# 1

# First Steps with XSLT

COPYRIGHTED MATERIAL

In this chapter you will get started with XSLT by developing two stylesheets. In the first stylesheet you'll see how to generate an XHTML web page from an XML document.

The second stylesheet illustrates how to transform one XML data format to another, in this case from the Atom 1.0 syndication format to RSS 1.0.

You'll learn about:

- Key XSLT elements and structure
- Built-in template rules
- XPath expressions for matching and selection
- Different ways to invoke a stylesheet processor

## Transforming an XML Document to a Web Page

Probably the most common application of XSLT is to generate one or more pages of a website from an XML source of some kind. For example, you might want to split a large file into chapters, each with a separate page, or display a news feed.

There are a couple of ways to accomplish this: You might want to rely on a browser's client-side processor to transform the content; alternatively, you could generate static content for a server to render.

Let's start with an example that relies on a browser's built-in processor. It is drawn from the case study that you will work on later in this book.

> *The case study in Chapter 11 illustrates the production of a website from a set of XML source documents that describe each of the XSLT elements and functions. The same information was used to produce the XSLT Quick Reference in Appendix C.*

If you haven't already done so, download the source code for this book from this book's web page at www.wrox.com. You'll be using the source files from now on in the examples that follow. When you've unzipped the download, open the folder for Chapter 1 and locate the file xml_stylesheet.xml.

Listing 1-1 shows a pared-down version of the source document describing the <xsl:stylesheet> element.

**Listing 1-1**

```
<?xml version="1.0" encoding="UTF-8"?>
<reference>
  <body>
     <title>xsl:stylesheet</title>
     <purpose>
        <p>The root element of a stylesheet.</p>
     </purpose>
     <usage>
        <p>The <element>stylesheet</element> is always the root element, even if
a stylesheet is included in, or imported into, another. It must have a
<attr>version</attr> attribute, indicating the version of XSLT that the
stylesheet requires.</p>
        <p>For this version of XSLT, the value should normally be "2.0". For a
stylesheet designed to execute under either XSLT 1.0 or XSLT 2.0, create a core
module for each version number; then use <element>xsl:include</element> or
<element>xsl:import</element> to incorporate common code, which should specify
<code>version="2.0"</code> if it uses XSLT 2.0 features, or
<code>version="1.0"</code> otherwise.</p>
       <p>The <element>xsl:transform</element> element is allowed as a synonym.</p>
        <p>The namespace declaration <code>xmlns:xsl="http//www.w3.org/1999/XSL/
Transform</code> by convention uses the prefix <code>xsl</code>.</p>
        <p>An element occurring as a child of the <element>stylesheet</element>
element is called a declaration. These top-level elements are all optional, and
may occur zero or more times.</p>
     </usage>
  </body>
</reference>
```

In the quick reference documents I use an XML grammar based on the Darwin Information Typing Architecture (DITA) reference content model. DITA is finding increasing support among the larger publishers of technical documentation. It differs considerably from the longer-established DocBook format, using a more modular approach, covering the concept/task/reference pattern often found in software help systems. You can look ahead to see details of the schema in Chapter 11. In addition to markup like <body>, <p> and <code>, which you'll recognize from XHTML, note that the root element in this example is <reference>. To keep the example simple, only the sections on <purpose> and <usage> are included, as are the inline <attr> (attribute) and <element> names. In later chapters I'll introduce more elements from the reference vocabulary.

## *Using a Browser*

To run any transform inside a browser, you need to add a processing instruction to the source document, which will give the browser the location of the stylesheet you want to use. This goes immediately after the XML declaration in xsl_stylesheet.xml. Save the update while you start work on the stylesheet.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="browser.xsl" type="text/xsl"?>
<reference>
...
</reference>
```

You may have used processing instructions in other XML applications; what appear to be attributes in the instruction are in fact known as *pseudo-attributes*. The `href` pseudo-attribute locates the stylesheet (`browser.xsl`). The file extension `.xsl` is a convention, which some applications may rely on for identification. The `type` pseudo-attribute defines the content type (text/xsl). In this case you use a relative URI, for a stylesheet in the same directory as the source document. (In this book I use the more general term URI, which is an identifier that may not imply a specific location, whereas the term URL implies a location from which you can obtain a representation of a resource such as an HTML page.

The W3C recommendation for this processing instruction is separate from the XSLT specifications, and is at `www.w3.org/TR/xml-stylesheet/`.

> *The semantics of pseudo-attributes is the same as that of the attributes used in* `<link rel="stylesheet">` *in HTML.*
>
> *The content type expressed need not be XSLT, and this processing instruction is often used to specify multiple CSS files to handle different types of media using the value* `'text/css'`*. The content type for XSLT 1.0 was never specified in the W3C recommendation. Microsoft invented the* `'text/xsl'` *value for Internet Explorer, which seems to have stuck in practice, though browsers may also accept other values, such as* `'text/xml'`*. The XSLT 2.0 recommendation formally registers the media type* `'application/xslt+xml'`*.*

## Built-in Rules

We can now process the sample by writing a bare-bones transform. It is not very exciting, but it illustrates the default behavior of a processor using a built-in template rule, specified in the XSLT specification.

XSLT defines built-in rules for processing templates, and the rule for document and element nodes ensures that the root element and all of its children will be handled recursively, even if there are no element-specific templates.

This book generally specifies XSLT version 2.0 for stylesheets. However, in the following Try It Out you'll create an XSLT 1.0 transform using a single root `<xsl:stylesheet>` element to demonstrate this built-in behavior.

### Try It Out A Root Element Stylesheet

To create the transform, follow these steps:

1. In the Oxygen IDE, mentioned in this book's Introduction, create a new document by choosing New ➢ Stylesheet (XSL) File.
2. In the dialog that appears, select 1.0 as the Stylesheet version.

3. Enter **browser.xsl** as the filename and click Finish. The new file should open with the following contents:

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

</xsl:stylesheet>
```

Because this stylesheet is an XML document, it must begin with an XML declaration, specifying the version number and the encoding. The root element in a stylesheet is `<xsl:stylesheet>`, though the synonym `<xsl:transform>` may also be used. You must always specify the XSLT namespace, and it is important to set the `version` attribute correctly to match the type of processing required.

After you've declared the namespace, all the XSLT element names require the namespace prefix, which is `xsl:` by convention. The prefix also makes it clear which element is referenced if other namespaces are in use.

Browsers that I have used will not complain about the version number, and many version 1.0 features are unchanged in version 2.0. In any case, it is good practice to document your intentions.

You can now process the sample. Open the `xsl_stylesheet.xml` file from a browser using the File ➢ Open menu command. You should see something like the output shown in Listing 1-2.

**Listing 1-2**

```
xsl:stylesheetThe root element of a stylesheet.The xsl:stylesheet element is
always the root element, even if a stylesheet is included in, or imported
into, another. It must have a version attribute, indicating the version of
XSLT that the stylesheet requires. For this version of XSLT, the value should
normally be "2.0". For a stylesheet designed to execute under either XSLT 1.0
or XSLT 2.0, create a core module for each version number; then use xsl:include
or xsl:import to incorporate common code, which should specify version="2.0" if
it uses XSLT 2.0 features, or version="1.0" otherwise.The xsl:transform element
is allowed as a synonym.The namespace declaration
xmlns:xsl="http//www.w3.org/1999/XSL
/Transform by convention uses the prefix xsl. An element occurring as a child of
the xsl:stylesheet element is called a declaration. These top-level elements are
all optional, and may occur zero or more times.
```

What has happened here? Without any further instructions, the processor has output all the text nodes from the source document.

*Safari 3.4 on Windows reports an empty stylesheet error and renders nothing, which suggests that something may not be quite right with the implementation of built-in rules. Google Chrome produces the same result, presumably because it is based on the same core engine.*

## *Defining an Output Method*

You can provide hints to the stylesheet processing by adding some output specifications to your stylesheet. Unless otherwise specified as HTML or XHTML, the output will be in XML format. It is also possible to add user-defined methods.

You can define the type of output in the declaration `<xsl:output>`. You saw the schema definition in this book's introduction, but here it is as a reminder. The attribute list is quite extensive, but for now I'd like to focus on just a few attributes:

```
<xs:element name="output" substitutionGroup="xsl:declaration">
   <xs:complexType mixed="true">
      <xs:complexContent mixed="true">
         <xs:extension base="xsl:generic-element-type">
            <xs:attribute name="name" type="xsl:QName"/>
            <xs:attribute name="method" type="xsl:method"/>
            <xs:attribute name="byte-order-mark" type="xsl:yes-or-no"/>
            <xs:attribute name="cdata-section-elements" type="xsl:QNames"/>
            <xs:attribute name="doctype-public" type="xs:string"/>
            <xs:attribute name="doctype-system" type="xs:string"/>
            <xs:attribute name="encoding" type="xs:string"/>
            <xs:attribute name="escape-uri-attributes" type="xsl:yes-or-no"/>
            <xs:attribute name="include-content-type" type="xsl:yes-or-no"/>
            <xs:attribute name="indent" type="xsl:yes-or-no"/>
            <xs:attribute name="media-type" type="xs:string"/>
            <xs:attribute name="normalization-form" type="xs:NMTOKEN"/>
            <xs:attribute name="omit-xml-declaration" type="xsl:yes-or-no"/>
            <xs:attribute name="standalone" type="xsl:yes-or-no-or-omit"/>
            <xs:attribute name="undeclare-prefixes" type="xsl:yes-or-no"/>
            <xs:attribute name="use-character-maps" type="xsl:QNames"/>
            <xs:attribute name="version" type="xs:NMTOKEN"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

In XSLT 1.0, the `method` attribute can take the values `"xml"`, `"html"`, or `"text"`. For instance, you would use the `"text"` method to output a CSV file, which you'll learn how to do in Chapter 9. You would use `"xml"` as a value for SVG output, and also for PDF because it requires transforming to the XSLFO format as an intermediate step. The XSLT 2.0 specification adds `"xhtml"`to the possible attribute values.

However, in the next XSLT 1.0 example you'll use `"xml"` as a value, as the output is XHTML.

> *The* `version` *attribute on the* `<xsl:stylesheet>` *element is rather confusing. It has absolutely nothing to do with the version of XSLT; rather, it refers to the version of XML to be output.*

You can define an `encoding` attribute, which specifies the preferred character encoding of the output document. All XSLT processors (and XML parsers) are required to support UTF-8 and UTF-16. Clearly, processing Chinese or Japanese content with UTF-8 encoding would produce corrupt output.

On this occasion, you'll set it to `'UTF-8'`. You can also add two more attributes specifying the XHTML `doctype-system` and `doctype-public` attribute values. These will result in the processor generating correct declarations in the output, before the `<html>` element:

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

```
      version="1.0">
   <xsl:output
      method="xml"
      encoding="UTF-8"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
      doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"/>
   ...
   </xsl:stylesheet>
```

An XSLT 2.0 stylesheetstylesheet may contain multiple `<xsl:output>` declarations and may include or import stylesheet modules that also contain `<xsl:output>` declarations. This enables you to use one or more stylesheets to output results using different methods. So you might, for instance, output both a CSV file and a web page in one pass.

If you use multiple declarations in this way, the `name` attribute must be specified on each `<xsl:output>` element to identify it. These names should match a set of `format` attribute values on `<xsl:result-document>` instructions, which I discuss in Chapter 7, in the stylesheet. The following snippet briefly illustrates how this might work:

```
<xsl:output name="web" method="xhtml" encoding="UTF-8"/>
<xsl:output name="csv" method="text"/>

<xsl:template match="/">
   <xsl:result-document format="web">
      ...
   </xsl:result-document>
   <xsl:result-document format="csv">
      ...
   </xsl:result-document>
</xsl:template>
```

*The form of* `name` *attributes on XSLT elements is defined as a lexical* QName *or namespace qualified name. It applies, for example, to templates, attribute sets, variables, parameters, and so on. Typically this is a simple name, but it may also be qualified with a namespace prefix such as* `<xsl:function name="xm:getentry-by-id">`*. If two qualified names are compared, the namespace URI that is declared with the prefix and the local name is used.*

## Main Template

The `<xsl:template>` element, which is covered in more detail in Chapter 3, is a basic building block of a stylesheet. This element is used to declare templates that match elements in the XML source, and to generate nodes in a result tree. Usually a stylesheet has a main template with the `match` attribute, set to `"/"`:

```
<xsl:template match="/">
   ...
   <xsl:apply-templates select="reference/body"/>
   ...
</xsl:template>
```

This value is an XPath expression that means ''match the root of the source tree.'' Note that this is *not* the same thing as the root element of the source document. The root of the source tree is outside of everything, including the containing top-level element.

This means that processing will begin right at the start of the XML source tree, outside the `<reference>` element. Path expressions in this context will be relative to this location.

The contained `<xsl:apply-templates>` instruction selects the `<body>` element in the source document (see Listing 1-1) for processing, showing the relative XPath expression `"reference/body"` in the `select` attribute. This means that the `<body>` element and everything inside it have been selected for processing. This instruction simply defines a set of nodes to be processed using the template rules for each source node to be matched.

> *You are not restricted to following the nested nodes as shown in this example. You might want to select all the paragraphs in the source document for processing, in which case you would use* `<xsl:apply-templates select= "//p"/>`*. There's more on XPath expressions in Chapter 2.*

## Literal Result Elements

You have two options when it comes to generating the element names that will be output. Usually, the most straightforward is to create what is called a *literal result element* by typing the element name with start and end tags straight into the stylesheet, and then populating the new elements with selected content from the source XML.

Literal result elements are treated as data to be copied from the result tree directly to the output. These elements can have any name, and the content may be XSLT instructions, nested literal result elements, or text. If you set attributes on the literal result elements, they will also be copied to the output.

This gives you considerable freedom to construct output from any source in your target XML vocabulary. For this XHTML page, you will start with something like the following skeleton:

```
<xsl:template match="/">
   <html>
   <head>
      <title>
         <xsl:value-of select="reference/body/title"/>
      </title>
   </head>
   <body>
      <p>The body goes here</p>
   </body>
</html>
</xsl:template>
```

The preceding code will render the following output:

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>xsl:stylesheet</title>
   </head>
   <body>
      <p>The body goes here.</p>
   </body>
<html>
```

## Selecting Source Content

To output source content, you select from the element to be transformed, using the `<xsl:value-of>` instruction. The `select` attribute defines the XPath expression to use. The next code snippet shows how:

```
<h1 class="section"><xsl:value-of select="title"/></h1>
```

As you learned in the introduction to this book, the `<xsl:value-of>` element is a sequence constructor, which is a series of XSLT instructions. This is the schema declaration:

```
<xs:element name="value-of" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="separator" type="xsl:avt"/>
        <xs:attribute name="disable-output-escaping" type="xsl:yes-or-no" default=
          "no"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

## Processing Specific Source Elements

The processor's built-in template rules have a lower priority than other templates, so by adding rules for individual elements, you can override the defaults.

Now you can add specific templates for the structural elements in the XML source: `<title>`, `<purpose>`, `<usage>`, and `<p>`. The `match` attribute identifies the element, and the output is specified with literal result elements. The `select` attribute value `"."` for the `<title>` element is an XPath expression that refers to the current node being processed. Because both the `<purpose>` and the `<usage>` elements can contain paragraphs, we apply processing to all the `<p>` content and its inline markup.

These templates are located at the top level like the main template you have just written, but their order is not significant. The XSLT processor treats the source elements in document order and will look in the templates for matches as it goes. I generally put them in rough document order in simple stylesheets:

```
<xsl:template match="title">
   <h1>
      <xsl:value-of select="."/>
   </h1>
</xsl:template>

<xsl:template match="purpose">
   <h2>Purpose</h2>
   <xsl:apply-templates select="p"/>
</xsl:template>

<xsl:template match="usage">
   <h2>Usage</h2>
   <xsl:apply-templates select="p"/>
</xsl:template
```

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

In the next template you match the XML source `<attr>` (attribute) and `<element>` names using the XPath union operator `"|"`, and output a containing `<code>` literal result element. The union operator performs a logical `OR`, matching either of the source element names:

```
<xsl:template match="attr | element">
   <code>
      <xsl:value-of select="."/>
   </code>
</xsl:template>
```

The output for an element name will look like this:

```
<p>An element occurring as a child of the
   <code>xsl:stylesheet</code> element is called a
   declaration. These top-level elements are all optional, and may
   occur zero or more times.
</p>
```

## Copying Content

When content in both the source and the output should be identical, you can simply copy the source nodes to the result. With the `<xsl:copy>` instruction, you copy the source `<code>` element name *and* its content to the output:

```
<xsl:template match="code">
      <xsl:copy>
         <xsl:apply-templates/>
      </xsl:copy>
</xsl:template>
```

Here is the schema definition:

```
<xs:element name="copy" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="copy-namespaces" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="inherit-namespaces" type="xsl:yes-or-no" default=
           "yes"/>
        <xs:attribute name="use-attribute-sets" type="xsl:QNames" default=""/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

There are two copy instructions in XSLT. The `<xsl:copy>` instruction is a *shallow copy*, and copies only the context node, but nothing under it. You specify the output in the sequence constructor inside `<xsl:copy>`. This instruction is most useful when copying element nodes.

It causes the current XML node in the source document to be copied to the output. The actual effect depends on whether the node is an element, an attribute, or a text node. For an element, the start and end element tags are copied; the attributes, character content, and child elements are copied only if `xsl:apply-templates` is used within `xsl:copy`.

In contrast, if you use `<xsl:copy-of>`, each new node will contain copies of all the children, attributes, and namespaces of the original node, recursively. This is often called a *deep copy*. This instruction has a `select` attribute, providing you with more flexibility in selection:

```
<xs:element name="copy-of" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:versioned-element-type">
        <xs:attribute name="select" type="xsl:expression" use="required"/>
        <xs:attribute name="copy-namespaces" type="xsl:yes-or-no" default="yes"/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

Listing 1-3 shows the completed stylesheet. Save this version as `local.xsl`.

**Listing 1-3**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0">
   <xsl:output method="html"
      encoding="UTF-8"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
      doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"/>

   <xsl:template match="/">
      <html>
         <head>
            <title>
               <xsl:value-of select="reference/body/title"/>
            </title>
         </head>
         <body>
            <xsl:apply-templates select="reference/body"/>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="title">
      <h1>
         <xsl:value-of select="."/>
      </h1>
   </xsl:template>

   <xsl:template match="purpose">
      <h2>Purpose</h2>
      <xsl:apply-templates select="p"/>
   </xsl:template>
```

```
    <xsl:template match="usage">
       <h2>Usage</h2>
       <xsl:apply-templates select="p"/>
    </xsl:template>

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

    <xsl:template match="attr | element">
       <code>
          <xsl:value-of select="."/>
       </code>
    </xsl:template>

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

</xsl:stylesheet>
```

## *Transforming Locally*

This time you'll run the stylesheet processor ''locally,'' rather than on the browser. The setup suggestions that follow make further use the Oxygen IDE . You'll also look briefly at invoking the Java command-line interface for the Saxon processor.

### Try It Out Configuring a Transformation

You can use the Oxygen IDE to set up a transformation, providing a range of configuration values. Essentially, you provide input and output values, which are associated with the source file and saved automatically for reuse. To avoid typing long paths, you can use editor variables recognized by the application, such as ''`${cfdu}`'' for ''current file directory URL.''

1. Open `xsl_stylesheet.xml` in the XML editor.
2. Choose XML ➢ Configure Stylesheet Transformation.
3. Click New in the dialog that opens, and name the scenario **xml2xhtml** in the Edit Scenario dialog that appears, as shown in Figure 1-1. By default, the variable **`"${currentFileURL}"`** is used for the XML URL setting.
4. On the XSLT tab, insert **${cfdu}local.xsl** in the XSL URL control.
5. Choose Saxon6.5.5 in the Transformer drop-down. Figure 1-1 shows the settings.
6. On the Output tab, accept the default setting Save as **`${cfn}.html`**, which will save the XHTML file in the same directory as the source, with the current filename.
7. Check Show in Browser and click OK.
8. In the main dialog, click Transform Now.

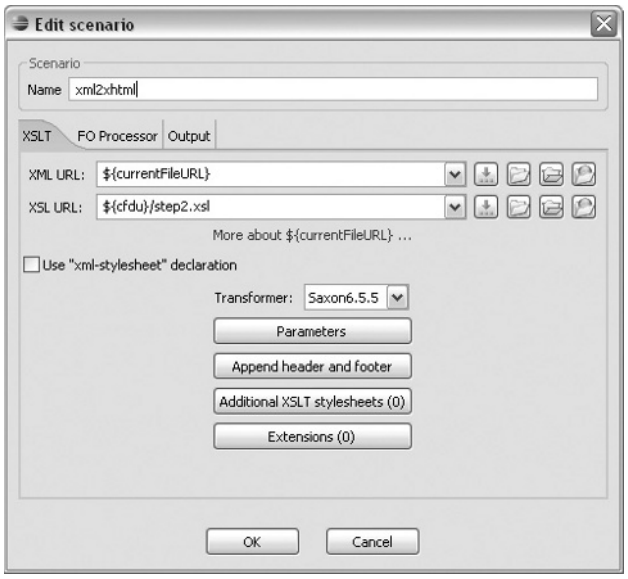


Figure 1-1

The transformed document should open in your browser.

Figure 1-2 shows the browser output, and Listing 1-4 shows the XHTML source code.

# xsl:stylesheet

## Purpose

The root element of a stylesheet.

## Usage

The `xsl:stylesheet` is always the root element, even if a stylesheet is included in, or imported into another. It must have a `version` attribute, indicating the version of XSLT that the stylesheet requires.

For this version of XSLT, the value should normally be `"2.0"`. For a stylesheet designed to execute under either XSLT 1.0 or XSLT 2.0, create a core module for each version number; then use `xsl:include` or `xsl:import` to incorporate common code. which should specify `version="2.0"` if it uses XSLT 2.0 features, or `version="1.0"` otherwise.

The `xsl:transform` element is allowed as a synonym.

The namespace declaration `xmlns:xsl="http//www.w3.org/1999/XSL/Transform` by convention uses the prefix `xsl`.

An element occurring as a child of the `xsl:stylesheet` element is called a declaration. These top-level elements are all optional, and may occur zero or more times.

Figure 1-2

**Listing 1-4**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title>xsl:stylesheet</title>
   </head>
   <body>
      <h1>xsl:stylesheet</h1>
      <h2>Purpose</h2>
      <p>The root element of a stylesheet.</p>
      <h2>Usage</h2>
      <p>The <code>stylesheet</code> is always the root element, even if a
stylesheet is included in, or imported into, another. It must have a <code>
version</code> attribute, indicating the version of XSLT that
the stylesheet requires.</p>
      <p>For this version of XSLT, the value should normally be <code>"2.0"</code>.
For a stylesheet designed to execute under either
XSLT 1.0 or XSLT 2.0, create a core module for each version number;
then use <code>xsl:include</code> or <code>xsl:import</code> to
incorporate common code, which should specify <code>version="2.0"</code> if it uses
XSLT 2.0 features, or <code>version="1.0"</code> otherwise.</p>
      <p>The <code>xsl:transform</code> element is allowed as a synonym.</p>
      <p>The namespace declaration <code>xmlns:xsl="http//www.w3.org/1999/XSL
/Transform</code> by convention uses the prefix <code>xsl</code>.</p>
      <p>An element occurring as a child of the <code>stylesheet</code> element is
called a declaration. These top-level elements are all optional, and may occur
zero or more times.</p>
   </body>
</html>
```

## *Using the Command Line*

Another way to invoke a stylesheet processor is to use a command-line interface. The specifics of the interface will vary according to which processor you use.

The next example uses the Saxon CLI for the open-source version on the `local.xsl`. If you intend to use the CLI frequently, you may prefer to run it from an open-source tool like jEdit (`www.jedit.org`), rather than from the file system console.

If you are using the Oxygen IDE and just want to experiment with the CLI, you will find the `.jar` file in the `lib` directory. If you are not using a bundled version of Saxon, you can download the Java version of the Saxon processor from SourceForge (`http://sourceforge.net/project/showfiles.php?groupid=29872`).

Unzip the download to a convenient directory. Add the `saxon9.jar` file to the classpath so that the command in the following Try It Out will locate the main program `net.sf.saxon`. The schema-aware version is `com.saxonica`.

Full documentation is available on the Saxonica site, which you should consult for installation and configuration instructions (`www.saxonica.com/documentation/contents.html`).

### Try It Out The Saxon CLI

Enter the following code on the command line and execute it:

```
java net.sf.saxon.Transform -s:xsl_stylesheet.xml -xsl:local.xsl
-o:xsl_stylesheet.html
```

The options in the example have the following meanings:

- `-s:filename`: The source XML file
- `-xsl:filename`: The XSL stylesheet to use
- `-o:filename`: The output filename

The remaining Saxon CLI options are extensive and quite powerful. If you are interested in pursuing the CLI approach, you should review the Saxon documentation at your leisure before using them.

# Transforming XML Data to XML

The next example illustrates how simple it can be to transform content from one XML format to another. A common transform problem is that two similar schemas will use different names for identical content values. Another problem is that in one case an attribute is used for a value, while another uses an element for the same purpose.

The next stylesheet uses two common metadata vocabularies that express information in roughly the same manner. One is the Atom 1.0 format, increasingly used for blogs and news feeds; the other is RSS 1.0, which uses a combination of the Dublin Core Metadata Initiative vocabulary and RDF/XML.

> *There is also a version 2.0 ''branch'' of RSS. Although it is often assumed that RSS 2.0 supersedes RSS 1.0, it doesn't; and the versions are incompatible in several ways. RSS 2.0 is also in widespread use, but we won't be using it in this chapter. If you want to explore the structure of RSS 2.0, go to* `http://cyber.law.harvard.edu/rss/rss.html`.

Of course, you wouldn't bother serializing either of these feeds to XML if the data were in a SQL database or an RDF triple store. However, if you are aggregating the data from feed URLs, or you have been provided with source data in XML, you won't have much choice.

## Atom and RSS Elements

The next two tables compare the feed elements and entry elements in the Atom 1.0 schema with the equivalents in RDF Site Summary (RSS) 1.0. The pros and cons of the different ways to describe metadata can be a contentious issue, but just now we need not be concerned with the details.

The following table lists the top-level elements that define the properties of the Atom feed in the `<feed>` and RSS 1.0 `<channel>` elements. The matches are quite weak at this level, perhaps reflecting the history

of how these structures were developed. The Atom specification provides a richer set of values on the whole.

| Atom | RSS 1.0 | Description |
|---|---|---|
| feed | channel | Root element of the feed document |
| title | title | Feed title |
| id | | Feed identifier |
| updated | | Date the feed was most recently updated |
| subtitle | | Feed subtitle |
| generator | | Application that generated the feed |
| link | link | URL for the HTML version of the feed |
| | description | Feed description |
| logo, icon | image | URI for a feed image |
| | items | RDF sequence acting as a table of contents |
| entry | item | Feed entry container |

The next table shows elements in the Atom <entry> and RSS 1.0 <item> elements. The ''item'' contents for RSS 1.0 are in the Dublin Core (DC) namespace. The matches between the schemas are much closer here, and the differences reflect the fact that the DC format has strong origins in the library community, and Atom was primarily developed for the requirements of web logs.

| Atom | DC | Description |
|---|---|---|
| id | identifier | Identifier of the resource |
| title | title | Name by which the resource is known |
| published | date | Date of publication |
| updated | | Date updated |
| author | | Container for name, e-mail, and URI elements |
| name | creator | The person or organization responsible for creating the resource |
| email | | Author's e-mail address |
| uri | | URI associated with the author |
| contributor | contributor | Contributor to a work; same structure as author. |

*Continued*

| Atom | DC | Description |
|---|---|---|
| summary | description | Description of the resource |
| | language | Principal language of the resource |
| content/@type | format | File format |
| | type | Defines either the genre or intellectual type of the resource |
| | publisher | Supplier of the resource |
| source | source | Identifier for source material for the resource, assuming it is derived from another format |
| content | | Container of or link to the content |
| | coverage | Locations or periods that are subjects of the resource |
| category | subject | Subjects of the resource |
| rights | rights | Rights information |
| | relation | Reference to a related resource |

Listing 1-5 shows part of an Atom feed from the `xml.com` website. We'll use this document as the source for the transformation. Some content, an additional namespace declaration, and stylesheet-processing instructions have been removed for clarity.

The listing shows the `<feed>` element and its content to the end of the first `<entry>` element. The code download is in the file `atom.xml`.

**Listing 1-5**

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
   <title>O'Reilly News: XML</title>
   <link rel="alternate" type="text/html" href="http://news.oreilly.com/"/>
   <id>tag:news.oreilly.com,2008-08-01://44</id>
   <updated>2008-12-17T07:32:30Z</updated>
   <subtitle>O'Reilly News - Spreading the knowledge of innovators</subtitle>
   <generator uri="http://www.sixapart.com/movabletype/">Movable Type Pro 4.21-en
     </generator>
   <link rel="self" href="http://feeds.oreilly.com/oreilly/xml"
      type="application/atom+xml"/>
   <entry>
      <title>Defining markup languages using Unicode properties</title>
      <link rel="alternate" type="text/html"        href="http://feeds.oreilly.com
         /~r/oreilly/xml/~3/487372046/defining-markup-languages-usin.html"/>
      <id>tag:broadcast.oreilly.com,2008://53.34679</id>
      <published>2008-12-17T03:01:23Z</published>
      <updated>2008-12-17T07:32:30Z</updated>
```

```
      <summary>Can we define a family of markup languages that used the Unicode
        properties and which could accept a fair imitation of XML and produce
        a SAX-like event stream?</summary>
      <author>
         <name>Rick Jelliffe</name>
      </author>
      <category term="xml" label="xml"
         scheme="http://www.sixapart.com/ns/types#tag"/>
      <content type="html" xml:lang="en" xml:base="http://broadcast.oreilly.com/">
        Can we define a family of markup languages that used the Unicode
        properties and which could accept a
        fair imitation of XML and produce a SAX-like event stream?
        &lt;img src="http://feeds.oreilly.com/~r/oreilly/xml/~4/487372046"
height="1" width="1"/&gt;</content>
   </entry>
   ...
</feed>
```

*This feed will be well out of date when you read this. To get a current version, go to* `http://feeds.oreilly.com/oreilly/xml`*, copy the source, and save it as a replacement. Alternatively, you can load the data directly from the feed site using the URL containing the feed source.*

## Developing the Stylesheet

As with the XML to HTML transform, we'll take the development one step at a time. The approach is essentially the same, with XML as the target rather than HTML. The vocabularies are, of course, different, but the matching process will work similarly. It is not too important at present to absorb the details of the Atom and RSS 1.0 formats, but if you would like to do so here are the relevant URLs:

**Atom 1.0** `www.atomenabled.org/developers/syndication/atom-format-spec.php`

**RSS 1.0** `http://web.resource.org/rss/1.0/spec`

I'll call the top-level elements ''feed elements,'' and the individual entries ''entry elements,'' using the Atom terminology.

### Preliminaries

Let's start with the basics of the stylesheet `rss_feed.xsl`. This time you'll set `"2.0"` as the value of the stylesheet's `version` attribute. Inside the `<xsl:stylesheet>` element are two namespaces to declare using the `rdf` and `dc` prefixes.

> **Always check the source file for a default namespace declaration: In this case it is `"http://www.w3.org/2005/Atom"`. You need to set the `xpath-default-namespace` attribute on the `<xsl:stylesheet>` element to this value; otherwise, *nothing* from the source file will be output.**

Next, declare the output method as `"xml"` and the encoding as `"UTF-8"`. In the main template, create the literal result elements `<rdf:RDF>` and `<channel>`, in that order, as the container for your output. The namespaces must be declared again on the `<rdf:RDF>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"

xpath-default-namespace="http://www.w3.org/2005/Atom">

   <xsl:output method="xml" encoding="UTF-8"/>
   <xsl:template match="/">
      <rdf:RDF
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
        <channel>
        ...
        </channel>
      </rdf:RDF>
   </xsl:template>
   ...
</xsl:styleheet>
```

### *Specifying Attributes*

Both the `<channel>` and `<item>` elements require the `rdf:about` attribute.

An attribute can be set directly on a literal result element if you know its value ahead of time, but in this case you need to use another approach, with the `<xsl:attribute>` instruction. This element should always come first in any set of sequence constructor instructions. You can use either the element content or the `select` attribute, but note that these approaches are mutually exclusive.

The XSLT 2.0 schema definition looks like this:

```
<xs:element name="attribute" substitutionGroup="xsl:instruction">
  <xs:complexType>
    <xs:complexContent mixed="true">
      <xs:extension base="xsl:sequence-constructor">
        <xs:attribute name="name" type="xsl:avt" use="required"/>
        <xs:attribute name="namespace" type="xsl:avt"/>
        <xs:attribute name="select" type="xsl:expression"/>
        <xs:attribute name="separator" type="xsl:avt"/>
        <xs:attribute name="type" type="xsl:QName"/>
        <xs:attribute name="validation" type="xsl:validation-type"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

To add the `rdf:about` attribute to the `<channel>` element, enter the `<xsl:attribute>` element right after the `<channel>` element, and use the `<xsl:value-of>` instruction to obtain the link URL `http://news.oreilly.com/` from the `href` attribute on the source feed's `<link>` element:

```
<channel>
   <xsl:attribute name="rdf:about">
      <xsl:value-of>feed/link/@href<.xsl:value-of>
   </xsl:attribute>
   ...
</channel>
```

## Completing the Feed Elements

Adding a title is straightforward, again using the element `<xsl:value-of>`; you could also have used `<xsl:copy-of>` because the elements are identical in each vocabulary:

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

There's nothing you can use to fill the `<description>` element except perhaps the feed's subtitle, but that might not be a good idea because it is optional in the Atom schema.

Next you come to the `<link>` element in the RSS feed . . . but wait — you've just used the required value in the `rdf:about` attribute. Let's backtrack and create a reusable template variable `$feedurl`.

You can also refine the selection to choose the link that has the `rel` attribute set to `'self'`, because there are two link elements in the source. To do this you use a predicate inside square brackets: `"[]"`. You'll learn more about predicates in the next chapter. Change the code to look like this:

```
<channel>
   <xsl:variable name="feedurl" select="feed/link[@rel='self']/@href"/>
   <xsl:attribute name="rdf:about">
      <xsl:value-of select="$feedurl"/>
   </xsl:attribute>
   <title>
      <xsl:value-of select="feed/title"/>
   </title>
   <link><xsl:value-of select="$feedurl"/></link>
   ...
</channel>
```

## Item Listing

To create an item listing to act as a table of contents, enter the literal result element `<items>` and an RDF sequence element, `rdf:Seq`. The sequence constructor `<xsl:for-each>` will take the processor to all of the matching nodes one by one, changing the context node as it goes. You will learn more about `<xsl:for-each>` in Chapter 4.

By selecting with the XPath expression `feed//entry` (using `"//"`), you operate on *all* the entry elements in the feed. For each entry, you add an RDF list item, `rdf:li`, and set its `rdf:resource` attribute value from the `<link>` element in each individual entry:

```
<items>
   <rdf:Seq>
```

```
      <xsl:for-each select="feed//entry">
         <rdf:li>
            <xsl:attribute name="rdf:resource">
             <xsl:value-of select="link/@href"/>
            </xsl:attribute>
         </rdf:li>
      </xsl:for-each>
    </rdf:Seq>
</items>
```

### *Entry Elements*

Still in the main template, you need to loop through the entries again to create a series of complete `<item>` elements in the output:

```
<xsl:for-each select="//entry">
   <xsl:apply-templates select="."/>
</xsl:for-each>
```

In a template matching `<entry>` elements, you can handle the translation from Atom to Dublin Core. Most of the translations are straightforward. Dublin Core doesn't have an equivalent of the `<atom:updated>` element, so you use that value in `<dc:date>`. The language can be obtained from the `<content>` element's `xml:lang` attribute. Another point to note is that there can be multiple categories in entries, just as there can be multiple `<dc:subject>` elements. Therefore, you need to select the `label` attribute on the `<category>` element inside another `<xsl:for-each>` loop that creates the subject elements.

In neither of these two schemas does the order of elements matter, or the number of occurrences, so you can simply let the source sequence drive the process:

```
<xsl:template match="entry">
   <item>
    <xsl:attribute name="rdf:about">
       <xsl:value-of select="id"/>
    </xsl:attribute>
    <link>
       <xsl:value-of select="link/@href"/>
    </link>
    <dc:identifier>
       <xsl:value-of select="id"/>
    </dc: identifier >
    <dc:language>
       <xsl:value-of select="content/@xml:lang"/>
    </dc:language>
    <dc:title>
       <xsl:value-of select="title"/>
    </dc:title>
    <dc:date>
       <xsl:value-of select="published"/>
    </dc:date>
    <dc:creator>
       <xsl:value-of select="author/name"/>
    </dc:creator>
```

```
      <dc:description>
         <xsl:value-of select="summary"/>
      </dc:description>
      <dc:format>
         <xsl:value-of select="content/@type"/>
      </dc:format>
      <xsl:for-each select="category">
         <dcsubject>
         <xsl:value-of select="./@label"/>
         </dc:subject>
         </xsl:for-each>
   </item>
</xsl:template>
```

The full stylesheet is shown in Listing 1-6.

**Listing 1-6**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xpath-default-namespace="http://www.w3.org/2005/Atom">
   <xsl:output method="xml"/>
   <xsl:variable name="site">testurl</xsl:variable>
   <xsl:template match="/">
      <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
         <channel>
            <xsl:variable name="feedurl" select="feed/link[@rel='self']/@href"/>
            <xsl:attribute name="rdf:about">
               <xsl:value-of select="$feedurl"/>
            </xsl:attribute>
            <title>
               <xsl:value-of select="feed/title"/>
            </title>
            <link>
               <xsl:value-of select="$feedurl"/>
            </link>
            <items>
               <rdf:Seq>
                  <xsl:for-each select="feed//entry">
                     <rdf:li>
                        <xsl:attribute name="rdf:resource">
                           <xsl:value-of select="id"/>
                        </xsl:attribute>
                     </rdf:li>
                  </xsl:for-each>
               </rdf:Seq>
            </items>
            <xsl:for-each select="//entry">
               <xsl:apply-templates select="."/>
            </xsl:for-each>
         </channel>
```

*Continued*

**Listing 1-6:** *(continued)*

```
      </rdf:RDF>
   </xsl:template>
   <xsl:template match="entry">
      <item>
         <xsl:attribute name="rdf:about">
            <xsl:value-of select="id"/>
         </xsl:attribute>
         <link>
            <xsl:value-of select="link/@href"/>
         </link>
         <dc:language>
            <xsl:value-of select="content/@xml:lang"/>
         </dc: language >
         <dc:title>
            <xsl:value-of select="title"/>
         </dc:title>
         <dc:date>
            <xsl:value-of select="updated"/>
         </dc:date>
         <dc:creator>
            <xsl:value-of select="author/name"/>
         </dc:creator>
         <dc:description>
            <xsl:value-of select="summary"/>
         </dc:description>
         <dc:format>
            <xsl:value-of select="content/@type"/>
         </dc:format>
         <xsl:for-each select="category">
            <dc:subject>
               <xsl:value-of select="./@label"/>
            </dc:subject>
         </xsl:for-each>
      </item>
   </xsl:template>
</xsl:stylesheet>
```

## *RSS 1.0 Results*

To run the transform, add a scenario in the Oxygen IDE, using `atom.xml` as the source, and `rss_feed.xsl` as the stylesheet.

Listing 1-7 shows a matching fragment of the transformed RSS 1.0 feed.

**Listing 1-7**

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <channel rdf:about="http://feeds.oreilly.com/oreilly/xml">
      <title>O'Reilly News: XML</title>
```

```
      <link>http://feeds.oreilly.com/oreilly/xml</link>
      <items>
         <rdf:Seq>
            <rdf:li rdf:resource="tag:broadcast.oreilly.com,2008://53.34667"/>
            <rdf:li rdf:resource="tag:broadcast.oreilly.com,2008://53.34679"/>
            <rdf:li rdf:resource="tag:broadcast.oreilly.com,2008://53.34620"/>
            <rdf:li rdf:resource="tag:broadcast.oreilly.com,2008://53.34524"/>
            <rdf:li rdf:resource="tag:broadcast.oreilly.com,2008://53.34508"/>
            ...
         </rdf:Seq>
      </items>
      <item rdf:about="tag:broadcast.oreilly.com,2008://53.34667">
         <link>http://feeds.oreilly.com/~r/oreilly/xml/~3/487860677
           /xforms-a-pause-for-reflection.html</link>
         <dc:language>en</dc: language >
         <dc:title>XForms, a pause for reflection</dc:title>
         <dc:date>2008-12-17T18:05:12Z</dc:date>
         <dc:creator>Philip Fennell</dc:creator>
         <dc:description>The other day I had what could only be described as a
'Roy Scheider moment', you know the bit in the film Jaws where the camera
tracks-in whilst zooming-out at the same time. Well, whilst debugging an
XForms enabled application, the Mozilla XForms plug-in had exposed the host
document, XForms and all, as the content of the empty xf:instance. How odd.
I mean, what good is that? That's when it struck me in a Roy Scheider sort
of way; this was Reflection, the ability of a program to look at itself and
change its behaviour.</dc:description>
         <dc:format>html</dc:format>
         <dc:subject>xforms</dc:subject>
         <dc:subject>xml</dc:subject>
         <dc:subject>xrx</dc:subject>
      </item>
      ...
   </channel>
</rdf:RDF>
```

# Summary

In this chapter you created two stylesheets, the first of which handled a typical document transformation from XML to XHTML.

The second transform was a little more complex, involving two different schemas. You used an XSLT version 2.0 stylesheet with XML output, and learned about using the `<xsl:for-each>` instruction to handle repeating uniform data structures.

You used three methods to invoke your first stylesheet: the `<?xsl-stylesheet?>` processing instruction, the Oxygen IDE, and the Saxon CLI.

Along the way, you learned about the main structural XSLT elements, defining output methods, matching nodes in source documents, and selecting content to transform. You also encountered some common XPath syntax, more of which is introduced in Chapter 2.

**Key Points**

- A stylesheet processor uses built-in rules for processing by default.
- You override these rules by using specific template rules to match elements in an XML source document with XPath expressions.
- You can specify different output methods in a stylesheet — XML, XHTML, HTML and text — and define the preferred character encoding too.
- Literal result elements and attributes are often used to define output structures.
- Output element content is usually specified by selecting values in the source with XPath expressions.
- Always check for default namespace declarations in source files and set the `xpath-default-namespace` attribute on the `<xsl:stylesheet>` element to this value.