <xsl:value-of select="."/>
  </xsl:non-matching-substring>
</xsl:analyze-string>
</xsl:template>
```


As XQuery, XSLT provides XPath 2.0 functions for regular expressions described before. In addition, XSLT provides the element <xsl:analyze-string> which can be used to replace characters with markup. The example shows an <xsl:analyze-string> element which matches the character "". For the matching substring, the <xsl:matching-substring> element is applied. Its content creates an element <myChar> with a @type attribute respectively. The non-matching substrings are handled within the <xsl:non-matching-substring> element. In the example, they are just added to the result document.

Regular expressions with XQuery

```
xquery version "1.0";
declare function local:expandPUAChar($string as xs:string,
   $char as xs:string) as
item()* {
  if (contains($string, $char))
  then (substring-before($string, $char),
      element myChar { attribute code {string-to-
    codepoints($char)} },
      local:expandPUAChar(substring-after($string, $char),
    $char))
  else $string
};
for $input in doc("replace-characters.xml")//text()
return local:expandPUAChar($input,"&#xE001;")
```


It is possible to achieve the same effect in XQuery. Nevertheless, the effort is a little bit higher … The sample query is a recursive, user-defined function which is evoked with a string and the character to be replaced. If the character is found in the string, the substring before it is added to the output and an element <myChar> is created instead of the character. The function is evoked again with the remaining substring. If the character is not found, the string is returned.

Topics – finally!

- Introduction
- The common underpinning: XPath 2.0 data model
- General processing of XQuery / XSLT
- String and number processing
- IRI processing
- Dates, timezones, language information
- Generating output: serialization

W3C®

Wrap up: Is it useful? Yes!

- QT: a power tool for i18n sensitive XML processing
- Quite hard to digest, but very tasty
- Some aspects of i18n related processing might be improved
- Remember:

It's still a set of working drafts ...

To summarize: XQuery and XSLT are a power tool for i18n sensitive processing of XML data. Some parts of this meal are really hard to digest. But it is worth it. Some aspects which are important for i18n sensitive processing might be improved. But always remember: It's still a set of working drafts, and there is still room for improvement!

W3C®

I18n Sensitive Processing with XQuery and XSLT

Felix Sasaki
World Wide Web Consortium