XSLT 1.0 Pocket Reference

XSLT is an essential tool for converting XML into other kinds of documents: HTML, PDF files, other XML vocabularies, and more. XSLT is also a critical technology for XML processing in Java, C#, VB.NET, Perl, Python, and others, as well as most web browsers and many authoring tools.

The *XSLT 1.0 Pocket Reference* will help you understand XSLT and use it effectively. With its concise approach, this handy pocket reference gets you up to speed quickly on XSLT 1.0 so you can transform XML like a seasoned pro. In addition to covering the basics of stylesheet structure, it also explains how to:

- Use template rules
- Create a result tree
- Apply conditional processing
- Transform multiple source documents
- Apply number formatting
- Extend XSLT’s capabilities with EXSLT
- Compare EXSLT with new features in the upcoming XSLT 2.0

The *XSLT 1.0 Pocket Reference* will deliver just what you need to get the job done in a timely fashion.
Contents

1. Data Model 1
   Node Types 1
   Node Properties 2

2. The XPath Language 6
   Data Types 6
   Expression Context 7
   Kinds of Expressions 8
   Node-Set Expressions 9
   Number Expressions 19
   String Expressions 19
   Boolean Expressions 20

3. How XSLT Works 23
   Stylesheet Structure 23
   Processing Model 24
   Template Rule Content 35
   Whitespace Stripping 39
   Serializing the Result Tree 40
   XSLT Elements by Use Case 41
4. Elements
Top-Level Elements 42
Instructions 43
Special Elements 44
QNames 45
Element Reference 45

5. Functions
XPath Functions 89
XSLT Functions 90
Data Type Conversions 90
Function Reference 94

6. Extending XSLT
Extension Functions 108
Extension Elements 109
EXSLT 110
EXSLT—Common 113
EXSLT—Dates and Times 116
EXSLT—Functions 129
EXSLT—Math 133
EXSLT—Random 136
EXSLT—Sets 138
EXSLT—Dynamic 140
EXSLT—Regular Expressions 148
EXSLT—Strings 152

Appendix A. XPath 1.0 Grammar 159
Appendix B. XSLT Pattern Grammar 164
Index 165
CHAPTER 1

Data Model

XSLT is a language for transforming XML documents. The input to an XSLT program (a "stylesheet") is one or more XML documents. The output is another document, which may be XML, HTML, or text. XSLT operates on an abstraction of XML, called the XSLT data model (the XPath data model with some additions). XSLT is "closed" over this data model. In other words, its data model applies both to its input and its output. In fact, it even models the stylesheet, which is itself expressed in XML.

TIP
Unless explicitly followed by "2.0," whenever this book speaks of “XSLT” or “XPath,” it is referring to the 1.0 versions of these languages.

Node Types

The XPath data model describes an XML document as a tree of nodes. There are seven types of nodes:

- root
- text
- element
- attribute
- processing instruction
- namespace
- comment

TIP
Easy Computing
In the XPath 1.0 data model, all XML documents have a single root node, which is an invisible container for the entire document. The root node is not an element.

**TIP**

XPath 2.0 uses the term “document node” instead of “root node.” Regardless of what it’s called, don’t confuse it with the “root element” or “document element,” which is an element: a child of the root node, or document node.

There is one element node for each element, one attribute node for each attribute (excluding namespace declarations), one comment node for each comment, and one processing instruction node for each processing instruction (PI) that occurs in an XML document. A contiguous sequence of character data, after expanding all entities and CDATA sections, is modeled as a single text node. Finally, there is a namespace node attached to each element for each namespace/prefix binding that is in scope on that element. Each element has its own unique set of namespace nodes, which always includes at least one namespace node that corresponds to the implicit mapping between the prefix “xml” and the URI “http://www.w3.org/XML/1998/namespace” (reserved for attributes such as xml:lang and xml:space).

**TIP**

Thus, even for a document that does not explicitly use namespaces, there will be as many namespace nodes as there are elements.

**Node Properties**

Table 1 lists four node properties and their applicability for each type of node. These properties deal with a node’s relationship to other nodes. If a table cell is grayed out, that means the property is not applicable for that node type.
In the XPath language, to access a node’s parent, child nodes, attributes, or namespace nodes, use the corresponding axis: parent, child, attribute, or namespace. See the section “Axes” in Chapter 2.

**TIP**

Attributes and namespace nodes are not children. An element is considered to be the parent of an attribute or namespace node, but the attribute or namespace node is not considered to be the element’s child.

The *descendants* of a node consist of the node’s children, its children’s children, and so on.

All nodes, regardless of their type, have a *string-value* and a *base URI*. Some types of nodes have an *expanded-name*, which consists of two strings: a local part and a namespace URI. Element nodes have an optional *unique ID*. For each of the string-typed node properties, Table 2 lists the node types it applies to and how its value is determined. Once again, if a table cell is grayed out, that means the property is not applicable for that node type.

---

**Table 1. Node relationship properties**

<table>
<thead>
<tr>
<th>Node type</th>
<th>Parent</th>
<th>Children</th>
<th>Attributes</th>
<th>Namespace nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td></td>
<td>Ordered list of 0 or more elements, PIs, comments, and text nodes</td>
<td></td>
<td>Unordered list of 1 or more namespace nodes</td>
</tr>
<tr>
<td>Element</td>
<td>Element or root</td>
<td>Unordered list of 0 or more attribute nodes</td>
<td>Unordered list of 1 or more namespace nodes</td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td>&quot; &quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>&quot; &quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>&quot; &quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Element</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Namespace</td>
<td>&quot; &quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The XPath language provides functions for directly accessing most of these properties. To access the string-value of a node, use the `string()` function.

<table>
<thead>
<tr>
<th>Node type</th>
<th>String-value</th>
<th>Expanded-name (local/URI)</th>
<th>Base URI</th>
<th>Unique ID</th>
<th>Unparsed entity URIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>Concatenation of descendant text nodes’ string-values, in document order</td>
<td>URI of the document entity</td>
<td></td>
<td>A set of mappings between declared entity names and their URIs</td>
<td></td>
</tr>
<tr>
<td>Element</td>
<td>&quot;</td>
<td>Local: local name</td>
<td>URI: namespace name</td>
<td>URI of external entity; otherwise, base URI of root</td>
<td>Value of attribute declared as type ID in DTD (optional)</td>
</tr>
<tr>
<td>PI</td>
<td>Text following PI target and whitespace</td>
<td>Local: PI target</td>
<td>URI: null</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>Content of comment</td>
<td>URI: null</td>
<td></td>
<td>Base URI of parent node</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>Character data (at least one character)</td>
<td>URI: null</td>
<td></td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Normalized attribute value</td>
<td>Local: local name</td>
<td>URI: namespace name</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>Namespace</td>
<td>Namespace URI</td>
<td>Local: namespace prefix</td>
<td>URI: null</td>
<td>&quot;</td>
<td></td>
</tr>
</tbody>
</table>
It’s not usually necessary to use `string()` explicitly, thanks to XPath’s automatic conversion of data types. See the “Data Type Conversions” section in Chapter 5.

To access the local and namespace URI parts of a node’s expanded-name, use the `local-name()` and `namespace-uri()` functions, respectively.

The base URI property is used for resolving relative URIs in a document, and it is used by XSLT’s `document()` function and the `xsl:import` and `xsl:include` elements. XSLT/XPath 1.0 does not provide a direct way to access the base URI property.

XPath 2.0, however, includes a function, `base-uri()`, for directly accessing the base URI of a given node. It also uses the `xml:base` attribute to determine the base URI of a node (unlike XSLT 1.0).

The unique ID property is queried by the `id()` function to retrieve elements according to their ID value. There is no function to access the unique ID property directly, but that is not normally necessary, since you can easily access an element’s attribute values using the attribute axis.

Finally, use the `unparsed-entity-uri()` function to retrieve the URI of an unparsed entity with a given name.

All of XPath and XSLT’s built-in functions are described in Chapter 5.
CHAPTER 2

The XPath Language

XPath 1.0 is the expression language embedded in XSLT 1.0. This chapter lists the primitives of XPath, including its four data types and how expressions of each type are evaluated.

Data Types

An XPath expression returns a value that has one of the following four data types:

- Node-set (an unordered collection of zero or more nodes without duplicates)
- Number (a floating-point number)
- String (a sequence of Unicode characters)
- Boolean (true or false)

Result Tree Fragments

XSLT adds one more data type to this list, the result tree fragment, which is created when an xsl:variable or xsl:param instruction uses content rather than the select attribute to define a variable. For example, given the following instruction, the value of $var will be a result tree fragment:

```xml
<xsl:variable name="var">
  <foo>text</foo>
  <bar/>
</xsl:variable>
```
A result tree fragment behaves like a node-set that contains one root node, except that certain operations allowed on node-sets are forbidden on result tree fragments. Result tree fragments can be converted to strings and copied to the result tree like regular node-sets, but the /, //, and [] operators are disallowed. Thus, the expression $var/foo is illegal. The only way to select the above foo element in XSLT 1.0 is to use an extension function that first converts the result tree fragment to a node-set, as in exsl:node-set($var)/foo. See Chapter 6.

**TIP**

XSLT 2.0 removes this restriction, abolishing the data type “result tree fragment” altogether. In XSLT 2.0, you can create “temporary trees” that are normal node-sets, so that, given the above example, the expression $var/foo by itself is legal and requires no extension functions.

**Expression Context**

All XPath expressions are evaluated in a context. The context consists of everything not present in the expression itself that may affect the resulting value of the expression. It consists of six parts:

- The **context node**
- The **context size**, an integer 1 or above (returned by the `last()` function)
- The **context position**, an integer 1 or above (returned by the `position()` function)
- A set of namespace/prefix declarations in scope for the expression
- A set of variable bindings
- A function library
XSLT specifies how the XPath context is initialized. See the “Processing Model” section in Chapter 3.

The namespace/prefix bindings and variable bindings are determined respectively by the namespace declarations and xsl:variable or xsl:param instructions in the XSLT stylesheet that are in scope for the attribute that contains the XPath expression. The function library consists of all of XPath and XSLT’s built-in functions, as well as any extension functions supported by the XSLT processor.

The context of a sub-expression may differ from the context of the whole expression. In particular, the expression context changes inside predicates. See the upcoming section “How Location Paths Are Evaluated.”

**Kinds of Expressions**

Table 3 shows an exhaustive list of the kinds of expressions in XPath 1.0. All XPath expressions fall into one of these eight categories.

<table>
<thead>
<tr>
<th>Expression category</th>
<th>Data type returned</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable reference</td>
<td>Any</td>
<td>$foo, $bar, etc.</td>
</tr>
<tr>
<td>Function call</td>
<td>Any</td>
<td>starts-with($str, &quot;a&quot;)</td>
</tr>
<tr>
<td>Function call</td>
<td>Any</td>
<td>true()</td>
</tr>
<tr>
<td>Function call</td>
<td>Any</td>
<td>round($num)</td>
</tr>
<tr>
<td>Parenthesized expression</td>
<td>Any</td>
<td>//para</td>
</tr>
<tr>
<td>Parenthesized expression</td>
<td>Any</td>
<td>foo</td>
</tr>
<tr>
<td>Literal</td>
<td>String</td>
<td>&quot;foo&quot;, 'bar', etc.</td>
</tr>
<tr>
<td>Number</td>
<td>Number</td>
<td>13, 24.7, .007, etc.</td>
</tr>
<tr>
<td>Node-set expression</td>
<td>Node-set</td>
<td>html/body/p[2]/text()</td>
</tr>
<tr>
<td>Node-set expression</td>
<td>Node-set</td>
<td>//person</td>
</tr>
<tr>
<td>Node-set expression</td>
<td>Node-set</td>
<td>//note</td>
</tr>
<tr>
<td>Node-set expression</td>
<td>Node-set</td>
<td>$ns[@id='xyz']</td>
</tr>
</tbody>
</table>
Node-Set Expressions

Variable references start with a dollar sign ($) and refer to variables or parameters in the XPath expression context. Function calls are explained in Chapter 5 in the “Function Reference” section. Parentheses can be placed around any expression in order to control operator precedence. The remaining kinds of XPath expressions are covered next under the node-set, string, number, and boolean categories.

Table 3. Examples of different XPath expressions (continued)

<table>
<thead>
<tr>
<th>Expression category</th>
<th>Data type returned</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic expression</td>
<td>Number</td>
<td>(($x - 5) * 2) div -3</td>
</tr>
<tr>
<td>Boolean expression</td>
<td>Boolean</td>
<td>$x &lt;= 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$pos mod 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$is-good and $is-valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>position() != last()</td>
</tr>
</tbody>
</table>

Node-Set Expressions

Node-set expressions include:

• Location paths
• Union expressions—union of two node-set expressions using the union (|) operator
• Filtered expressions—a predicate applied to any expression using the predicate operator ([...])
• Path expressions (any expression composed with a location path using the / or // operators)

The location path is the most important kind of expression in XPath.

Location Paths

A location path is an expression that selects nodes relative to the context node. There are two kinds of location paths: relative and absolute.
A relative location path consists of one or more steps separated by `/`. An absolute location path consists of `/`, followed by an optional relative location path. A `/` by itself selects the root node of the document that contains the context node.

**TIP**

This is usually the root node of the source tree, but it may be a different root node—for example, when using the `document()` function to select secondary input documents.

An absolute location path (other than `/` by itself) is evaluated by evaluating the relative location path that follows the `/`, using the root node as the context node.

**Steps**

A step consists of three parts: an axis, a node test, and zero or more predicates (see Figure 1).

![Figure 1. Syntax of a step](image)

The axis specifier in the previous example is `child::`, the node test is `paragraph`, and the predicate is `[string-length(.) > 100]`. From left to right, here is how a step is evaluated:

1. The axis identifies a set of nodes relative to the context node.
2. The node test acts as a filter on that set.
3. Each of any number of optional predicates in turn acts as a filter on the result identified by the preceding predicates and node test.
Axes

The axis of a step determines in what “direction” nodes are selected, starting at the context node. It identifies a set of possible nodes to select in relation to the context node. There are 13 XPath axes. Table 4 lists what each axis contains, relative to the context node.

Table 4. What each XPath axis contains

<table>
<thead>
<tr>
<th>Axis</th>
<th>What it contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>child</td>
<td>The children of the context node.</td>
</tr>
<tr>
<td>descendant</td>
<td>The descendants of the context node; i.e., children, children’s children, etc.</td>
</tr>
<tr>
<td>parent</td>
<td>The parent of the context node, if there is one.</td>
</tr>
<tr>
<td>ancestor</td>
<td>The ancestors of the context node; i.e., parent, parent’s parent, etc.</td>
</tr>
<tr>
<td>following-sibling</td>
<td>All nodes with the same parent as the context node that come after the context node in document order (empty if the context node is an attribute or namespace node).</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>All nodes with the same parent as the context node that come before the context node in document order (empty if the context node is an attribute or namespace node).</td>
</tr>
<tr>
<td>following</td>
<td>All nodes after the context node in document order, excluding descendants, attributes, and namespace nodes.</td>
</tr>
<tr>
<td>preceding</td>
<td>All nodes before the context node in document order, excluding ancestors, attributes, and namespace nodes.</td>
</tr>
<tr>
<td>attribute</td>
<td>The attributes of the context node (empty if context node is not an element).</td>
</tr>
<tr>
<td>namespace</td>
<td>The namespace nodes of the context node (empty if context node is not an element).</td>
</tr>
<tr>
<td>self</td>
<td>Just the context node itself.</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>The context node and descendants of the context node.</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>The context node and ancestors of the context node.</td>
</tr>
</tbody>
</table>
Node tests

The node test part of a step tests each node on the given axis for its name or node type. If a node passes the test, it is included in the resulting node-set. Otherwise, it is filtered out. There are two basic kinds of node test—name tests and node type tests:

- Name tests
  - Wildcard (any name): *
  - Namespace-qualified wildcard (any local name within a particular namespace): xyz:*, abc:*, etc.
  - QName (specific name): foo, xyz:foo, etc.

- Node type tests
  - Any node: node()
  - Specific type: text(), comment(), processing-instruction()
  - Specific PI target: processing-instruction('foo'), etc.

Name tests only select element nodes, except on the attribute and namespace axes where they only select attributes and namespace nodes, respectively. For example, child::* only returns elements (on the child axis), but attribute::* only returns attributes.

**WARNING**

One notable consequence of this rule is that self::* will be empty if the context node is not an element node. For example, to select all attributes except those named foo, you might think that this would work: @*[not(self::foo)]. No; name tests on the self axis only select elements. Instead, you will have to use @*[not(local-name()='foo')].

A QName name test is expanded using the set of namespace/prefix bindings in the XPath expression context. The default namespace of the stylesheet (declared by xmlns) is not included in the context. This means that if a QName name
function calls, 161
lexical structure of expressions, 162
location paths, 159
location steps, 159
namespaces, 163
node set expressions, 161
node tests, 160
numeric expressions, 161
predicates, 160
qualified namespace names, 163
whitespace and name characters, 163
XPath context, 37
XPath data model, 1
XPath language, 6–22
xs:date data type, 117
xs:dateTime data type, 117
xs:duration data type, 117
xs:gDay data type, 117
xs:gMonth data type, 117
xs:gMonthDay data type, 117
xs:gYear data type, 117
xs:gYearMonth data type, 117
xs:time data type, 117
xsl:import element, 5
xsl:include element, 5
xsl:namespace-alias element, 42
xsl:namespace-set element, 42, 50
xsl:call-template instruction, 43, 45, 51
xsl:choose instruction, 43, 44, 51
xsl:comment instruction, 43, 52
xsl:copy instruction, 43, 53
xsl:copy-of instruction, 43, 54
xsl:decimal-format element, 42, 54–57
decimal-separator attribute, 56
digit attribute, 56
grouping-separator attribute, 56
infinity attribute, 55
minus-sign attribute, 55
name attribute, 55
NaN attribute, 55
pattern-separator attribute, 56
percent attribute, 56
per-mille attribute, 56
zero-digit attribute, 56
xsl:element instruction, 43, 57
name attribute, 57
namespace attribute, 57
use-attribute-sets attribute, 58
xsl:fallback instruction, 43, 58
xsl:for-each instruction, 43, 44, 60
xsl:if instruction, 43, 61
xsl:import element, 42, 61
xsl:include element, 42, 63
xsl:key element, 42, 64
xsl:message instruction, 43, 66
xsl:namespace-alias element, 42
xsl:number instruction, 43, 69
count attribute, 69
format attribute, 70
from attribute, 69
grouping-separator attribute, 56
infinity attribute, 55
name attribute, 55
NaN attribute, 55
pattern-separator attribute, 56
percent attribute, 56
per-mille attribute, 56
zero-digit attribute, 56
xsl:element instruction, 43, 57
name attribute, 57
namespace attribute, 57
use-attribute-sets attribute, 58
xsl:fallback instruction, 43, 58
xsl:for-each instruction, 43, 44, 60
xsl:if instruction, 43, 61
xsl:import element, 42, 61
xsl:include element, 42, 63
xsl:key element, 42, 64
xsl:message instruction, 43, 66
xsl:namespace-alias element, 42
xsl:number instruction, 43, 68–74
count attribute, 69
format attribute, 70
from attribute, 69
xsl:number instruction
  (continued)
grouping-separator, 72
grouping-size attribute, 72
lang attribute, 71
letter-value attribute, 72
level attribute, 69
numbering hierarchical sections and subsections, 72
numbering the items in a list, 72
separator tokens, 70
value attribute, 69
xsl:otherwise element, 44, 74
xsl:output element, 42, 75–77
cdata-section-elements attribute, 77
doctype-public attribute, 76
doctype-system attribute, 76
encoding attribute, 76
indent attribute, 77
media-type attribute, 77
method attribute, 75
omit-xml-declaration attribute, 76
output method parameters, 76
standalone attribute, 76
version attribute, 76
xsl:param element, 42, 44, 45, 77
xsl:preserve-space element, 42, 78
example, 82
xsl:processing-instruction element, 43, 78
xsl:sort element, 44, 78–80
case-order attribute, 80
data-type attribute, 80
lang attribute, 80
order attribute, 80
parameters, 79
select attribute, 79
xsl:strip-space element, 42, 78, 80
example, 82
xsl:stylesheet element, 44, 82
exclude-result-prefixes attribute, 82
extension-element-prefixes attribute, 82
xsl:template element, 42, 45, 83–84
match attribute, 83
mode attribute, 84
name attribute, 83
priority attribute, 84
xsl:text instruction, 43, 84
xsl:transform element, 44, 85
xsl:value-of instruction, 43, 86
xsl:variable element, 42, 43, 87
xsl:when element, 44, 87
test attribute, 52
xsl:with-param element, 44, 45, 88
select attribute, 88
XSLT extending, 108–158
functions, 90
how it works, 23–41