

A Hands-On Introduction to XSLT and XPath



Learning XSLT

O'REILLY®

Michael Fitzgerald

Building New Documents with XSLT

In the first chapter of this book, you got acquainted with the basics of how XSLT works. This chapter will take you a few steps further by showing you how to add text and markup to your result tree with XSLT templates.

First, you'll add literal text to your output. Then you'll work with *literal result elements*, that is, elements that are represented literally in templates. You'll also learn how to add content with the text, element, attribute, attribute-set, comment, and processing-instruction elements. In addition, you'll get your first encounter with attribute value templates, which provide a way to define templates inside attribute values.

Outputting Text

You can put plain, literal text into an XSLT template, and it will be written to a result tree when the template containing the text is processed. You saw this work in the very first example in the book (*msg.xml* in Chapter 1). I'll go into more detail about adding literal text in this section.

Look at the single-element document *text.xml* in *examples/ch02* (this directory is where all example files mentioned in this chapter can be found):

```
<?xml version="1.0"?>

<message>You can easily add text to your output.</message>
```

With *text.xml* in mind, consider the stylesheet *txt.xsl*:

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

  <template match="/">Message: <apply-templates/></template>

</stylesheet>
```

When applied to *text.xml*, here is what generally happens, although the actual order of events may vary internally in a processor:

1. The template rule in *txt.xsl* matches the root node (*/*), the beginning point of the source document.
2. The implicit, built-in template for elements then matches *message*.
3. The text “Message: ” (including one space) is written to the result tree.
4. *apply-templates* processes the text child node of a *message* using the built-in template for *text*.
5. The built-in template for *text* picks up the text node “You can easily add text to your output.”
6. The output is serialized.

Apply *txt.xsl* to *text.xml* using Xalan:

```
xalan text.xml txt.xsl
```

This gives you the following output:

```
Message: You can easily add text to your output.
```

The *txt.xsl* stylesheet writes the little tidbit of literal text, “Message: ”, from its template onto the output, and also grabs some text out of *text.xml*, and then ultimately puts them together in the result tree. You can do the same thing with the XSLT instruction element *text*.

Using the text Element

Instead of literal text, you can use XSLT’s *text* instruction element to write text to a result tree. Instruction elements, you’ll remember, are elements that are legal only inside templates. Using the *text* element gives you more control over result text than literal text can.

The template rule in *lf.xsl* contains some literal text, including whitespace:

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

  <template match="/">Message:
    <apply-templates/>
  </template>

</stylesheet>
```

When you apply *lf.xsl* to *text.xml* with Xalan like this:

```
xalan text.xml lf.xsl
```

the whitespace—a linefeed and some space—is preserved in the result:

```
Message:
  You can easily add text to your output.
```

The XSLT processor sees the whitespace in the stylesheet as literal text and outputs it as such. The XSLT instruction element `text` allows you to take control over the whitespace that appears in your template.

In contrast, the stylesheet *text.xsl* uses the `text` instruction element:

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

  <template match="/">
    <text>Message: </text>
    <apply-templates/>
  </template>

</stylesheet>
```

When you insert `text` like this, the only whitespace that is preserved is what is contained in the `text` element—a single space. Try it to see what happens:

```
xalan text.xml text.xsl
```

This gives you the same output you got with *txt.xsl*, with no hidden whitespace:

```
Message: You can easily add text to your output.
```

Back in the stylesheet *txt.xsl*, recall how things are laid out in the `template` element:

```
<template match="/">Message: <apply-templates/></template>
```

The literal text “Message: ” comes immediately after the `template` start tag. The reason is that if you use any literal text that is not whitespace in a `template`, an XSLT processor interprets adjacent whitespace in the `template` element as significant. Any whitespace that is considered significant is preserved and sent along to output.

To see more of how whitespace effects literal text in a result, look at the stylesheet *whitespace.xsl*:

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

  <template match="/">
    Message:

    <apply-templates/>

    ...including whitespace!

  </template>

</stylesheet>
```

Now, process it against *text.xml* to see what happens:

```
xalan text.xml whitespace.xsl
```

Observe how the whitespace is preserved, both from above and below the `apply-templates` element:

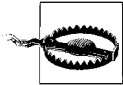
Message:

You can easily add text to your output.

...including whitespace!

If no nonwhitespace literal text follows `apply-templates` (that is, if you removed “...including whitespace!” from within template in *whitespace.xml*), the latter whitespace would not be preserved.

Whitespace is obviously hard to see. I recommend that you make a copy of *whitespace.xml* and experiment with whitespace to see what happens when you process it.



Netscape and Mozilla, by the way, preserve the whitespace-only text nodes in output from *whitespace.xml*, but IE does not. Use *whitespace-pi.xml* to test this in a browser if you like, but keep in mind that such output can vary as browser versions increment upward.

If you use text elements, the other whitespace within template elements becomes insignificant and is discarded when processed. You’ll find that whitespace is easier to control if you use text elements. The *control.xml* stylesheet uses text elements to handle the whitespace in its template:

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

<template match="/">
  <text>Message: </text>
  <text>

</text>
<text>
</text>
  <apply-templates/>
  <text>

          ...and whitespace, too!</text>
</template>

</stylesheet>
```

The *control.xml* stylesheet has four text elements, two of which contain only whitespace, including one that inserts a pair of line breaks. Because you can see the start and end tags of text elements, it becomes easier to judge where the whitespace is, making it easier to control. To see the result, process it with *text.xml*:

```
xalan text.xml control.xml
```

As an alternative, you could also insert line breaks by using *character references*, like this:

```
<text>&#10;&#10;</text>
```

This instance of the text element contains character references to two line breaks in succession. A character reference begins with an ampersand (&) and ends with a semicolon (;). In XML, you can use decimal or hexadecimal character references. The decimal character reference `
` represents the linefeed character using the decimal number 10, preceded by a pound sign (#). A hexadecimal character reference uses a hexadecimal number preceded by a pound sign and the letter *x* (`
`). You can also use `
` or `
`, which are equivalent hexadecimal character references to the decimal reference `
`.

Why Linefeeds?

You might be wondering why I use a linefeed line-end character (`
`) instead of a carriage return (``) or carriage return/linefeed combination. The reason is because when a document is processed with a compliant XML processor, the line ends are all changed to linefeeds anyway. In other words, if an XML processor encounters a carriage return or a carriage return/linefeed combination, these characters are converted into linefeeds during processing. You can read about this in Section 2.11 of the XML specification.

The disable-output-escaping attribute

The text element has one optional attribute: `disable-output-escaping`. XSLT does not require processors to support this attribute (see Section 16.4 of the XSLT specification), but most do. This attribute can have one of two values, either `yes` or `no`. The default is `no`, meaning the same whether the `disable-output-escaping` attribute is present or if its value is `no`. What does this attribute do? Hang on—this is going to take a bit of explaining.

In XML, some characters are forbidden in certain contexts. Two notable characters that fit into this category are the left angle bracket or less-than sign (`<`) and the ampersand (`&`). It's fine to use these characters in markup, such as when beginning a tag with `<`. You can't, however, use a `<` in character data (the strings that appear between tags) or in an attribute value. The reason is that the `<` is a road sign to an XML processor. When an XML processor munches on an XML document, if it sees a `<`, it says in effect, "Oh. We're starting a new tag here. Branch to the code that handles that." Therefore, you can see why we aren't allowed to use `<` directly in XML, except in markup.

There is a way out, though. XML provides several ways to represent these characters by escaping them with an entity or character reference whenever you want to use them where they are normally not allowed. Escaping a character essentially hides it from the processor. The most common way to escape characters like < and & is by referencing predefined entities. You'll find XML's built-in, predefined entity references listed in Table 2-1.

Table 2-1. Predefined entities in XML 1.0

Character	Entity reference	Numeric character reference
< (less-than)	<	<
& (ampersand)	&	&
> (greater-than)	>	>
" (quotation)	"	"
' (apostrophe)	'	'

The greater-than entity is provided so that XML can be compatible with Standard Generalized Markup Language (SGML). The > character alone is permissible in character data and in attribute values, escaped or not. (For SGML compatibility, you always need to escape the > character if it appears as part of the sequence]]>, which is used to end CDATA sections. CDATA sections are described in more detail in Chapter 3.)



XML, by the way, is a legal subset of SGML, an international standard. SGML is a product of the International Organization for Standardization (ISO), and you can find the SGML specifications on the ISO web site, <http://www.iso.ch>. But have your credit card ready: you have to pay for most ISO specifications (sometimes dearly), unlike W3C specifications, which are free to download.

The " and ' entities allow you to include double and single quotes in attribute values. A second matching quote should indicate the close of an attribute value. If not escaped, a misplaced matching quote signals a fatal error, if not followed by well-formed markup. (See Section 1.2 of the XML specification.) I say *matching* because if an attribute value is surrounded by double quotes, it can contain single quotes in its value (as in "'value'"). The reverse is also true, that is, single quotes can enclose double quotes ('"value"').

You have to escape an ampersand in character content because the ampersand itself is used to escape characters in entity and character references! If that's confusing, a few examples should clear things up. I'll now show you how the disable-output-escaping attribute works.

The little document *escape.xml* contains the name of a famous publisher:

```
<title>O'Reilly</title>
```

The stylesheet *noescape.xml* adds some new text to this title using the default, which is to *not* disable output escaping:

```
<stylesheet version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
<output method="xml" omit-xml-declaration="yes"/>

<template match="/">
  <publisher xmlns="">
    <value-of select="title" xmlns="http://www.w3.org/1999/XSL/Transform"/>
    <text disable-output-escaping="no" xmlns="http://www.w3.org/1999/XSL/Transform">
& Reilly & Associates</text>
  </publisher>
</template>

</stylesheet>
```

noescape.xml uses the `xml` output method. You can't see the effect of output escaping when the output method is `text`, so you have to use either the `xml` or `html` methods. You'll learn more about output methods later in this chapter and in Chapter 3.

This stylesheet also redeclares the XSLT namespace several times (on the `value-of` and `text` elements). You'll see how to circumvent this cumbersome practice with a namespace prefix in "Adding a Namespace Prefix," later in this chapter.

To see output escaping in action, process *escape.xml* with this command:

```
xalan escape.xml noescape.xml
```

Here is the result:

```
<publisher>O'Reilly & Associates</publisher>
```

`disable-output-escaping` with a value of `no` has the same effect as having no attribute at all, that is, the output is escaped and `&` is preserved in the result.

The following stylesheet, *escape.xml*, disables output escaping:

```
<stylesheet version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
<output method="xml" omit-xml-declaration="yes"/>

<template match="/">
  <publisher xmlns="">
    <value-of select="title" xmlns="http://www.w3.org/1999/XSL/Transform"/>
    <text disable-output-escaping="yes" xmlns="http://www.w3.org/1999/XSL/Transform">
& Reilly & Associates</text>
  </publisher>
</template>

</stylesheet>
```

Process this:

```
xalan escape.xml escape.xml
```

and you get:

```
<publisher>O'Reilly & Associates</publisher>
```

In *escape.xml*, escaping is turned off so that `&` is not preserved. You get only the ampersand in the result. The `publisher` element, which appears in both *escape.xml* and *noescape.xml*, is a literal result element. Let me explain what that is.

Literal Result Elements

A *literal result element* is any XML element that is represented literally in a template, is not in the XSLT namespace, and is written literally onto the result tree when processed. Such elements must be well-formed within the stylesheet, according to the rules in XML 1.0.

The example stylesheet *tedious.xml*, which produces XML output, contains an instance of the `msg` literal result element from a different namespace:

```
<stylesheet version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
  <output method="xml" indent="yes"/>
  <template match="/">
    <msg xmlns="http://www.wyeast.net/msg">
      <apply-templates xmlns="http://www.w3.org/1999/XSL/Transform"/>
    </msg>
  </template>
</stylesheet>
```

Here is *literal.xml*:

```
<?xml version="1.0"?>
<message>You can use literal result elements in stylesheets.</message>
```

If you apply this stylesheet to *literal.xml*:

```
xalan literal.xml tedious.xml
```

you will get this output:

```
<?xml version="1.0" encoding="UTF-8"?>
<msg xmlns="http://www.wyeast.net/msg">You can use literal result elements in
stylesheets.</msg>
```

Because this stylesheet uses the XML output method, XML declaration was written to the result tree. The literal result element, along with its namespace declaration, was also written.

Adding a Namespace Prefix

In *tedious.xml*, the `msg` element has its own namespace declaration. This is because the XSLT processor would reject the stylesheet if it did not have a namespace declaration. The `apply-templates` element that follows must also redeclare the XSLT namespace because the processor will produce unexpected results without it. (Try it and you'll see.)

Ok, ok. This is getting a little confusing. If you had to add a namespace declaration to every literal element and then to following XSLT elements, that would add up to a lot of error-prone typing. So, it's time to start using a prefix with the XSLT namespace.

The conventional prefix for XSLT is `xsl`, but you can choose another one if you like. Here is a rewrite of *tedious.xsl* that uses the `xsl` prefix with the XSLT namespace declaration. It's called *notsotetious.xsl*:

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

  <xsl:template match="/">
    <msg>
      <xsl:apply-templates/>
    </msg>
  </xsl:template>

</xsl:stylesheet>
```

This version of the stylesheet drops the namespace declaration for `msg` because it's no longer required to have one. Likewise, you don't have to redeclare the XSLT namespace for `apply-templates` either.

If you apply *notsotetious.xsl* to *literal.xml*:

```
xalan literal.xml notsotetious.xsl
```

it produces:

```
<?xml version="1.0" encoding="UTF-8"?>
<msg>You can use literal result elements in stylesheets.</msg>
```

When you use a prefix with a namespace declaration on the XSLT document element `stylesheet`, as in *notsotetious.xsl*, you don't have to repeat the declaration on any other element in the document that uses the same prefix—you only have to declare it once. Throughout the rest of the book, I'll usually use an `xsl` prefix in a stylesheet.

Here is another simple example of a literal result element, expanded with a few more details. The template in the stylesheet *literal.xsl* contains a literal result element `paragraph`:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <paragraph><xsl:apply-templates/></paragraph>
  </xsl:template>

</xsl:stylesheet>
```

QNames and NCNames

An element or attribute name that is qualified by a namespace is called a *qualified name*, or QName for short. In normal XSLT, two examples of QNames are `stylesheet` or `xs1:stylesheet`. Both are (or should be) qualified by the namespace name `http://www.w3.org/1999/XSL/Transform`. A QName may have a prefix, such as `xs1`, which is separated by a colon from its local part or local name, as in `stylesheet`. A QName may also consist only of a local part. If a local part is qualified with a namespace, and there is no prefix, it should be qualified by a *default namespace declaration*. You'll learn about default declarations in "Applying namespaces," later in this chapter.

An element or attribute name that is not qualified with a namespace is unofficially called a *non-colonized name*, or, officially, an NCName. As spelled out in XML 1.0, a colon was allowed in XML names, even as the first character of a name. For example, names like `doc:type` or even `:type` were and still are legal, even if they are not qualified with a namespace. But there was little notion of namespaces in early 1998 when XML 1.0 came out, so if a colon occurred in a name, it was considered a legal name character. Nevertheless, XML names with colons that are not namespace-qualified are undefined in XSLT and don't work. Avoid them and be happier!

The XML namespaces specification created the term *NCName*. It is an XML name minus the colon, and it makes way for the special treatment of the colon in XML namespace-aware processing. If an XML processor is not up to date and does not support namespaces (most do so now), colons will not be treated specially in names. You can read more about QNames and NCNames in Sections 3 and 4 of the XML namespaces specification.

If namespaces sound somewhat confusing to you, you are in good company. Namespaces in XML are here to stay, but they are admittedly befuddling and difficult to explain.

The output element specifies the `xml` output method, instead of the `text` method, and turns indentation on (`indent="yes"`). When the `xml` output method is set, XSLT processors will write an XML declaration on the first line of the result tree (as you saw earlier).

When the output element's `indent` attribute has a value of `yes`, the processor will add some indentation to make the output more human-readable. The amount of indentation will vary from processor to processor because the XSLT specification states only that, in regard to indentation, an "XSLT processor may add additional whitespace when outputting the result tree" (see Section 16). The modal *may add* gives implementers some free rein on how they put indentation into practice. Some implementers, in fact, don't implement indentation at all, although they are allowed to do so.

Apply `literal.xsl` to `literal.xml` with the command:

```
xalan literal.xml literal.xsl
```

and you will see the following results:

```
<?xml version="1.0" encoding="UTF-8"?>
<paragraph>You can use literal result elements in stylesheets.</paragraph>
```

Using the stylesheet, the processor replaced the document element `message` from the source tree with the literal result element `paragraph` in the result tree. In its output, Xalan also included an encoding declaration in the XML declaration.

The encoding declaration takes the form of an attribute specification (`encoding="UTF-8"`). The encoding declaration provides an encoding name, such as `UTF-8`, that indicates the intended character encoding for the document. The encoding name is not case sensitive; for example, both `UTF-8` or `utf-8` work fine. Xalan uses uppercase when outputting an encoding declaration, while Saxon uses lowercase. You'll learn more about encoding declarations and character encoding in Chapter 3.

Literal Result Elements for HTML

Taking this a few steps further, the stylesheet `html.xml` produces HTML output using literal result elements:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" indent="yes"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>HTML Output</title>
      </head>
      <body>
        <p><xsl:apply-templates/></p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The output method is now `html`, so no XML declaration will be written to the output. Indentation is the default for the `html` method, though it is shown explicitly in the output element (`indent="yes"`). The tags for the resulting document are probably familiar to you, and they are near the minimum necessary for an HTML document to display anything. For reference, you can find the current W3C specification for HTML Version 4.01 at <http://www.w3.org/TR/html401/>.

Now, use Xalan to apply the stylesheet to `literal.xml`, and save the result in a file:

```
xalan -o literal.html literal.xml html.xml
```

This transformation will construct the following result tree and save it to the file `literal.html`:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
<title>HTML Output</title>
</head>
<body>
<p>You can use literal result elements in stylesheets.</p>
</body>
</html>
```

By default, Xalan's indentation depth is zero, but as a general rule, start tags begin on new lines. Saxon's default indentation depth is three spaces, with start tags on new lines as well.

The META tag

Xalan automatically adds a META tag to the head element. This META tag is an apparent attempt to get Hypertext Transfer Protocol (HTTP) to bind or override the value of the META tag's content attribute (`text/html; charset=UTF-8`) to the Content-Type field of its response header. In other words, if you request this document with HTTP, such as with a web browser, the server that hosts the document will issue an HTTP response header, and one of the fields or lines in that header should be labeled Content-Type, as shown here:

```
HTTP/1.1 200 OK
Date: Thu, 01 Jan 2003 00:00:01 GMT
Server: Apache/1.3.27
Last-Modified: Thu, 31 Dec 2002 23:59:59 GMT
ETag: "8b6172-c7-3e3878a8"
Accept-Ranges: bytes
Content-Length: 199
Connection: close
Content-Type: text/html; charset=UTF-8
```

I cannot guarantee that the content of the META tag will wind up in the Content-Type header field, though that's what it logically seems to be trying to do. You can tell Xalan to not output the META tag by using the `-m` option on the command line. For example, the command:

```
xalan -m literal.xml html.xsl
```

will produce HTML output without the META tag:

```
<html>
<head>
<title>HTML Output</title>
</head>
<body>
<p>You can use literal result elements in stylesheets.</p>
</body>
</html>
```

The `apply-templates` element in *html.xsl* brought the content of message from *literal.xml* into the content of the `p` element in the resulting HTML. If you open the document *literal.html* in the Mozilla Firebird web browser, it should look like Figure 2-1. (Firebird is a leaner and faster branch of Mozilla.)

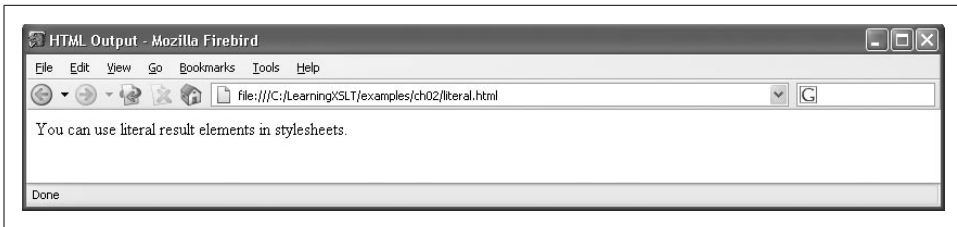


Figure 2-1. Displaying *literal.html* in Mozilla Firefox

XHTML Literal Result Elements

The XML document *doc.xml* uses a minimal set of elements to express a rather simple document structure:

```
<?xml version="1.0"?>

<doc styletype="text/css">

<css>
  h1 {font-family: sans-serif; font-size: 24pt}
  p {font-size: 16pt}
</css>

<title>Using Literal Result Elements</title>

<heading>What Is a Literal Result Element?</heading>

<paragraph>You can use literal result elements in
stylesheets. A literal result element is any non-XSLT element,
including any attributes, that can be written literally in a
template, and that will be pushed literally onto the
result tree when processed.</paragraph>

</doc>
```

The document element *doc* in *doc.xml* is the container, so to speak, for the whole document. This element has a single attribute, *styletype*, that ostensibly provides a content type for a CSS stylesheet. The *css* element holds a few CSS rules, which don't apply to any elements in *doc.xml*, but they'll come in handy later when you move to XHTML. The *title*, *heading*, and *paragraph* elements that follow have fairly obvious roles. Now look at the stylesheet *doc.xsl*, which you can use to transform *doc.xml* into XHTML:

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

<xsl:template match="doc">
  <html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title><xsl:apply-templates select="title"/></title>
    <style type="{@styletype}">
```

```

        <xsl:apply-templates select="css"/>
    </style>
</head>
<body>
    <h1><xsl:apply-templates select="heading"/></h1>
    <p><xsl:apply-templates select="paragraph"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>

```

The output method is XML again, because XHTML is really a vocabulary of XML. (XSLT 1.0 does not support a specific `xhtml` output method, but XSLT 2.0 does.) With indentation on (yes), the output will be more readable. The literal result element for `html` has a namespace declaration for XHTML 1.0.

As a vocabulary of XML, XHTML 1.0 has requirements that go beyond those of HTML, an SGML vocabulary. For example, all XHTML tags must be in lowercase, and must be closed properly, either with an end tag or in the form of an empty element tag. Attribute values must be enclosed in matching double or single quotes. In other words, because XHTML *is* XML, it must be well-formed.

Looking back at *doc.xsl*, what about the braces in the value of `style`'s `type` attribute? That's called an *attribute value template* in XSLT.

Attribute value templates

An attribute value template provides a way to bring computed data into attribute values. Think for a moment why such a syntax is needed. You know that the markup character `<` is not allowed in attribute values. That's a rule from the XML 1.0 specification. So, you couldn't use something like a `value-of` element in an attribute value. And you can't use entity references such as `<`; as you normally would in an attribute value of a literal result element because an XSLT processor will interpret these references as literal text. These are a few reasons why XSLT provides this special syntax.

The following line in *doc.xsl* contains an attribute value template:

```
<style type="{@styletype}">
```

Because it is processing the `doc` element, and eventually all its children, the processor uncovers the attribute `styletype` on `doc`. In the stylesheet, the braces (`{}`) enclose the attribute value template. Everything in the braces is computed rather than copied through. The at sign (`@`) syntax comes from XPath and indicates that the following item in the location path is an attribute you're looking for in the context node. The XSLT processor then picks up the value of the `styletype` attribute from the source tree and places it at this same spot in the output, giving you:

```
<style type="text/css">
```

in the result tree. (You can read more about attribute value templates in Section 7.6.2 of the XSLT specification.)

Now process this transformation and save the result in the file:

```
xalan -o doc.html doc.xml doc.xsl
```

The resulting file *doc.html* will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using Literal Result Elements</title>
<style type="text/css">
  h1 {font-family: sans-serif; font-size: 24pt}
  p {font-size: 16pt}
</style>
</head>
<body>
<h1>What Is a Literal Result Element?</h1>
<p>You can use literal result elements in stylesheets.
A literal result element is any non-XSLT element,
including any attributes, that can be written literally in a
template, and that will be pushed literally onto the
result tree when processed.</p>
</body>
</html>
```

Figure 2-2 shows what *doc.html* looks like in Netscape 7.1. Actually, you can either open *doc.html* or *doc-pi.xml* and you'll be looking at essentially the same document.

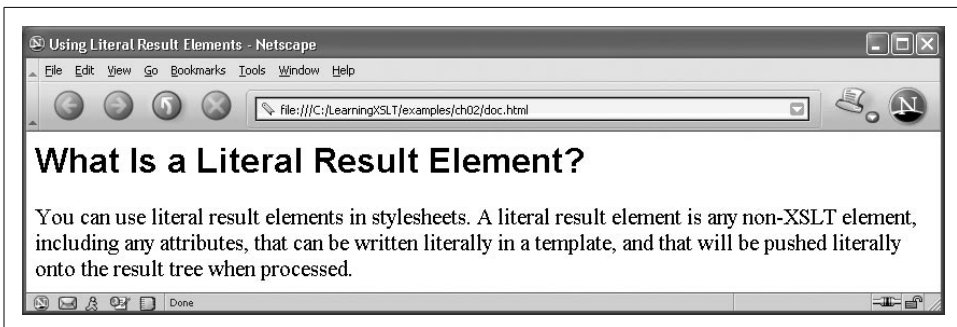


Figure 2-2. Displaying *doc.html* in Netscape 7.1

Applying namespaces

Before moving on, I want to call your attention to the namespace declaration in *doc.html*. This, which originated in a literal result element in *doc.xsl*, is considered a default namespace declaration:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

The URI *http://www.w3.org/1999/xhtml*, by the way, is the official namespace for XHTML 1.0. No prefix appears on any element or attribute in the resulting document. A default namespace declaration applies to the element on which it was declared, and also to any child elements that follow that element, but default declarations never apply to attributes.

There is little to no risk of having a name conflict between attribute names. For example, take two elements that both can have an attribute with the same name. With or without a namespace declaration, there won't be a name conflict because an attribute's domain, so to speak, is limited to the element that owns it. You can only use an attribute once on a given element—attribute names must be unique within the element. If, however, two attributes have the same name, and one is qualified with a namespace prefix (a QName with a prefix), those names won't conflict. For example, in the following fragment, the `invoice` start tag has two attributes:

```
<invoice order="293-7756-11" new:order="2003-08-31-4556">
```

There are two `order` attributes, but because one is qualified with a prefix, the names won't collide, and you don't break the rule of using an attribute more than once. For more details, see Section 5.2 of the XML namespaces specification.

Using the Element Called `element`

Literal result elements aren't the only way to create elements on the result tree. You can also use the XSLT instruction `element`. The following document, *element.xml*, is similar to *literal.xml*, which you saw earlier in this chapter:

```
<?xml version="1.0"?>

<message>You can use the element element to create elements on the result tree.
</message>
```

Unlike *literal.xml*, the stylesheet *element.xsl* uses `element` instead of a literal result element to create a new element in the output:

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

  <xsl:template match="message">
    <xsl:element name="{concat('my', name())}"><xsl:apply-templates/></xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

`element` has three attributes. The `name` attribute is required as it obviously specifies a name for the element. In this example, the `name` attribute uses an attribute value template to compute a name for the element. In other words, the name of the element is computed by using the `concat()` and `name()` functions to contrive a new name based on the name of the current node. This is useful when you don't have the name of a node until you actually perform the transformation (at runtime).

You don't have to use an attribute value template in the value of name—you could use any legal XML name you want in the value. Computing the name, however, is one justification for using `element`. Another justification is using attribute sets, which you'll learn about presently. Otherwise, you might as well use a literal result element, but the choice remains yours.

The namespace attribute

`element` has two other attributes beside `name`: `namespace` and `use-attribute-sets`, which are optional. I'll discuss `namespace` here, and I'll explain how to work with `use-attribute-sets` in “Reusing a Set of Attributes,” a little later in this chapter.

The `namespace` attribute identifies a namespace name to associate with the element. If `element`'s `name` attribute contains a QName with a prefix, the processor will usually associate the namespace name in the `namespace` attribute with the prefix in the QName, though it is not required to do so (see Section 7.1.2 of the XSLT spec). You can use either a namespace URI in `namespace` or you can compute the namespace with an attribute value template. The stylesheet *namespace.xsl* uses a namespace URI:

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

  <xsl:template match="/">
    <xsl:element name="doc:paragraph" namespace="http://www.example.com/documents">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

Apply this stylesheet to *element.xml*:

```
xalan element.xml namespace.xsl
```

and you will see what I'm talking about:

```
<?xml version="1.0" encoding="UTF-8"?>
<doc:paragraph xmlns:doc="http://www.example.com/documents">You can use the element
element to create elements on the result tree.</doc:paragraph>
```

When the XSLT processor encounters the namespace name *http://www.example.com/documents* in `namespace` and the QName `doc:paragraph` in `name`, it associates the prefix `doc` with the namespace name *http://www.example.com/documents* in the namespace declaration, as you can see. (I should say it *usually* associates the `doc` prefix with the namespace URI, unless there is a clash.)

Likewise, if you declare this namespace name and prefix on the document element in the stylesheet, as in *rootns.xsl*:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:doc="http://www.example.com/documents">
  <xsl:output method="xml" indent="yes"/>
```

```

<xsl:template match="/">
  <xsl:element name="doc:paragraph"><xsl:apply-templates/></xsl:element>
</xsl:template>

</xsl:stylesheet>

```

Transforming *element.xml* against *rootns.xsl* using:

```
xalan element.xml rootns.xsl
```

will produce the same result as transforming *element.xml* against *namespace.xsl*:

```

<?xml version="1.0" encoding="UTF-8"?>
<doc:paragraph xmlns:doc="http://www.example.com/documents">You can use the element
element to create elements on the result tree.</doc:paragraph>

```

This section has only covered a few basics about `element`. You will get to see `element` at work in a larger example in the later section, “One Final Example.” Now let’s add an attribute or two to the `paragraph` element with the `attribute` instruction.

Adding Attributes

To add a single, nonliteral attribute to `paragraph` in a result tree, all you have to do is add an XSLT `attribute` element as a child of `element`. The stylesheet *attribute.xsl* does just that:

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <xsl:element name="paragraph">
      <xsl:attribute name="priority">medium</xsl:attribute>
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>

```

Like `element`, `attribute` can have name and namespace attributes. Again, the name attribute, which specifies the name of an attribute for the result tree, is required, while namespace is not. The namespace attribute works pretty much like it does in `element`. The values of both name and namespace can be computed by using an attribute value template, just as in `element`.

Apply *attribute.xml* (which contains no attributes) to *attribute.xsl* with:

```
xalan attribute.xml attribute.xsl
```

to produce a result with a `priority` attribute:

```

<?xml version="1.0" encoding="UTF-8"?>
<paragraph priority="medium">You can use the attribute element to create attributes
on the result tree.</paragraph>

```

The next stylesheet, *attributes.xsl*, adds two more attributes to paragraph for a total of three attributes. One of the additional attributes will have a namespace, and one will not:

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

<xsl:template match="/">
  <xsl:element name="paragraph">
    <xsl:attribute name="priority">medium</xsl:attribute>
    <xsl:attribute name="date">2003-09-23</xsl:attribute>
    <xsl:attribute name="doc:style" namespace="http://www.example.com/documents">
      classic</xsl:attribute>
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

When transforming *attribute.xml* with *attributes.xsl*:

```
xalan attribute.xml attributes.xsl
```

it produces this result:

```
<?xml version="1.0" encoding="UTF-8"?>
<paragraph priority="medium" date="2003-09-23" xmlns:doc="http://www.example.com/
documents" doc:style="classic">You can use the attribute element to create attributes
on the result tree.</paragraph>
```

There is another way to specify multiple attributes besides listing them one after another: you can use an attribute set.

Reusing a Set of Attributes

The top-level attribute-set element in XSLT allows you to label a group of attributes with a name. Then you can reference and reuse that group of attributes by supplying the name in the use-attribute-sets attribute of element. The attribute element has a required name attribute, and it also has an optional use-attribute-sets attribute (such as element) so that you can chain attribute sets together. The next section, “Chaining attribute sets,” shows you how.

The stylesheet *attribute-set.xsl* implements this feature:

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

<xsl:attribute-set name="paragraph">
  <xsl:attribute name="priority">medium</xsl:attribute>
  <xsl:attribute name="date">2003-09-23</xsl:attribute>
  <xsl:attribute name="doc:style" namespace="http://www.example.com/documents">
    classic</xsl:attribute>
</xsl:attribute-set>
<xsl:template match="/">
```

```

    <xsl:element name="paragraph" use-attribute-sets="paragraph">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>

```

The `attribute-set` element is a top-level element in XSLT, meaning that it is only allowed as a child of the stylesheet's document element. Also, the `attribute-set` element allows only attribute elements as children. This named group of attributes is linked to the element `paragraph` by the `use-attribute-sets` attribute. You can also see that even though an element and an attribute set have the same name (`paragraph`), it poses no naming conflict within XSLT.

If you process *attribute-set.xml* against *attribute.xml* with:

```
xalan attribute.xml attribute-set.xml
```

you will get about the same result as processing it against *attributes.xml*:

```

<?xml version="1.0" encoding="UTF-8"?>
<paragraph priority="medium" date="2003-09-23" xmlns:doc="http://www.example.com/
document" doc:style="classic">You can use the attribute element to create attributes
on the result tree.</paragraph>

```

Chaining attribute sets

As I mentioned earlier, you can also chain attribute sets together. The stylesheet *chain.xml* shows you how to do this:

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>

  <xsl:attribute-set name="doc" use-attribute-sets="paragraph">
    <xsl:attribute name="doc:style" namespace="http://www.example.com/documents">
      classic</xsl:attribute>
    </xsl:attribute-set>
  <xsl:attribute-set name="paragraph">
    <xsl:attribute name="priority">medium</xsl:attribute>
    <xsl:attribute name="date">2003-09-23</xsl:attribute>
  </xsl:attribute-set>

  <xsl:template match="/">
    <xsl:element name="paragraph" use-attribute-sets="doc">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>

```

This stylesheet has two `attribute-set` elements that are chained together by means of the `use-attribute-sets` attribute. The element definition links to the attribute set named `doc`, which in turn links to the attribute set named `paragraph`.

When you process these using:

```
xalan attribute.xml chain.xml
```

the only difference you might see in the result is that the attributes may appear in a different order:

```
<?xml version="1.0" encoding="UTF-8"?>
<paragraph priority="medium" date="2003-09-23" xmlns:doc="http://www.example.com/
documents" doc:style="classic">You can use the element element to create elements on
the result tree.</paragraph>
```

This is not a problem because attributes are unordered in XML. Although a processor may attempt to keep track of the order of attributes, it is not obligated to do so by the XML 1.0 specification.

Finally, an attribute-set element need not have any content, that is, it does not have to have attribute children. This means that you can do the following (*chaining.xml*):

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

<xsl:attribute-set name="para" use-attribute-sets="paragraph"/>
<xsl:attribute-set name="paragraph">
  <xsl:attribute name="priority">medium</xsl:attribute>
  <xsl:attribute name="date">2003-09-23</xsl:attribute>
  <xsl:attribute name="doc:style" namespace="http://www.example.com/documents">
classic</xsl:attribute>
</xsl:attribute-set>

<xsl:template match="/">
  <xsl:element name="paragraph" use-attribute-sets="para">
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

The attribute-set element named *para* does not have any attribute children; however, it links to the attribute-set named *paragraph* with its *use-attribute-sets* attribute. This has the effect of, in essence, renaming *paragraph* to *para* and producing the same result as *chain.xml*. Here's the command:

```
xalan attribute.xml chaining.xml
```

Another thing to keep in mind is that *use-attribute-sets* is not a required attribute, neither on attribute-set nor on element. So, a stylesheet like *unchain.xml* is legal:

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

<xsl:attribute-set name="para">
  <xsl:attribute name="doc:style" namespace="http://www.example.com/documents">
classic</xsl:attribute>
</xsl:attribute-set>
```

```

<xsl:attribute-set name="paragraph">
  <xsl:attribute name="priority">medium</xsl:attribute>
  <xsl:attribute name="date">2003-09-23</xsl:attribute>
</xsl:attribute-set>

<xsl:template match="/">
  <xsl:element name="paragraph" use-attribute-sets="para">
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>

```

And when processed against *attribute.xml* with:

```
xalan attribute.xml unchain.xsl
```

it produces a result with only one attribute:

```

<?xml version="1.0" encoding="UTF-8"?>
<paragraph xmlns:doc="http://www.example.com/documents" doc:style="classic">You can
use the attribute element to create attributes on the result tree.</paragraph>

```

As you may have guessed already, you can use attribute-sets creatively to add attributes to, or omit them from, a result tree.

Outputting Comments

Comments allow you to hide advisory text in an XML document. You can also use comments to label documents, or portions of them, which can be useful for debugging. When an XML processor sees a comment, it may ignore or discard it, or it can make the text content of comments available for other kinds of processing. The text in comments is not the same as the text found between element tags, that is, it is not character data. As such, comments can contain characters that are otherwise forbidden, like < and &. XML comments are formed like this:

```
<!-- This element holds the current date & time -->
```



Comments are markup and can go anywhere in an XML document, except directly inside the pointy brackets of other kinds of markup. This means, for example, that you can't place a comment inside of a start tag of an element.

The only legal XML characters that a comment must not contain are the sequence of two hyphen characters (--), as this pair of characters signals the end of a comment. Other than that, you are free to use any legal XML character in a comment. (Again, to check on what characters are legal in XML, and where they are legal, see Sections 2.2 through 2.4 of the XML specification.)

To insert a comment into a result tree, you can use the XSLT instruction element `comment`, as demonstrated in the *comment.xsl* stylesheet:

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

  <xsl:template match="/">
    <xsl:comment> comment & msg element </xsl:comment>
    <msg><xsl:apply-templates/></msg>
  </xsl:template>

</xsl:stylesheet>
```

The output method is XML. If it were text, the comment would not show up in the output. Because comments in XML can contain markup characters, you can include an ampersand in a comment, among otherwise naughty characters, though it must first be represented by an entity reference (`&`) in the stylesheet.

Process this stylesheet against *comment.xml* with Xalan:

```
xalan comment.xml comment.xsl
```

You will get the following results:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- comment & msg element -->
<msg>You can insert comments in your output.</msg>
```

Outputting Processing Instructions

It must come as no surprise that you can add processing instructions, or PIs, to the result tree with the `processing-instruction` element. This element is formed like this:

```
<xsl:processing-instruction name="xml-stylesheet">href="new.css"
  type="text/css"</xsl:processing-instruction>
```

A `processing-instruction` element requires one attribute, `name`, which identifies the target name for the PI. The value of this attribute must be an NCName, and, as such, must not be a QName and cannot contain a colon. In other words, you can't qualify a target name with a namespace.

The content of the `processing-instruction` element contains the pair of pseudo-attributes `href` and `type` that are necessary to apply the CSS stylesheet *processing.css* to the resulting XML document:

```
paragraph {font-size: 24pt; font-family: serif}
code {font-family: monospace}
```

These rules will apply to the `paragraph` and `code` elements in the result tree. Provided that you view the result tree in a browser, any `paragraph` elements will be rendered with a best-fit serif font, in 24-point type, while any `code` elements will be rendered in a monospace font. (Courier is an example of a monospace font.) You'll get a chance to see the effects of these style rules later on in this section.

In the example that follows, I'll discuss more than just PIs. I'll also talk about a different kind of content in an XML document, and why you have to use more than one template to get at it. Consider for a moment the following XML document, *processing.xml*, which contains mixed content:

```
<?xml version="1.0"?>

<message>You can add processing instructions to a document with the <courier>
processing-instruction</courier> element.</message>
```

Mixed Content

The message element in *processing.xml* contains *mixed content*. Mixed content freely mixes character data and element content together. That's why you see tags for the courier element mixed with text in message. Any elements that appear in mixed content are allowed to appear in any order, although they, of course, must also be well-formed. In this context, well-formed elements must either have both start and end tags or must be empty element tags, and the characters used for text and names must follow XML 1.0 rules. (See Section 3.2.2 of the XML specification for more details about mixed content.)

processing.xsl handles the mixed content in *processing.xml*:

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

<xsl:template match="/">
  <xsl:processing-instruction name="xml-stylesheet">href="processing.css" type="text/
css"</xsl:processing-instruction>
  <xsl:element name="doc">
    <xsl:element name="paragraph"><xsl:apply-templates/></xsl:element>
  </xsl:element>
</xsl:template>

<xsl:template match="courier">
  <xsl:element name="code"><xsl:apply-templates/></xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Using Multiple Template Rules

For the first time in this book, you are seeing a stylesheet (*processing.xsl*) that has more than one template rule. (Remember, a template rule consists of a pattern to match and a constructor telling the processor what to do when the pattern is matched.) The way you design your templates tells the XSLT processor what to look for in a document, and then what to do if and when it finds what you've asked it to find.

In the stylesheet *processing.xsl*, the first template matches the root node in the document using */*. When the processor encounters *apply-templates* in this template, it matches any children of the root node in the source. When applied to *processing.xml*, the built-in templates for elements and text match the *message* element and its child text content.

The next template rule is invoked whenever it encounters a *courier* element in the source tree. There is only one *courier* element in *processing.xml*, so it is only invoked once. If there were more *courier* elements, it would be invoked for each occurrence of *courier*. This template also has an *apply-templates* child, which uses the built-in templates to find the text content of *courier* (you could try *value-of* here with the same outcome). As a result, the processor surrounds the character content of *courier* with *code* elements, and returns control to the template that invoked it.

The original template, seeing nothing else to do, picks up where the other template left off and takes care of its other work. With the processor holding onto the work that the other template did in a temporary tree, the built-in template for text nodes yanks the character data out of *message* and surrounds it with *paragraph* tags.

Somewhere along the way, it surrounds all the elements with the new root element *doc*. It creates a new PI, too, based on the instructions given by the *processing-instruction* element. Once that work is done, and the XSLT processor sees that there is nothing left to do, it writes its work out to the result tree, pulls down the shades, locks the door, and calls it quits.

What can go in a template rule?

It's obvious that the *template* element can hold a template rule, but other XSLT elements can hold templates as well. Generally speaking, a template consists of one or more XSLT elements that can create a result tree. These templates are not template rules *per se* because they don't have to match a pattern—they just contain sequence constructors. Literal result elements and literal text, as well as the *apply-templates*, *attribute*, *element*, *comment*, *processing-instruction*, and *text* elements can all be contained in templates.

The 15 elements that can contain templates (but don't match patterns) are:

- *attribute*
- *comment*
- *copy*
- *element*
- *fallback*
- *for-each*
- *if*
- *message*
- *otherwise*
- *param*
- *processing-instruction*
- *template*
- *variable*
- *when*
- *with-param*

A lot of elements in this list are probably new to you. It would consume pages to tell you what elements can go where in all possible templates, so for now please take this discussion of what templates are on faith. The concept of template rules and templates will continue to unfold throughout the book, all in due time. Meanwhile, Appendix A of Doug Tidwell's *XSLT*, also published by O'Reilly, provides an excellent XSLT reference and lists in detail what elements can contain, including those that follow in the template category.

Creating the PI and Putting It to Work

Now you can run the processor and see for yourself what the result actually is:

```
xalan -o proc.xml processing.xml processing.xsl
```

Here is what *proc.xml* looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="processing.css" type="text/css"?>
<doc>
<paragraph>You can add processing instructions to a document with the <code>
processing-instruction</code> element.</paragraph>
</doc>
```

Xalan placed the XML stylesheet PI in the document prolog (before the document element *doc*) because of where the *processing-instruction* element was placed in the first template. PIs can go anywhere in an XML document except inside other markup, so you can move the *processing-instruction* element to the second template if you want, and see where it comes out in the output.

The problem is, if the stylesheet PI does not appear in the prolog, the rendering engine (a browser in this case) won't apply the *processing.css* stylesheet. The point is that the order of templates, and the order of the content of templates, matters in regard to the output of those templates.

Figure 2-3 shows *proc.xml* displayed in IE.

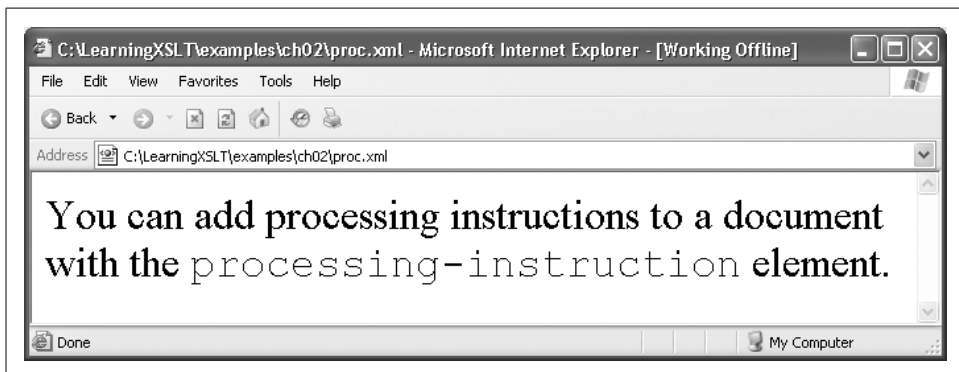


Figure 2-3. The document *proc.xml* displayed in IE using *processing.css*

One Final Example

Finally, to wrap things up, here is an example stylesheet that shows you, once again, how to perform most of the techniques discussed in this chapter. The example starts out with the rather short document containing mixed content, *final.xml*:

```
<?xml version="1.0"?>

<message>You can add processing instructions to a document with the <courier>
processing-instruction</courier> element.</message>
```

There isn't much to it, but you can augment *final.xml* with the well-rounded XSLT stylesheet, *final.xsl*:

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

<xsl:attribute-set name="atts">
  <xsl:attribute name="noteworthy">true</xsl:attribute>
  <xsl:attribute name="priority">medium</xsl:attribute>
</xsl:attribute-set>

<xsl:template match="/">
  <xsl:processing-instruction name="xml-stylesheet">href="final.css" type="text/css"
</xsl:processing-instruction>
  <xsl:comment> final.xml as processed with final.xsl </xsl:comment>
  <doc>
    <heading>Final Summary</heading>
    <paragraph>Following is a summary of how you can build documents with XSLT:
</paragraph>
    <paragraph>You can add text either literally or with the <code>text</code> element.
</paragraph>
    <paragraph>You can use literal result elements in stylesheets.</paragraph>
    <xsl:element name="paragraph">You can use <xsl:element name="code">element
</xsl:element> elements in stylesheets.</xsl:element>
    <xsl:comment> you can add a line break &amp; some spaces with the text element
</xsl:comment>
    <xsl:text>

    </xsl:text>
    <xsl:element name="paragraph"><xsl:attribute name="noteworthy">true</xsl:attribute>
You can add attributes to elements with the <xsl:element name="code">attribute
</xsl:element> element.</xsl:element>
    <xsl:element name="paragraph" use-attribute-sets="atts">You can even add sets of
attributes to elements with the <xsl:element name="code">attribute-set</xsl:element>
top-level element.</xsl:element>
    <paragraph>You can add comments with the <code>comment</code> element.</paragraph>
    <xsl:element name="paragraph"><xsl:text>And last but not least: </xsl:text>
<xsl:apply-templates select="message"/></xsl:element>
  </doc>
</xsl:template>
```

```

<xsl:template match="courier">
  <xsl:element name="code"><xsl:apply-templates/></xsl:element>
</xsl:template>

</xsl:stylesheet>

```

Processing *final.xml* with *final.xsl*, you can serialize the result tree and place it in a file:

```
xalan -o finally.xml final.xml final.xsl
```

The XML document *finally.xml* turns out like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="final.css" type="text/css"?>

<!-- final.xml as processed with final.xsl -->
<doc>
<heading>Final Summary</heading>
<paragraph>Following is a summary of how you can build documents with XSLT:
</paragraph>
<paragraph>You can add text either literally or with the <code>text</code> element.
</paragraph>
<paragraph>You can use literal result elements in stylesheets.</paragraph>
<paragraph>You can use <code>element</code> elements in stylesheets.</paragraph>
<!-- you can add a line break & some spaces with the text element -->

  <paragraph noteworthy="true">You can add attributes to elements with the <code>
attribute</code> element.</paragraph>
<paragraph noteworthy="true" priority="medium">You can even add sets of attributes to
elements with the <code>attribute-set</code> top-level element.</paragraph>
<paragraph>You can add comments with the <code>comment</code> element.</paragraph>
<paragraph>And last but not least: You can add processing instructions to a document
with the <code>processing-instruction</code> element.</paragraph>
</doc>

```

When *finally.xml* is displayed in Mozilla, it depends on the CSS stylesheet *final.css* to figure out how to render heading, paragraph, and code elements:

```

heading {display: block; font-size: 16pt; font-family: sans-serif; margin: 8pt 15pt}
paragraph {display: block; font-size: 12pt; font-family: serif; margin: 5pt 15pt}
code {display: inline; font-size: 11pt; font-family: monospace}

```

The display property with value of block gives the heading and paragraph elements a block- or box-like appearance on the browser canvas or rendering space. The display value of inline for code means that elements should be displayed inline with other text. The sans-serif font family for heading indicates that you want the browser to select a sans-serif font on a best-match basis, just as with monospace for code elements. The margin property sets the top and right margins for the element to either 8 and 15 points, or 5 and 15 points, respectively.

With this CSS applied in Mozilla, *finally.xml* is displayed in Figure 2-4.

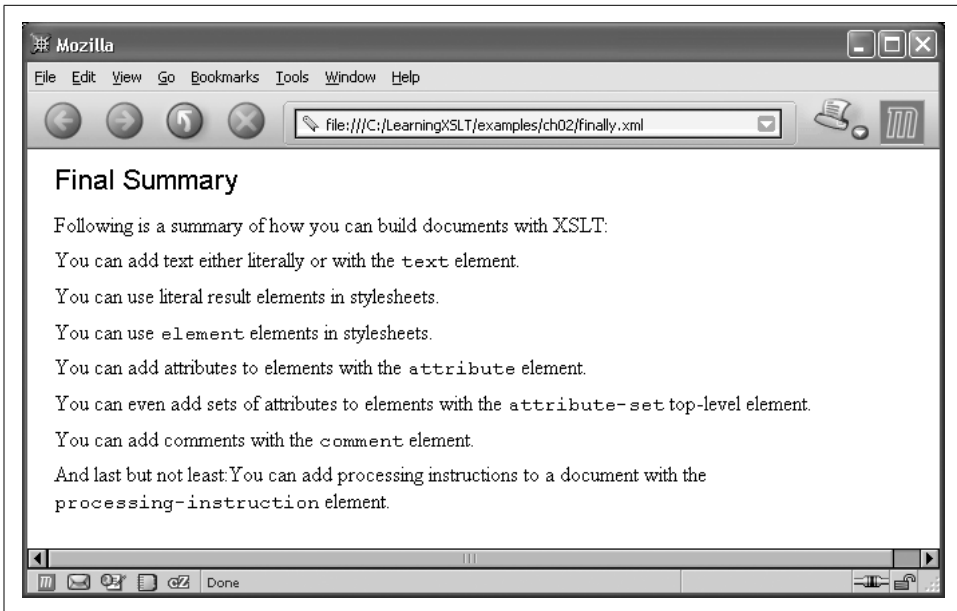


Figure 2-4. The document *finally.xml* displayed in Mozilla with *final.css*

Summary

In this chapter, you have learned the techniques that allow you to build a new result tree document. You learned about literal result elements and the XSLT instruction elements `text`, `element`, `attribute`, `attribute-set`, `comment`, and `processing-instruction`. You also learned about XHTML's relationship to HTML, and came to grips with some of the fundamentals of how template rules are evaluated and processed (more to come on that topic). You are now ready to explore ways that you can finely tune a result tree with the `output` element. You'll find out how in Chapter 3.