XSLT 1.0 Multiple Namespace Issues

Finnbarr P. Murphy
(fpm@fpmurphy.com)

XSLT and XPath assume that XML documents conform to the XML Namespaces recommendation whereby XML namespaces are identified by a Uniform Resource Identifier (URI). One form of a URI is a URL, e.g. *http://blog.fpmurphy.com*. Another form of URI is a URN, e.g. *urn:biztalk-org:biszalk:biztalk_1*. A namespace URI does not need to actually point to anything. Whilst in theory a namespace URI is intended to be globally unique, in practice it just needs to be unique within the local scope in which you are using it.

There are two main reasons to use XML namespaces: to avoid name collisions, and to facilitate name recognition. Namespaces allow you to use multiple markup vocabularies within the same document without having to worry about name collisions. Just as important in my opinion is that fact that XML namespaces facilitate recognition of elements or attributes based only on their associated URI.

XML namespaces are declared with an *xmlns* attribute, which associates a prefix with the namespace. Note that, except for a couple of exceptions documented in the recommendations, colons (':') are reserved for namespace prefixes. Consequently, any element type or attribute name in an XML namespace is assumed to be uniquely identified by a two-part name: the name of its XML namespace and it's local name. This two-part naming system is what is defined in the XML namespaces recommendation.

Prefixes can be either *null* or *non-null*. A namespace declaration for a *non-null* prefix is written thus:

```
xmlns:prefix="http://mycompany.com/namspace"
```

while a namespace declaration for a *null* prefix is written as:

```
xmlns="http://mycompany.com/namespace"
```

The null prefix namespace is more commonly known as the *default* namespace and the declaration is known as the *default namespace declaration*. A default namespace makes an XML document easier to read because it allows you to just give the local name of an element rather than using namespace prefixes. Compare this example document:

```
<employee status="Guru" xmlns="http://www.example.com/ns/employee">
    <first>John</first>
    <last>Kane</last>
    <department>IT</department>
    <country>IE</country>
</employee>
```

against this document:


```
<emp:employee status="Guru" xmlns:emp="http://www.example.com/ns/employee">
    <emp:first>John</emp:first>
    <emp:last>Kane</emp:last>
    <emp:department>IT</emp:department>
     <emp:country>IE</emp:country>
</emp:employee>
```

They are equivalent documents as far as XML is concerned. The default namespace version is far easier to read and understand but can quickly lead to unintended results. In some cases, specifically documents that are validated against a DTD or interpreted by non-namespace-aware applications, you might be forced to use the default namespace. A typical case is (X)HTML documents.

Consider the following trivial XML document:

```
<foo xmlns="http://example.com"
       xmlns:ns1="http://example.com/ns1">
   <bar>dog</bar>
</foo>
```

How many namespace nodes do you think are in the above document? Two? Three? The correct answer is six. Each of the two elements has three namespace nodes attached to it: a default namespace, the XML namespace, and the namespace bound to the prefix *ns1*. This is equivalent representation of the above document using all namespace declarations:

```
<foo xmlns="http://example.com"
     xmlns:ns1="http://example.com/ns1"
     xmlns:xml="http://www.w3.org/XML/1998/namespace">
  <bar xmlns="http://example.com"
       xmlns:ns1="http://example.com/ns1"
       xmlns:xml="http://www.w3.org/XML/1998/namespace">
       dog
  </bar>
</foo>
```

Why is this? It is because the scope of a namespace is the element on which it appears and all its children and descendants, excluding any subtree where the same prefix is associated with a different URI, i.e. overwritten. Any *name* with this prefix is automatically associated with the given namespace URI.

What exactly is a *name*? A *name* can be the name of an element, an attribute, a variable , a function, a template, keys, etc. It has three properties: a *prefix*, a *local part* and a *namespace* URI. For example, consider the name *xsl:element*. The prefix is *xsl* and the local part is *element*. The namespace URI is the innermost element that contains a namespace declaration for the *xsl* prefix. If a name has no prefix, then its namespace is the default namespace in the case of an element or a null URI otherwise. The exception to this rule is in XPath expressions where the default namespace is not used to qualify unprefixed names even if the name is an element. The various components of a *name* can be retrieved using either *name()*, *local-name()* or *namespace-uri()*. Note, however, that no function is specified to return a namespace prefix.

Most of the time you will not be directly working with namespace nodes. They just get copied along with the element nodes that you copy from the source tree or stylesheet. When either the *xsl:copy* or the *xsl:copy-of* instruction is applied to an element node, it copies all of the element's namespace nodes along with it, thereby ensuring that the same namespace bindings are in scope

in the result tree as were in scope for that element in the source tree. Similarly, whenever you use a literal result element in a stylesheet, it is treated as an instruction to copy that element from the stylesheet into the result tree together with any namespace nodes that are in scope. However there are times when you will want to explicitly control the namespaces and namespace declarations of the output document.

Whenever you use *xsl:element* to create an element or *xsl:attribute* to create an attribute, the processor will automatically create the necessary namespace nodes in the result tree, and the corresponding namespace declarations, to ensure that the result is well-formed as far as namespaces are concerned.

Here is an example of a local namespace where the *pm* namespace is applied against the title element only:

```
<poems>
   <poem>
      <pm:title xmlns:pm="http://example.com/poem">
       The Love Song of J. Alfred Prufrock
      </pm:title>
      <author>T.S. Elliot</author>
   </poem>
  ...
</poems>
```

and here is an example of multiple namespaces within an XML document where one namespace is defined against the root element, and another against a child element.

```
<pm:poems xmlns:pm="http://example.com/poem">
    <pm:poem>
        <pm:title>The Waste Land</pm:title>
        <pm:author>T.S. Elliot</pm:author>
        <pub:name xmlns:pub="http://example.com/publisher">Boni and Liveright</pub:name>
        <pub:email>pub@boniliverright.com</pub:email>
    </pm:poem>
    ...
</pm:poems>
```

Here is an example of overriding a namespace declaration:

```
<pm:poems xmlns:pm="http://example.com/poem">
    <pm:poem>
        <pm:title xmlns:pm="http://example.com/greatpoem">The Waste Land</pm:title>
        <pm:author>T.S. Elliot</pm:author>
        <pub:name xmlns:pub="http://example.com/publisher">Boni and Liveright</pub:name>
        <pub:email>pub@boniliverright.com</pub:email>
    </pm:poem>
    ...
</pm:poems>
```

Here the overwritten namespace applies only to the *title* element.

Attributes don't take on the default namespace and are not usually placed in a namespace because they are traditionally thought to be "owned" by their parent element. In most cases, placing an attribute in a namespace adds no value. The exception is the case of *global attributes* such as XML Schema types, e.g. *xsi:type*, which are placed in a namespace to avoid namespace collision with any of an element's other attributes. However namespaces can also be applied to attributes. Here

is example where attributes have the same local name but different namespaces:

```
<product xmlns="http://www.example.com/prod"
              xmlns:app="http://www.example.com/app">
     <product_number>100</product_number>
     <product_size app:type="R12"  type="EU">10</product_size>
</product>
```

and here is another slightly different example:

```
<product xmlns="http://www.example.com/prod"
              xmlns:prod="http://www.example.com/prod">
     <product_number>100</product_number>
     <product_size prod:type="R12"  type="EU">10</product_size>
</product>
```

The default namespace is not used for XPath 1.0 expressions where an unprefixed name in an expression always means *not in a namespace*. For example, the expression *foo* means “select all child elements that have local name *foo* and that are not in a namespace.”

Namespace declarations are not exposed as attributes in the XPath data model. So if namespace declarations aren’t attributes, what are they? It turns out that namespace declarations have no direct representation in the XPath data model. Instead, each element has a set of one or more namespace nodes attached to it that represents the set of namespaces that are in scope for that element. Moreover, even if you do not use namespaces, each element still has the implicit XML namespace node attached to it.

Consider the following trivial XML document.

```
<poem xmlns:red="http://www.example.com/red"
      xmlns:green="http://www.example.com/green">
   <green:author>T.S. Elliot</green:title>
   <red:title>The Waste Land</red:title>
   <green:line>April is the cruelest month, breeding </green:line>
   <red:line>Lilacs out of the dead land, mixing</red:line>
   <green:line>Memory and desire, stirring</green:line>
   <line>Dull roots with spring rain.</line>
</poem>
```

It contains the first four lines of a famous poem by one of the most influential English-language poets of the early 20th century, T.S.Elliot. If you have not read his poems, you should take some time out and do so. His poetry is extremely powerful and moving and lead to Elliot being awarded the Nobel Prize in Literature in 1948.

Let us use the following stylesheet to examine the various components of this document’s namespace with a view to getting a better understanding of namespaces.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:red="http://www.example.com/red"
                xmlns:green="http://www.example.com/green"
                version="1.0">
    <xsl:output method="text"/>
    <xsl:template match="poem">
     Namespace nodes:
         <xsl:for-each select="namespace::*">
```



```
            <xsl:value-of select="name()"/><xsl:text> </xsl:text>
         </xsl:for-each>
         <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="red:line">
     Matched a red element:
        name          <xsl:value-of select="name()"/>
        local-name    <xsl:value-of select="local-name()"/>
        namespace-uri <xsl:value-of select="namespace-uri()"/>
        contents      <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="green:line">
     Matched a green element:
        name          <xsl:value-of select="name()"/>
        local-name    <xsl:value-of select="local-name()"/>
        namespace-uri <xsl:value-of select="namespace-uri()"/>
        contents      <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="line">
     Matched default namespace element:
        name          <xsl:value-of select="name(.)"/>
        local-name    <xsl:value-of select="local-name(.)"/>
        namespace-uri <xsl:value-of select="namespace-uri(.)"/>
        contents      <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="*"/>
</xsl:stylesheet>
```

Here is the output of the transformation:

```
     Namespace nodes:
         xml green red

     Matched a green element:
        name          green:line
        local-name    line
        namespace-uri http://www.example.com/green
        contents      April is the cruelest month, breeding

     Matched a red element:
        name          red:line
        local-name    line
        namespace-uri http://www.example.com/red
        contents      Lilacs out of the dead land, mixing

     Matched a green element:
        name          green:line
        local-name    line
        namespace-uri http://www.example.com/green
        contents      Memory and desire, stirring

     Matched default namespace element:
        name          line
        local-name    line
        namespace-uri
        contents      Dull roots with spring rain.
```

As you can see no *namespace-uri* data is outputted for the default namespace.

Consider the following stylesheet. Here the green namespace is declared to be *http://www.example.com/black* and the prefix was changed to be *grn* instead of *green*.

```
```



```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:grn="http://www.example.com/black"
                version="1.0">
     <xsl:output method="text"/>
     <xsl:template match="poem">
          <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="grn:line">
     Matched a green element:
        name          <xsl:value-of select="name()"/>
        local-name    <xsl:value-of select="local-name()"/>
        namespace-uri <xsl:value-of select="namespace-uri()"/>
        contents      <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="line">
     Matched a default namespace element:
        name          <xsl:value-of select="name()"/>
        local-name    <xsl:value-of select="local-name()"/>
        namespace-uri <xsl:value-of select="namespace-uri()"/>
        contents      <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="*"/>
</xsl:stylesheet>
```

Here is what is output when this stylesheet is used to transform our example document:

```

     Matched a default namespace element:
        name          line
        local-name    line
        namespace-uri
        contents      Dull roots with spring rain.
```

As you can see no elements in the green namespace were outputted due to the mismatch in the green namespace URI. This should underline the importance of ensuring that the input document and stylesheet URIs match correctly.

Here is how to output all the line elements regardless of their associated namespace:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:grn="http://www.example.com/green"  version="1.0">
     <xsl:output method="text"/>
     <xsl:template match="poem">
          <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="*[local-name()='line']">
     Matched an element:
         name          <xsl:value-of select="name()"/>
         local-name    <xsl:value-of select="local-name()"/>
         namespace-uri <xsl:value-of select="namespace-uri()"/>
         contents      <xsl:apply-templates/>
     </xsl:template>
     <xsl:template match="*"/>
</xsl:stylesheet>
```

XSLT 1.0 provides a crude method to control what namespaces are excluded from the output of a transformation. You can use the *exclude-result-prefixes* attribute of the *xsl:stylesheet* element to suppress specific namespaces from an output document:

```

```



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ns1="http://www.example.com/ns1"
      exclude-result-prefixes="ns1">
      ......
</xsl:stylesheet>
```

To suppress multiple namespaces from the output document specify them separated by whitespace:

```
exclude-result-prefixes="ns1 ns2 ns3"
```

Our next example uses *exclude-result-prefixes* to output the following new (and very simple) stylesheet:

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

from this XML document:

```
<v name="myNewVarName">myNewValue</v>
```

using this stylesheet:

```
<xsl:stylesheet version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:dummy="my:dummyNS" exclude-result-prefixes="dummy">
    <xsl:output omit-xml-declaration="yes" indent="yes"/>
    <xsl:namespace-alias result-prefix="xsl" stylesheet-prefix="dummy"/>
    <xsl:template match="/*">
        <dummy:stylesheet version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <dummy:variable name="{@name}">
             <xsl:value-of select="."/>
        </dummy:variable>
        </dummy:stylesheet>
   </xsl:template>
</xsl:stylesheet>
```

Note that *xsltproc* does not output the new stylesheet correctly; it includes the dummy namespace in its output. However Saxon 9 and other XSLT 1.0 processors that I tested the example on behave correctly.

For our next example, consider the following XML document. It has the usual default namespace and a namespace declared by the *http://www.example.com/schema/orders* URI.

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns:o="http://www.example.com/schema/orders">
  <request completionCode="0" customerID="1"/>
  <list total="1">
    <o:order CustomerID="1" orderNo="1" status="Shipped">
      <o:billToZipCode>32940</o:billToZipCode>
      <o:shipToZipCode>60000</o:shipToZipCode>
```



```
      <o:totals tax="1.80" subTotal="30.00" shipping="6.75"/>
    </o:order>
  </list>
</root>
```

Suppose you wanted to transform the above document by adding multiple namespaces to the document as follows:

```
<?xml version="1.0"?>
<root xmlns:o="http://www.example.com/schema/orders"
        xmlns:ns="http://www.example.com/schema/response"
        xmlns:nr="http://www.example.com/schema/request"
        xmlns:nd="http://www.example.com/schema/details">
  <nr:request completionCode="0" customerID="1"/>
  <ns:list total="1">
    <nd:order CustomerID="1" orderNo="1" status="Shipped">
      <nd:billToZipCode>32940</nd:billToZipCode>
      <nd:shipToZipCode>60000</nd:shipToZipCode>
      <nd:totals tax="1.80" subTotal="30.00" shipping="6.75"/>
    </nd:order>
  </ns:list>
</root>
```

Here is how to do the transformation using literal-result elements rather than by means of the *xsl:element* constructor:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:o="http://www.example.com/schema/orders"
    xmlns:ns="http://www.example.com/schema/response"
    xmlns:nr="http://www.example.com/schema/request"
    xmlns:nd="http://www.example.com/schema/details">
    <xsl:output indent="yes"/>
    <xsl:template match="@* | node()">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()"/>
        </xsl:copy>
    </xsl:template>
    <xsl:template match="root">
        <root>
            <xsl:apply-templates/>
        </root>
    </xsl:template>
    <xsl:template match="request">
        <nr:request>
            <xsl:apply-templates select="@* | node()"/>
        </nr:request>
    </xsl:template>
    <xsl:template match="list">
        <ns:list>
            <xsl:apply-templates select="@* | node()"/>
        </ns:list>
    </xsl:template>
    <xsl:template match="o:order">
        <nd:order>
            <xsl:apply-templates select="@* | node()"/>
        </nd:order>
    </xsl:template>
    <xsl:template match="o:billToZipCode">
        <nd:billToZipCode>
            <xsl:apply-templates select="@* | node()"/>
        </nd:billToZipCode>
```



```
    </xsl:template>
    <xsl:template match="o:shipToZipCode">
        <nd:shipToZipCode>
            <xsl:apply-templates select="@* | node()"/>
        </nd:shipToZipCode>
    </xsl:template>
    <xsl:template match="o:totals">
        <nd:totals>
            <xsl:apply-templates select="@* | node()"/>
        </nd:totals>
    </xsl:template>
    <xsl:template match="errors" />
</xsl:stylesheet>
```

and here is how to do the same transformation using the *xsl:element* constructor:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:o="http://www.example.com/schema/orders"
   xmlns:ns="http://www.example.com/schema/response"
   xmlns:nr="http://www.example.com/schema/request"
   xmlns:nd="http://www.example.com/schema/details">
  <xsl:output indent="yes"/>
  <xsl:template match="o:*">
     <xsl:element name="nd:{local-name()}" namespace="http://www.example.com/schema/detail
s">
        <xsl:apply-templates select="node()|@*"/>
     </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <xsl:element name="ns:{name()}" namespace="http://www.example.com/schema/response">
       <xsl:apply-templates select="node()|@*"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="@*">
     <xsl:copy-of select="."/>
  </xsl:template>
  <xsl:template match="/root">
    <xsl:element name="{name()}">
       <xsl:copy-of select="document('')/*/namespace::*[name()='nr' or name()='nd' or name
()='ns' or name()='o']"/>
       <xsl:apply-templates select="node()|@*"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="request">
    <xsl:element name="nr:{name()}" namespace="http://www.example.com/schema/request">
       <xsl:apply-templates select="node()|@*"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="o:*[ancestor-or-self::order]">
    <xsl:element name="nd:{name()}" namespace="http://www.example.com/schema/details">
       <xsl:apply-templates select="node()|@*"/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

This particular technique was demonstrated by Dimitre Novatchev in response to a question on Stack Overflow. In this technique, the identity template is replaced by the first three templates in the above stylesheet. However I recommend that you stick to the literal-results method unless you are quite proficient in XSLT.

For our next example, consider the following SOAP document:


```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ns1="h
ttp://integration.xyz.com/common/Errora.xsd">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>Error.800</faultcode>
            <faultstring>Communicating error</faultstring>
            <faultactor>System12</faultactor>
            <detail>
                <ns1:errorDetailItem>
                    <ns1:providerError>
                        <ns1:providerErrorCode>Error.800</ns1:providerErrorCode>
                        <ns1:providerErrorText>Failed to establish a backside connection</
ns1:providerErrorText>
                    </ns1:providerError>
                    <ns1:providerError>
                        <ns1:providerErrorCode>Error.800</ns1:providerErrorCode>
                        <ns1:providerErrorText>Communicating  error</ns1:providerErrorText
>
                        <ns1:errorSystem>System12</ns1:errorSystem>
                    </ns1:providerError>
                </ns1:errorDetailItem>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The required output from the transformation is the following:

```
<ns1:providerErrorCode>Error.800</ns1:providerErrorCode>
<ns1:providerErrorText>Failed to establish a backside connection</ns1:providerErrorText>
<ns1:providerErrorText>Communication error</ns1:providerErrorText>
```

The transformation is achieved using both a stylesheet and judicious use of the *sed* utility. Here is the stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://integration">
  <xsl:output method="xml" omit-xml-declaration="yes" />
  <xsl:template match="/">
     <xsl:apply-templates select="//ns1:errorDetailItem" />
  </xsl:template>
  <xsl:template match="//ns1:errorDetailItem" >
     <xsl:apply-templates select="//ns1:providerError[1]/ns1:providerErrorCode" />
     <xsl:apply-templates select="//ns1:providerErrorText" />
  </xsl:template>
  <xsl:template match="//ns1:providerErrorCode" >
     <xsl:element name="{local-name()}">
        <xsl:value-of select="."/>
     </xsl:element>
     <xsl:text>
</xsl:text>
  </xsl:template>
  <xsl:template match="//ns1:providerErrorText" >
     <xsl:element name="{local-name()}">
        <xsl:value-of select="."/>
     </xsl:element>
     <xsl:text>
</xsl:text>
```



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

and here is how the transformation was achieved:

```
$ xsltproc soap.xsl soap.xml | sed -e 's/provider/ns1:provider/g'
<ns1:providerErrorCode>Error.800</ns1:providerErrorCode>
<ns1:providerErrorText>Failed to establish a backside connection</ns1:providerErrorText>
<ns1:providerErrorText>Communicating  error</ns1:providerErrorText>
```

Sometimes the use of external utilities to modify namespace prefixes is warranted – so always consider this option when faced with transforming documents with multiple namespaces.

For our final example, consider the following document:

```
<work-order-list>
   <wo:job xmlns:wo="http:hello" freshness:timestamp="2006-01-12" xmlns:freshness="http://
freshness"
        history:timestamp="2006" xmlns:history="http:history"  xmlns:mnr="http:mnr">
       <wo:work-order id="1" status="k" freshness:timestamp="2006-01-13" />
   </wo:job>
   <wo:job xmlns:wo="http:hello" freshness:timestamp="2006-01-13" xmlns:freshness="http://
freshness"
        history:timestamp="2006" xmlns:history="http:history"  xmlns:mnr="http:mnr">
       <wo:work-order id="2" status="k" freshness:timestamp="2006-01-14" />
   </wo:job>
   <wo:job xmlns:wo="http:hello" freshness:timestamp="2006-01-14" xmlns:freshness="http://
freshness"
        history:timestamp="2006" xmlns:history="http:history"  xmlns:mnr="http:mnr">
       <wo:work-order id="3" status="k" freshness:timestamp="2006-01-15" />
   </wo:job>
</work-order-list>
```

The requirement is to transform this document into the following output:

```
<?xml version="1.0"?>
<work-order-list>
   <job timestamp="2006">
       <work-order id="1" status="k" timestamp="2006-01-13"/>
   </job>
   <job timestamp="2006">
       <work-order id="2" status="k" timestamp="2006-01-14"/>
   </job>
   <job timestamp="2006">
       <work-order id="3" status="k" timestamp="2006-01-15"/>
   </job>
</work-order-list>
```

Here is a stylesheet which meets this requirement:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:freshness="http://freshness"
  xmlns:history="http:history"
  xmlns:wo="http:hello">
  <xsl:output method="xml" indent="no"/>
```



```
  <xsl:template match="/|comment()|processing-instruction()">
    <xsl:copy>
      <!-- go process children (applies to root node only) -->
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="*">
    <xsl:element name="{local-name()}">
      <!-- go process attributes and children -->
      <xsl:apply-templates select="@*|node()"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="wo:*">
    <xsl:element name="{name()}" namespace="{namespace-uri()}">
      <!-- go process attributes and children -->
      <xsl:apply-templates select="@*|node()"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="@history:*">
    <xsl:attribute name="{name()}" namespace="{namespace-uri()}">
      <xsl:value-of select="."/>
    </xsl:attribute>
  </xsl:template>
  <xsl:template match="@freshness:*">
    <xsl:attribute name="{name()}" namespace="{namespace-uri()}">
      <xsl:value-of select="."/>
    </xsl:attribute>
  </xsl:template>
  <xsl:template match="@*">
    <xsl:attribute name="{local-name()}">
      <xsl:value-of select="."/>
    </xsl:attribute>
  </xsl:template>
</xsl:stylesheet>
```

You have probably noticed by now that this post only covers XSLT 1.0 and not XSLT 2.0. This was intentional as most people are still using XSLT 1.0 processors. Finally, if you want to learn more about XML namespaces, a great place to start is to read Ronald Bourret's FAQ. Another excellent discussion on namespaces is Evan Lenz's Namespaces In XSLT article. In particular, Lenz covers the enhancements to namespace processing introduced in the XSLT 2.0 recommendation.

