XML Namespaces on XPath Queries and XSLT

Sinan Uşşaklı,
Microsoft Corporation
May 2, 2006

This document explains the use of namespaces in XML, XSLT and XPath by giving several examples to cover each subject.

What is an XML Namespace?

Most developers are familiar with the concept of namespaces in programming languages such as C#, Java, and other popular object oriented languages. The concept of Namespaces in programming languages and XML are more or less likely the same. The namespace in programming languages define the set for a group of classes. Each class, defined in a namespace will live in that namespace and can directly interact with other classes that are in the same namespace domain. If a class needs to interact with another class living in another namespace domain, it needs to know about that domain in order to have any interaction.

We can think of classes (or identifiers) as cities in the world, and namespaces as states/countries. When we refer the City Vernon, we might be referring to any Vernon from: *BC Canada, Ont. Canada, France, Alabama, California, Connecticut, Florida, Illinois, Indiana, Michigan, New Jersey, New York, Ohio, Texas, Utah*. Namespaces gives us the capability of using the same name

on different domain under same scope. As two cities in the same state or country would make confusions, two classes (or identifiers) would make the same problem in a domain (namespace) therefore a class name should be unique in a namespace.

The reason of why XML Namespaces exist is the same. We may define a tag name that has different meaning in different contexts. For example address of a person, which would contain street address and other physical location properties, has completely different meaning and structure than address of a computer, which would be a URL or an IP. If we want to have both person addresses and computer addresses in same XML structure, then the only way to distinguish these addresses would be namespaces. Following example is an XML document that has two `address`es, one of them is a person's `address` the other is a computer `address`.

```
<list xmlns:ps="http://tempuri.org/person.xsd"
xmlns:pc="http://tempuri.org/computer.xsd">
  <ps:address>
    <ps:streetAddress>1 Microsoft Way</ps:streetAddress>
    <ps:postCode>98052</ps:postCode>
    <ps:country>USA</ps:country>
  </ps:address>
  <pc:address>
    <pc:IP>192.168.0.1</pc:IP>
  </pc:address>
</list>
```

Figure 1.

Here the namespace prefix `ps` and `pc` has the scope inside `list` element. The namespace prefix `ps` is associated with `http://tempuri.org/person.xsd` and prefix `pc` associated with `http://tempuri.org/computer.xsd`. Each element name that has these prefixes is bound to the corresponding namespace.

It is possible to re-declare the default namespace under each address element's scope and define the binding accordingly. The choice of prefixes instead of namespace declarations is completely up on the developer: do whichever makes the XML document more readable.

```
<list>
  <address xmlns="http://tempuri.org/person.xsd">
    <streetAddress>1 Microsoft Way</streetAddress>
    <postCode>98052</postCode>
    <country>USA</country>
  </address>
  <address xmlns="http://tempuri.org/computer.xsd">
    <IP>192.168.0.1</IP>
  </address>
</list>
```

Figure 2.

The namespace of an element is declared by the attribute name `xmlns`, and the value of the attribute should be a valid URI. This URI actually is just a naming convention for that namespace, it doesn't necessarily point to an actual resource. Two namespaces are considered to be the same if the character by character string comparisons of URIs are the same.

In this example, the `list` element is not bound to any namespace. The first `address` element is bound to `http://tempuri.org/person.xsd` the other is bound to `http://tempuri.org/computer.xsd` namespace. Notice that the namespace for the first `address` is declared at the `address` element and all of the descendant nodes of `address` will

have the declared namespace. Therefore, it is not necessary to explicitly declare the namespaces at each descendant node.

One example where namespace declarations will be useful is when we are collecting XML data from different sources. One example might be data from a Biztalk server and combined with other data from an ERP server. When we try to merge the data, the elements from these resources might conflict. Separating the elements by namespace declarations would ease queries against the merged document, and prohibits any conflicts.

What is the Default Namespace?

If a namespace is set with no namespace prefix, then the default namespace is set. For example the default namespace is not declared on `list` element on **Figure 2**, but it declared as `http://tempuri.org/person.xsd` at the first `address`, and declared as `http://tempuri.org/computer.xsd` at the second `address`. These changes are only in effect on the scope of each address.

The element `postCode` in the following example is not part of default namespace `http://tempuri.org/address.xsd` applied to `address`.

```
<address xmlns="http://tempuri.org/address.xsd"
xmlns:pc="http://tempuri.org/postcode.xsd">
  <streetAddress>1 Microsoft Way</streetAddress>
  <pc:postCode pc:type="US">98052</pc:postCode>
  <country>USA</country>
</address>
```

Figure 3.

Namespace Re-declaration

We can redeclare or undeclare a namespace. In order to declare a namespace, change the bound namespace URI to another URI. If the target URI is an empty string, that is un-declaration. Both default namespaces and namespaces associated with a prefix can be redeclared. However, undeclaring a namespace prefix is not allowed.

```
<list xmlns="http://ns1" xmlns:ps="http://foo">
  <address xmlns="http://bar">
    <postCode type="US">98052</postCode>
  </address>
  <address xmlns="">
    <postCode type="US">98053</postCode>
  </address>
  <ps:address xmlns:ps=""> <!-- error! -->
    <ps:postCode type="US">98054</ps:postCode>
  </ps:address>
  <ps:address xmlns:ps="http://bar">
    <ps:postCode type="US">98054</ps:postCode>
  </ps:address>
</list>
```

Figure 4.

In this example, default namespace is bound to `http://ns1`, and namespace prefix `ps` is associated with `http://foo`. At the first `address` element's scope the default namespace is redeclared to `http://bar`. At the second `address`, the default namespace is undeclared. At the third `address` we see an error, the namespace prefix ps is being tried to undeclared, this will lead to a runtime error. At the last address, `ps` is redeclared to `http://bar`.

Name Identity

In order to uniquely identify a name in Xml, the pair: name of the element and the namespace of the element are used. Two elements with the same name and bound to the same namespace are considered to be identical.

```
<root xmlns="http://foo">
  <p:n xmlns:p="http://foo">1</p:n>
  <p:n xmlns:p="http://bar">2</p:n> <!-- different than 1 and 3 -->
  <k:n xmlns:k="http://foo">3</k:n>
  <n>4</n>
</root>
```

Figure 5.

In this example, the elements that have the value of 1, 3 and 4 have identical names. Identity of an element depends on the namespace that the element is in.

Attributes and Namespaces

Attributes do not bind to any namespaces unless they are prefixed. Therefore an attribute never binds to a default namespace. Consider following examples:

Attribute in default namespace

```
<list xmlns="http://tempuri.org/list.xsd">
  <address>
    <postCode type="US">98052</postCode>
  </address>
</list>
```

Figure 6.

In this XML document the `address` and `postCode` is bound to `http://tempuri.org/list.xsd`, but the attribute `type` is bound to empty namespace.

Prefixed attribute

```
<list xmlns="http://tempuri.org/list.xsd"
xmlns:ps="http://tempuri.org/person.xsd">
  <ps:address>
    <ps:postCode ps:type="US">98052</ps:postCode>
  </ps:address>
</list>
```

Figure 7.

In this XML document the `address`, `postCode` and attribute `type` is bound to `http://tempuri.org/person.xsd`, as all of them are prefixed.

Attribute in prefixed element

```
<list xmlns="http://tempuri.org/list.xsd"
xmlns:ps="http://tempuri.org/person.xsd">
  <ps:address>
    <ps:postCode type="US">98052</ps:postCode>
  </ps:address>
</list>
```

Figure 8.

In this XML document the `address` and `postCode` is bound to `http://tempuri.org/person.xsd`, but the attribute `type` is bound to empty namespace.

Namespaces in XSLT

XSLT is an XML based language for transforming an XML document to another document which is usually, but not necessarily another XML document. Its syntax is XML based; therefore it is no big surprise that namespaces are widely used in XSLT. We can group all places where namespace are used in XSLT three groups:

1. In the syntax of the language.
    a. XSLT recognizes its instructions by their namespace. All instructions should have "http://www.w3.org/1999/XSL/Transform" namespace.
    b. Names of internal XSLT objects (templates, modes, attribute sets, keys, decimal-formats, variables and parameters) are also qualified names. If they have a prefix then the prefix is expanded into URI using standard NS resolving rules. The default namespace is not used for unprefixed names.
2. In the XPath expressions to query nodes from source document. The default namespace is not used any for unprefixed names in XPath expressions.
3. In the names of constructed elements. In contradiction, the default namespace is used for unprefixed names there.

In all cases where name is expected to be qualified name and it has a prefix this prefix should be defined in context of this name:

1. Qualified name is name of element or attribute. Consider the following example:

```
<xsl:copy my:att="q:bar" />
```

`xsl` and `my` should be defined for `xsl:copy`, or else this document will not be a valid XML document.

2. Qualified name is in the value of attribute. Consider the following example:

```
<xsl:variable name="n1:foo" select="n2:bar/foo" my:att="q:bar" />
```

`n1` and `n2` should be defined in context of `xsl:variable`, but namespace prefix `q` in both samples is not required to be defined to any resource. `my:att` is not XSLT attribute and its context is not expected to be a valid qualified name.

Namespaces in XSLT

As XPath queries needs to get information about namespaces, we need to provide the namespace information to XPath. As we also mentioned in the default namespace rules section, we should transfer the namespace of the item that we are selecting.

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <foo:root xmlns:foo="http://foo">
      <xsl:element name="foo:{/default:elements/default:block}"
           xmlns:default="http://default">eleVal</xsl:element>
    </foo:root>
  </xsl:template>
</xsl:stylesheet>
```

XML Document:

```
<elements xmlns="http://default">
  <block>blockValue1</block>
  <block xmlns="http://foo">blockValue2</block>
```

```
  <bl:block xmlns:bl="http://bar">blockValue3</bl:block>
</elements>
```
Output:
```
<foo:root xmlns:foo="http://foo">
  <foo:blockValue1>eleVal</foo:blockValue1>
</foo:root>
```

Figure 9. Querying the XML Document for a element name using XPath, where namespaces exist.

The source XML document in this example has a default namespace `http://default`. We will query one of its nodes to create an element in our XSLT query. To do this, we need to pass the appropriate namespace to the XPath query that will select the correct item.

The steps to define the element name we are going to use in our `xsl:element` declaration is: resolve any namespaces and prefixes, run the XPath query, get the result string and create the element name.

In order to query against elements with namespaces, we have to pass the namespace context to XPath query, therefore we are binding the prefix `default` to `http://default`, and using this prefix for querying the document with `/default:elements/default:block`. When the XPath query returns with a string (in our case it is `blockValue1`) where we will use it as the element name, we are binding the element name to namespace `http://foo` with `foo` prefix. The result is `<foo:blockValue1>eleVal</foo:blockValue1>`.

Namespaces in XPath

Using XPath for quering documents in C#

The XPath language can be used outside XSLT to select nodes in the documents which support IXPathNavigable interface. .NET Framework provides three such documents: XmlDocument, XPathDocument and XmlDataDocument.

Here is an example of creating XPathNavigator, executing an XPath query and iterating through results:

```
// Create document and load XML to it. It can be XmlDocument,
// XPathDocument or any other document that support IXPathNavigable
XmlDocument doc = new XmlDocument();
doc.Load("myXmlFile.xml");

// Create XPathNavigator for this document. CreateNavigator() is the
// only method IXPathNavigable interface defines
XPathNavigator nav = doc.CreateNavigator();

// select all nodes with name "order" which has price attribute
// with a value > 10
string xpathExpression = "//order[@price>10]";

// Select sequence of nodes (node-set):
XPathNodeIterator iterator = nav.Select(xpathExpression);

// Iterate through node-set:
while (iterator.MoveNext())
{
```

```
    XPathNavigator curNode = iterator.Current;
    Console.WriteLine(curNode.LocalName);

    // XPathNavigator created for XmlDocument can give you
    // underlying XmlNode it points to
    // Not all navigators will do this. Property UndelineObject of
    // XPathNavigator create for XPathDocument always returns null
    XmlNode node = (XmlNode)curNode.UnderlyingObject;
}

// Alternately you can select one (first) node:
XPathNavigator firstNode = nav.SelectSingleNode(xpathExpression);
```

Figure 10.

In addition to executing XPATH expressions with help of XPathNavigator XmlDocument offers a simplified API for selecting nodes.

```
XmlDocument doc = new XmlDocument();
doc.Load("myXmlFile.xml");

string xpathExpression = "//order[@price>10]";

// Select sequence of nodes (node-set):
XmlNodeList nodeList = doc.SelectNodes(xpathExpression);

// Iterate through node-set:
foreach (XmlNode node in nodeList)
{
    Console.WriteLine(node.LocalName);
}

// Alternately you can select one (first) node:
XmlNode firstNode = doc.SelectSingleNode(xpathExpression);
```

Figure 11.

In both samples we executed an XPath expression that does not have namespaces. If ethe lements in your document belong to some namespace you would need to specify the namespace of the element in your expression.

For example to query `block` element from the previous example, XPath expression may be `//ns:elements/ns:block` where the prefix `ns` is bound to the namespace `http://default`. In the case of XSLT, the XPath engine uses the namespace context of the element where the expression is written to resolve prefixes to namespaces. Since C# is not an XML-derived language and doesn't have the concept of XML namespaces, the context must be passed to the XPath engine explicitly with the query.

Namespace context for XPathQuery in .NET framework 1.0 and 1.1

The following sample demonstrates how namespace context may be passed to the XPath engine n the .NET Framework 1.0 :

Person.Xml:

```
<list xmlns="http://tempuri.org/list.xsd"
xmlns:ps="http://tempuri.org/person.xsd">
  <ps:address>
    <ps:streetAddress>1 Microsoft Way</ps:streetAddress>
    <ps:postCode type="US">98052</ps:postCode>
    <ps:country>USA</ps:country>
  </ps:address>
</list>

XmlDocument xDoc = new XmlDocument();
xDoc.Load("person.xml");
//Also:
//XPathDocument xpDoc = new XPathDocument("person.xml");
//XPathNavigator nav = xpDoc.CreateNavigator();
XPathNavigator nav = xDoc.CreateNavigator();

XPathExpression expr = nav.Compile("//person:postCode");
XmlNamespaceManager xn = new XmlNamespaceManager(nav.NameTable);
xn.AddNamespace("local", "http://tempuri.org/list.xsd");
xn.AddNamespace("person", "http://tempuri.org/person.xsd");
xn.AddNamespace("empty", "");
expr.SetContext(xn);
XPathNodeIterator iter = nav.Select(expr);

while (iter.MoveNext()) {
    Console.WriteLine(iter.Current.InnerXml);
}
Console.ReadLine();
```

Figure 12. Creating XPathNavigator from a XmlNode or using XPathDocument class

This example will use Person.xml file. The namespace bindings works exactly as we mentioned in the Namespaces section: `list` is bound to the default namespace, `address` and `postCode` is bound to person, `type` is bound to an empty namespace. In order to select them, our XPath queries with the `XmlNamespaceManager` settings (see figure) should be as follows:

`list`	`//local:list`
`address`	`//person:address`
`postCode`	`//person:postCode`
`type`	`//@empty:type`

Figure 9. Possible XPath Queries for Figure 8

In the example above, we are specifying the namespaces for each prefix we are creating. Prefixes used in the XPath expression (`person`) do not need to be the same as used in the source document (`ps`).

Namespace context for XPath queries in .NET Framework 1.0 and 1.1

The first question everybody asks is when looking on example above is “Why this is so complex?”.

All namespaces you may want to use in your XPath query may already present in the source document and bound to some prefixes. Why not allow to using this prefixes in the expression automaticaly?

There are several reasons for this:

1. The same prefix may be bound to diferent namespaces in deferent nodes of document (which should system use in this case?);
2. To find all namespaces and prefixes engine would need to traverse entire document – this may be too expensive;
3. When source document redefines default namespace, no prefix exist in source document to address this new namespace, therefore elements in this namespace would be unreachable by XPath.

Another side of complexity is in API as well. XmlNamespaceManager provides no functionality that is not required for XPath engine that complicates writing custom XmlNamespaceManagers. Select() and SelectNodes() don't accept XmlNamespaceManager directly and require you to compile XPath string before use.

.NET Framework 2.0 adds additional interfaces and method overloads to simplify XPath usage. In the next sections we describe these improvements as well as provide samples and code snippets to make XPath usage easier.

Namespace context for XPath queries in .NET Framework 2.0

As we said above, in order to execute XPath expression, the engine needs a way of resolving prefixes to namespaces. .NET 2.0 adds a simplified interface for doing this: IXmlNamespaceResolver. It also adds several new method overloads to XPathNavigator that accept IXmlNamespaceResolver as an argument.

Quering XPath became easier but you still need the way to create something that implements the IXmlNamespaceResolver interface. XmlNamespaceManager does this, so you can use code from the previous sample to create and fill a namespace manager. More interesting is that XPathNavigator implements IXmlNamespaceResolver by itself. It will resolve prefixes to namespaces in context of XML node it is positioned on.

So consider that we know that all namespaces are defined on the topmost node of a source document. We now can use an XPathNavigator positioned on this node as a namespace resolver for our queries.

Person.Xml:

```
<list xmlns="http://tempuri.org/list.xsd"
xmlns:ps="http://tempuri.org/person.xsd">
  <ps:address>
    <ps:streetAddress>1 Microsoft Way</ps:streetAddress>
    <ps:postCode type="US">98052</ps:postCode>
    <ps:country>USA</ps:country>
  </ps:address>
</list>
```

```
XmlDocument xDoc = new XmlDocument();
xDoc.Load("person.xml");
XPathNavigator nav = xDoc.CreateNavigator();

XPathNavigator context = nav.Clone();
context.MoveToFirstChild();

XPathNodeIterator iter = nav.Select("//person:postCode", context);

while (iter.MoveNext()) {
    Console.WriteLine(iter.Current.InnerXml);
```

```
}
```

Figure 13.

Please note that we have to call the `MoveToFirstChild()` method because the real root node in the document is not a topmost node -- we see in the text that the only prefix defined on the root node is 'xml' itself.

Consider an environment that we have an XML document and transformations over it, where we control all of the prefix definitions, and there are no redefinitions of prefixes anywhere on the document. We can use the following example if we don't want to update our prefix-namespace bindings in our code if the resource strings of namespaces in XML document change, and prefixes stay the same.

DocumentNsResolver

Previous sample will be helpful for the documents when all prefixes used in the query are defined on the topmost element.

It's possible, but not common, that prefixes will be defined and redefined on other elements of the document. To simplify selecting nodes in such documents we decided to write helper class `DocumentNsResolver` that implements `IXmlNamespaceResolver` that collects the namespace prefix couples by visiting all namespace declarations in the document.

All concerns that we expressed above regarding such class is of course valid here, and we address them as follows:

1. The same prefix can be bound to different namespaces in the different places in the same document. – In the `DocumentNsResolver` the first definition wins but `DocumentNsResolver.RedefinedPrefixes` property allows you to check for this condition.
2. Construction requires `DocumentNsResolver` to do an entire tree traversal and may be time consuming. – this is not that expensive,because the document is already cached anyway. Create it once and use it for multiple queries.
3. Default namespace doesn't have a prefix "by default". – The `DocumentNsResolver.DefaultPrefix` property can be used to set a prefix `name` that can be used to bind to the default namespace. By default, this property set to the string 'default-prefix'.

Here is an example how DocumentNsResolver can be used for creating namespace context.

```
XmlDocument xDoc = new XmlDocument();
xDoc.Load("person.xml");
XPathNavigator nav = xDoc.CreateNavigator();

DocumentNsResolver nsResolver = new DocumentNsResolver(nav);
XPathNodeIterator nl = nav.Select("//ps:postCode", nsResolver);
```

Using XPath with XmlDocument in .NET Framework 2.0

With the introduction of .NET 2.0, we introduced XPathNavigator. To provide similar functionality over XmlDocument, we are introducing the XmlNamespaceManager helper class that you can find at Appendix 2. XmlNamespaceManager simplifies XPath queries over XmlDocument, utilizes the new class XPathNavigator, and offers better performance than XmlDocument.Select().

XmlDocument does not accept `IXmlNamespaceResolver` for resolving prefixes. XmlDocument.SelectNodes() function requires passing a XmlNamespaceManager for prefix to namespace resolution. DocumentNsManager extends XmlNamespaceManager and contains DocumentNsResolver as prefix-namespace resolver.

The following sample shows how to use DocumentNsManager to pass namespace context to SelectNodes() method.

```
XmlDocument xDoc = new XmlDocument();
xDoc.Load("xs\\person.xml");

DocumentNsManager nsManager = new DocumentNsManager(xDoc);
XmlNodeList nl = xDoc.SelectNodes("//ps:postCode", nsManager);
foreach (XmlNode n in nl)
    Console.WriteLine(n.InnerText);
```

XPath

XPath expressions always work against a context. A context consists of a context node (a node in XML Document); context position and size; a set of variable bindings that is a mapping of variable names to variable values; a function library that is a mapping from a function name to functions, and the set of namespaces that is a mapping of namespace prefixes to actual namespaces. Therefore we need to provide this information to XPath for it to work correctly.

The following query uses the XMLDocument.SelectNodes() method to query the XmlDocument. Notice that we are creating an XmlNamespaceManager from the NameTable of the document, and adding the namespaces that we would like to use in our XPath query with the prefixes that we will use in our queries. By passing the XmlNamespaceManager we created to the XPath query, we are setting the namespaces context of XPath. We are passing the node to the XPath query by `x.SelectNodes`. So the two crucial attributes of context (the context node and namespaces) are set, the rest are left as default.

```
XmlDocument x = new XmlDocument();
x.Load("person.xml");
XmlNamespaceManager xn = new XmlNamespaceManager(x.NameTable);
xn.AddNamespace("local", "http://tempuri.org/list.xsd");
xn.AddNamespace("person", "http://tempuri.org/person.xsd");
xn.AddNamespace("empty", "");
XmlNodeList list = x.SelectNodes("//person:postCode",xn);
foreach (XmlNode n in list)
{
    Console.WriteLine(n.InnerXml);
}
```

Figure 12. Using SelectNodes()

This example will use the Person.xml file in Figure 8. The namespace bindings works exactly as we mentioned in Namespaces section: `list` is bound to the default namespace, `address` and `postCode` is bound to person, `type` is bound to empty namespace.

XmlDocument.SelectNodes() can be mimicked with using XPathNavigator, which will perform better. The following example can be used instead of XmlDocument.SelectNodes().

```
XmlNodeList SelectNodes(XmlNode From, XmlNode Context, string
Expression)
{
    XPathNavigator navi = From.CreateNavigator();
```

```
    XPathNodeIterator it = navi.Select(Expression,
                                       Context.CreateNavigator());
    return new XPathNodeList(it);
}



XmlNodeList nl = SelectNodes(x, x.DocumentElement, "//ps:postCode");
foreach (XmlNode node in nl)
{
    Console.WriteLine(node.InnerXml);
}
```

Figure 19. Using the XPathNavigator for selecting nodes

In this sample, the Select() function accepts two XmlNodes, the first one is for specifying where the select operation will be done, the second one is for specifying the context of namespaces. On our XML example, as the `ps` prefix is defined on the the `list` element that is the document element, we are passing it. The most common place to define prefix tags would be the document element, passing the document element would be okay for majority of the XML files that we work with.

Workarounds with Namespaces

In this section we would like to give some widely accepted examples that are used for selecting nodes with namespaces.

Selecting elements regardless of the namespace declaration

If we would like to select all nodes regardless of the namespaces or prefixes they are bound to we could use the following query.

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="text()" />
  <xsl:template match="/">
    <root>
      <xsl:apply-templates/>
    </root>
  </xsl:template>
  <xsl:template match="*[local-name() = 'block']">
    <xsl:copy />
  </xsl:template>
</xsl:stylesheet>
```

XML Document:

```
<elements xmlns="http://default">
  <element>elementValue1</element>
  <block>blockValue1</block>
  <block xmlns="http://foo">blockValue2</block>
  <bl:block xmlns:bl="http://bar">blockValue3</bl:block>
</elements>
```

Output:

```
<root>
  <block xmlns="http://default" />
  <block xmlns="http://foo" />
  <bl:block xmlns:bl="http://bar" xmlns="http://default" />
```

```
</root>
```

Figure 20.

In this example `*[local-name() = 'block']` will match all elements regardless of their namespaces, whose local name (element's name) is equal to block.

Selecting elements with specific name and namespace declaration

In the above example, if we change the match part of the template with `*[local-name()='block' and namespace-uri()='http://bar']`, the output will be the following. In this example we are matching will match all elements with name `block` and at namespace `http://bar`.

Output:

```
<root>
  <bl:block xmlns:bl="http://bar"
xmlns="http://default">element</bl:block>
</root>
```

Figure 21.

Also if we are in control of the XML document, and we would like to stick with prefixes, we can use `*[name()='bl:block']` at the match parameter of the template. However, this would not be very stable, as the prefix and or the binding of the prefix could change over time.

Conclusion

Namespaces in XML is a W3C standard for providing uniquely named elements and attributes in an XML instance. A Namespace is used to distinguish elements bound to different contexts. The namespace declarations on XML Namespaces are a URI which is just a virtual identifier for that namespace that does not need to map to an actual resource. We may redeclare namespaces to different resources in our document, however we cannot undeclare prefixes. The attributes of elements are not bound to any namespace unless prefixed. Identity of an element depends on the namespace that the element is in, and the name of the element.

Namespaces affect XSLT in three ways: the syntax of the XSLT depends on the namespaces; the XPath expressions in XSLT needs a context to run against the source, but XPath queries are not affected from the default namespace; lastly, the names of constructed elements does depend on the default namespace.

XPath can be used with XmlDocument and IXPathNavigator. In these cases, XPath requires a context to associate namespaces that the elements and attributes are bound to. This could be done by setting the context of XPathExpression, assigning a namespace resolver either to XPathNavigator's select method or XmlDocument's SelectNodes method.

Finally we talked about some common workarounds for selecting nodes in XPath.

We hope that this document helped you to visualize practical the usage of namespaces over XSLT and XPath. You may always read more about these topics in the W3C's Recommendation and documents at Microsoft's web sites.

Appendix 1 – DocumentNsResolver

```
using System;
using System.Xml.XPath;
using System.Collections.Generic;
using System.Xml;
class DocumentNsResolver : IXmlNamespaceResolver
{
    Dictionary<string, string> nsCollection = new Dictionary<string, string>();
    List<string> dupPrefixes = new List<string>();
    string defaultPrefix = "default-prefix";

    public DocumentNsResolver(XmlNode node) : this(node.CreateNavigator(), true) { }
    public DocumentNsResolver(XmlNode node, bool moveToRoot) : this(node.CreateNavigator(), moveToRoot) { }
    public DocumentNsResolver(XPathNavigator nav) : this(nav, true) { }

    public DocumentNsResolver(XPathNavigator nav, bool moveToRoot)
    {
        if (moveToRoot)
        {
            nav = nav.Clone();
            nav.MoveToRoot();
        }

        // xmlns:xml="http://www.w3.org/XML/1998/namespace" is always implicitly defined
        nsCollection.Add("xml", "http://www.w3.org/XML/1998/namespace");

        // Enumerate all elements in the subtree:
        XPathNodeIterator it = nav.SelectDescendants(XPathNodeType.Element, /*matchSelf:*/true);
        while (it.MoveNext())
        {
            XPathNavigator cur = it.Current;

            // Enumerate all namespace declared on current element
            if (cur.MoveToFirstNamespace(XPathNamespaceScope.Local))
            {
```

```
                do
                {
                    string prefix = cur.LocalName;
                    string ns = cur.Value;
                    string prevNS;
                    if (!nsCollection.TryGetValue(prefix, out prevNS))
                    {
                        //add the prefix-namespace couple to the nsCollection
                        nsCollection.Add(prefix, ns);
                    }
                    else
                    {
                        // We already added this prefix. Check that it's bound to the same namespace
                        // if not -- add it to the dupPrefixes collection
                        if (prevNS != ns)
                        {
                            // prefix redefined to diferent namespace
                            dupPrefixes.Add(prefix);
                        }
                    }
                }
                while (cur.MoveToNextNamespace(XPathNamespaceScope.Local));
                cur.MoveToParent();
            }
        }
    }

    public string DefaultPrefix
    {
        get { return this.defaultPrefix; }
        set { this.defaultPrefix = (value ?? string.Empty); }
    }

    public List<string> RedefinedPrefixes { get { return dupPrefixes; } }

    public IDictionary<string, string> GetNamespacesInScope(XmlNamespaceScope scope)
    {
        return nsCollection;
```

```
    }

    public string LookupNamespace(string prefix)
    {
        if (prefix == this.defaultPrefix)
        {
            prefix = string.Empty;
        }
        string namesp;
        if (nsCollection.TryGetValue(prefix, out namesp))
        {
            return namesp;
        }
        else
        {
            return null;
        }
    }

    public string LookupPrefix(string namespaceName)
    {
        //XPath never asks for the prefix of a Namespace.
        throw new NotSupportedException();
    }
}
```

Appendix 2 – DocumentNsManager

```
using System;
using System.Xml.XPath;
using System.Collections.Generic;
using System.Collections;
using System.Xml;
public class DocumentNsManager : XmlNamespaceManager
{
    IXmlNamespaceResolver resolver;

    public DocumentNsManager(XmlNode node) : this(new DocumentNsResolver(node)) { }
    public DocumentNsManager(XmlNode node, bool moveToRoot) : this(new DocumentNsResolver(node,
moveToRoot)) { }
    public DocumentNsManager(XPathNavigator nav) : this(new DocumentNsResolver(nav)) { }
    public DocumentNsManager(XPathNavigator nav, bool moveToRoot) : this(new DocumentNsResolver(nav,
moveToRoot)) { }

    public DocumentNsManager(IXmlNamespaceResolver resolver)
        : base(new NameTable())
    {
        this.resolver = resolver;
    }

    public override IEnumerator GetEnumerator()
    {
        return GetNamespacesInScope(XmlNamespaceScope.All).Keys.GetEnumerator();
    }

    public override IDictionary<string, string> GetNamespacesInScope(XmlNamespaceScope scope)
    {
        return resolver.GetNamespacesInScope(scope);
    }

    public override string LookupNamespace(string prefix)
```

```
    {
        return resolver.LookupNamespace(prefix);
    }

    public override string LookupPrefix(string uri)
    {
        return resolver.LookupPrefix(uri);
    }

    public override bool HasNamespace(string prefix)
    {
        return resolver.LookupNamespace(prefix) != null;
    }

    // Not Supported:
    public override XmlNameTable NameTable { get { throw new NotSupportedException(); } }
    public override string DefaultNamespace { get { throw new NotSupportedException(); } }

    public override void PushScope() { throw new NotSupportedException(); }
    public override bool PopScope() { throw new NotSupportedException(); }

    public override void AddNamespace(string prefix, string uri) { throw new NotSupportedException(); }
    public override void RemoveNamespace(string prefix, string uri) { throw new NotSupportedException(); }
}
```

Appendix 3 - XPathNodeList

```
class XPathNodeList : XmlNodeList {
    List<XmlNode> list = new List<XmlNode>();

    public XPathNodeList(XPathNodeIterator nodeIterator) {
        while (nodeIterator.MoveNext()) {
            XmlNode node = (XmlNode)nodeIterator.Current.UnderlyingObject;
            if (node == null) {
                throw new Exception("Not supported XPathNodeIterator");
            }
            list.Add(node);
        }
    }

    public override int Count { get { return list.Count; } }

    public override XmlNode Item(int index) {
        if (index < 0 || list.Count<= index) {
            return null;
        }
        return list[index];
    }

    public override IEnumerator GetEnumerator() {
        return (IEnumerator)list.GetEnumerator();
    }
}
```

References

1. Bray, T., Hollander, D. Layman, A. (1999). Namespaces in XML, http://www.w3.org/TR/REC-xml-names/
2. Clark, J. (1999). XSL Transformations, http://www.w3.org/TR/1999/REC-xslt-19991116
3. Clark, J. (1999). XML Path Language. http://www.w3.org/TR/1999/REC-xpath-19991116
4. Obasanjo, D. (2002). XML Namespaces and How They Affect XPath and XSLT. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml05202002.asp

5. Microsoft XML Team's WebLog. http://blogs.msdn.com/xmlteam/
6. Wikipedia (2006). http://en.wikipedia.org/wiki/XML_namespace