CHAPTER 1

A brief tutorial

1.1 What is XSLT (and XSL, and XPath)? 3
1.2 A simple XSLT stylesheet 8
1.3 More element and attribute manipulation 13
1.4 Summing up the tutorial 16

1.1 What is XSLT (and XSL, and XPath)?

Extensible Stylesheet Language Transformations (XSLT) is a language that lets you convert XML documents into other XML documents, into HTML documents, or into almost anything you like. When you specify a series of XSLT instructions for converting a class of XML documents, you do so by creating a "stylesheet," an XML document that uses specialized XML elements and attributes that describe the changes you want made. The definition of these specialized elements and attributes comes from the World Wide Web Consortium (W3C), the same standards body responsible for XML and HTML.

Why is XSLT necessary? XML's early users were excited about their new ability to share information, but they gradually realized that sharing this information often assumed that both sharing parties used the same schema or DTD—a lot to assume. Assembling a schema that both parties could agree on was a lot of trouble, especially if they didn't need to exchange information often. XSLT solves this problem by providing an easy, W3C-sanctioned way to convert XML documents that conform to one schema into documents that conform to others, making information much easier to pass back and forth between different systems.

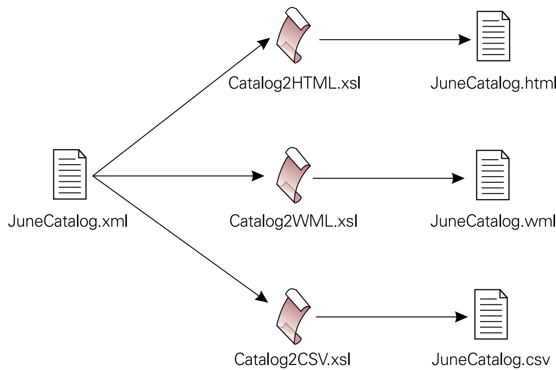

Figure 1.1
XSLT stylesheets can automate the conversion of the same input into multiple output formats.

XSLT was originally part of the Extensible Stylesheet Language (XSL). In fact, XSLT is still technically a part of XSL. The XSL specification describes XSL as a language with two parts: a language for transforming XML documents and an XML vocabulary for describing how to format document content. This vocabulary is a collection of specialized elements called "formatting objects," which specify page layout and other presentation-related details about the text marked up with these elements' tags: font family, font size, margins, line spacing, and other settings.

Because a powerful formatting language should let you rearrange your input document in addition to assigning these presentation details, the original XSL specification included specialized elements that let the stylesheet delete, rename, and reorder the input document's components. As they worked on this collection of elements, the W3C XSL Working Group saw that it could be useful for much more than converting documents into formatting object files—that it could convert XML documents into almost *anything* else. They called this transformation language XSLT and split it out into its own separate specification, although the XSL specification still said that everything in the XSLT specification was considered to be part of the XSL specification as well.

One great feature of XSLT is its ability, while processing any part of a document, to grab information from any other part of that document. The mini-language developed as part of XSLT for specifying the path through the document tree from one part to another is called "XPath." XPath lets you say things like "get the `revisionDate` attribute value of the element before the current element's `chapter` ancestor element." This ability proved so valuable that the W3C also broke XPath (see figure 1.2) out into its own specification so that other W3C specifications could incorporate this language. For example, an XLink link can use an XPath expression as part of an XPointer expression that identifies one end of a link.

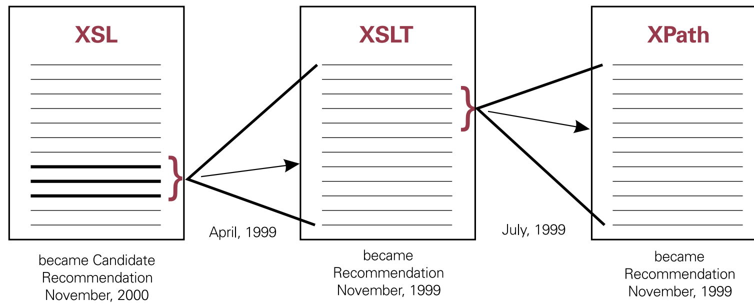

Figure 1.2 The W3C released the first Working Draft of XSL in August 1998, split XSLT out into its own Working Draft in April of 1999, then split XPath out from XSLT into its own working Draft in July 1999.

1.1.1 XSLT and alternatives

Other ways exist for transforming XML documents. These options fall into two categories:

- XML-related libraries added to general purpose programming languages such as Java, Perl, Visual Basic, Python, and C++
- languages such as Omnimark and Balise designed specifically for manipulating XML (and, typically, SGML) documents

So why use XSLT? For one thing, it's a standard. This doesn't necessarily make XSLT a good language, but it does mean that multiple vendors worked on it together, each contributed to its design, and each has committed to supporting it in their products. XSLT wasn't invented by some guy who started a company to sell it and then added and dropped features and platform support over the years as it fit the needs of the company's bigger customers. XSLT's features and platform support reflect a broad range of interests, and the wide availability of open source implementations make it easy for most programmers to put together their own customized XSLT processors with any features they may want to add.

Being a W3C standard also means that XSLT fits in with other W3C standards and that future W3C standards will also fit in with it. The phrase "standards-driven" is nearing tiresome buzzword status these days as more products take advantage of XML; unfortunately, many ignore important related W3C standards. For example, one XML transformation product has various specialized element types whose names you're not allowed to use for your own element types. If this product declared and used a namespace for their specialized element types they wouldn't need to impose such arbitrary constraints on your application development—for example, if they declared a URI and a prefix for this set of element and attribute names, and then used that prefix in the document with those names to prevent an XML parser from confusing them with other elements and attributes that may have the same name.

Another advantage of XSLT over other specialized XML transformation languages is that a series of XSLT document transformation instructions are themselves stored as an XML document. This gives XSLT implementers a big head start because they can use one of the many available XML parsers to parse their input. It also means that developers learning XSLT syntax don't need to learn a completely new syntax to write out their instructions for the XSLT processor. They must just learn new elements and attributes that perform various tasks.

1.1.2 Documents, trees, and transformations

Speaking technically, an XSLT transformation describes how to transform a source tree into a result tree. Informally, we talk about how XSLT lets you transform documents into other documents, but it's really about turning one tree in memory into another one. Why?

Most XSLT processors read a document into the source tree, perform the transformations expressed by the XSLT stylesheet to create a result tree, and write out the result tree as a file, with the net result of converting an input XML document into an output document. Nothing in the XSLT specification requires these processors to read and write disk files; by leaving files and input/output issues out of it, the spec offers more flexibility in how XSLT is used. Instead of an XML file sitting on your hard disk, the input may come from a Document Object Model (DOM) tree in memory or from any process capable of creating a source tree—even another XSLT transformation whose result tree is the source tree for this new transformation. (The DOM is a W3C standard for representing and manipulating a document as a tree in memory.) How would you tell a processor to treat the result tree of one transformation as the source tree of another? See the documentation for your XSLT processor. As I said, the XSLT spec deliberately avoids input and output issues, so that's up to the people who designed each processor.

Similarly, the processor doesn't have to write out the result tree as a disk file, but can store it as a DOM tree, pass it to a new XSLT stylesheet that treats that result tree as the source tree of a new transformation to perform, or pass it along for use by another program via some means that hasn't been invented yet. Figure 1.3 shows these relationships when using XSLT to create an HTML file from a `poem` document type.

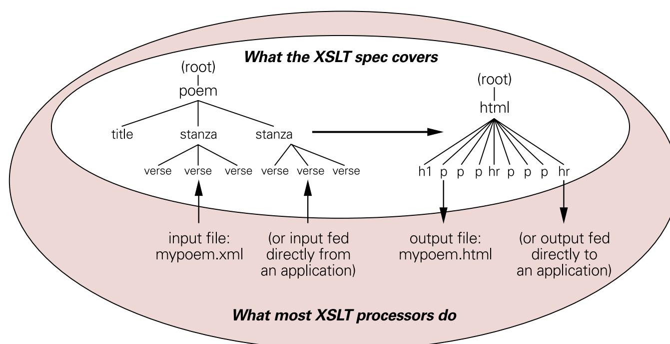

Figure 1.3 Document trees, XSLT, and XSLT processors

Dealing with an input tree instead of an input document also gives you an important advantage that XML developers get from DOM trees: at any given point in your processing, the whole document is available to you. If your program sees a word in the document's first paragraph that's defined in the glossary at the end, it can go to the glossary to pull out the term's definition. Using an event-driven model such as the Simple API for XML (SAX) to process a document instead of a tree-based model like XSLT uses, your program would process each XML element as it read the element in. While doing this, if you want to check some information near the end of your document when reading an element in the beginning, you need to create and keep track of data structures in memory, which makes your processing more complicated.

TIP When a discussion of XSLT issues talks about a source tree and a result tree, you can think of these trees as temporary representations of your input and output documents.

Not all nodes of a document tree are element nodes. A diagram of the tree that would represent this document

```
<?xml-stylesheet href="article.xsl" type="text/xsl"?>
<article>
  <!-- here is a comment -->
  <title author="bd">Sample Document</title>
  <para>My 1st paragraph.</para>
  <para>My 2nd paragraph.</para>
</article>
```

shows that there are nodes for elements, attributes, processing instructions, comments, and the text within elements. (There are also nodes for namespaces, but this document has no namespace nodes.)

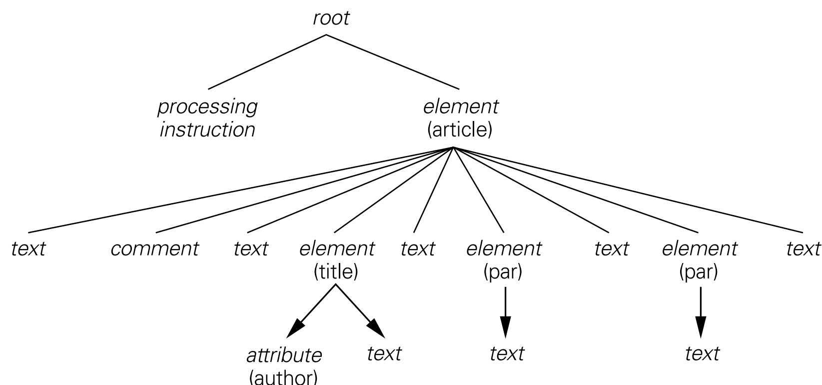

Figure 1.4 A document tree with several different node types

It's easy to match up the parts of the tree with the parts of the corresponding document, except that it might appear that the document has too many "text" nodes. The tree diagram shows text between the comment and the `title` element, and text

between the two `para` elements; where is this text in the document? You can't see this text, but it's there: it's the carriage returns that separate those components of the document. If the two `para` elements had been written as one line, like this,

```
<para>My 1st paragraph.</para><para>My 2nd paragraph.</para>
```

no text node would exist between those two elements, and you wouldn't see a text node between them in the tree diagram. (See section 6.11, "Whitespace: preserving and controlling," page 229 for more on this.)

You also might wonder why, if `article` is the root element of the document, it's not the root node of the tree. According to XSLT's view of the data (its "data model"), the root element is a child of a predefined root node of the tree (shown as a slash in the diagram) because it may have siblings. In the example above, the processing instruction is not inside the `article` element, but before it. It is therefore not a child of the `article` element, but its sibling. Representing both the processing instruction and the `article` element as the children of the tree's root node makes this possible.

1.2 A SIMPLE XSLT STYLESHEET

An XSLT transformation is specified by a well-formed XML document called a stylesheet. The key elements in a stylesheet are the specialized elements from the XSLT namespace. (A namespace is a unique name for a given set of element and attribute names. Their use is usually declared in an XML document's document element with a short nickname that the document uses as a prefix for names from that namespace.) When an XSLT processor reads one of these stylesheets, it recognizes these specialized elements and executes their instructions.

XSLT stylesheets usually assign "xsl" as the prefix for the XSLT namespace (ironically, XSL stylesheets usually use the prefix "fo" to identify their "formatting objects"), and XSLT discussions usually refer to these element types using the "xsl" prefix. This way, when something refers to the `xsl:text` or `xsl:message` elements you can assume that they mean the text and message element types from the XSLT namespace and not from somewhere else.

An XSLT stylesheet doesn't have to use "xsl" as the namespace prefix. For example, if the stylesheet below began with the namespace declaration

```
xmlns:harpo="http://www.w3.org/1999/XSL/Transform"
```

the stylesheet's XSLT elements would need names like `harpo:text` and `harpo:message` for an XSLT processor to recognize them and perform their instructions.

The following stylesheet demonstrates many common features of an XSLT stylesheet. It's a well-formed XML document with a root element of `xsl:stylesheet`. (You can also use name `xsl:transform` for your stylesheet's root element, which means the same thing to the XSLT processor.):

```
<!-- xq15.xsl: converts xq16.xml into xq17.xml -->
```

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

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

  <xsl:template match="price">
  </xsl:template>

  <!-- Copy all the other elements and attributes, and text nodes -->
  <xsl:template match="*|@*|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

1.2.1 Template rules

An XSLT stylesheet has a collection of template rules. Each template rule has a pattern that identifies the source tree nodes to which the pattern applies and a template that is added to the result tree when the XSLT processor applies that template rule to a matched node.

In a stylesheet document, the template rules that comprise a stylesheet are represented as `xsl:template` elements; the stylesheet above has three. The value of each `xsl:template` element's `match` attribute is the pattern that gets matched against source tree nodes. The element's content—that is, everything between its start- and end-tag—is the template that gets added to the result tree for each source tree node that corresponds to the match pattern. An `xsl:template` element essentially tells the XSLT processor, "as you go through the source tree, when you find a node of that tree whose name matches the value of my `match` attribute, add my contents to the result tree."

For example, the first template rule in the preceding stylesheet tells the XSLT processor what to do when it sees a `year` element node as the child of another node in the source tree. (The "year" attribute value is actually an abbreviation of "child::year.") The template rule has one element as its template to add to the result tree: a `vintage` element. This element contains an `xsl:apply-templates` element that tells the processor to apply any relevant templates to the children of the matched element node (in this case, `year`). The ultimate result of this template is the contents of the input `year` element surrounded by `vintage` tags—in effect, renaming the source tree's `year` element to a `vintage` element for the result tree. Figure 1.5 shows where the pattern and template are in one example of a template rule.

```
                                         pattern
          <xsl:template match ="year">
template {  <vintage>
              <xsl:apply-templates/>
            </vintage>
          </xsl:template>
```

Figure 1.5 The two parts of a template rule

The specialized elements in a template from the XSLT namespace are sometimes called "instructions," because they are instructions to the XSLT processor to add something to the result tree. What does this make the elements in the template that don't use the "xsl" namespace prefix, such as the `vintage` element? The stylesheet is a legal, well-formed XML document, and the `vintage` element is an element in that stylesheet. Because this element is not from the XSLT namespace, the XSLT processor will pass it along just as it is to the result tree. In XSLT, this is known as a "literal result element."

Like all template rules, the second `xsl:template` rule in the stylesheet on page 9 tells the XSLT processor "if you find a source tree node whose name matches the value of my `match` attribute, add my contents to the result tree." The string "price" is the pattern to match, but what are the template's contents? There are no contents; it's an empty element. So, when the XSLT processor sees a `price` element in the source tree, the processor will add nothing to the result tree—in effect, deleting the `price` element.

Because the stylesheet is an XML document, the template rule would have the same effect if it were written as a single-tag empty element, like this:

```
<xsl:template match="price"/>
```

XSLT has other ways to delete elements when copying a source tree to a result tree, but a template rule with no template is the simplest.

Unlike the first two template rules, the third one is not aimed at one specific element type. It has a more complex match pattern that uses some XPath abbreviations to make it a bit cryptic but powerful. The pattern matches any element, attribute, or text node, and the `xsl:copy` and `xsl:apply-templates` elements copy any element, attribute, or text node children of the selected nodes to the result tree. Actually, the pattern doesn't match *any* element—an XSLT processor uses the most specific template it can find to process each node of the source tree, so it will process any `year` and `price` elements using the stylesheet's templates designed to match those specific tree nodes. Because the processor will look for the most specific template it can find, it doesn't matter whether the applicable template is at the beginning of the stylesheet or at the end—the order of the templates in a stylesheet means nothing to an XSLT processor.

TIP If more than one `xsl:template` template rule is tied for being most appropriate for a particular source tree node, the XSLT processor may output an error message or it may just apply the last one to the node and continue.

The values of all of the `xsl:template` elements' `match` attributes are considered "patterns." Patterns are like XPath expressions that limit you to using the child and attribute axes, which still gives you a lot of power. (see chapter 2, "XPath," on page 23, for more on axes and the abbreviations used in XPath expressions and patterns.) The "year" and "price" strings are match patterns just as much as "*|@*|text()" is, even though they don't take advantage of any abbreviations or function calls.

That's the whole stylesheet. It copies a source tree to a result tree, deleting the `price` elements and renaming `year` elements to `vintage` elements. For example, the stylesheet turns this `wine` element

```
<wine grape="chardonnay">
  <product>Carneros</product>
  <year>1997</year>
  <price>10.99</price>
</wine>
```

into this:

```
<?xml version="1.0" encoding="utf-8"?>
<wine grape="chardonnay">
  <product>Carneros</product>
  <vintage>1997</vintage>

  </wine>
```

Although the `price` element was deleted, the carriage returns before and after it were not, which is why the output has a blank line where the `price` element had been in the input. This won't make a difference to any XML parser.

This is not an oversimplified example. Developers often use XSLT to copy a document with a few small changes such as the renaming of elements or the deletion of information that shouldn't be available at the document's final destination.

1.2.2 Running an XSLT processor

The XSLT specification intentionally avoids saying "here is how you apply stylesheet A to input document B in order to create output document C." This leaves plenty of flexibility for the developers who create XSLT processors. The input, output, and stylesheet filenames might be entered in a dialog box; or they might be entered at a command line; or they might be read from a file.

Many XSLT processors are designed to give a range of options when identifying the inputs and outputs. Some are programming libraries which you can invoke from a command line or call from within a program. For example, an XSLT processor supplied as a Java class library often includes instructions for using it from a Windows or Linux command line, but its real power comes from your ability to call it from your own Java code. This way, you can write a Java program that gets the input, stylesheet, and output filenames either from a dialog box that you design yourself, from a file sitting on a disk, or from another XML file that your application uses to control your production processes. Your program can then hand this information to the XSLT processor.

In fact, the input, stylesheet, and output don't even have to be files. Your program may create them in memory or read them from and write them to a process communicating with another computer. The possibilities are endless, because the input and output details are separate from the transformation.

One example of a Java library that you can use from the command line or as part of a larger Java application is Xalan (pronounced "Zalan"). This XSLT processor was

written at IBM and donated to the Apache XML project (http://xml.apache.org). To run release 2.0 of this particular XSLT processor from the command line with an input file of winelist.xml, a stylesheet of winesale.xsl, and an output file of winesale.xml, enter the following as one line (to fit on this page, the example below is split into two lines):

```
java org.apache.xalan.xslt.Process -in winelist.xml
  -xsl winesale.xsl -out winesale.xml
```

(This assumes that the appropriate Java libraries and system paths have been set up; directions come with each Java processor.) For examples of how to run other XSLT processors, see appendix A, "XSLT quick reference" on page 259.

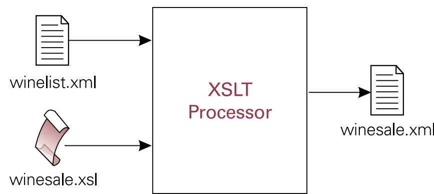

Figure 1.6
The XSLT processor reads an input XML file and an XSLT stylesheet and outputs another file based on the stylesheet's instructions.

1.2.3 An empty stylesheet

What would an XSLT processor do with a stylesheet that contained no template rules? In other words, what effect would an empty stylesheet, such as the following, have on an input document?

```
<!-- xq21.xsl: converts xq22.xml into xq23.xml -->

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

XSLT has several built-in default templates that tell the XSLT processor to output the text content (or, in XML terms, the PCDATA) of the elements, leaving out the attributes and markup. For example, the processor would turn this

```
<winelist date="20010626">

<wine grape="chardonnay">
  <product>Carneros</product>
  <year>1997</year>
  <price>10.99</price>
</wine>
</winelist>
```

into this:

```
<?xml version="1.0" encoding="UTF-8"?>

  Carneros
  1997
  10.99
```

TIP An XSLT processor's default behavior adds an XML declaration as well. This can be overridden with the optional `xsl:output` element if you want to create HTML, plain text, or other non-XML output.

The built-in templates also tell the XSLT processor to apply any relevant templates to the children of the elements being processed. (Otherwise, when the `wine-list` element in the above example gets processed, no reason would exist for the XSLT processor to do anything with the `wine` element.) If the stylesheet has no template for one of the children, the most appropriate template for that child element may be the same built-in template that the processor applied to the child's parent. The XSLT processor will do the same thing to the child: add any character data nodes to the result tree and apply the most appropriate templates to any grandchildren elements.

1.3 MORE ELEMENT AND ATTRIBUTE MANIPULATION

In our first stylesheet, we saw that an `xsl:apply-templates` element with no attributes tells the XSLT processor to apply any relevant templates to all the matched node's children. By using this element type's `select` attribute, you can be pickier about exactly which children of a node should be processed and in what order.

For example, this stylesheet

```
<!-- xq25.xsl: converts xq26.xml into xq27.xml -->
<xsl:template match="wine">
  <wine>
    <price><xsl:apply-templates select="price"/></price>
    <product><xsl:apply-templates select="product"/></product>
  </wine>
</xsl:template>
```

will turn this XML element

```
<wine grape="chardonnay">
  <product>Carneros</product>
  <year>1997</year>
  <price>10.99</price>
</wine>
```

into this:

```
<wine>
  <price>10.99</price>
  <product>Carneros</product>
</wine>
```

The stylesheet performs two important operations on this element:

- It moves the `price` element before the `product` element.
- It deletes the `year` element.

The first technique that we saw for deleting an element—using an empty template for that element type—is often simpler than adding `xsl:apply-templates` elements for each of an element's children (except the ones you want to delete). If you're

reordering the children anyway, as with the preceding example, omitting an `xsl:apply-templates` element for the elements in question can be an easier way to delete them.

1.3.1 Manipulating attributes

We've seen how to delete and rename elements. How do you delete and rename attributes? For example, how would you delete the following `wine` element's `price` attribute and rename its `year` attribute to `vintage`?

```
<wine price="10.99" year="1997">Carneros</wine>
```

We want the result to look like this:

```
<wine vintage="1997">Carneros</wine>
```

(Because an XML declaration is optional, it won't make any difference if that shows up as well.) The first template rule in the following stylesheet makes both of these changes:

```
<!-- xq30.xsl: converts xq28.xml into xq29.xml -->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">

<xsl:template match="wine">
<wine vintage="{@year}"> <!-- price attribute omitted -->
    <xsl:apply-templates/>
  </wine>
</xsl:template>

<!-- Copy all the other source tree nodes. -->
<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates
         select="@*|node()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Deleting the `price` attribute was easy: the template just left the attribute out of the `wine` start-tag in the template. To rename the `year` attribute to `vintage`, the `year` start-tag includes the attribute specification `vintage="{@year}"`. The part between the quotation marks says "put the value of the source tree `wine` element's `year` attribute here." The `@` character is shorthand for the XPath notation that means "get the value of the attribute with this name," and the curly braces tell the XSLT processor that the expression they contain is an attribute value template—not a literal string to appear in the result tree exactly as shown, but an expression to be evaluated and replaced with the result of the evaluation. If this attribute specification had said `vintage="{2+2}"`, the XSLT processor would have added `vintage="4"` to the result tree. In the example, the processor understands the meaning of `@` and plugs in the appropriate attribute value between the quotation marks on the result tree.

1.3.2 Attribute value templates

You can do a lot with attribute value templates. For example, these templates make converting elements to attributes simple. The `@` character makes it easy to insert an attribute value where that value will be used as element content in the result. To demonstrate this, let's convert the `grape` attribute in the following to a `product` subelement of the `wine` element. While we're at it, we'll convert the `year` subelement to a `vintage` attribute.

```
<wine grape="Chardonnay">
  <product>Carneros</product>
  <year>1997</year>
  <price>10.99</price>
</wine>
```

The result should look like this:

```
<wine vintage="1997">
  <product>Carneros</product>
  <category>Chardonnay</category>
  <price>10.99</price>
</wine>
```

The following template converts the `grape` attribute into a `category` subelement by using the `@` character, and the message uses the `xsl:value-of` element to put each `grape` attribute value between a pair of `category` start- and end-tags. (As with attribute value templates, an XSLT processor takes what the `xsl:value-of` element hands it in its `select` attribute, evaluates it, and adds the result to the appropriate place on the result tree.)

```
<!-- xq33.xsl: converts xq31.xml into xq32.xml. -->
<xsl:template match="wine">
  <wine vintage="{year}">
    <product><xsl:apply-templates select="product"/></product>
    <category><xsl:value-of select="@grape"/></category>
    <price><xsl:apply-templates select="price"/></price>
  </wine>
</xsl:template>
```

To convert an element to an attribute, the same template uses an attribute value template—the curly braces around "year"—to put the source tree `wine` element's `year` subelement after `vintage=` in the result tree's `wine` element.

Another great trick is selective processing of elements based on an attribute value. Because an XSLT processor applies the most specific template it can find in the stylesheet for each source tree node, it will apply the first template in the following stylesheet for each `wine` element that has a value of "Cabernet" in its `grape` attribute, and the second for all the other `wine` elements. (The `[@grape='Cabernet']` part that specifies this is a special part of a match pattern or XPath expression called a "predicate.") The first template copies the element, while the second doesn't. The output will therefore only have wines with "Cabernet" as their `grape` value.

```
<!-- xq34.xsl -->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">

<xsl:template match="wine[@grape='Cabernet']">
  <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

<xsl:template match="wine"/>

<xsl:template match="@*|node()|processing-instruction()|comment()">
  <xsl:copy>
    <xsl:apply-templates
         select="@*|node()|processing-instruction()|comment()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

How useful is this? Think about the importance of a database system's ability to extract subsets of data based on certain criteria. The creation of such customized reports can be the main reason for developing a database in the first place. The ability to generate customized publications and reports from your XML documents can give you similar advantages in a system that uses those documents, because the more you can re-use the document components in different permutations, the more value the documents have.

For related information, see

- chapter 2, "XPath," on page 23 for more on the use of expressions in square brackets ("predicates") to filter out a subset of the nodes that you want
- section 3.5, "Converting elements to attributes for the result tree," page 55
- section 3.8, "Deleting elements from the result tree," page 63
- section 3.14, "Converting attributes to elements," page 79

1.4 SUMMING UP THE TUTORIAL

So far, this brief tour has only given you a taste of XSL's capabilities—yet we've already covered the features that will let you do four-fifths of your XSLT work! We've shown you how to:

- delete elements
- rename elements
- reorder elements
- delete attributes
- rename attributes
- convert elements to attributes
- convert attributes to elements
- process elements based on an attribute's value

These are the most basic changes that you'll want to make when converting XML documents that conform to one schema or DTD into documents that conform to another. If data is shared between two organizations that designed their data structures independently, those organizations probably have many types of information in common—after all, that's why they're sharing it. Yet, it's also likely that they assigned different names to similar information, or ordered their information differently, or stored extra information that the other organization doesn't need (or hasn't paid for!). XSLT makes most of these conversions painless and quick.

Before moving on, let's review what the XSLT processor is doing now that you've seen it in action a few times. Imagine that an XSLT processor has just started processing the children of the `chapter` element in the following document,

```
<book><title>Paradise Lost</title>
  <chapter><title>The Whiteness of the Whale</title>
    <para>He lights, if it were Land that ever burned</para>
    <para>With solid, as the Lake with liquid fire</para>
  </chapter>
<chapter><title>The Castaway</title>
    <para>Nine times the Space that measures Day and Night</para>
    <para>To mortal men, he with his horrid crew</para>
  </chapter>
</book>
```

and it's using a stylesheet with the following two template rules to process it:

```
<!-- xq37.xsl -->

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

  <xsl:template match="chapter/title">
    Chapter title: <xsl:apply-templates/>
  </xsl:template>
```

(The match pattern "chapter/title" in the second template element shows that this template rule is for the `title` elements that are children of `chapter` elements. The first is for all the other `title` elements.) The diagram in figure 1.7 shows the steps that take place. The chapter title's content is the only text node shown in the source tree; the rest are omitted to simplify the diagram.

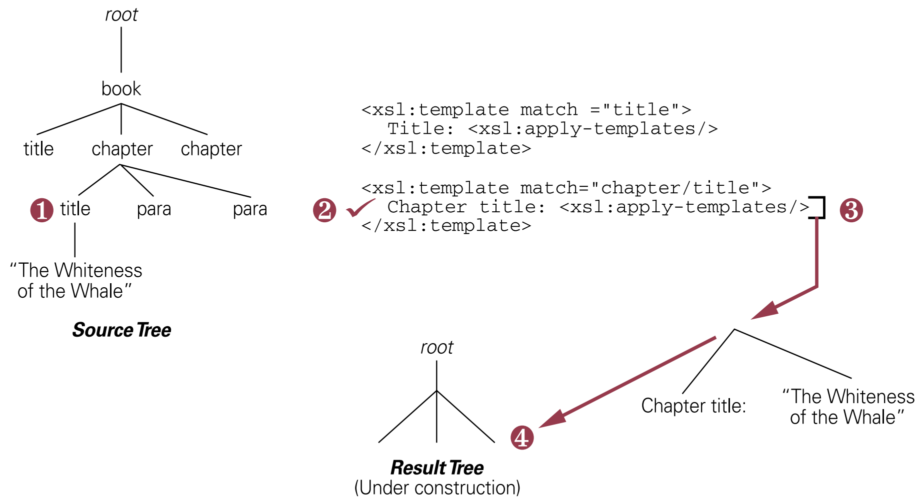

Figure 1.7 How the XSLT processor handles an element node

1. It finds the first child of the `chapter` element, a `title` element node.
2. It checks the stylesheet for a matching template rule. It finds two, and picks the most specific one it can find.
3. Once it's found the best template rule for the node, it gets the template rule's template to add to the result tree.
4. It adds the template, which consists of a text node ("Chapter title:") and the result of processing the template's `xsl:apply-templates` element to the title element's lone text node child: the string "The Whiteness of the Whale."

Of course, XSLT can do much more than what we've seen so far. If you'd like more background on key XSLT techniques before you dive into stylesheet development, the following sections of part 2, "XSLT user's guide: How do I work with...," on page 21 of this book are good candidates for "Advanced Beginner" topics:

- chapter 2, "XPath," on page 23
- chapter 3, "Elements and attributes," on page 47
- section 3.6, "Copying elements to the result tree," page 57
- section 5.1, "Control statements," page 110
- section 6.1, "HTML and XSLT," page 187
- section 6.5, "Non-XML output," page 202
- section 6.6, "Numbering, automatic," page 205
- section 6.9, "Valid XML output: including DOCTYPE declarations," page 225

If your stylesheet absolutely depends on these potentially unavailable elements, the `xsl:message` element can do more than just output a message about the problem: it can abort the processing of the source document with its `terminate` attribute. (See section 5.4.1, “Runtime messages, aborting processor execution,” page 134, for more on the use of `xsl:message`.)