Using XPath

Querying an XML Document with XPath Expressions

garth@ggilmour.com

The XPath Standard

- XPath began as part of a standard called XSL
  - A single standard which could query, transform and format XML documents for publication (DSSL in SGML)
- Each component proved useful by itself
  - An expression based language could simplify code
    - In the same way that regular expressions simplify parsing
  - A transformation language could produce web pages
    - Who needs a formatting standard when you have HTML/CSS?
- The XSL standard was split three ways
  - The querying component became XPath
  - The transformation component became XSLT
  - The formatting objects were placed in XSL-FO

The XPath Standard

- XPath lets you address nodes within an XML document
  - This is a prerequisite for data extraction and transformation
- XPath is only half of a query language
  - You can retrieve data but not manipulate it
  - That functionality is provided by XSLT and XQuery
- XPath expressions are an alternative to SAX or DOM
  - They eliminate a lot of tedious and error prone coding
  - The approach is similar to regular expressions in Perl

XPath Data Model

- The XML parser views a document as a tree of nodes
  - Joins between multiple physical files are seamless
- The most common tree structures are DOM and XPath
  - The structure of the tree is termed the data model
- The XPath data model is organised as follows:
  - Seven node types make up the tree
  - Nodes are arranged around thirteen axis
    - Axis are either *forwards* or *reverse*
  - XPath expressions return one of four data types
    - String, number, boolean and node set
  - Nodes are found in document or proximity order

XPath Node Types

Node Type	Description
Root	The very top of the node tree (one root node per document)
Element	Created from a start tag, end tag and the enclosed content
Text	A continuous block of characters found in an element
Comment	The characters inside a pair of ‘<!-- -->’ delimiters
Processing Instruction	An instruction to the XML Parser
Attribute	An name/value pair attached to an element
Namespace	Represents an XML namespace

Directions in the XPath Node Tree

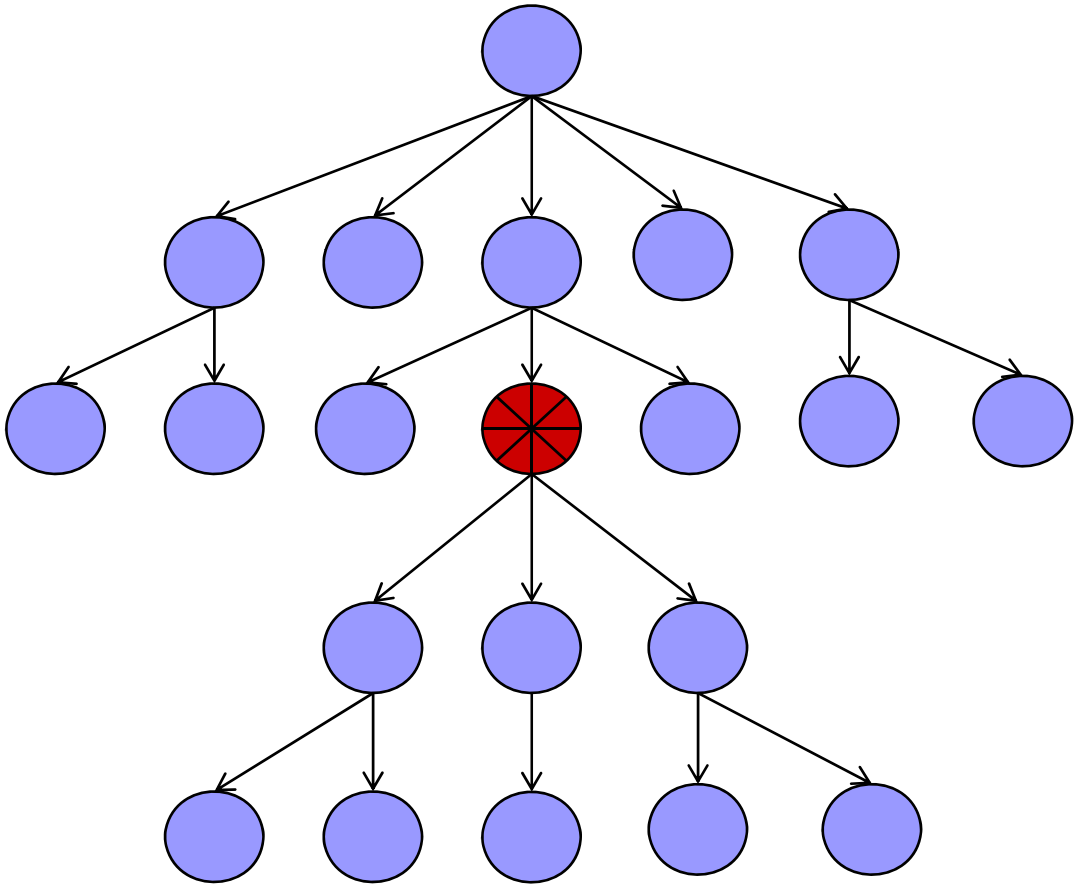
Axis Name	Description
self	Our current position within the node tree
parent	The parent of the current node (the root node has no parent)
ancestor	Obtained by collecting parent nodes from the current node to the root
ancestor-or-self	All ancestor nodes and the current node
child	All the nodes that are declared immediately inside the current node
descendant	The child nodes plus their children and their childrens children etc…
descendant-or-self	All descendant nodes and the current node
preceding-sibling	All children of the parent node that appear before the current node
following-sibling	All children of the parent node that appear after the current node
preceding	All nodes which occur before the current node
following	All nodes which occur after the current node
attribute	All attribute nodes attached to the current node
namespace	All namespace nodes attached to the current node

The Self Axis

```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The self axis always contains the current node
  - ‘self::*’ selects the current node if it is an element
  - ‘self::age’ selects the current node if it is an ‘age’ element
  - ‘self::node()’ always selects the current node
    - The abbreviation is ‘.’
  - ‘count(self::node())’ returns 1
- How the current node is determined is not specified
  - It is set by the client program

The Self Axis

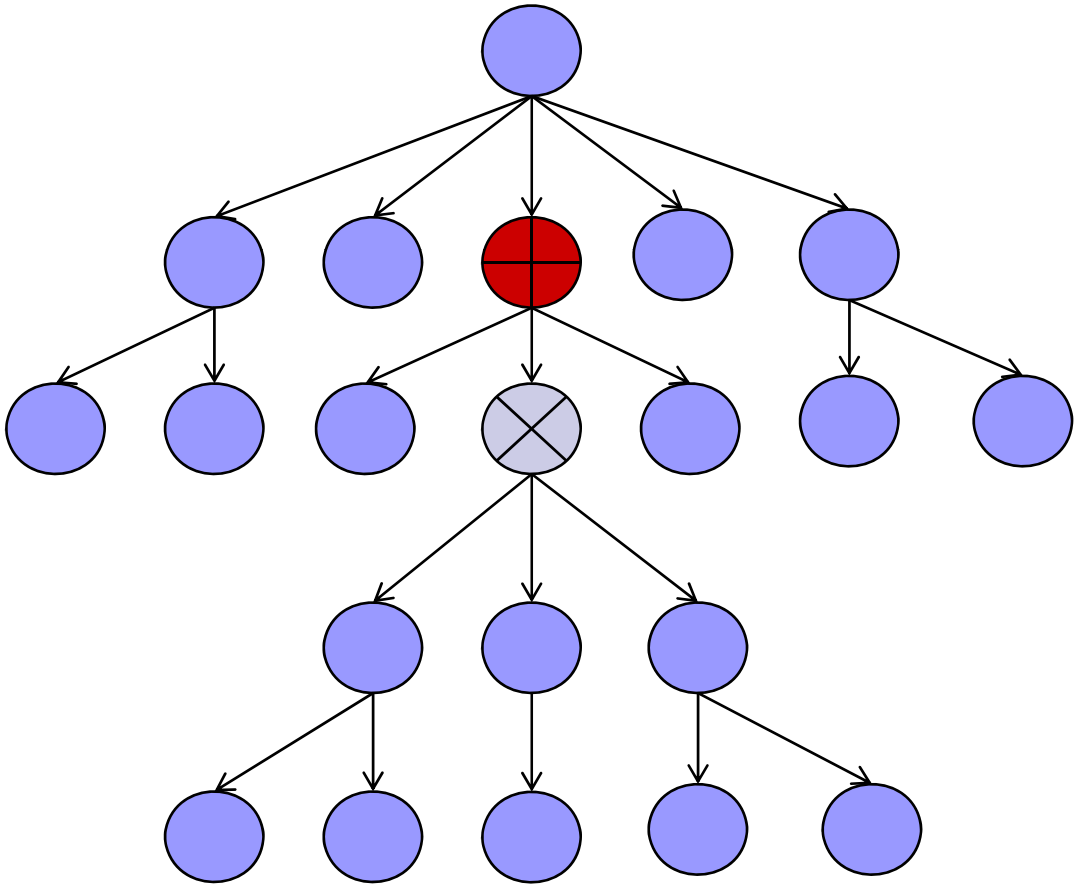


The Parent Axis

```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The parent axis contains the parent of the current node
  - ‘parent::node()’ always selects the parent node
    - The abbreviation is ‘..’
  - Root nodes don't have parents
    - ‘/..’ selects the parent of the root and hence is always an empty node set
  - Attribute and namespace nodes do not have parents
    - Because they are attached

The Parent Axis

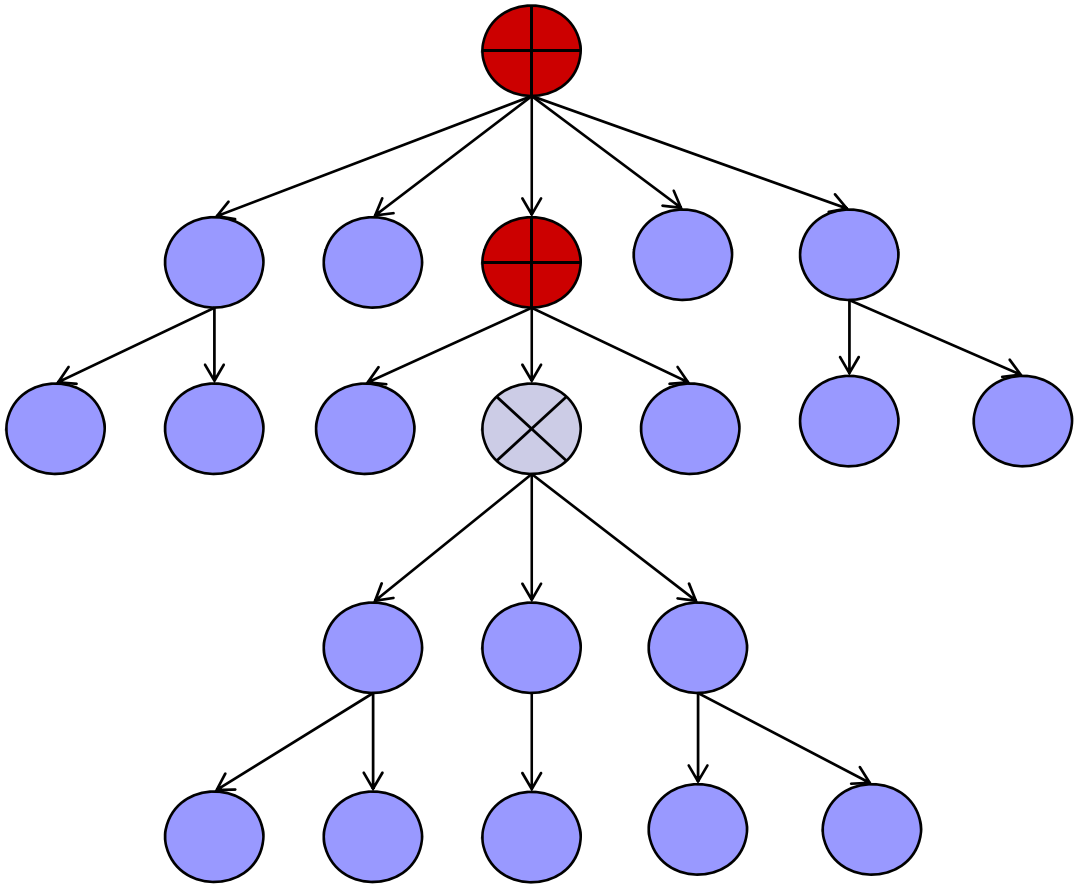


The Ancestor Axis

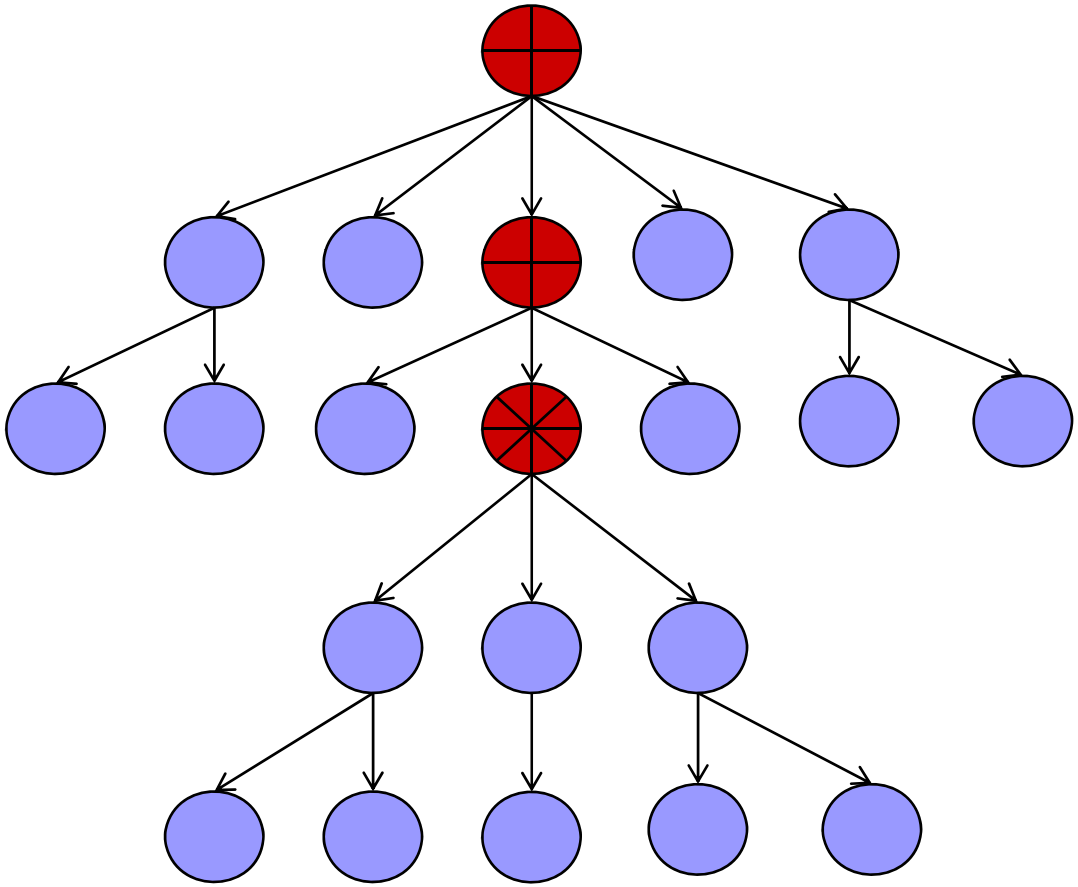
```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The ancestor axis selects parent nodes recursively
  - Starting from the current node and stopping with the root node
  - The ancestors of ‘h’ are ‘f’, ‘a’ and the root node
    - ‘count(ancestor::*)’ would return 2 for ‘h’ and 3 for the text node it contains
  - The ancestor-or-self axis includes the current node

The Ancestor Axis



The Ancestor-Or-Self Axis

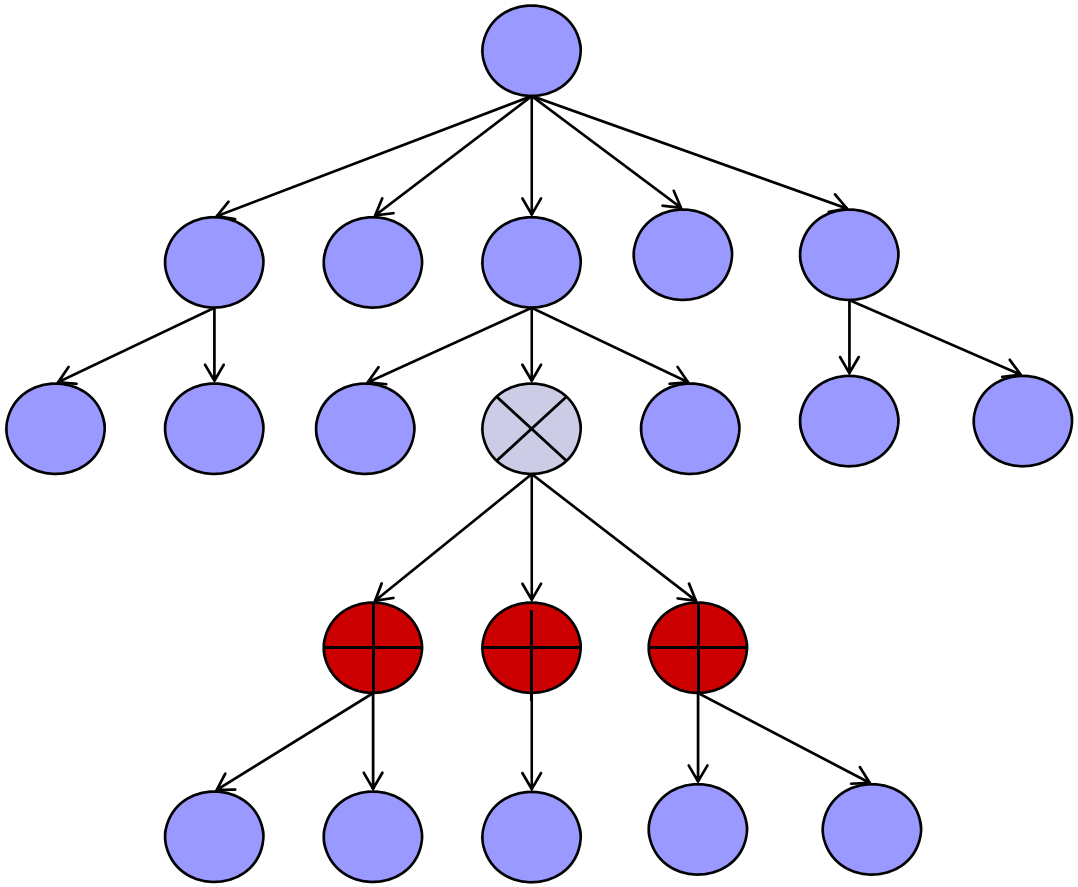


The Child Axis

```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The child axis contains children of the current node
  - But not any nested children
  - Children may be elements, comments or PI's
    - 'child::*' selects element children of the current node
    - 'child::node()' selects all children of the current node
  - The children of element node 'f' are 'g','h','i', 'j' and 'k'

The Child Axis

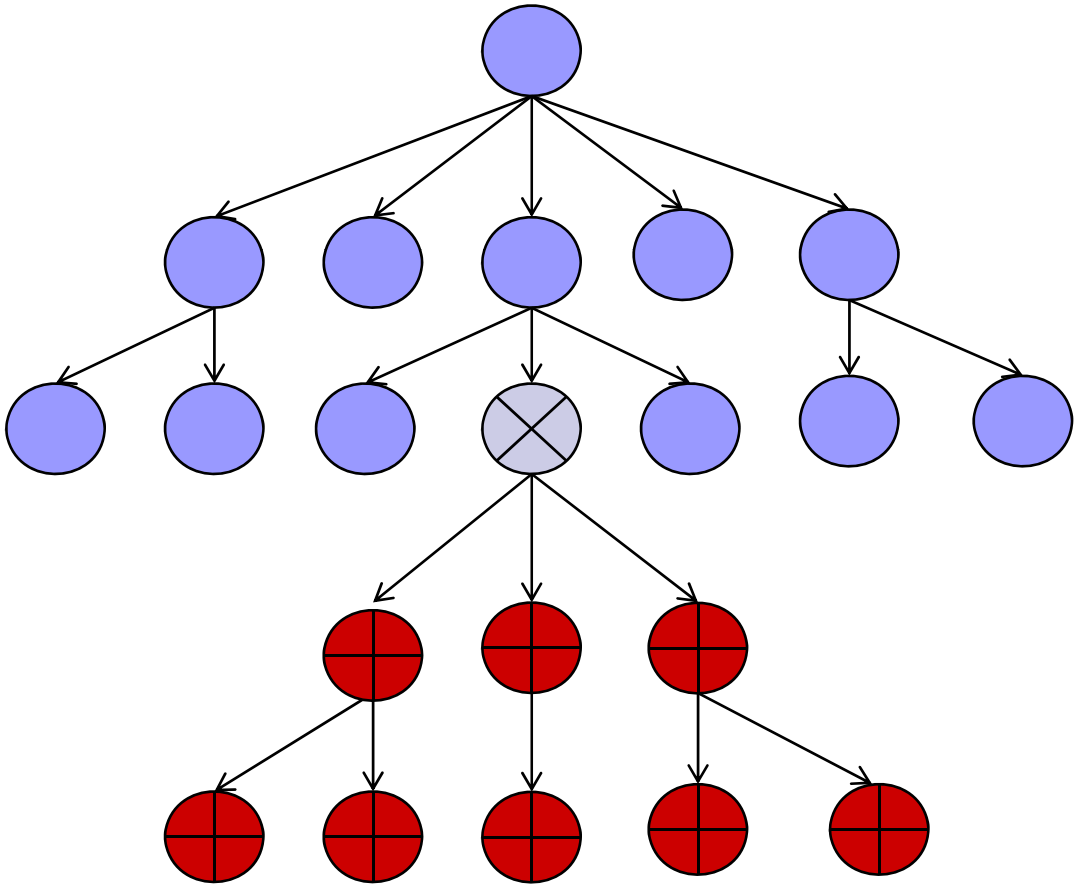


The Descendant Axis

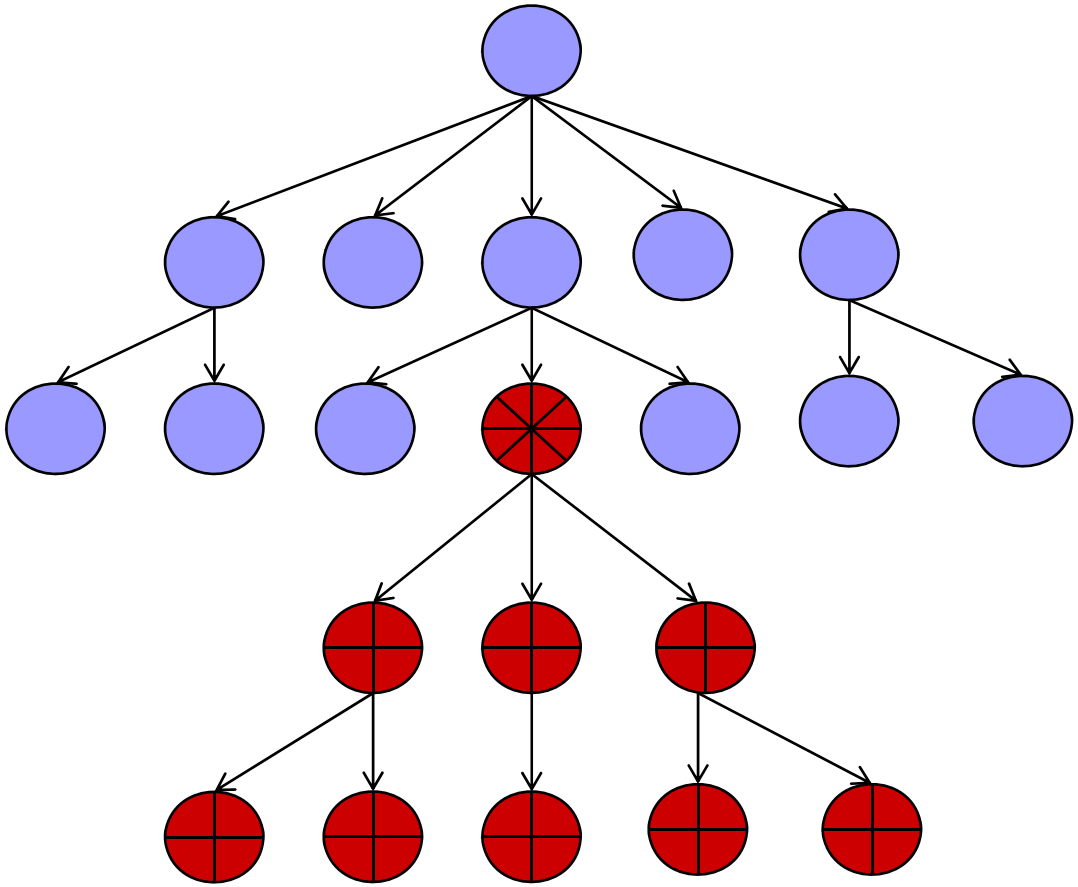
```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The descendant axis contains all the nodes declared inside the current node
  - The descendants of ‘f’ are 5 element and 5 text nodes
    - Excluding the issue of whitespace only text nodes
    - ‘count(descendant::node())’ returns 10
    - ‘count(descendant::*)’ would return 5
  - The descendant-or-self axis includes the current node

The Descendant Axis



The Descendant-Or-Self Axis

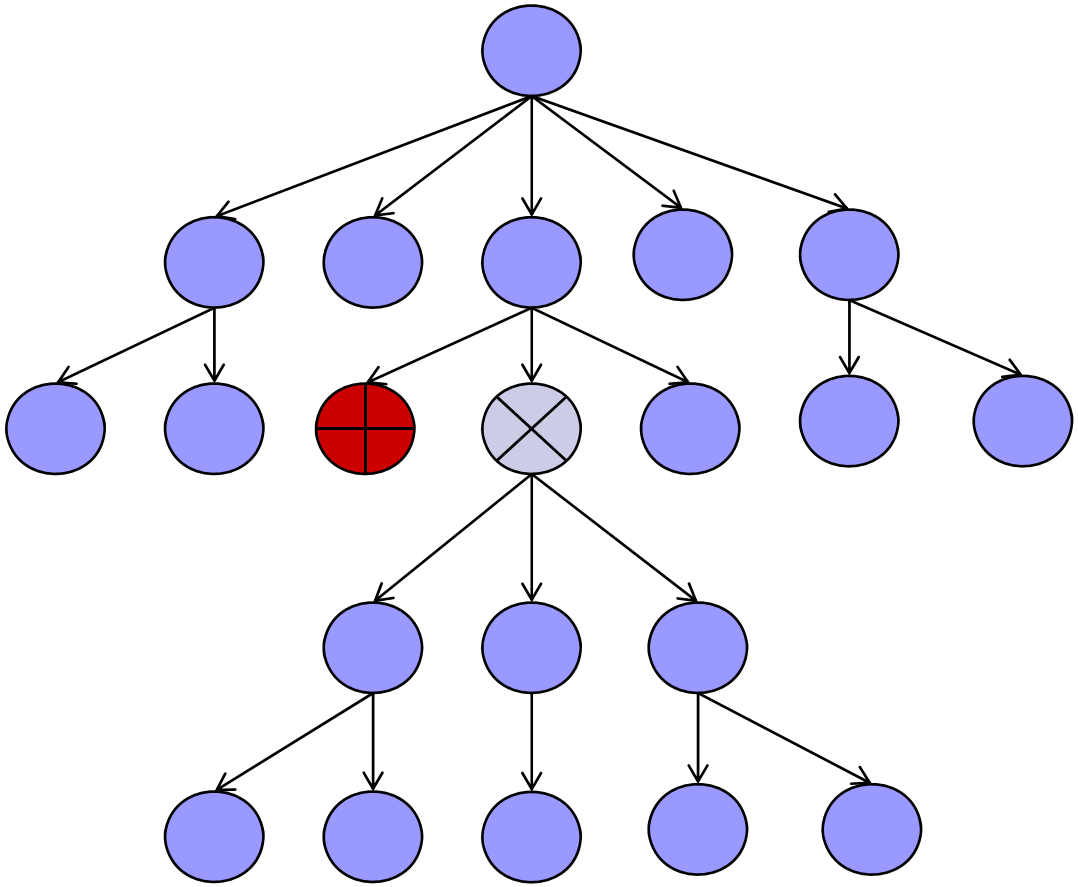


The Preceding Sibling Axis

```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The preceding sibling axis contains siblings which occur before the current node
  - The ordering is referred to as ‘document order’
  - The preceding siblings of ‘i’ are ‘g’ and ‘h’
  - These could be selected via ‘preceding-sibling::node()’ or ‘preceding-sibling::*’
    - Because in this case all the preceding siblings are elements

The Preceding Sibling Axis

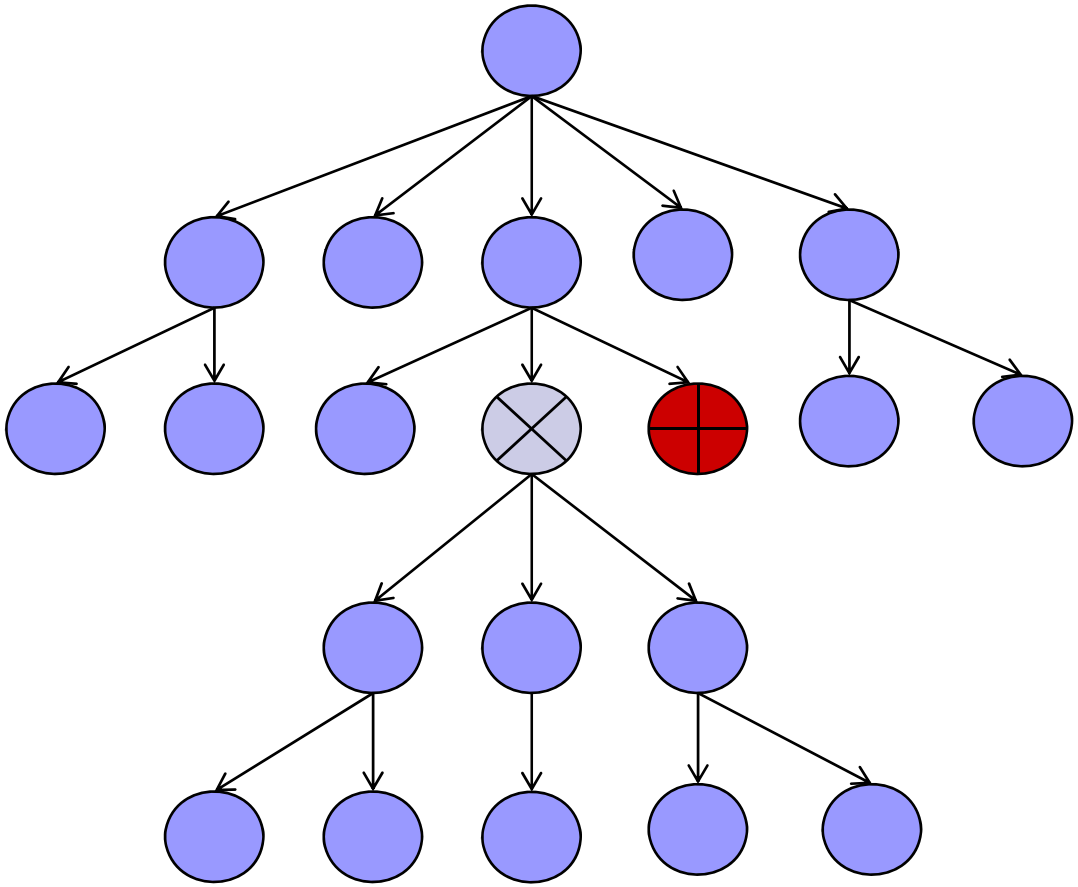


The Following Sibling Axis

```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- The following sibling axis contains siblings which occur after the current node
  - The ordering is referred to as ‘document order’
  - The following siblings of the ‘i’ element are ‘j’ and ‘k’
  - These could be selected via ‘following-sibling::node()’ or ‘following-sibling::*’
    - Because in this case all the following siblings are elements

The Following Sibling Axis

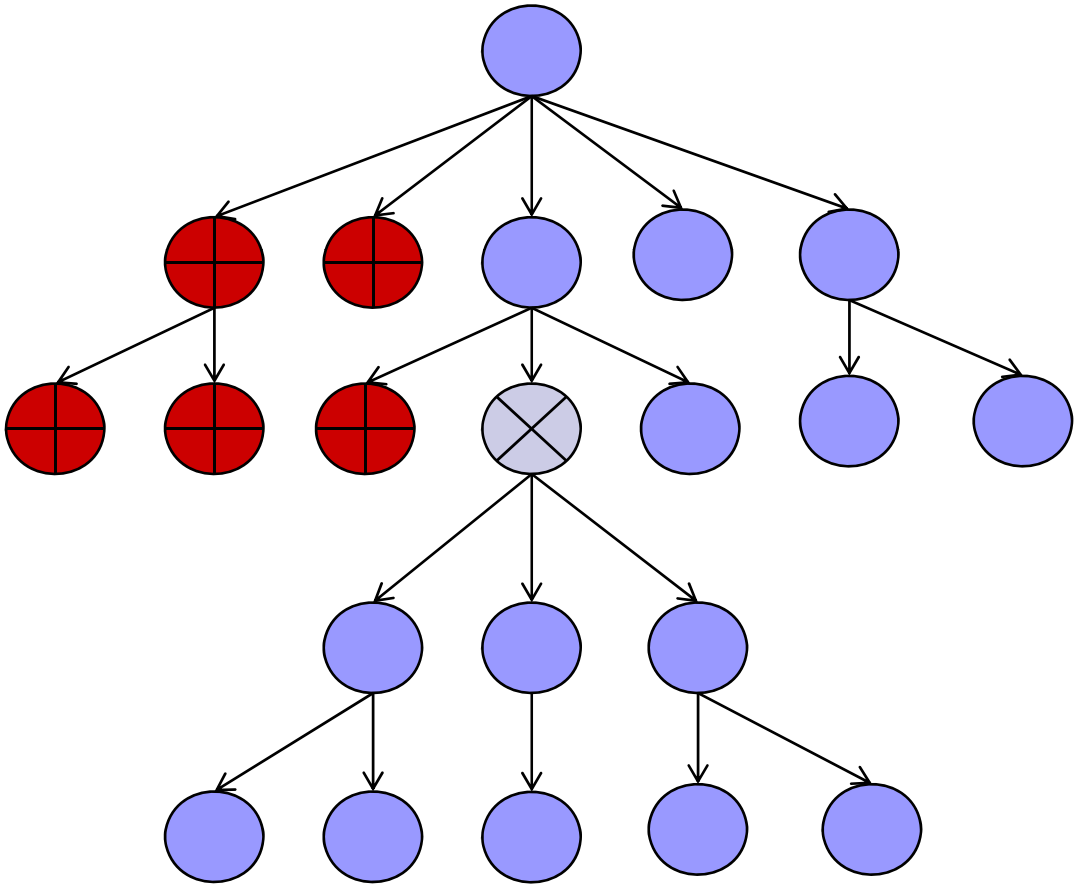


The Preceding and Following Axis

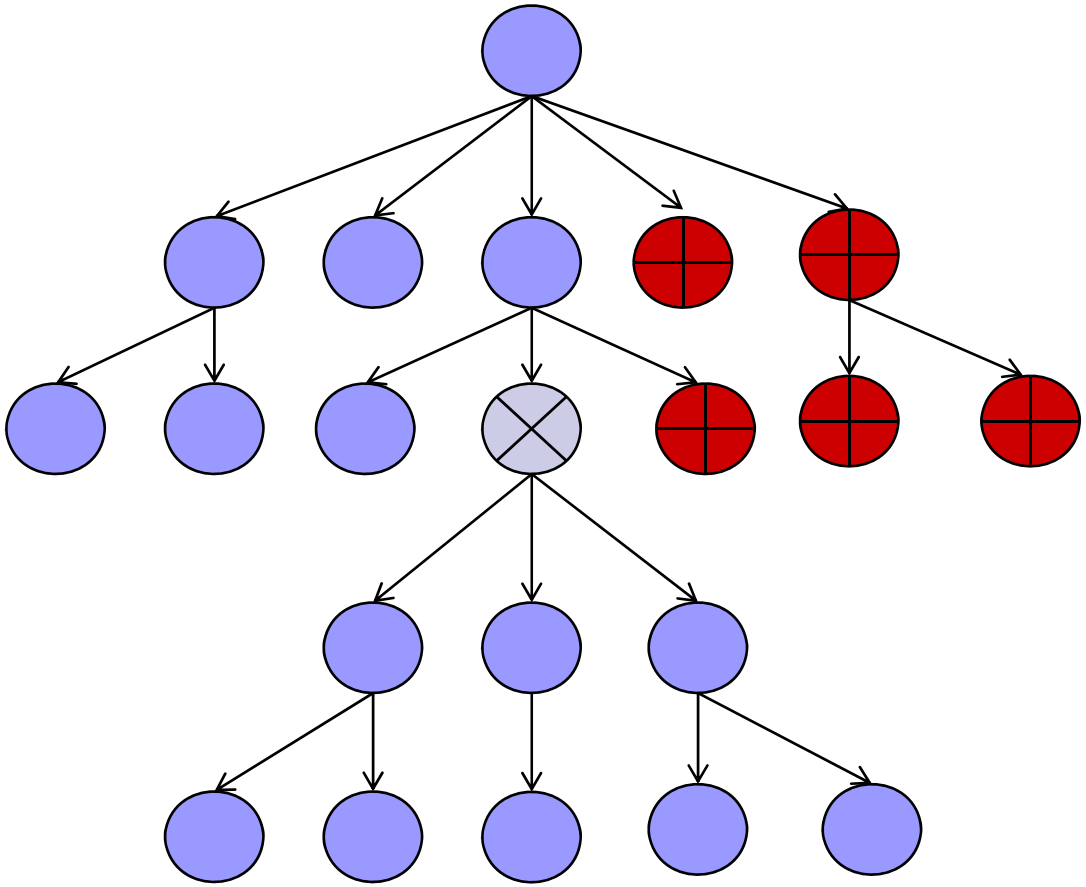
```
<a>
    <b>
        <c>
            <d>Text One</d>
            <e>Text Two</e>
        </c>
    </b>
    <f>
        <g>Text Three</g>
        <h>Text Four</h>
        <i>Text Five</i>
        <j>Text Six</j>
        <k>Text Seven</k>
    </f>
</a>
```

- These axes are hard to explain but very rarely used
  - The preceding axis contains all the nodes that come before the current node
    - For ‘g’ the xpath ‘preceding::*’ selects ‘a’,’d’,’c’ and ‘b’
  - The following axis contains all the nodes that come after the current node
    - For ‘e’ the expression ‘following::*’ selects ‘f’,’g’,’h’,’i’,’j’ and ‘b’

The Preceding Axis



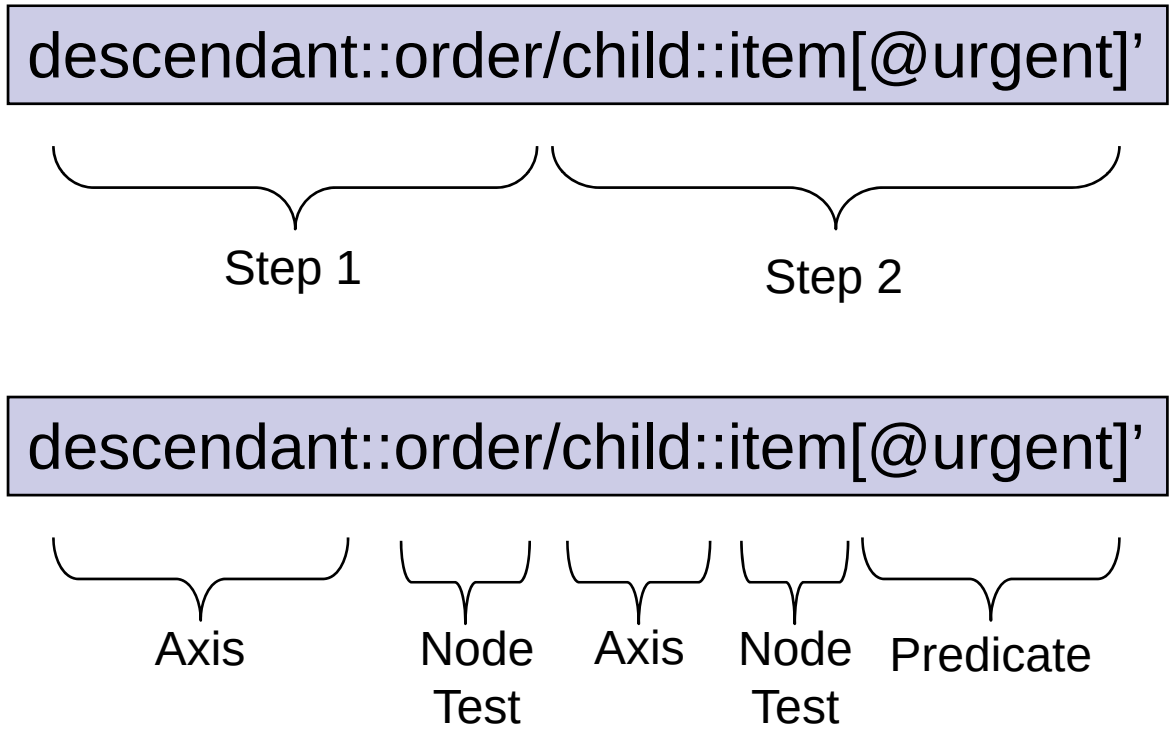
The Following Axis



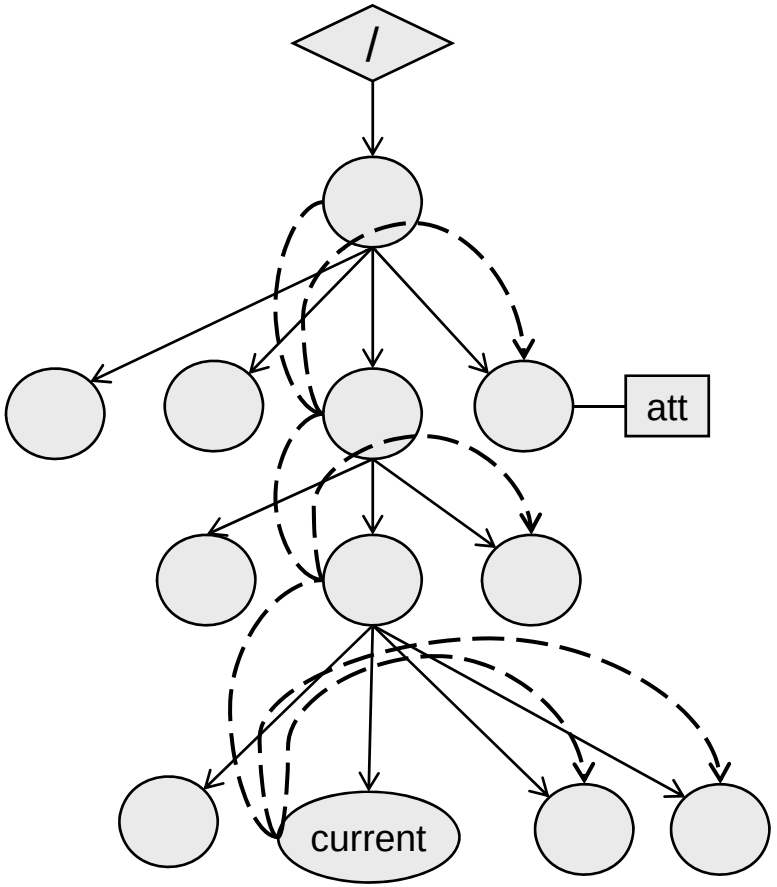
XPath Expressions

- An XPath expression is made up of steps
  - Every step contains
    - The axis to search along (child is the default)
    - The type of nodes to select
    - Zero of more predicates to filter the node set
  - For example consider ‘descendant::order/child::item[@urgent]’
    - The first step selects all elements named ‘order’ on the descendant axis and stores them in a node set
    - The second step selects all those children of the selected nodes which are item elements, these are stored in a new node set
    - Finally the predicate at the end of the second step filters out all the nodes which do not have an ‘urgent’ attribute

XPath Expressions

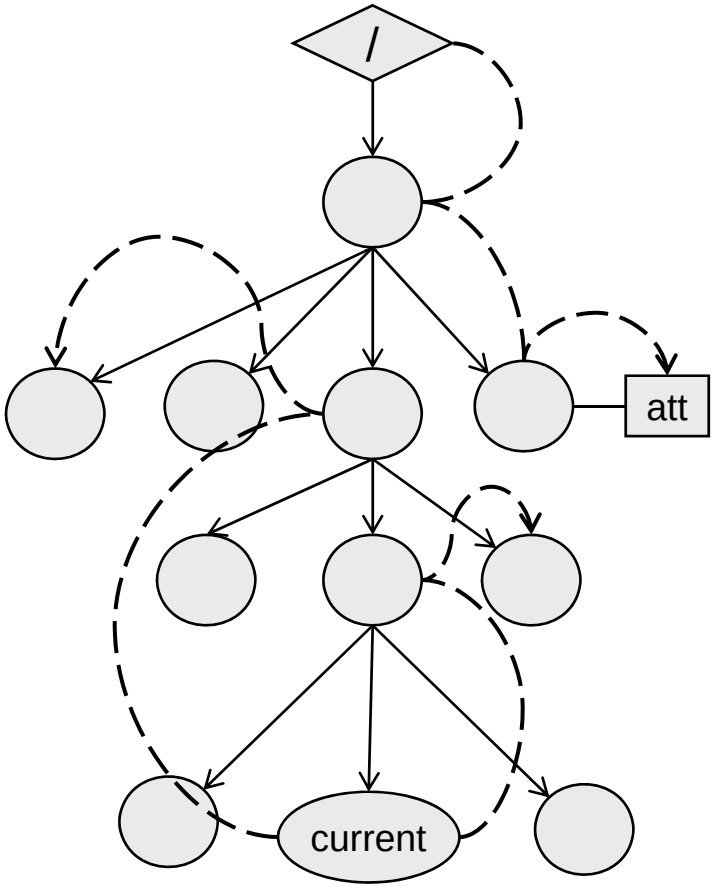


Using Steps To Find Many Nodes



ancestor-or-self::*/following-sibling::*

Using Steps To Find A Single Node



ancestor::*[2]/preceding-sibling::*[2]

parent::*/following-sibling::*[1]

/child::*/child::*[4]/attribute::att

Node Tests

- The node test can be a name or the wildcard (‘*’)
  - These select attribute nodes on the attribute axis, namespace nodes on the namespace axis and elements on all other axes
  - Both the name and the wildcard can be qualified by a namespace prefix
    - For example ‘svg:*’ or ‘math:Triangle’
- Built in functions return other Nodes on the current axis
  - The processing-instruction() function selects all PI’s
    - The function can take the name of a PI as a parameter
  - The comment() functions selects comment nodes
  - The text() function selects text nodes
  - The node() function selects nodes of any type

Simple XPath Expressions

```
child::*                              //All element nodes on the child axis
child::node()                         //All nodes on the child axis
child::text()                         //All text nodes on the child axis
child::comment()                      //All comment nodes on the child axis
child::processing-instruction()       //All PI nodes on the child axis
child::processing-instruction('abc')  //The PI child node called 'abc'

parent::order                         //The parent node if it is an 'order' element
attribute::cost                       //The attribute of the current node called 'cost'
preceding-sibling::*[text()]          //All preceding siblings containing text nodes
ancestor::*[attribute::cost]          //All ancestors with a 'cost' attribute
ancestor::*/attribute::cost           //The 'cost' attribute nodes of ancestors

/invoice                              //The document element if of type 'invoice'
//invoice                             //Every invoice element in the document
//attribute::cost                     //Every cost attribute in the whole document
```

Syntax and Abbreviations

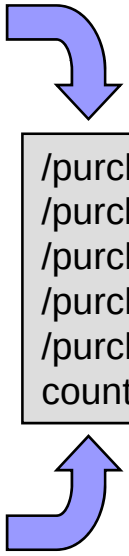
- An XPath expression can be absolute or relative
  - Absolute expressions begin with ‘/’ and search from the root
  - Relative expressions search from the current node
- Two separate expressions can be combined with ‘|’
  - This represents a union and NOT a logical OR
- Commonly used features are abbreviated
  - The default axis is child and is usually left out
  - The ‘@’ character is short for ‘attribute::’
  - The ‘.’ character selects the current node i.e. ‘self::node()’
  - The ‘..’ characters select the parent node i.e. ‘parent::node()’

Abbreviated XPath Expressions

*	//All element nodes on the child axis
node()	//All nodes on the child axis
text()	//All text nodes on the child axis
comment()	//All comment nodes on the child axis
processing-instruction()	//All PI nodes on the child axis
processing-instruction('abc')	//The PI child node called 'abc'
parent::order	//The parent node if it is an 'order' element
@cost	//The attribute of the current node called 'cost'
preceding-sibling::*[text()]	//All preceding siblings containing text nodes
ancestor::*[@cost]	//All ancestors with a 'cost' attribute
ancestor::*/@cost	//The 'cost' attribute nodes of ancestors
../item	//All item children of the parent node
./*/*	//All element grandchildren of the current node
/*/*	//All children of the document element
../*/@cost	//All 'cost' attributes of children of the parent

What Do These Expressions Do ?

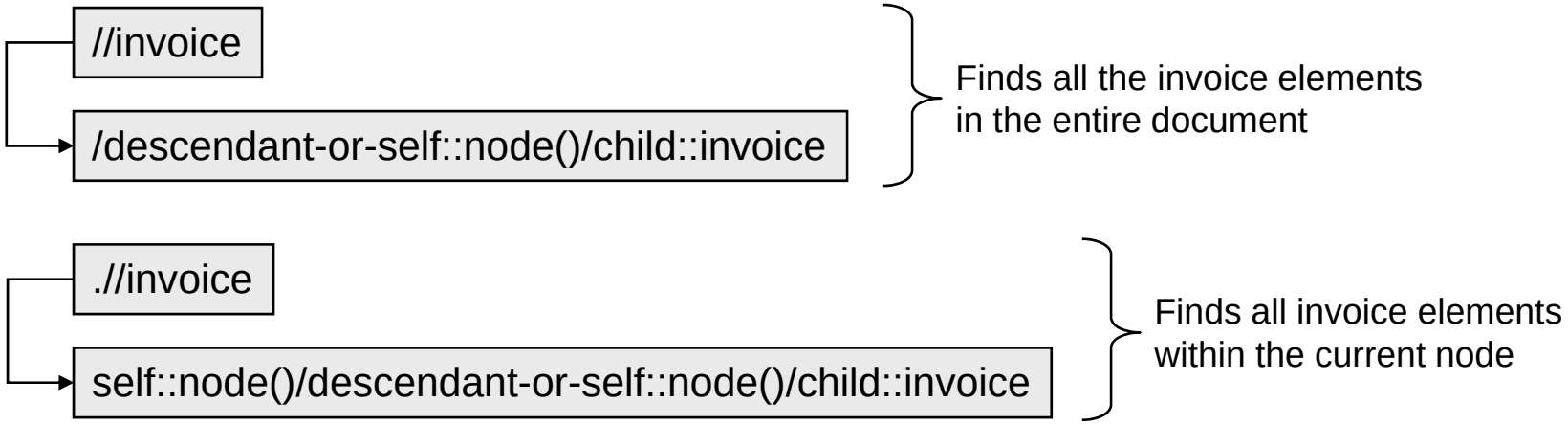
```
<purchaseOrder id="ABC123">
   <customer id="DEF456">
      <name>MegaCorp</name>
      <address postcode="BT37 ABC">
         <city>Beflast</city>
         <street no="10">Arcatia Road</street>
      </address>
      <paymentOptions>
         <category>Retail</category>
         <daysToPay>30</daysToPay>
         <creditLimit>20000</creditLimit>
      </paymentOptions>
   </customer>
   <itemsList>
      <item index="1" id="12" quantity="12">
         <description>Hard Disk</description>
      </item>
      <!-- Other items omitted -->
   </itemsList>
</purchaseOrder>
```



```
/purchaseOrder/customer/address/@postcode
/purchaseOrder/customer/paymentOptions/category
/purchaseOrder/itemsList/item[1]/description/text()
/purchaseOrder/itemsList/item[last()]/description/text()
/purchaseOrder/itemsList/item/description
count(/purchaseOrder/itemsList/item)
```

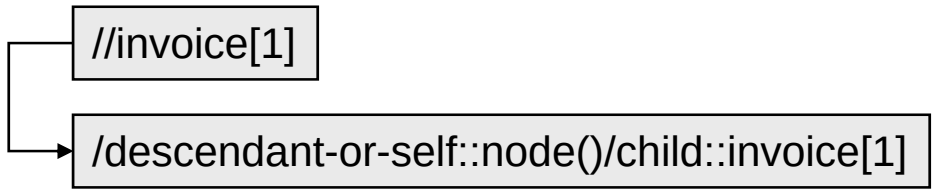
Searching Large Areas of the Tree

- The symbol '//' is short for '/descendant-or-self::node()/'
  - This is very powerful and can be used at any step
  - Avoid using it casually as it makes the engine do a lot of work



A Common Error

- Most developers expect '//invoice[1]' to find the first 'invoice' element in the document
  - Instead it finds all invoice elements which are the first invoice element child of their parent
  - Parenthesis can be used to separate the predicate from the search, producing the correct result

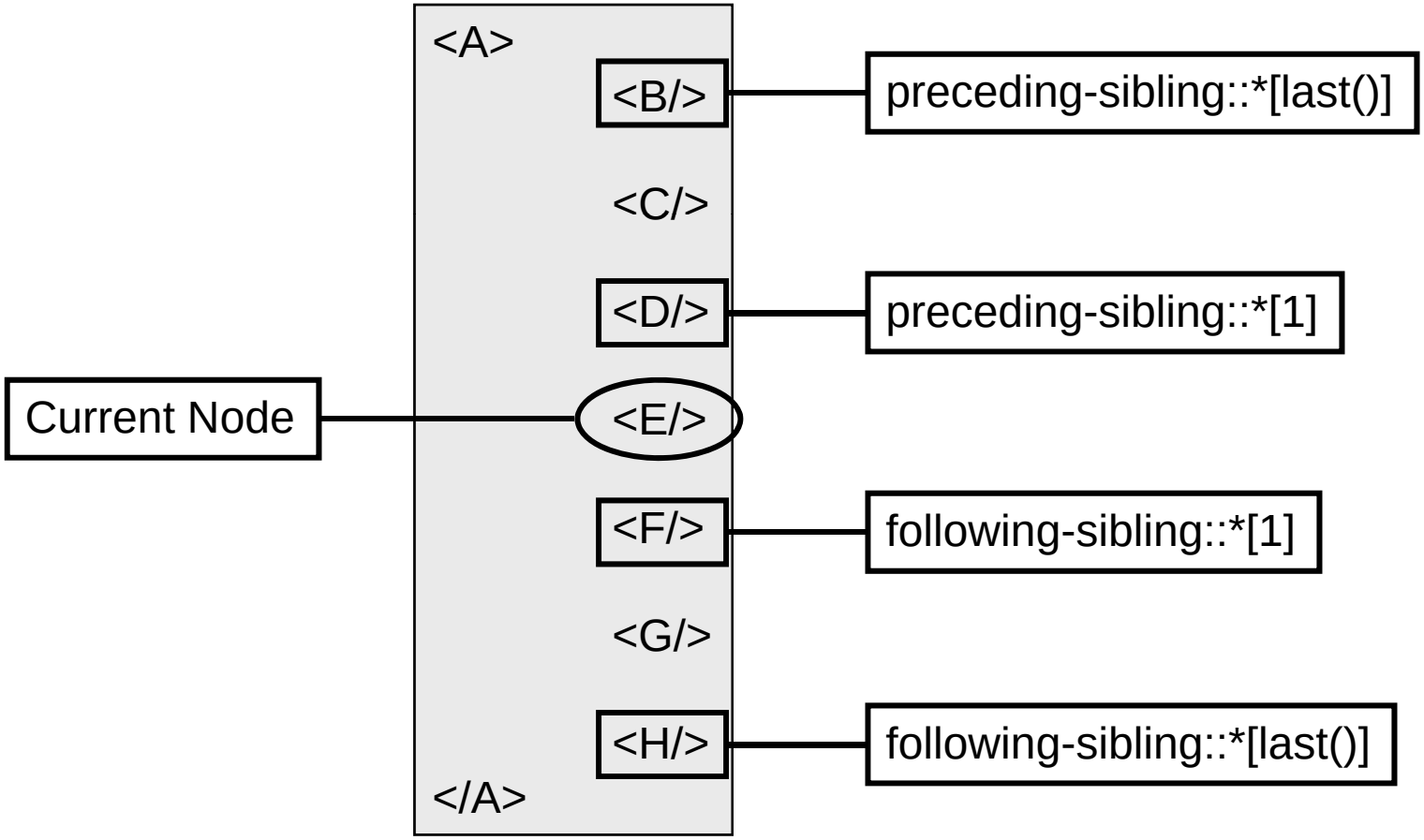



```
<accounts>
    <customer details="…">
        <invoice/>
        <invoice/>
    </customer>
    <customer details="…">
        <invoice/>
    </customer>
    <customer details="…">
        <invoice/>
        <invoice/>
    </customer>
<accounts>
```

Predicates and Positioning

- Predicates often test for position
  - ‘child::item[3]’ is short for ‘child::item[position() = 3]’
- Following axes are indexed in document order
  - So ‘following-sibling::*[1]’ returns the closest sibling to the current element in the rest of the document
  - And ‘following-sibling::*[last()]’ returns the furthest
    - The closest to the end of the document
- Preceding axes are indexed in reverse document order
  - So ‘preceding-sibling::*[1]’ returns the closest sibling to the current element in the document so far
  - And ‘preceding-sibling::*[last()]’ returns the furthest
    - The closest to the start of the document

Predicates and Positioning



Predicates and Node Tests

- Predicates can be used to test the name of an element
  - Normally you would simply hard code the element name
  - However in XSLT and XQuery you may need to iterate over a set of elements and perform behaviour according to their type
- The two common ways to perform the test are:
  - Use the ‘local-name’ function e.g. ‘//*[local-name() = ‘item’]’
  - Use the ‘self’ axis e.g. ‘//*[self::item]’
    - This is short for ‘//*[count(self::item) > 0]’

```
<xsl:for-each select="//Invoice | //Item">
    <xsl:if test="self::Invoice"> <!-- Do A --> </xsl:if>
    <xsl:if test="self::Item"> <!-- Do B --> </xsl:if>
</xsl:for-each>
```

Predicates in XPath Expressions

```
/*[@total]                          //The doc element if it has a 'total' attribute
//item[1]                          //The first 'item' child of any element
(//item)[1]                        //The first 'item' element in the document
//order/item[@cost > 5000]         //All 'items' in orders with a 'cost' over 5000
//comment()[parent::item]          //All comments in 'item' elements

//All preceding sibling 'order' elements with more than 5 'item' children
preceding-sibling::order[count(item) > 5]

//The closest preceding sibling 'order' element with a 'totalCost' attribute
preceding-sibling::order[@totalCost][1]

//The closest preceding sibling 'order' element IF it has a 'totalCost' attribute
preceding-sibling::order[1][@totalCost]

//The 'cost' attribute of the sixth 'item' child of the fifth 'order' child of the doc element
/porder/order[5]/item[6]/@cost
```

How Ordering Affects Predicates

- The order of the predicates is vital
  - The expression ‘descendant::item[@urgent][1]’ is not the same as ‘descendant::item[1][@urgent]’
    - The former selects the first element from the set of descendant elements named ‘item’ which have an ‘urgent’ attribute
    - The latter selects the first descendant ‘item’ element IF it has an ‘urgent’ attribute
- Braces can be used to separate a step from a predicate
  - The expression //item[1] selects all the ‘item’ elements which are the first child of their parent
    - Remember it expands to ‘/descendant-or-self::node()/child::item[1]’
  - Whereas (//item)[1] selects the first ‘item’ element in the whole document, which is normally what you are looking for

XPath Functions

- XPath includes a library of functions
  - To manipulate strings and numbers
  - To investigate nodes and convert data types
- The most useful functions are:
  - The count function, which counts a node set
    - For example '//order[count(item) = 20]'
  - The sum function, which converts each item in a node set to a number and totals them
    - For example 'sum(//orders/@totalCost)'
  - The translate function, which operates like Perls 'tr'
    - For example 'translate("aeiou","AEIOU",$var1)' returns a new string with the same contents as $var1, but with all the vowels capitalised
  - The format-number function, which pretty-prints numbers

Running Expressions in Java 1.5

- XPath has always been supported in Java
  - By using extension libraries like JDOM and Jaxen
- Java 1.5 adds support for XPath to JAXP
  - Via the types in the package “javax.xml.xpath”
- An XPath expression engine is found indirectly
  - Via the ‘XPathFactory’ factory class
  - The ‘XPath’ interface represents the engine
- Expressions can be run in two ways
  - By passing the expression as a String into ‘XPath.evaluate’
  - By creating and using an ‘XPathExpression’ object

Running Expressions in Java 1.5

- There are two choices when evaluating an expression
  - How will the input XML be represented?
  - In what format should the result be returned?
- There are three ways of supplying the XML
  - As an ‘InputSource’ object
  - As an ‘org.w3c.dom.Document’ object
  - As an ‘org.w3c.dom.Node’ object
    - This is essential if your XPath is a relative expression
- There are five ways of representing the queries result
  - These are a ‘Node’, ‘NodeList’, ‘Double’, ‘String’ or ‘Boolean’
  - As represented by the values declared in ‘XPathConstants’

```
InputSource input = new InputSource(new FileReader("input/purchase_order.xml"));
DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = domFactory.newDocumentBuilder();
Document document = builder.parse(input);

XPathFactory xpathFactory = XPathFactory.newInstance();
XPath xpath = xpathFactory.newXPath();

XPathExpression [] absoluteExpressions = new XPathExpression[6];
absoluteExpressions[0] = xpath.compile("/purchaseOrder/customer/address/@postcode");
absoluteExpressions[1] = xpath.compile("/purchaseOrder/customer/paymentOptions/category");
absoluteExpressions[2] = xpath.compile("/purchaseOrder/itemsList/item[1]/description/text()");
absoluteExpressions[3] = xpath.compile("/purchaseOrder/itemsList/item[last()]/description/text()");
absoluteExpressions[4] = xpath.compile("/purchaseOrder/itemsList/item/description");
absoluteExpressions[5] = xpath.compile("count(/purchaseOrder/itemsList/item)");

Attr result1 = (Attr)absoluteExpressions[0].evaluate(document,XPathConstants.NODE);
Element result2 = (Element)absoluteExpressions[1].evaluate(document,XPathConstants.NODE);
Text result3 = (Text)absoluteExpressions[2].evaluate(document,XPathConstants.NODE);
Text result4 = (Text)absoluteExpressions[3].evaluate(document,XPathConstants.NODE);
NodeList result5 = (NodeList)absoluteExpressions[4].evaluate(document,XPathConstants.NODESET);
Double result6 = (Double)absoluteExpressions[5].evaluate(document,XPathConstants.NUMBER);
```

```
InputSource input = new InputSource(new FileReader("input/purchase_order.xml"));
DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = domFactory.newDocumentBuilder();
Document document = builder.parse(input);

XPathFactory xpathFactory = XPathFactory.newInstance();
XPath xpath = xpathFactory.newXPath();

String contextString = “/purchaseOrder/itemsList/item[@index = 5]”;
Node contextNode = (Node)xpath.evaluate(contextString,document,XPathConstants.NODE);

XPathExpression [] relativeExpressions = new XPathExpression[6];
relativeExpressions[0] = xpath.compile("count(preceding-sibling::item)");
relativeExpressions[1] = xpath.compile("count(following-sibling::item)");
relativeExpressions[2] = xpath.compile("preceding-sibling::item[1]/description");
relativeExpressions[3] = xpath.compile("following-sibling::item[1]/description");
relativeExpressions[4] = xpath.compile("description/text()");
relativeExpressions[5] = xpath.compile("../item/description");

Double result1 = (Double)relativeExpressions[0].evaluate(contextNode,XPathConstants.NUMBER);
Double result2 = (Double)relativeExpressions[1].evaluate(contextNode,XPathConstants.NUMBER);
Element result3 = (Element)relativeExpressions[2].evaluate(contextNode,XPathConstants.NODE);
Element result4 = (Element)relativeExpressions[3].evaluate(contextNode,XPathConstants.NODE);
Text result5 = (Text)relativeExpressions[4].evaluate(contextNode,XPathConstants.NODE);
NodeList result6 = (NodeList)relativeExpressions[5].evaluate(contextNode,XPathConstants.NODESET);
```

XSLT Stylesheets

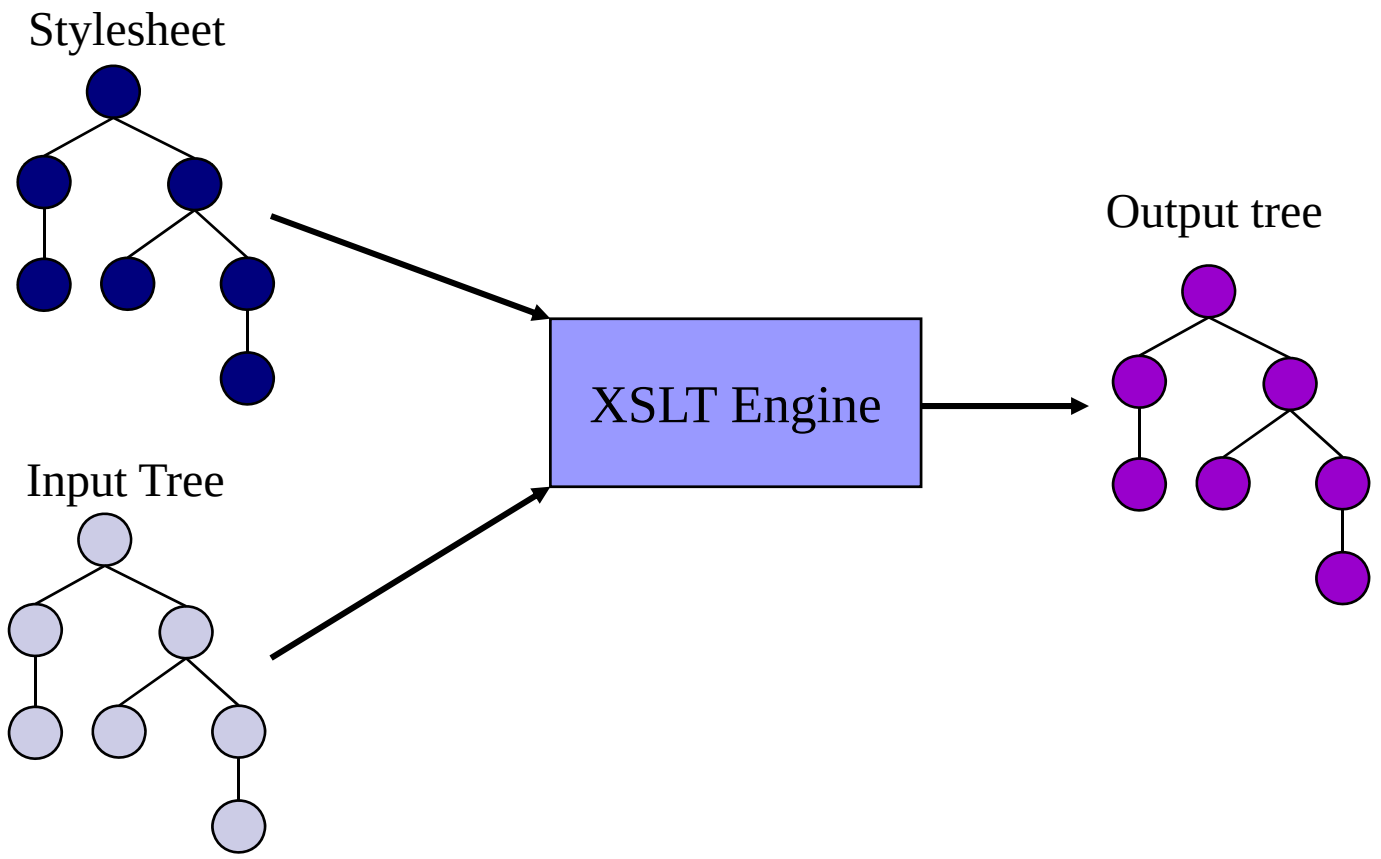
Transforming XML in your Web Application

garth@ggilmour.com

The XSLT Standard

- XSLT enables you to transform XML
  - For example from XML to HTML
- Your Web Apps can be free of presentation
  - Everything is encapsulated into stylesheets
- XSLT is advocated as an alternative to ASP/JSP
  - In truth they are complimentary technologies
- XSLT needs to address parts of the input XML
  - This ability is provided by the XPATH standard
- XQuery has recently joined the XML family
  - It performs a similar job to XSLT but in a DB oriented way

The XSLT Processor



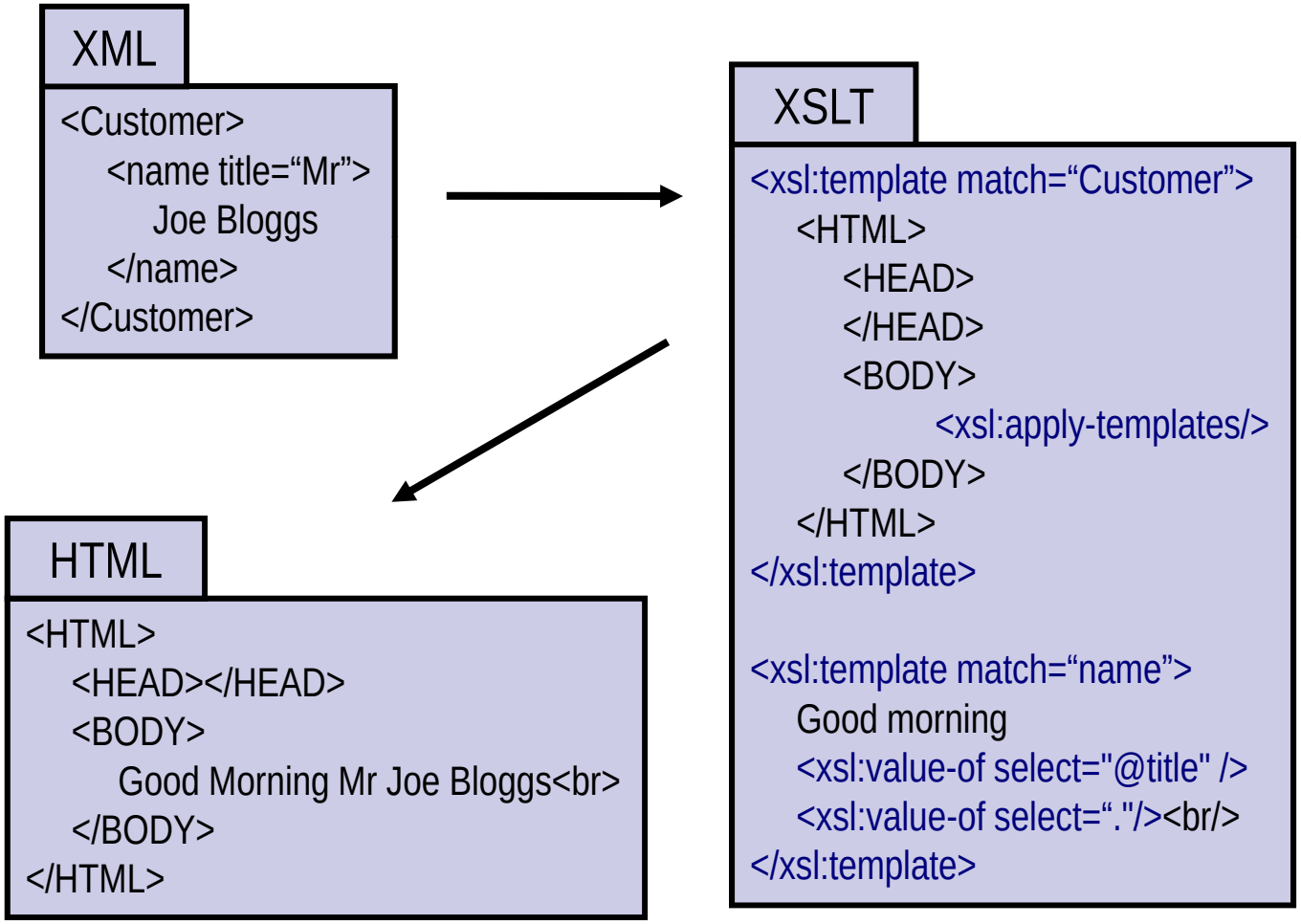
The XSLT Processor

- One logical document is the input
  - Organised according to the XPATH data model
  - The physical file structure is irrelevant
  - Most of the DTD is not important
    - Fixed and default attribute values can affect the transformation
  - Other documents can be opened from within the stylesheet
- One logical stylesheet provides the transformation rules
  - Again this is organised according to the XPATH data model
  - Multiple stylesheets can be combined
    - Via the include and import instructions

The XSLT Processor

- The output from XSLT may be:
  - An XML Document
    - Processor tries to ensure output is well formed
  - An HTML document
    - Processor follows SGML lexical conventions
  - A text document
    - No precautions are taken and no characters are escaped
- Output is serialised or passed to another process
  - If it is serialised you have little control over which alternatives are chosen for escaping, indentation etc…
  - XSLT was designed to produce an in memory tree for use by another process
    - Such as an XSL-FO Engine or Web Browser

A Simple Transformation



Applying Transformations in Code

- Both Java and .NET support XSLT
  - You can apply stylesheets from within code
- Java intentionally hides the type of the engine
  - The static ‘TransformerFactory.newInstance’ method finds the default engine on the classpath and returns a reference to it
  - This is used to build ‘Transformer’ objects that run stylesheets
- .NET supports XSLT via the types in ‘System.Xml.Xsl’
  - ‘XslTransform’ objects apply stylesheets
    - The input is an ‘XPathDocument’
    - The output is written down a stream
  - The transformation can be customized
    - Via stylesheet arguments and extension objects

Applying a Stylesheet in Java Code

```
public static void main(String[] args) throws Exception {
	String sep = File.separator;
	File inputFile = new File("input" + sep + "much_ado_about_nothing.xml");
	File xsltFile = new File("input" + sep + "shakespeare.xslt");
	File outputFile = new File("output" + sep + "play.html");

	TransformerFactory tf = TransformerFactory.newInstance();

	StreamSource inputSource = new StreamSource(inputFile );
	StreamSource stylesheetSource = new StreamSource(xsltFile);

	StreamResult consoleResult = new StreamResult(System.out);
	StreamResult fileResult = new StreamResult(outputFile);

	Transformer t = tf.newTransformer(stylesheetSource);

	t.transform(inputSource,consoleResult);
	t.transform(inputSource,fileResult);
}
```

Applying a Stylesheet in C# Code

```
static void Main(string[] args) {
	//A document class optimised for XPath
	XPathDocument inputDoc = new XPathDocument("..\\..\\muchAdoAboutNothing.xml");

	//The transformer object
	XslTransform stylesheet = new XslTransform();
	stylesheet.Load("..\\..\\shakespeare.xslt");

	//Perform a simple transformation
	// Parameter two could be an XsltArgumentList object to pass parameters
	// Parameter four would be an XmlResolver if our XSLT used 'import' or 'include'
	stylesheet.Transform(inputDoc,null,File.Open("output.html",FileMode.Create),null);

	Console.WriteLine("Transformation Complete");
}
```