XPath

Authors:

- Tobias Schlitt <toby@php.net>
- Jakob Westhoff <jakob@php.net>

License: CC-by-nc-sa

XPath

Introduction 2
- The XML tree model . . . 2
  - Clarification of terms . . . 2
  - An XML tree . . . 3
  - Node relations . . . 5
- The idea behind XPath . . . 6
- Possibilities . . . 7

Addressing 7
- Absolute vs. relative addressing . . . 8
  - Absolute addressing . . . 8
  - Relative addressing . . . 9
- Addressing specialties . . . 9
  - Wildcards . . . 9
  - Matching all descendants . . . 10
- Addressing attributes . . . 11
- Union selections . . . 13

XPath axes 14
- Examples of axis usage . . . 15
  - Ancestor axis . . . 15
  - Attribute axis . . . 16
  - Descendant axis . . . 16
  - Following axis . . . 17
  - Following-sibling axis . . . 17
  - Namespace axis . . . 17
  - Parent axis . . . 18
  - Preceding axis . . . 18

Preceding-sibling axis . . . 19

Functions, operators and conditions **19**
Conditions . . . 20
Operators . . . 21
Functions . . . 21
Node set functions . . . 22
String functions . . . 22
Boolean functions . . . 24
Number functions . . . 24

XPath and XSLT **24**
XSLT introduction . . . 24
Coherence between XPath and XSLT . . . 25
Use case of XPath inside a XSL transformation . . . 25
In detail inspection of the transformation . . . 27
Demonstration of complex XPath usage in XSLT . . . 28

Complex XPath example **30**

Conclusion **31**

Introduction

This paper will give an overview on XPath[1] an addressing language for XML[2] documents. XPath is a W3C recommendation currently in version 1.0. XPath was created in relation to the XSL recommendation and is intended to be used with XSLT[3] and XPointer[4]. Beside that, XPath can be used in a variety of programming languages, commonly in combination with a DOM[5] API.

Note, that the XML declaration has been left out of any examples in this document for clarity reasons.

The XML tree model

XPath is a language to address a set of nodes in an XML document. To understand how XPath works, an understanding of the tree structure of XML documents is necessary. This section will give a short introduction on how an XML document is structured.

Clarification of terms

This section gives a rough introduction into terms that are used in this chapter and overall the whole document. Only the most important terms are described here, more specialized ones are introduced when needed. The XML standard describes several types of items that might occur in a document. The most commonly known item type are elements:

```
<title>Some book title</title>
```

The above example shows a simple element named *title*. Each element consists at least of a start-tag. If the element has content (the children of the element), an end-tag is also present. In case an element does not have children, it can also consist just of a start tag, which is noted using a special syntax:

```
<title />
```

This element does not have any content. The usage of an end-tag is omitted. Beside plain text (an atomic value), an element may also have one or more elements as its children:

```
<book>
  <title>Some book title</title>
  <author>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
  </author>
</book>
```

This example document consists of the root element (also called document element) *book*. Every XML document must have exactly one document element. The root element in this has two child elements: *title* and *author*. The *author* element in addition has two child elements *firstname* and *lastname*. Those two and the *title* node have an atomic value as their content.

Elements belong to the category "nodes". An XML document only consists of nodes and "atomic values". The other type of nodes are "attribute nodes":

```
<title level="1">Some book title</title>
```

The element node shown above has an attribute node attached it:

```
level="1"
```

Both, the attribute node and the element node have atomic values assigned as their content. The content of the title element is "Some book title", the content of the attribute node level is "1". The following example shows a simple XML document and a tree visualization of its items. Items are the highest level category in XML. Nodes and atomic values are both considered items.

An XML tree

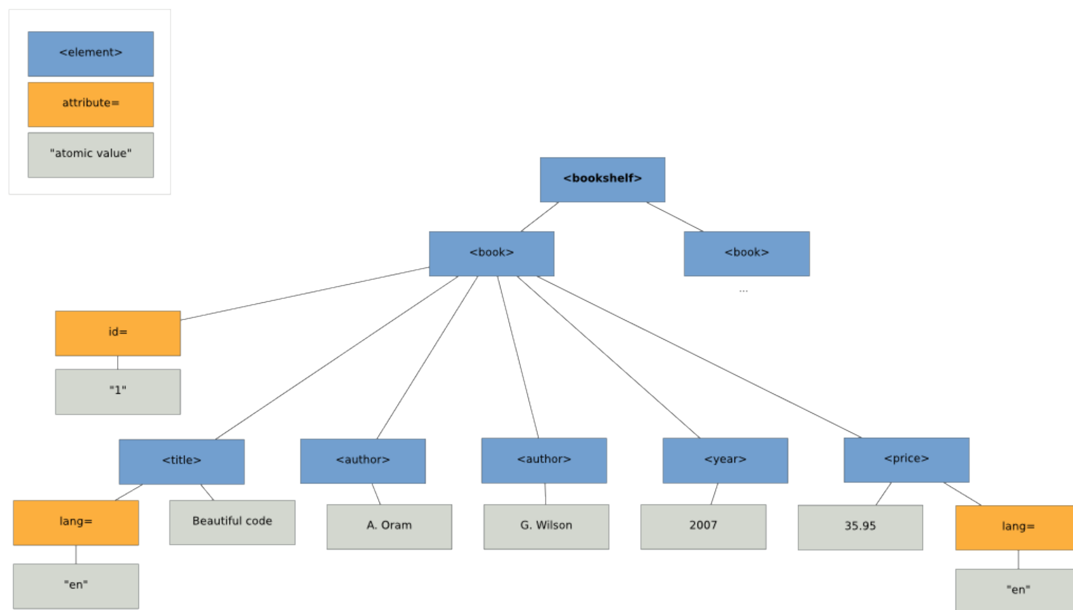
A typical XML document looks like this:

```
<bookshelf>
  <book id="1">
    <title lang="en">Beautiful code</title>
    <author>A. Oram</author>
    <author>G. Wilson</author>
    <year>2007</year>
    <price currency="Euro">35.95</price>
  </book>
  <book id="2">
    <title lang="de">eZ Components - Das Entwicklerhandbuch</title>
    <author>T. Schlitt</author>
    <author>K. Nordmann</author>
    <year>2007</year>
    <price currency="Euro">39.95</price>
  </book>
</bookshelf>
```

This document consists of several

- Items
- Nodes
- Elements
- Attributes
- Atomic values

Visualized in a tree, the document looks like this:



The chart shown above visualizes the tree structure of the XML document. Element nodes (aka tags) are displayed in blue color, attribute nodes are orange and atomic values are gray colored. The *bookshelf* tag is the root of the tree, the document element. It has two child element nodes, which each contain similar children. Therefore the second *book* node is skipped. The displayed *book* node has two different kinds of child nodes: One attribute node and 4 element nodes. The attribute node itself has an atomic value attached as its value. The 4 child nodes of the *book* element are structured the same way.

Node relations

You already got to know the child relation between nodes and the document element, the root node of each XML document. These terms should also be familiar to you from other tree structures. Beside those, tree terminology is used when talking about XML.

parent
: All child nodes of an element have this element as their parent.

sibling
: All nodes of the same type, that have a common parent, node are siblings.

descendants
: All children of a node and their children recursively are collected in the group of a nodes descendants. This are all nodes that reside below a certain node.

ancestors

In contrast to the descendants, the ancestors are all nodes that reside above the affected node: His parent and his parents parents recursively. The root node is an ancestor of every node, except of itself.

To illustrate this a bit more, here are some of the relations in the example document: The document element *bookshelf* has a lot of descendants: Two *book* elements, two *id* attributes, two *title* elements, two *lang* attributes four author *elements* and many more. Some of these are also descendants of the *element* with *id* 1. The ancestors of the *author* element with the value "A. Oram" are *book* (the direct parent node) and *bookshelf* (the parents parent). The element *title*, the two *author* elements, the *year* and *price* elements are siblings. The two *book* elements are siblings, too. The *bookshelf* element does not have any siblings, no descendants and no parent.

The idea behind XPath

The XPath language allows you to query a well-defined set of nodes from an XML document. If you write an XPath expression and have it processed by an XPath interpreter engine, you will always get a set of XML document items as the result.

To illustrate this, here is a short introduction example, that might be executed on the XML document shown in the section An XML tree:

```
/bookshelf/book/title
```

If you give this expression to a XPath processor together with the XML document, you will received a set of *title* elements, containing the title for each *book* element in the *bookshelf*.

The XPath syntax is inspired by the typical way to address file system nodes (files and directories) on Unix and Unix-like systems. The simple expression shown above consists of three element names, divided by slashes (/). Each slash creates a new context. Subsequent expression parts operate on this context. A context may consist of a single element, multiple elements or no elements.

For example, the expression above evaluates as follows:

1. / Create the first context, the document itself.
2. *bookshelf* Select all *bookshelf* elements found on this level. Only one, the document root element.
3. / Create a new context of the found "bookshelf" elements as basis for the next evaluation. The next evaluation takes place on the children of the elements in the context.
4. *book* Select all "book" elements found as children of the context elements. This means all "book" elements which are direct children of the "bookshelf" element.

5. / Create a new context and add the found “book” elements to it. This context (in contrast to step 3) contains two elements.
6. *title* Select all “title” elements found as direct children of the current context. Since the context contains two “book” elements, both are considered. This means all “title” elements which are direct children of any of the “book” elements in the context are selected.

Since the expression does not define any further evaluation steps, the set of *title* elements is the return value.

Possibilities

XPath is usually not used on its own, but in combination with other XML processing techniques. Most common are XSLT and DOM. The items retrieved from an XPath query are then manipulated using these techniques. An XSL Templates contains a “match” attribute, which holds an XPath expression. This expression determines the XML items to which the template will be applied.

DOM is an API specification for unified working with XML documents. It defines an object oriented tree structure for the parsed document and the methods to navigate through the document and manipulate it’s items. DOM is implemented for a lot of programming languages like C[6], C++[7], Java[8], PHP[9] and even Haskell[10]. Since recursive processing of DOM tree structures from a programming language alone is quite cumbersome, DOM Level 3 defines an API for the evaluation of XPath expressions. The XPath evaluator the returns the item set that matched the evaluation as a DOMNodeList object.

XPath allows to address any arbitrary item in a XML Document. You can address sets of elements, attributes and atomic values, as well as sub-parts of such data. Beside that, the XPath specification provides a large variety of functions that can be used to extract certain portions of atomic values and perform mathematical processing.

Addressing

The purpose of XPath is to address nodes (elements, attributes) in XML documents. The basic syntax of XPath expressions and how they typically evaluate was already presented in the last chapter. This chapter introduces you to further possibilities XPath offers you. You will get to know more syntax elements an XPath expression may contain and how to use them to create simple expressions for selecting nodes.

The subsequent chapters will give you a deeper view on XPath and introduce even more complex operations you can use in an expression. However, this chapter should last to basically understand how an XPath expression is built and how you can use it for simple addressing purposes. The example document, presented in the first chapter, will deal for the most examples in this chapter, although it might slightly be altered to suite different needs.

Absolute vs. relative addressing

In the previous chapter you already learned how to address element nodes from the document basis. The following simple expression was analyzed to explain XPath evaluation:

```
/bookshelf/book/title
```

This will grab a list of all *title* elements from the document, which are children of a *book* element, which is a child of a *bookshelf* element. Overall, it will grab all *title* elements. So far we assumed, that the starting / refers to the document basis and the first element name provided therefore refers to the document element. This is not necessarily true. XPath is always embedded into some other technical environment. As you already know, a DOM API of a programming language is very common. This allows you, to define an arbitrary node of a document as the root for your XPath expression.

For example, you might grab one of the *book* elements from the document using DOM functions and then evaluate an XPath expression on this element. If you do so with the expression shown above, it will return an empty node list, since the expression did not match anything. Now consider the following, slightly extended, version of the example document:

```
<bookstore>
  <bookshelf>
    <!-- ... -->
  </bookshelf>
  <bookshelf>
    <!-- ... -->
  </bookshelf>
  <!-- ... -->
</bookstore>
```

Now *bookshelf* is not the document element anymore. If you use the XPath expression from above on the document root now, you would again receive an empty list. Still, you could evaluate the expression on one of the *bookshelf* elements from the document. However, there is nothing really special about this behavior: It's all about context. Normally, an XPath expression takes the document basis as the initial context, but in some cases, you can change this to an arbitrary node.

The following sub-section introduce you to different styles of addressing elements in XPath. You already know the absolute style of addressing elements absolutely and will see how you can also use relative addresses.

Absolute addressing

The examples you have seen so far all used absolute addressing. An absolute item address uses the single slash as a separator. It means, that the next match in an expression must occur as a direct child to the items matched so far. For the example document shown in the last section, the XPath expression would not produce any results if you evaluate it

on the document element. Since it is *bookstore* and not *bookshelf*, the expression would fail in the first match and return an empty item set.

Relative addressing

Beside the absolute addressing XPath allows to define relative addresses. The scheme to define such addresses is again very similar to typical file accesses on the Unix command line. It is possible to select the current item through . and the parent item for the current matches using ... In the *bookshelf* document from above, for example the following expressions would work:

```
/bookshelf/book/title/.

/bookshelf/book/title/..

/bookshelf/book/title/../author
```

The first expression will simply select the *title* elements of the books again. The expression is just an alternative to the expression you already got to know to access the *title* elements:

```
/bookshelf/book/title
```

The second expression will select the parent elements of the titles. There are the *book* elements in the example document. The expression would work almost the same as:

```
/bookshelf/book
```

Beside a slide difference: If the document would contain a *book* element that does not have a title, it would not be listed, using the relative expression. The last expression selects *author* elements from books. But, as the second one, only of books that have a *title* element.

Addressing specialties

Just selecting a path through the document tree is kind of cumbersome and limited. You don't always know the concrete path to a certain element or you want to re-use an expression on different document types. XPath offers you two features for such purposes, which will be explained in the following two sub-sections.

Wildcards

The wildcard used in XPath is the *, which can be used instead of an element name. This character defines that the name of the element to match does not matter, but there must be an element. File system paths also use this wildcard, but with a slightly different meaning. The already discussed expression to fetch all *title* elements could also be written like this:

```
/bookshelf/*/title
```

```
/*/book/title
```

```
/*/*/title
```

All variants would fetch the same items as the already known expressions. A difference would occur, for example, if a *bookshelf* could also contain newspapers:

```
<bookshelf>
  <book id="1">
    <title lang="en">Beautiful code</title>
    <author>A. Oram</author>
    <author>G. Wilson</author>
    <year>2007</year>
    <price currency="Euro">35.95</price>
  </book>
  <book id="2">
    <!-- ... -->
  </book>

  <newspaper id="3">
    <title>Ct Magazin - Ausgabe 3/2008</title>
  </newspaper>

</bookshelf>
```

In this case, the first and third expressions would also fetch the *title* of the *magazine*, while the second one would only deliver titles from books. The star wildcard also works when addressing attributes, which will be shown in the next main section.

Matching all descendants

So far, we only presented XPath expressions which worked on direct children of certain elements. In addition you can address all descendant elements of the current context using two slashes instead of a single one:

```
//title
```

The semantic of this expression is, to find all *title* elements which reside below the current context. In other words, it finds all descendant *title* elements. For the *bookshelf* example, the result would be the same as for the absolute addressed expression:

```
/bookshelf/book/title
```

If we change the example XML slightly, the result of the expressions would differ:

```
<bookshelf>
  <title>Computer science books</title>
  <book id="1">
    <title lang="en">Beautiful code</title>
    <author>A. Oram</author>
    <author>G. Wilson</author>
    <year>2007</year>
    <price currency="Euro">35.95</price>
  </book>
  <book id="2">
    <title lang="de">eZ Components - Das Entwicklerhandbuch</title>
    <author>T. Schlitt</author>
    <author>K. Nordmann</author>
    <year>2007</year>
    <price currency="Euro">39.95</price>
  </book>
</bookshelf>
```

The already known expression would still find all *title* elements of all books. In contrast, the relative expression would also find the *title* of the *bookshelf*, since this is also a descendant of the document base. It is possible to use a descendants match anywhere in an XPath expression. For example, the expression:

```
/bookshelf//title
```

Would also result in the same item list as the previously shown expression did. The following expression instead results in the same list as the original expression:

```
/bookshelf/book//title
```

Addressing attributes

So far you know how to address element nodes (aka tags), using XPath. Beside these, XPath also allows you to address attribute nodes. You will learn how to do this in the following sections. Addressing atomic values does not make much sense, since these are always direct children of either an element or an attribute. If you need to access the atomic value of a node, select it via XPath and access it from the technology from which you are using XPath (possibly DOM). However, in the next chapter you will see, that while you cannot select atomic values, you still have access to them through functions and comparators.

While element matching in an XPath expression is simply done by using the name of the desired element, attribute matches need to be prefixed by the "@" character. Beside that, such matches are used in an expression not different from element addressing. For example, it is possible to select all "id" attribute items of book elements, using one of the following expressions:

```
/bookshelf/book/@id

//book/@id

//book//@id

//@id
```

The first expression uses absolute addressing of the desired items. The document root element *bookshelf* is selected, below that, all *book* children are selected and of these, the *id* attribute is fetched. The second version uses relative addressing to fetch all *book* elements and the selects the attribute *id* from these. The difference from this expression to the forth one is, that the latter one would also fetch *id* attributes from descendant elements of a *book*. The last version finally uses a relative address to select all *id* attributes all over the document. In fact, for the example *bookshelf* document, all of these expressions deliver the same set of items. In contrast, they do not behave like this for the following example. It is a slightly modified version of the *bookshelf* example:

```
<bookshelf id="computer_science_books">
  <title>Computer science books</title>
  <book id="1">
    <title lang="en">Beautiful code</title>
    <author id="a_oram">A. Oram</author>
    <author id="g_wilson">G. Wilson</author>
    <year>2007</year>
    <price currency="Euro">35.95</price>
  </book>
  <book id="2">
    <title lang="de">eZ Components - Das Entwicklerhandbuch</title>
    <author id="t_schlitt">T. Schlitt</author>
    <author id="k_nordmann">K. Nordmann</author>
    <year>2007</year>
    <price currency="Euro">39.95</price>
  </book>
</bookshelf>
```

The first and second expressions, evaluated on this document, would result in the same set of items that would have been returned from the original document. A different result would be retrieved from the third expression. Since this one selects all *id* attributes that are a descendant of any *book* element, the *id* attributes from the *author* elements would also be selected. The resulting attribute items would be:

1. id="1"
2. id="a_oram"
3. id="g_wilson"

4. id="2"
5. id="t_schlitt"
6. id="k_nordmann"

Even more items would be selected by the last expression. This one selects all *id* attributes from all over the document. Therefore the *id* attribute of the *bookshelf* is returned in addition, as the first item. As already mentioned, attribute matches can also be used together with the * wildcard. This allows you to select all attribute items at once. The following examples illustrate the usage of the wildcard with attributes:

```
/bookshelf/book/title/@*
```

```
/bookshelf/book//@*
```

The first expression will only return the *lang* attributes from the *title* elements, since they are the only available attributes there. The second expression will return different attribute items: The *id* attributes from the *book* elements, since they are direct children of the current context. The *lang* attributes from the *title* elements and the *currency* attribute from the *price* elements will be returned, since these are descendants of the current context. The whole list would be:

1. id="1"
2. lang="en"
3. currency="Euro"
4. id="2"
5. lang="de"
6. currency="Euro"

Union selections

Sometimes it makes sense to not only select nodes by a single expression, but to union the item sets returned by multiple expressions. Instead of evaluating two or more expressions in a row and work on them separately, you can instruct the XPath engine to unify them in a single return set. You connect two expressions by the pipe char | to have them united.

```
/bookshelf/book|/bookshelf/magazine
```

```
/bookshelf/book/title|/bookshelf/book/author
```

The first expression would select all *book* elements from a *bookshelf* and add all *magazine* elements to the result set, too. So, you can use this expression on both, the original example document and the enhanced version from a previous section, which also contained magazines. The second expression will select all *title* elements of any books and

all *author* elements. Note, that you won't see any relationship between these. They are just selected in the order they appear in the document. It is also possible to mix element and attribute selection in such a combined expression, since both are just nodes.

XPath axes

As already explained a XML document represents a tree structure. Simple navigational constructs to find and select elements have already been mentioned in this paper. The following section will cover a more advanced navigating scheme called "XPath Axes". These axes are similar to axes of a Cartesian coordinate system. Using them it is possible to specify the exact location of any point inside the system. Or in our case the exact location of any element.

The tree is treated as being inside a multi dimensional space, where every axis represents a relation between some of the nodes. The XPath standard defines 13 axes[11]. These axis are named as follows:

Axis	**Description**
ancestor	All ancestors of the current node. This includes the parent node, as well as the parent of the parent and so on. Therefore this axis will always include the root node
ancestor-or-self	All ancestors, as well as the current node itself
attribute	Attributes of the current element. If the current node is not an element this axis will be empty.
child	Every direct child of the current node.
descendant	Every descendant of the current node. This includes children of children and so on.
descendant-or-self	All descendants or the current node itself.
following	Every node following the current node. Descendants, attributes and namespace nodes are excluded from this axis
following-sibling	All siblings following the current node.
namespace	All namespace nodes defined on the current node.
parent	The parent node of the current node.
preceding	All nodes before the current node. Ancestors, attributes and namespace nodes are excluded
preceding-sibling	All siblings preceding the current node
self	The current node itself

Every step inside a location specified as an XPath query actually consists of three parts: An axis, a node test and one or more predicates, which are optional and narrow

down the node set to a more fine grained subset[12]. The syntax is defined as follows:

```
axisname::nodetest[predicates]
```

Because in most cases the child axis will be used. Therefore it is defined as the default one. This means if the axis is omitted the child axis will be used automatically. Therefore a simple location like this one.

```
/bookstore/book/title[@lang="eng"]
```

Is treated like it has been written like this:

```
/child::bookstore/child::book/child::title[@lang="eng"]
```

Examples of axis usage

All of the examples below are applied to the following XML document:

```
<groupmembers>
  <member>
    <firstname>Jakob</firstname>
    <lastname>Westhoff</lastname>
    <x:nationality xmlns:x="http://westhoffswelt.de/EXTENDED_MEMBER_NAMESPACE">
        german
    </x:nationality>
  </member>
  <member>
    <firstname>Tobias</firstname>
    <lastname>Schlitt</lastname>
  </member>
  <member age="42">
    <firstname>John</firstname>
    <lastname>Doe</lastname>
  </member>
</groupmembers>
```

Ancestor axis

Used XPath query:

```
/groupmembers/member/firstname[text() = 'Jakob']/ancestor::member
```

Result:

```
<member>
  <firstname>Jakob</firstname>
  <lastname>Westhoff</lastname>
  <x:nationality xmlns:x="http://westhoffswelt.de/EXTENDED_MEMBER_NAMESPACE">
     german
  </x:nationality>
</member>
```

The beginning steps of the location, which are not preceded by a special axis are using the child axis by default and therefore select every *firstname* element with the text “Jakob”. In our case this is only one element. The following *ancestor:member* step selects every ancestor node of the type member. The same result could have been achieved only utilizing the child axis by using the following expression:

```
/groupmembers/member[firstname='Jakob']
```

But the slightly more complex path inside the tree shows quite well which elements are on the ancestor axis.

Attribute axis

Used XPath query:

```
/groupmembers/member[firstname = 'John' and lastname = 'Doe']/attribute::age
```

Result:

```
42
```

The member node which contains personal information about “John Doe” is selected first. The attribute axis is finally used to select the attribute of the type *age*.

Descendant axis

Used XPath query:

```
/groupmembers/member/descendant::*
```

Result::

```
<firstname>Jakob</firstname>
<lastname>Westhoff</lastname>
<x:nationality xmlns:x="http://westhoffswelt.de/EXTENDED_MEMBER_NAMESPACE">
   german
</x:nationality>
<firstname>Tobias</firstname>
<lastname>Schlitt</lastname>
<firstname>John</firstname>
<lastname>Doe</lastname>
```

The descendants of every member type node is selected by this query. As you can see the surrounding member nodes are not included in this selection, because they are no direct descendants of themselves. To include the member nodes “descendant-or-self” should be used.

Following axis

Used XPath query:

```
/groupmembers/member[2]/following::member
```

Result:

```
<member age="42">
  <firstname>John</firstname>
  <lastname>Doe</lastname>
</member>
```

The second member element in the document is selected first. This would be the member “Tobias Schlitt”. Finally the *following* axis is used to select all following elements of the type member. Only one member element is following the current node. Therefore this one is printed out. Attention needs to be brought to the fact that the node test for the type member is necessary. Otherwise all following nodes would have been selected. Therefore the firstname and lastname nodes would have been outputted twice.

Following-sibling axis

Used XPath query:

```
/groupmembers/member[2]/following-sibling::*
```

Result:

```
<member age="42">
  <firstname>John</firstname>
  <lastname>Doe</lastname>
</member>
```

Because only siblings which follow the current node are selected using this axis, we don’t need the node test for member anymore, as in the example above. In this case only elements on the same level as the current one are selected.

Namespace axis

Used XPath query:

```
/groupmembers/member[firstname = 'Jakob']/nationality/namespace::*
```

Result:

```
xmlns:x
```

On the selected nationality node the namespace x is defined. Therefore the corresponding namespace node is returned by this query.

Parent axis

Used XPath query:

```
/groupmembers/member/firstname[text() = 'Jakob']/parent::*
```

Result:

```
<member>
  <firstname>Jakob</firstname>
  <lastname>Westhoff</lastname>
  <x:nationality xmlns:x="http://westhoffswelt.de/EXTENDED_MEMBER_NAMESPACE">
    german
  </x:nationality>
</member>
```

In contrast to the ancestor axis example a node test for the type *member* is not necessary in this query. Because the parent axis only includes the direct parent, which is the enveloping member element in this case.

Preceding axis

Used XPath query:

```
/groupmembers/member[2]/preceding::member
```

Result:

```
<member>
  <firstname>Jakob</firstname>
  <lastname>Westhoff</lastname>
  <x:nationality xmlns:x="http://westhoffswelt.de/EXTENDED_MEMBER_NAMESPACE">
     german
  </x:nationality>
</member>
```

The second member element does only have one preceding member, which is the first node in the document. The node test for the *member* type is needed because, all child nodes of the outputted member element are preceding the current node as well. Therefore they would have been outputted twice without the node test.

Preceding-sibling axis

Used XPath query:

```
/groupmembers/member[2]/preceding-sibling::*
```

Result:

```
<member>
  <firstname>Jakob</firstname>
  <lastname>Westhoff</lastname>
  <x:nationality xmlns:x="http://westhoffswelt.de/EXTENDED_MEMBER_NAMESPACE">
     german
  </x:nationality>
</member>
```

The difference between the preceding and preceding-sibling axes is the same as with the following and following-sibling axes. The use of preceding-sibling only selects elements on the same level as the current one. Therefore the node test for the member type as in the example above is not needed anymore.

Functions, operators and conditions

By know you are already an XPath professional. You can address elements and attribute nodes. You have access to parents and children and can create complex matches of parent nodes by matching their children through relative addressing. You know about the different axis XPath knows in XML documents and can utilize them to create even more complex queries. This knowledge usually lasts to give you a handy tool at hand that allows you to select a set of XML nodes from a document and process these in your favorite programming language. However, there are many cases, where you still need a lot of hand work in the language to extract a sub set of the selected nodes: What, if you only want nodes that have a certain attribute value? What if you need only the first child of a certain element?

Functions and comparators give you the necessary power to achieve such goals in pure XPath and therefore allow you to also realize such tasks in pure XSL. In this section we illustrate examples again on a variation of the bookshelf XML file you already know:

```
<bookshelf id="computer_science_books">
  <title>Computer science books</title>
    <book id="1">
    <title lang="en">Beautiful code</title>
    <author id="a_oram">A. Oram</author>
    <author>G. Wilson</author>
    <year>2007</year>
    <price currency="Euro">35.95</price>
```

```
    </book>
    <book id="2">
      <title lang="de">eZ Components - Das Entwicklerhandbuch</title>
      <author>T. Schlitt</author>
      <author id="k_nordmann">K. Nordmann</author>
      <price currency="Euro">39.95</price>
    </book>
  </bookshelf>
```

Conditions

By now, you already saw how to select a set of XML nodes, which match a certain expression. You can match absolute and relative paths throughout the document and even perform some more advanced checks, using nasty tricks. For example, if you want to select all authors, which have an *id* attribute, you could use the following expression:

/bookshelf/book/author/@id/..

This one selects the *bookshelf*, all *books* and then the *author* elements of the books. After that, the *id* attribute of the *author* element is matched. If no such attribute is available, it cannot be included in the next context. In the next context the parents of the found *id* attribute nodes are selected, which are the *author* elements again. But only those, which really have an attribute named *id*. An alternative to this, somewhat hackish, expression is, to use a condition with the *author* match:

```
/bookshelf/book/author[@id]
```

The square braces indicate, that the context created by the match right in front of it should be limited to nodes, which satisfy the condition. The condition is given inside of the braces. In the example, the condition is that an attribute (indicated by the @) must be available. You will learn in the next sections how to match against values inside the condition and how to invert the match using boolean negation.

It is not possible to match a child element condition, but you already know how to do this: It works similar to the alternative attribute match shown at the start of this section. The wildcard * also works in conditions. For example you can use

```
/bookshelf/book/author[@*]
```

to match all *author* elements which have any attributes set. While the conditions in the examples shown above basically return boolean values, it is also possible to give a numeric value in the condition:

```
/bookshelf/book/author[1]
```

In this case, only the first author of every *book* element is selected. The section about functions will give you more examples of how to use numeric conditions.

Operators

Inside of matches, XPath allows you to use a set of comparison operators, which you should already be familiar with from your favorite programming language. The following table shows an overview of the operators and explains shortly, what they do:

Operator(s)	Explanation
+, -, *	Mathematical operations (addition, subtraction and multiplication)
div	Mathematical division
mod	Mathematical modulo operator
=	Check for equality (mathematical and string)
!=	Check for inequality (mathematical and string)
<, <=	Less than and less than or equal check
>, >=	Greater than and greater than or equal check
or	Logical or
and	Logical and

There is no logical negation operator, but a function which returns the negated value. See the next section for details. Using these operators you can already create some more powerful conditions to limit the set of matched nodes. Some examples on how to use operators:

```
/bookshelf/book/title[@lang = 'de']/..
```

This expression matches only book titles, which are in German language and returns the *book* elements, where such a title is found. So in fact, it returns all German books. It is not only possible to operate on attribute values, but also on tags:

```
/bookshelf/book[price < 39]
```

Using this expression you will receive all *book* elements, which have a *price* tag as a child, that contains a number that is smaller than 39. If you want to know the price of book number 1, you could use the following expression:

```
/bookshelf/book[@id = 1]/price
```

Functions

Finally, XPath offers a large set of functions, that allow you to operate on atomic values before and while comparing them in an expression. Functions usually reside in a namespace in XPath version 2.0, which is named “fn”. Since XPath 1.0 is the current

recommendation, we leave this namespace out in further examples. It should usually be save to leave it out when using them in practice anyway.

We already referred to one function in the previous chapters: *not()*. This function returns the boolean negation of a submitted boolean value. Using this function, you can finally invert the attribute-existence check, we performed earlier:

```
/bookshelf/book/author[not( @id )]
```

This expression returns all "author" elements, that do not have an attribute "id". Due to the variety of functions, we have limited the examples shown in this chapter to a small subset, which illustrates the usage. The W3C recommendation contains a complete function reference[13]. XPath functions are categorized and you will see at least one example of each category in following.

Node set functions

The functions in this category work on the node set which is currently matched. The most important ones are:

last()
: This function returns the number of the last node of a certain type.

count()
: This function returns the number of nodes in the current context.

position()
: This function does actually not work on a set of nodes, but always returns the position of the current nodes. You can use this functions to fetch for example only the last author of a book, or the second last one:

```
/bookshelf/book/author[last() - 1]
```

Another possibility is to return only the authors at an odd position:

```
/bookshelf/book/author[position() mod 2 = 1]
```

String functions

XPath provides a large variety of string functions. Some you might be interested in, right now are:

concat()
: The concat() function concatenates 2 or more strings and returns their concatenation.

string-join()
: You give an arbitrary number of strings to this function, as some kind of list. In addition you give it a single separator string. The latter one will be used to to separate the other strings while being concatenated. An example:

```
string-join( ( 'Milk', 'Butter', 'Flower' ), ', ' )
```

Will return 'Milk, Butter, Flower'.

substring()
: This function extracts a part from a string, defined by a start index and an optional length value.

string-length()
: Returns the length of the given string.

escape-uri()
: This method escapes the given string so that it can safely be used in an URI. This is especially useful, if you need to give a complete URI as a parameter to another one.

starts-with()
: You give this function a two strings. It returns true, if the first string starts with the second one, false otherwise. The function ends-with() is the counterpart for this function, checking for the end of a string.

contains()
: This function returns, if the second given string occurs in the first one.

replace()
: This function receives three strings: The second is a match string, as for contains(), the third is a string that is used to replace every occurrence of the match. The replacement takes place in the first string.

Some example expressions for illustration:

```
/bookshelf/book/author[substring( ., 1, 1 ) = 'T']
```

This expression selects all authors whose names begin with a *T*. Note the reference to the current node, using the already known .. The following expression selects only authors whose name contains an abbreviation (indicated by a . in the name):

```
/bookshelf/book/author[contains( ., '.' )]
```

The last expression returns only books, of which the titles are longer than 15 characters:

```
/bookshelf/book/title[string-length( . ) > 15]/..
```

Boolean functions

The boolean functions are mostly used to realize certain boolean expressions. The most important ones are:

not()
: To negate a given boolean expression.

true()
: Returning the boolean value "true". Its counterpart function is false().

An example for the not() function was already given at the beginning of this chapter.

Number functions

The number function category contains the most common functions other programming languages know:

floor()
: Returns the largest integer number that is not greater than the given parameter.

ceiling()
: As the counterpart to floor(), this function returns the smallest integer number that is not smaller than the given parameter.

round()
: Returns the closest integer number of the given parameter.

An example:

```
/bookshelf/book/price[round( . ) = 40 and @currency = 'Euro']/..
```

This expression returns all books which cost about 40 Euros.

XPath and XSLT

This chapter gives you a practical example on how to use XPath in XSLT.

XSLT introduction

XSLT is an abbreviation for "Extensible Stylesheet Language Transformations"[14]. This transformation language is completely XML based. Its main purpose is to define transformation rules which can be applied to any XML document. The target document may be another XML structure, any other kind of human-readable output or arbitrary binary data. Attention needs to be given to the fact that this transformation does not change the input document directly. The input is read, transformed and finally written to the transformed output file. The original document remains untouched. This easily allows to process one input file with several different style sheets to create a vast majority of output formats.

Coherence between XPath and XSLT

XPath and XSLT belong to a larger group of XML tools called “Extensible Stylesheet Language” or XSL for short[15]. XSLT makes heavy usage of XPath it is therefore discussed more detailed here. To be able to define transformation rules on XML documents a way of addressing nodes and sub trees inside the document is needed. As mentioned before XPath has been developed to do just this. Consequentially it is used for this purpose.

Use case of XPath inside a XSL transformation

To present you with a use case of XPath a really simple XSL transformation will be shown and discussed in detail. This will help to understand how XPath can be used to ease the work on XML documents.

```
<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">

    <html>
      <head>
        <title>PG Members Example</title>
      </head>
      <body>
        <h1>List of PG members</h1>
        <table>
          <thead>
            <tr>
              <td>Firstname</td>
              <td>Lastname</td>
            </tr>
          </thead>
          <tbody>

            <xsl:for-each select="groupmembers/member">
              <tr>
                <td>
                  <xsl:value-of select="firstname" />
                </td>
                <td>
                  <xsl:value-of select="lastname" />
                </td>
              </tr>
            </xsl:for-each>
```

```
          </tbody>
        </table>
      </body>
    </html>

  </xsl:template>
</xsl:stylesheet>
```

The Stylesheet is applied on the following snippet of XML:

```
<groupmembers>
  <member>
    <firstname>Jakob</firstname>
    <lastname>Westhoff</lastname>
  </member>
  <member>
    <firstname>Tobias</firstname>
    <lastname>Schlitt</lastname>
  </member>
  <member>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
  </member>
</groupmembers>
```

The result of the transformation will look like this:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>PG Members Example</title>
  </head>
  <body>
    <h1>List of PG members</h1>
    <table>
      <thead>
        <tr>
          <td>Firstname</td>
          <td>Lastname</td>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Jakob</td>
```

```
            <td>Westhoff</td>
          </tr>
          <tr>
            <td>Tobias</td>
            <td>Schlitt</td>
          </tr>
          <tr>
            <td>John</td>
            <td>Doe</td>
          </tr>
        </tbody>
      </table>
    </body>
  </html>
```

For this transformation the program xsltproc[16] has been used. The generated output with the given style sheet generated valid XHTML out of our initially defined grouplist.

In detail inspection of the transformation

We will now take a closer look at elements of the XSLT file.

```
<?xml version="1.0" encoding="utf-8"?>
```

As with every well-formed XML document this one is started with a general type definition line which supplies the used XML version as well as the utilized charset. In this case we are using XML version 1.0 and have stored our style sheet content int UTF-8[17].

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

The *stylesheet* element needs to enclose every XSLT. It is needed to declare general information like for example the used version. In our case it is also used to declare the XSL namespace. Every element containing XSLT processing information needs to be in a defined namespace. In most cases the abbreviation for this namespace will be *xsl*. But as far as the reference to the correct URI is made the namespace name can be freely chosen.

```
<xsl:template match="/">
```

The following line is the first one which is of real interest to us, because an XPath expression is used in it. The / inside of the match attribute is actually an XPath expression which selects every document root. In more complex XSLT definitions you will most likely define more than one template, where each template handles only parts of the document. In our small example this is not needed. Therefore the match on the document root ensures that the template is called once on the complete document.

Most of the following elements are just copied to the output document, because they are not inside the defined *xsl* namespace. The next element which is of interest to us is *for-each*:

```
<xsl:for-each select="groupmembers/member">
```

for-each is a special XSLT element which iterates over a set of selected nodes. In our case this iteration should be done over all members inside the source XML. Therefore a XPath expression is used to retrieve this subset of elements. Because no / is used at the beginning of the expression we are addressing the elements from the current element on. In our case this is the document root, because of the template matching rule explained just before. On more complicated structures the difference of relative to absolute addressing can be used to handle nested elements with ease.

```
<xsl:value-of select="firstname" />
```

To copy the *value-of* a specified source element into the target tree the XSLT element of the same name is used. In our case we just select the node *firstname* relative to the current node, which is the currently processed member element. The exact same method is used to handle the lastname.

Demonstration of complex XPath usage in XSLT

In the discussed example quite simple XPath expressions have been used for demonstration purposes. In productive environments they are likely to be a lot more complex. To show the power of XSLT and XPath a short transformation snippet from the phpUnderControl[18] project is discussed in detail below.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="2.0">

  <xsl:template name="is.odd.or.even">

  <xsl:param name="c" select="1" />
  <xsl:param name="n" select="." />

  <xsl:choose>
    <xsl:when test="name($n) = 'testcase' and $n/preceding-sibling::*">

    <xsl:call-template name="is.odd.or.even">
      <xsl:with-param name="c" select="$c + 1" />
      <xsl:with-param name="n" select="$n/preceding-sibling::*[1]" />
    </xsl:call-template>

    </xsl:when>
```

```
      <xsl:when test="name($n) = 'testsuite' and $n/preceding-sibling::*">

      <!-- Get preceding node -->

      <xsl:variable name="preceding" select="$n/preceding-sibling::*[1]" />

      <!-- Count child nodes of preceding -->

      <xsl:variable name="child.count" select="count($preceding//testcase) +
                    count($preceding//testsuite)" />

      <xsl:call-template name="is.odd.or.even">
        <xsl:with-param name="c" select="$c + 1 + $child.count" />
        <xsl:with-param name="n" select="$preceding" />
      </xsl:call-template>

      </xsl:when>

      <xsl:when test="(name($n) = 'testcase' or name($n) = 'testsuite') and contains($n

      <xsl:call-template name="is.odd.or.even">
        <xsl:with-param name="c" select="$c + 1" />
        <xsl:with-param name="n" select="$n/.." />
      </xsl:call-template>

      </xsl:when>

      <xsl:otherwise>

      <xsl:if test="$c mod 2 = 1">
        <xsl:text>oddrow</xsl:text>
      </xsl:if>

      </xsl:otherwise>
    </xsl:choose>

    </xsl:template>
  </xsl:stylesheet>
```

This style sheet is most likely to be used in future versions of phpUnderControl to create lists with a different behavior for odd and even rows. This kind of differentiation is often used in modern web application to allow an easy optical separation of different data rows, for example by alternating the color between odd and even rows.

As you can see this XSLT is a lot more sophisticated than the one which was shown

before. It makes heavy usage of the functional programming features XSLT offers. The transformation specific details will not be discussed. Nevertheless one of the used XPath queries will be explained to make the possibilities of this language clearer.

```
count($n/preceding-sibling::*[1]//testcase) + count($n/preceding-sibling::*[1]//tests
```

The *$n* in this expression is a variable containing a part of the used node path. Variables are not part of the XPath specification. They can only be used inside a XSLT style sheet. This expression sums up the number of testcase and testsuite elements, which are somewhere under the first preceding sibling of the node *$n*. It might look quite complex at first, but splitted into all of its parts it is quite readable.

```
$n/preceding-sibling::\*[1]
```

Select the first preceding sibling of $n. The sibling might be of any element type.

```
//testcase//testsuite
```

Select all testcase and testsuite elements below the current one. The level of these nodes is not constricted to the level directly below the active one. Every level below the current one satisfies this expression.

```
count()
```

Count is a XPath function which simply returns the number of the elements in the node set instead the node set itself.

Most complex XPath expression can be splitted up like this to make them more easy to understand.

Complex XPath example

To show the possibilities of XPath this section will explain a complex expression in detail, showing that quite sophisticated selections are realizable with XPath.

The problem the following expression tries to solve is rooted in the world of the Resource Description Framework (short RDF). The RDF specification allows elements from arbitrary name spaces inside its description element. Therefore it is a common task to retrieve a list of all used name spaces. This particular task often means to analyze every child of description elements to create a comprehensive collocation. Most of the times it is a lot faster to select the needed nodes directly using an XPath expression.

The following query can be used to get all nodes of all the different name spaces inside a RDF documents description element.

```
//*[ name(.) = 'rdf:Description']
   /*[ namespace-uri(.) != namespace-uri(..) and
       namespace-uri(.) != `` and
       namespace-uri(.) != namespace-uri(preceding-sibling::*)
     ]
```

To easily handle this expression it is split up into its main parts.

```
//\*[ name(.) = 'rdf:Description']
```

This part selects every node at any level of the document which name is rdf:Description. For this to work the the rdf namespace name needs to be registered and mapped to the correct namespace URI.

```
namespace-uri(.) != namespace-uri(..)
```

Because only name spaces which are different from the original *rdf* namespace are relevant we check that the current namespace-URI is different from the one which exists one level up. One level up lies the *rdf:Description* element, therefore this does exactly what it is supposed to do.

```
namespace-uri(.) != ''
```

Someone might add elements which are in no defined namespace. Therefore nodes with an empty namespace-URI are ignored completely.

```
namespace-uri(.) != namespace-uri(preceding-sibling::*)
```

This one is a little more tricky. It ensures that we are only receiving one node of each namespace. It does this by checking any preceding sibling of the current node for the same namespace as the one which is currently processed. If such a node is found the current one can be skipped, because one node inside this namespace already exists.

Conclusion

As shown by the various small and complex examples XPath presents a sophisticated way of selecting sets of nodes inside any arbitrary XML document. It can be used to retrieve all sorts of different sets, ranging from simple directly addressed elements to quite complex selections. XPath axis round off the feature set. They provide all necessary means to explicitly define the way of navigating the XML tree structure. This allows the selection of complex node sets from different levels of the document. Regardless of all this features XPath impresses by its simple and clear syntactical structure. Even the most complex expressions will always consist of one or more *steps* which form a location. These *steps* have a clearly defined structure which is quite simple, if you take into account the power of XPath. Combined with other XSL parts like XSLT XPath is even more powerful and can be used as part a functional programming language to easily transform

a XML document into any output format.

[1] W3C Recommendation: XPath http://www.w3.org/TR/xpath

[2] W3C Standard: XML http://www.w3.org/XML/

[3] W3C Recommendation: XSL http://www.w3.org/TR/xsl/

[4] W3C Recommendation: XPointer http://www.w3.org/TR/xptr/

[5] W3C Standard: DOM http://www.w3.org/DOM/

[6] DOM for C# http://gdome2.cs.unibo.it/

[7] DOM for C++ http://xerces.apache.org/xerces-c/program-dom.html

[8] DOM for Java http://www.w3.org/2003/01/dom2-javadoc/index.html

[9] DOM for PHP http://php.net/dom/

[10] DOM for Haskell http://www.fh-wedel.de/~si/HXmlToolbox/

[11] W3C XPath Specification, XPath axes http://www.w3.org/TR/xpath#axes

[12] W3C XPath Specification, Location http://www.w3.org/TR/xpath#section-Location-Steps

[13] XPath function reference http://www.w3.org/TR/xpath#corelib

[14] W3C XSLT Specification http://www.w3.org/TR/xslt

[15] W3Schools, xsl language introduction http://w3schools.com/xsl/xsl_languages.asp

[16] Xmlsofts xsltproc http://www.xmlsoft.org/

[17] UTF-8 is a special encoding of Unicode characters. More information on this topic can be found here: http://en.wikipedia.org/wiki/UTF-8

[18] phpUnderControl is a continuous integration tool for PHP, which makes heavy use of XSLT to transform XML based log files into readable and optical pleasant HTML result views. More information can found at the official website http://phpundercontrol.org