

# Removing XML whitespace in structured FrameMaker documents

WHITE PAPER



**Simon Bate**  
Senior Technical  
Consultant

**An ongoing frustration with using structured FrameMaker to generate great PDF files from DITA or other XML files is that structured FrameMaker does not ignore whitespace, resulting in excess spaces in paragraphs and table cells and unnecessary space between paragraphs. These spaces are annoying and time consuming to remove. This paper describes an XSL transform that removes whitespace from XML documents. You can incorporate the transform into a FrameMaker structured application to remove whitespace automatically.**

## Introduction

Many XML editors add indentation and carriage returns to XML files so that they're easier to read (often called *pretty printing*). For the most part, this extra whitespace is fine because XML tools normally ignore it. For example:

```
<section>
  <title>Match processing instruction nodes</title>
  <p>As mentioned earlier, FrameMaker occasionally reacts badly to
  non-FrameMaker processing instructions. To forestall any problems, the template that
  handles processing instructions ignores all processing instructions except those from
  FrameMaker.</p>
  <p>Using the
  <commandname>&lt;xsl:choose&gt;</commandname>statement
  rather than
  <commandname>&lt;xsl:if&gt;</commandname>, so that it
  would be easier to modify the transform in the future, should we want to handle
  additional processing instructions.</p>
```



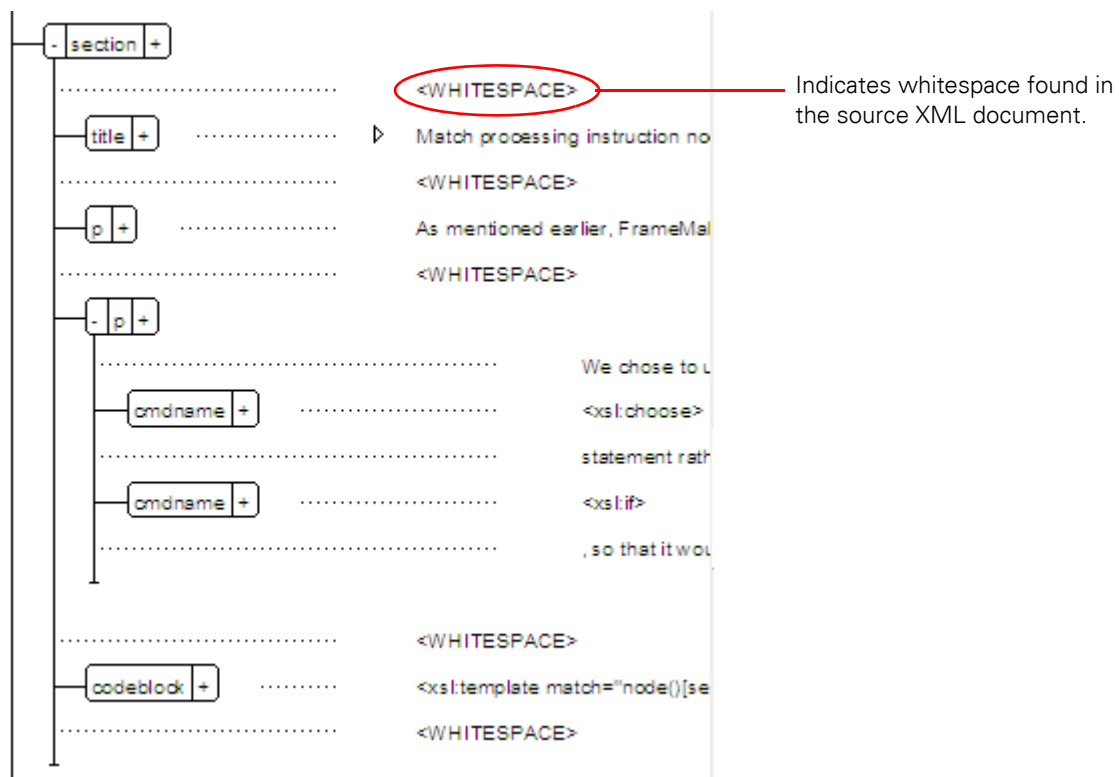
Unfortunately, FrameMaker does not ignore whitespace when it imports XML documents. The whitespace that made the XML easier to read becomes extra spaces within text and between lines.

### Match processing instruction nodes

As mentioned earlier, FrameMaker occasionally reacts badly to non-FrameMaker processing instructions. To forestall any problems, the template that handles processing instructions ignores all processing instructions except those from FrameMaker.

Using the `<xsl:choose>` statement rather than `<xsl:if>`, so that it would be easier to modify the transform in the future, should we want to handle additional processing instructions.

Here's how the document looks in the structure view. In some cases, these extra `<WHITESPACE>` nodes cause FrameMaker 9 to crash when generating PDF files:



Because extra whitespace is found in so many XML files, I created an XSL transform to remove the whitespace before reading files into FrameMaker. You can run this transform separately or you can integrate it into a FrameMaker structured application. Here is the same block of text with no whitespace (actually the text is all on one line, with no line breaks).

```
...<section><title>Match processing instruction nodes</title><p>As mentioned earlier, FrameMaker occasionally reacts badly to non-FrameMaker processing instructions. To forestall any problems, the template that handles processing instructions ignores all processing instructions except those from FrameMaker.</p><p>Using the <cmdname>&lt;xsl:choose&gt;</cmdname>statement rather than <cmdname>&lt;xsl:if&gt;</cmdname>, so that it would be easier to modify the transform in the future, should we want to handle additional processing instructions.</p>...
```

Of course, there are some elements that preserve whitespace, such as the DITA `<pre>` or `<codeblock>` elements. The transform is aware of these and deals with them correctly.

In addition to handling whitespace, the transform also deletes all non-FrameMaker processing instructions. Although FrameMaker is supposed to ignore all non-FrameMaker processing instructions, some processing instructions cause FrameMaker to fail on reading XML files.

The entire stylesheet is presented at the end of this document and is available for download from [Scriptorium's web site](#).

**NOTE:** This white paper assumes a basic understanding of XSL. However, it is possible to use the stylesheet without knowledge of XSL. Just start at the section named “Using the stylesheet” on page 8.

## The basic concepts

What is whitespace exactly? In XML, the following characters are considered to be whitespace characters: tab (`&#x09;`), newline (`&#x0A;`), carriage return (`&#x0D;`), and space (`&#x20;`).

The global variable `WHITESPACE` in our template contains these entities:

```
<xsl:variable name="WHITESPACE">
  <xsl:text>&#x09;&#x0A;&#x0D;&#x20;</xsl:text>
</xsl:variable>
```

You could define the variable using the actual characters, but using entities makes the XSL more readable. Later on, the stylesheet uses the `WHITESPACE` variable in an `XSL contains()` function to test if individual characters are whitespace characters.

XSL provides a good way of dealing with most whitespace: the `normalize-space()` function. This function replaces all contiguous whitespace in a string with a single space character (`&#x20;`). If any whitespace characters are found at the beginning or end of the string, they are deleted. Carried to its logical extreme, these rules mean that a string containing only whitespace is reduced to an empty string.

Normalizing the spaces is useful, but there are places where the leading and trailing whitespace are important. When an inline element occurs before or after a text node, the space between the text node and the element is important. Consider the phrase this is a `<b>test</b>`. If the space after the article “a” is deleted, the resulting string is “this is **atest**.” Additionally, a text node that consists of only whitespace must retain at least one space because the space might fall between two elements that must be separated. For example, the phrase `<b>test</b> <i>phrase</i>` must retain its space, or the result is “**testphrase**.”

Additionally, whitespace at the beginning of most elements that can contain CDATA must be deleted (`<p>` or `<li>` elements, in particular). Not deleting the leading whitespace leads to odd indentation on the first line of these elements.

Finally, the stylesheet must also allow for elements that are intended to preserve space (such as the DITA `<pre>` or `<codeblock>` elements). The stylesheet must pass these through without modification.



## Stylesheet organization

The stylesheet does an XML-to-XML copy (called an *identity transform*) that provides special-case processing for text nodes and processing instructions. The templates used in the stylesheet perform these functions:

- ❖ Root template — Handles the document root, adds a DOCTYPE declaration, and hands off processing to other templates.
- ❖ Match processing instruction nodes — Handles all processing instruction nodes.
- ❖ Match text nodes — Handles all text nodes. This is the heart of the stylesheet, where the `normalize-space()` function is applied.
- ❖ Match attributes — Handles attributes. Removes attributes added by the parser.
- ❖ Match other nodes (elements and comments) — Identifies elements that preserve whitespace and handles them differently.
- ❖ Keepspace template — Copies elements, preserving all whitespace.

The stylesheet uses an `<xsl:output>` statement to ensure that the output is not indented.

```
<xsl:output method="xml" indent="no" xml:space="default" encoding="UTF-8"/>
```

For testing purposes, there are two versions of the `<xsl:output>` statement:

- ❖ The first one contains `indent="yes"` and is commented out.
- ❖ The second (shown above) contains `indent="no"`.

When you use `indent="no"`, most of the output appears on one line. To debug the transform, you might want to see the output on multiple lines. In that case, uncomment the first `<xsl:output>` statement and comment out the second one. Just remember to restore the correct form of the `<xsl:output>` statement before putting your transform into production.

## Root template

The root template handles the document root, inserts a DOCTYPE declaration (through a named template), and invokes `<xsl:apply-templates>` for all nodes, using the identity mode. This mode is used for almost all other templates (other than `keepspace`).

```
<xsl:template match="/">
  <xsl:message>Handling whitespace.</xsl:message>
  <xsl:call-template name="add_doctype"/>
  <xsl:apply-templates select="node()" mode="identity"/>
</xsl:template>
```

The message “Handling whitespace.” is useful in initial testing. When things are up and running correctly, you can delete or comment out this message.

For more information about the template that adds the DOCTYPE declaration, see my blog post [Adding a DOCTYPE declaration on XML output](#).

## Match processing instruction nodes

As mentioned earlier, FrameMaker occasionally reacts badly to non-FrameMaker processing instructions. To forestall any problems, the template that handles processing instructions deletes all processing instructions, except those from FrameMaker.

Using the `<xsl:choose>` statement rather than `<xsl:if>` makes it easier to modify the transform if you want to handle additional processing instructions in the future.

```
<xsl:template match="node()[self::processing-instruction()]" mode="identity">
  <xsl:choose>
    <xsl:when test="name() = 'Fm'">
      <xsl:copy/>
    </xsl:when>
    <xsl:otherwise><!-- Do nothing. --></xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

## Match text nodes

The template that matches text nodes is the most complex template in the stylesheet. The template defines two variables (`fore_space` and `aft_space`). If there is whitespace at the beginning or end of the text node, these variables are assigned a single space. If the text node contains no whitespace or if it contains only whitespace, the variable is assigned an empty string.

However, when creating `fore_space`, the template makes additional judgments. If the text node is the first node in the current element (`position() = 1`), the variable is assigned an empty string. Also, if the text node is a single space character, the assumption is that this is a space between two inline elements. In this case, the variable is assigned a single space.

**NOTE:** This assumption about the single space is the least certain assessment made in the stylesheet. It would be far better if the script could examine the DTD's content model and determine if the node's parent allowed CDATA (and thus, allowed a space). Unfortunately, that's not possible. One possible refinement might be to add a "class contains" predicate to see if the node's parent element inherits behavior from `topic/p`: (`parent::*[contains(@class,' topic/p ')]`). You could add this as part of the match attribute or as a separate `<xsl:choose>` statement within the template.

The pieces are then reassembled at the end of the template. First the `fore_space` variable is output, followed by the text node with its spaces normalized, and then the `aft_space` variable.

```
<xsl:template match="node()[self::text()]" mode="identity">
  <!-- Create a variable for the whitespace before the text node. -->
  <xsl:variable name="fore_space">
    <xsl:choose>
      <!-- If after normalizing space the length is zero, consider contents. -->
      <xsl:when test="string-length(normalize-space()) = 0">
        <!-- If this is a single space between elements, allow it. -->
        <xsl:choose>
          <xsl:when test="string-length(.) = 1 and string(.) = ' '">
            <xsl:value-of select="' '"/>
          </xsl:when>
          <xsl:otherwise>
```



```

        <xsl:value-of select="""/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:when>
  <!-- When the text node is the beginning of a paragraph, value is empty string. -->
  <xsl:when test="position() = 1">
    <xsl:value-of select="""/>
  </xsl:when>
  <!-- When the first character in the text node is a whitespace char, value is space. -->
  <xsl:when test="contains($WHITESPACE,substring(.,1,1))">
    <xsl:value-of select=" "/>
  </xsl:when>
</xsl:choose>
</xsl:variable>
<!-- Create a variable for the whitespace after the text node.
Note that in this case, we're not worried about the last text node in a block element. -->
<xsl:variable name="aft_space">
  <xsl:choose>
    <!-- If after normalizing space the length is zero, value is empty string. -->
    <xsl:when test="string-length(normalize-space(.)) = 0">
      <xsl:value-of select="""/>
    </xsl:when>
    <!-- When the last character in the text node is a whitespace char, value is space. -->
    <xsl:when test="contains($WHITESPACE,substring(.,string-length(.),1))">
      <xsl:value-of select=" "/>
    </xsl:when>
  </xsl:choose>
</xsl:variable>
<!-- Build the new version of the text node, using the fore_space, normalized string, and aft_space. -->
<xsl:value-of select="$fore_space"/>
<xsl:value-of select="normalize-space(.)"/>
<xsl:value-of select="$aft_space"/>
</xsl:template>

```

## Match attributes

Because the DITA DTD defines a default class attribute value for each element, the parser automatically adds this attribute to each element when validating. As a result, when structured FrameMaker reads your document, each element will have a class attribute. These class attributes have no meaning to structured FrameMaker and can potentially confuse people working on the files, so the stylesheet ignores them.

The template matches all attributes and uses an `<xsl:choose>` statement to handle class attributes. The template copies any attributes that are not named class.

```

<xsl:template match="@*" mode="identity">
  <xsl:choose>
    <xsl:when test="name() = 'class'">
      <!-- Do nothing. -->
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

```

There are other attributes you might want to handle, such as domains and ditaarch:DITAArchVersion.

## Match other nodes

This template handles any node that isn't a processing instruction or text node (essentially, elements and comments).

The template examines the `xml:space` attribute to test if the element preserves space (used in `<codeblock>` and `<pre>` elements). If `xml:space` is set to preserve, the element is copied and its contents are processed using the `keep-space` mode, which does a literal copy, including all whitespace in the element's contents. If you're not processing DITA content, you may have to modify this test look for the elements in which space should be preserved.

If the element doesn't preserve space, the element is copied and its contents are processed using the `identity` mode.

```
<xsl:template match="node()[not(self::processing-instruction()) and not(self::text())]" mode="identity">
  <xsl:choose>
    <!-- If the element specifies xml:space="preserve",
         cannot mess with the whitespace, so use keep-space template. -->
    <xsl:when test="@xml:space = 'preserve'">
      <xsl:copy>
        <xsl:apply-templates select="@*|node()" mode="keep-space"/>
      </xsl:copy>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy>
        <xsl:apply-templates select="@*|node()" mode="identity"/>
      </xsl:copy>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

**NOTE:** Rather than test the name of the node to a series of possible element names (`name(.)='codeblock'`), you could use a “class contains” predicate: `*[contains(@class,' topic/pre ')]`.

If you need to modify or eliminate specific comment nodes, you can create a separate template to handle comment nodes.

## Keep-space template

The `keep-space` template implements a standard identity transform that recursively copies all attributes and nodes, while ignoring whitespace in text nodes.

```
<xsl:template match="@*|node()" mode="keep-space">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()" mode="keep-space"/>
  </xsl:copy>
</xsl:template>
```

This template makes the assumption that elements preserving space do not contain processing instructions. If necessary, you could create templates similar to the processing instruction template that used the `keep-space` mode.



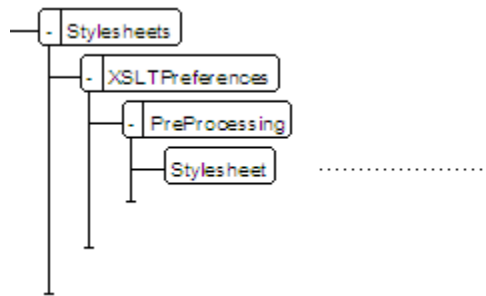
## Using the stylesheet

To use this stylesheet, you can either run it by itself, or you can add it to a FrameMaker structured application.

If you choose to run the stylesheet by itself, read my blog post [Ignoring DOCTYPE in XSL Transforms using Saxon 9B](#). Otherwise you'll have to find some other way of removing the DOCTYPE declarations before processing.

To add the stylesheet to a structured application, edit the structapps.fm file in structured FrameMaker and find the application you want to modify (typically DITA-Topic-FM). If you want to be able to import XML files with and without whitespace modification, you can make a copy of the DITA-Topic-FM application and name the copy something like DITA-NoWhitespaceTopic-FM.

In the application you're modifying, add the following elements after the ReadWriteRules element:



The Stylesheet element contains the path to the handle\_whitespace.xsl stylesheet. Typically, it should be in \$STRUCTDIR\xml\dita\app\DITA-Topic-FM\handlewhitespace.xsl.

**NOTE:** Normally in structured FrameMaker, you can make these changes to the structapps.fm file in the FrameMaker installation folder (typically C:\Program Files\Adobe\Framemaker{version}\Structure). However, things seem to be broken in FrameMaker 9.0. At current writing (FM9.0p250), you must make these changes in the structapps.fm file in each users' Application Data folder (C:\Documents and Settings\{user}\Application Data\Adobe\Framemaker9). We hope this bug is corrected soon.

## The complete stylesheet

Here's the complete stylesheet. You can also download the stylesheet from [Scriptorium's web site](#).

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:variable name="WHITESPACE"><xsl:text>&#x09;&#x0A;&#x0D;&#x20;</xsl:text></xsl:variable>

  <!-- DEBUGGING use indent="no" in production -->
  <!-- <xsl:output method="xml" indent="yes" xml:space="default" encoding="UTF-8"/> -->
  <xsl:output method="xml" indent="no" xml:space="default" encoding="UTF-8"/>

```

```

<xsl:template match="/">
  <xsl:message>Handling whitespace.</xsl:message>
  <xsl:call-template name=" add_doctype "/>
  <xsl:apply-templates select="node()" mode="identity"/>
</xsl:template>
<!-- FrameMaker doesn't like non-FrameMaker processing instructions. This template removes them.-->
<xsl:template match="node()[self::processing-instruction]" mode="identity">
  <xsl:choose>
    <xsl:when test="name() = 'Fm'">
      <xsl:message>Copying processing instruction
        (name is "<xsl:value-of select="name()"/>").</xsl:message>
      <xsl:copy/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:message>Ignoring processing instruction
        (name is "<xsl:value-of select="name()"/>").</xsl:message>
      <!-- Do nothing. -->
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- Have to make sure that space before and after is treated correctly.
  If the text node is in a block that contains inline elements (<ph>, <b>, and so on),
  the spaces must be preserved at the beginning and end of strings. On the other hand,
  it's equally important to make sure the FIRST whitespace in a block IS stripped.
-->
<xsl:template match="node()[self::text]" mode="identity">
  <!-- Create a variable for the whitespace before the text node. -->
  <xsl:variable name="fore_space">
    <xsl:choose>
      <!-- If after normalizing space the length is zero, consider contents. -->
      <xsl:when test="string-length(normalize-space(.)) = 0">
        <!-- If this is a single space between elements, allow it. -->
        <xsl:choose>
          <xsl:when test="string-length(.) = 1 and string(.) = ' '">
            <xsl:value-of select="' '"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select=""/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>
      <!-- When the text node is the beginning of a paragraph, value is empty string. -->
      <xsl:when test="position() = 1">
        <xsl:value-of select=""/>
      </xsl:when>
      <!-- When the first character in the text node is a whitespace char, value is space. -->
      <xsl:when test="contains($WHITESPACE,substring(.,1,1))">
        <xsl:value-of select="' '"/>
      </xsl:when>
    </xsl:choose>
  </xsl:variable>

```



```

<!-- Create a variable for the whitespace after the text node.
Note that in this case, we're not worried about the last text node in a block element. -->
<xsl:variable name="aft_space">
  <xsl:choose>
    <!-- If after normalizing space the length is zero, value is empty string. -->
    <xsl:when test="string-length(normalize-space(.)) = 0">
      <xsl:value-of select=""/>
    </xsl:when>
    <!-- When the last character in the text node is a whitespace char, value is space. -->
    <xsl:when test="contains($WHITESPACE,substring(.,string-length(.),1))">
      <xsl:value-of select=" " />
    </xsl:when>
  </xsl:choose>
</xsl:variable>
<!-- Build the new version of the text node, using the fore_space, normalized string, and aft_space. -->
<xsl:value-of select="$fore_space"/>
<xsl:value-of select="normalize-space(.)"/>
<xsl:value-of select="$aft_space"/>
</xsl:template>

<xsl:template match="@*" mode="identity">
  <xsl:choose>
    <xsl:when test="name() = 'class' ">
      <!-- Do nothing. -->
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="node()[not(self::processing-instruction()) and not(self::text())]" mode="identity">
  <xsl:choose>
    <!-- If the element specifies xml:space="preserve",
cannot mess with the whitespace, so use keep-space template. -->
    <xsl:when test="@xml:space = 'preserve' ">
      <xsl:copy>
        <xsl:apply-templates select="@*[node()]" mode="keep-space"/>
      </xsl:copy>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy>
        <xsl:apply-templates select="@*[node()]" mode="identity"/>
      </xsl:copy>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- A standard identity transform for where the whitespace is important. -->
<!-- Presumes that processing instructions don't exist in these elements. -->
<xsl:template match="@*[node()]" mode="keep-space">
  <xsl:copy>
    <xsl:apply-templates select="@*[node()]" mode="keep-space"/>
  </xsl:copy>
</xsl:template>

```

```

<!-- Utility function to add the doctype.  Handles DITA "standard" topic types.
      If you specialize, you may have to add others.  NOTE whitespace is important here.
      If you pretty print this stylesheet, make sure there's a single CR before each
      "<!DOCTYPE" (with no spaces before "<") and a CR after the ">".
-->
<xsl:template name="add_doctype">
  <xsl:choose>
    <xsl:when test="/concept">
      <xsl:text disable-output-escaping="yes">
&lt;!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd"&gt;
      </xsl:text>
    </xsl:when>
    <xsl:when test="/reference">
      <xsl:text disable-output-escaping="yes">
&lt;!DOCTYPE reference PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd"&gt;
      </xsl:text>
    </xsl:when>
    <xsl:when test="/task">
      <xsl:text disable-output-escaping="yes">
&lt;!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd"&gt;
      </xsl:text>
    </xsl:when>
    <xsl:when test="/topic">
      <xsl:text disable-output-escaping="yes">
&lt;!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd"&gt;
      </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:message>Unknown root element, using topic doctype.</xsl:message>
      <xsl:text disable-output-escaping="yes">
&lt;!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "http://docs.oasis-open.org/dita/v1.1/OS/dtd/
topic.dtd"&gt;
      </xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
</xsl:stylesheet>

```

## About Scriptorium

Scriptorium Publishing provides expert advice on how to develop, deploy, and manage content. Our typical customer has thousands of pages of information, which needs to be delivered in print, PDF files, HTML, and other media, often in dozens of languages. Our mission is to automate formatting and production tasks, usually through XML technologies, so that authors can write more efficiently.

Our consultants have experience in traditional publishing workflows, including typesetting, book design, copyfitting, and production editing. This understanding influences our approach to creating state-of-the-art publishing systems with modern tools and technologies, such as XML, HTML, DITA, the DITA Open Toolkit, XSLT, XSL-FO, FrameMaker, Ant, Perl, FrameScript, Flash, InDesign, XMetaL, oXygen, and many more.



Our customers include federal and state government as well as companies in defense, consumer electronics, telecommunications, health care, pharmaceutical, and other industries.

If you are facing a difficult publishing challenge, we want to hear from you. Contact us at [info@scriptorium.com](mailto:info@scriptorium.com) or 919-481-2701 x105.

Scriptorium Publishing is based in the Research Triangle area of North Carolina and has been in business since 1997.

Scriptorium Publishing Services, Inc.  
PO Box 12761  
Research Triangle Park, NC 27709-2761  
USA  
[info@scriptorium.com](mailto:info@scriptorium.com)  
919-481-2701  
[www.scriptorium.com](http://www.scriptorium.com)

