# XML Programming: Web Applications and Web Services with JSP and ASP

ALEXANDER NAKHIMOVSKY
TOM MYERS

Apress™

ISBN (pbk): 1-59059-003-1

Printed and bound in the United States of America 12345678910



Technical Reviewer: Slava Paperno
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Erin Mulligan
Copy Editor: Tom Gillen, Gillen Editorial, Inc.
Production Editor: Kari Brooks
Compositor: Impressions Book and Journal Services, Inc.
Artist: Kurt Krames
Indexer: Carol Burbo
Cover Designer: Kurt Krames
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.



The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 2

# Well-Formed Documents and Namespaces

With basic definitions and examples behind us, we can move on to a detailed discussion of the specifications. In this chapter, we concentrate on documents without DTDs because they have a simpler structure. Although occasionally mentioned in this chapter, DTDs and other approaches to validation (such as XML Schema and RELAX NG) will be introduced in Chapter 3.

In outline, this chapter proceeds as follows:

- HTML vs. XHTML
- XHTML modularization and XHTML Basic
- well-formed XML documents
- names and namespaces
- global attributes and XLink
- namespace URI and RDDL (XHTML Basic + XLink)

We will start with a comparison of HTML and XHTML.

## HTML, XML, and XHTML

HTML is by far the most familiar markup language. We will review its main features in comparison with XHTML to emphasize, one last time, the following basic facts.

- HTML is a specific language defined in the SGML framework.
- XML is not a language but a framework for defining languages.
- XML is a revision of SGML.

The main difference between XML languages and HTML and other SGML languages is that XML documents can be parsed without a DTD, whereas SGML documents (whether in HTML or any other SGML language) can be parsed only with the help of the DTD. This is because, in SGML languages, the end tag of an element can frequently be omitted even if the element is not empty: in HTML, you don't have to close off your s with a . For HTML empty elements, the end tag is always optional: nobody puts <br></br> in a Web page.

## HTML vs. XHTML

Listing 2-1 provides an example of a perfectly grammatical HTML document (paralist.htm); it uses CSS within a style attribute to specify the font properties for the first  element:

*Listing 2-1. An HTML Document*

```
<html>
<head><title>HTML Example</title></head>
<body bgcolor="#ffffef">
  <h1>Heading</h1>
  <p style="color:maroon;font-size:2em">a paragraph with <em>italics</em>
 followed by a list
  <ul>
    <li>item one
    <li>item two
  </ul>
  <p>Another paragraph with a line break <br> in the middle.
</body>
</html>
```

What would the element tree for this document look like? Figure 2-1 shows one possibility.

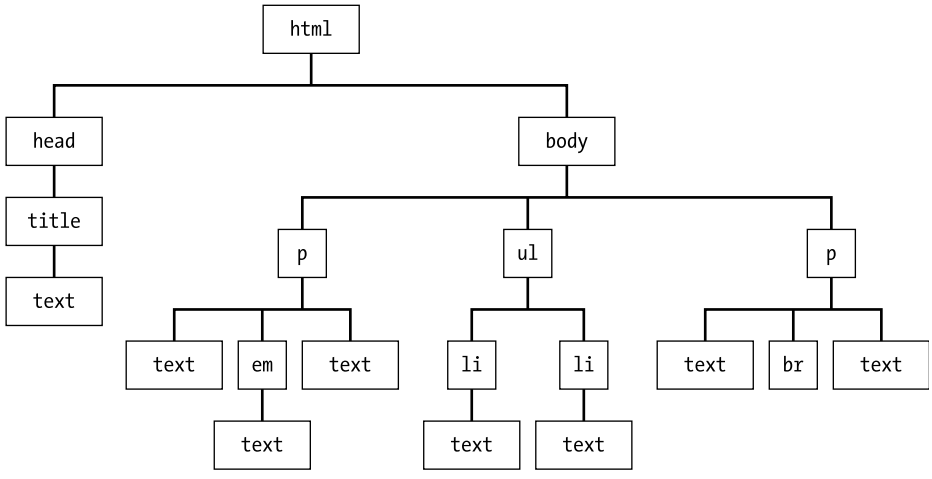


*Figure 2-1. Element tree of an HTML document*

Is this the only possible tree? Note that the <p> elements don't have an end tag, so it would be consistent with the markup to make the <ul> element a child of the first <p>. In fact, we could even make the second <p> a child of the first. Is there a "correct" structure among these possibilities? The question is not academic because the page uses CSS, and a CSS style defined on an element is inherited by the element's children. If <ul> is a child of <p>, its font will be large and maroon; otherwise, it will be small and black. Obviously, we can't leave this decision to the browser's parser: we need a rule. There is, indeed, such a rule; in fact, for every HTML element, there is a rule that stipulates which elements it can contain. The rule for <p> lists many possible children, but <ul> is not among them. As Figure 2-2 shows, the browser knows and obeys the rule.

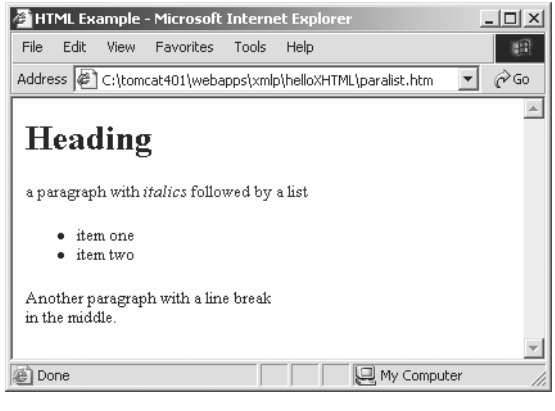


*Figure 2-2. HTML document in the browser*

The rules of HTML are stated in the HTML DTD, which is part of the W3C HTML recommendation. (The latest and final version is 4.01.) HTML DTDs are very similar to XML DTDs, and we are not going to discuss their minor differences. The essential point is that, to process an HTML document and build its tree, the browser's parser needs to know the grammar of HTML (the HTML DTD). The corresponding XML document terminates each element with an end tag and can be parsed without a grammar. (Empty elements consist of a single tag terminated with the "/>" sequence, as in `<br />`.) Modified in this way, HTML becomes XHTML, officially described by W3C as "a Reformulation of HTML 4 in XML 1.0".

To parse the document shown in Listing 2-2, the browser wouldn't need the grammar anymore.

*Listing 2-2. An XHTML Document*

```
<html>
<head><title>HTML Example</title></head>
<body bgcolor="#ffffef">
  <h1>Heading</h1>
  <p style="color:maroon;font-size:2em">a paragraph with <em>italics</em>
 followed by a list</p>
  <ul>
    <li>item one</li>
    <li>item two</li>
  </ul>
  <p>Another paragraph with a line break <br/> in the middle.</p>
</body>
</html>
```

## HTML As a Language

In Chapter 1, we defined a language as consisting of a vocabulary and a grammar. Is HTML a language in the sense of this general definition? It most certainly is because it has a fixed vocabulary of element and attribute names whose usage is controlled by specific grammar rules. Is HTML a formal language or an interpreted one? This question is a little trickier: HTML itself does not define the meaning of its expressions, but they are always given a meaning in a stylesheet (either the default stylesheet that comes with the browser or a custom stylesheet supplied by the user). For instance, the meaning of the `<h1>` tag in the following line is something like: "display the text 'Heading' in a large font, bold face" (for example, 24 point Times New Roman).

```
<h1>Heading </h1>
```

As an HTML author who is also well versed in CSS, you can change that meaning by redefining the style as in the following line of code.

```
<h1 style="font-family:algerian;color:red;font-size:4em">Heading</h1>
```

Figure 2-3 shows the resulting document, paralist2.htm, in the browser.


*Figure 2-3. HTML Document with a different stylesheet*

However, your creativity has limits: all possible meanings are about how the text within the tags is to be displayed in the browser. This is what HTML expressions within the body are mostly about. The important point here is that the meaning of HTML tags and attributes is determined by a stylesheet and a specific application—the browser—that interprets it.

By contrast, XML is not a language. It doesn't have a specific markup vocabulary or a specific grammar. Instead, it has a DTD language for defining vocabularies and grammars. In addition, XML makes no assumptions at all about the meaning of its markup languages. They can be intended for any application, and the application alone determines the meaning of the markup.

## SGML/HTML = XML/XHTML

In a sense, it is unfair to compare HTML to XML: it's like comparing a cookie to a cookie cutter. Or, to change the metaphor, XML is not HTML's sibling but more like its (youthful and vigorous) uncle. Figure 2-4 illustrates the relationship.

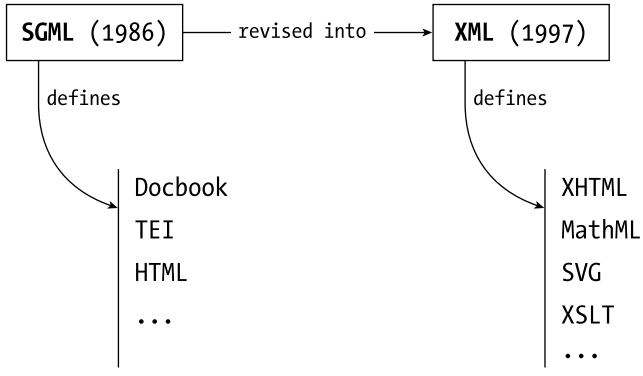


*Figure 2-4. SGML and XML*

A good comparison would be between HTML and XHTML. More precisely, we want to compare an HTML document with the corresponding XHTML document; Listing 2-1 and 2-2 give us material for comparison. The most important difference between them, from which more-specific differences follow, is that an XHTML document can be checked for well-formedness and parsed in the absence of a grammar. We have expressed this difference by saying that elements must form a tree in all SGML languages as well as in all XML languages, but, in XML languages, additionally, the document's markup must explicitly show the tree structure.

HTML and XHTML have several other, less important differences, such as XML is case sensitive and XHTML tags are defined to be in lowercase. (A good place to look for a summary of all such differences is Section 4 of the XHTML spec that lists its differences with HTML 4.0, `www.w3.org/TR/xhtml1/#diffs`.) However, most remaining differences between XHTML and HTML pages result from the behavior of the parser that processes them rather than from any differences in their grammar and underlying framework.

## *Parsers with Attitude*

HTML parsers are famously tolerant of ungrammatical Web pages: they will display a page without complaint even if it lacks the `<html>` and `<head>` elements, has unquoted attribute values, and shows other violations of HTML rules. For instance, the page shown here (paralistbad.htm) will display correctly in both IE and Netscape browsers, possibly with differences in whitespace:

```
<p style=color:maroon;font-size:2em>a paragraph followed by a list
  <li>item one
  <li>item two
```

As you can imagine, this means a lot of work both for parser writers, and for parsers themselves. All SGML parsers are complex, but HTML parsers are even more so because they try to anticipate and correct users' mistakes. XML is more economical on all counts. (One of XML's design goals was that it shall be easy to write programs that process XML documents.) XML parsers, especially nonvalidating ones, are small and relatively easy to write both because XML is simpler than SGML and because XML's attitude to syntax errors is totally negative.

Error handling by XML parsers is not only strict, it is also uniform. The W3C XML recommendation, in a section on conformant processors, precisely specifies what a processor must do in response to different kinds of errors. There is, as we mentioned in Chapter 1, a test suite that is designed to test the parser's compliance with the XML specification, especially in its error handling. (See `http://oasis-open.org/committees/xml-conformance/xml-test-suite.shtml`.)

There are two main reasons for this strict attitude. First, XML parsers are frequently used to mediate between computer applications or components within an application. XML data is often generated by a program and consumed by another program that performs computations on it. In this sort of configuration, ill-formed data must be inadmissible. (In particular, it must be inadmissible to feed the same XML data into two different browsers and see one of them succeed and the other fail to parse it.) Second, XML was developed from the start in anticipation of small mobile devices. A parser sitting in a cell phone, wristwatch, or remote sensor cannot afford the megabytes of memory that are needed to anticipate and accommodate grammatical error.

**NOTE** *Dave Raggett, a longtime staff member at W3C, wrote a remarkable program called Tidy (*`http://www.w3.org/People/Raggett/tidy/`*). It performs several functions on HTML documents: fixes grammatical errors, points out deprecated features, and converts the HTML document to XHTML. We will use Tidy in Chapter 7.*

## Why XHTML?

Why use XHTML instead of HTML? The main reason is that the entire array of XML technologies becomes available to you. If you want your Web page to be produced by an XSLT program, you have to make your template HTML material conformant to XML rules because an XSLT program is an XML document, and if you enter, for example, `<br>` instead of `<br />`, the parser will object.

Since 1999, HTML has been in effect mothballed while XHTML has been an area of active development. A quick look at the list of W3C recommendations shows *XHTML 1.1—Module-Based XHTML* (May 2001) and *XHTML Basic for*

*Small Devices* (December 2000). If you look inside *MathML 2.0* (February 2001), you will see that "it is designed to be used as a *module* in documents marked up with the XHTML family of markup languages" (Appendix A2). Outside W3C, XHTML has been used to define RDDL (Resource Directory Description Language), a promising new idea that we will introduce in the section on namespaces.

> **NOTE** *All new XHTML-based or XHTML-related languages rely on XHTML modularization, a framework for dividing the large vocabulary of HTML into small modules that can be independently reused in various ways. We present XHTML modularization in Chapter 3.*

## XML Documents Without a DTD

XML documents have logical and physical structure. Logically, a document consists of elements, attributes, and other less important items, such as comments. Some types of items are processed away in parsing, and the rest are represented as a tree of nodes. The *Infoset* recommendation regulates what gets preserved in the tree.

Physically, a document is a unit of storage (such as a file or a string) for character data and markup that can include other such units by reference. A generic name for "units of storage" is "entity." Most entities have to be declared in a DTD before they can be used, but two groups of entities can appear in documents without a DTD. We explain about entities and CDATA sections before presenting a complete summary and outline of an XML document without a DTD.

### *Character Entities and Five Predeclared Entities*

Character entities represent individual characters by their Unicode numbers, either decimal or hexadecimal. They are used for markup characters and characters that are difficult to enter from the keyboard. To refer to a character entity within a document, place it between an ampersand and a semicolon. For instance, the copyright symbol (©), whose Unicode number is x00A9, can be entered into your document in three ways:

```
&#169; &#xA9; &#x00A9;
```

The ampersand itself can also be entered as a character reference (&), but it has a special name in addition to the numeric code. Such special names are

predeclared for five characters. (See Table 2-1.) To refer to a character by its special predeclared name, place it between an ampersand and a semicolon.

*Table 2-1. Five Predeclared Entities*

| CHARACTER | ENTITY NAME | REFERENCE | DECIMAL CODE | HEX CODE |
|---|---|---|---|---|
| & | amp | &amp; | &#38; | &#x0026; |
| < | lt | &lt; | &#60; | &#x003C; |
| > | gt | &gt; | &#62; | &#x003E; |
| " | quot | &quot; | &#34; | &#x0022; |
| ' | apos | &apos; | &#39; | &#x0027; |

## CDATA Sections

If you have a section in your document that contains a great number of markup characters (such as XML source or Java code), you may want to enter the entire section as a CDATA section that is not parsed by the processor. The syntax is as follows,

```
<![CDATA["<" & ">" are "angle brackets"]]>
```

Within XHTML, it is common to put the contents of `<script>` and `<style>` elements in CDATA sections.

The markup of CDATA sections is removed in parsing. The boundaries of CDATA sections leave no trace in the XPath tree model but may be preserved in DOM. They are not preserved in the document's infoset, and the next version of DOM will conform to the *Infoset* recommendation and align itself with XPath.

## Summary, Outline, and EBNF Productions

In summary, documents without a DTD can contain the following kinds of material,

- Declaration: Optional but highly recommended, as in

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

- Elements and attributes: This is the informational core of the document.
- Comments:

```
<!-- this is a comment -->
```

- Processing instructions (PIs) (in XML, they are rarely used except to provide a stylesheet reference):

```
<?xml-stylesheet href="style0.css" type="text/css"?>
```

- Character entity references.
- References to five predeclared entities: `lt`, `gt`, `quot`, `apos`, and `amp`.
- CDATA sections: A document without a DTD follows this outline.

```
<![CDATA[if(a<b && b<c) return;]]>.
```

- XML declaration: Nothing can precede it, not even a comment or whitespace.
- Miscellaneous optional material: Comments and PIs, whitespace as desired. (In a document with a DTD, the DTD or a DTD reference would appear here.)
- The start tag of the root element.
- All other material, well formed.
- The end tag of the root element.

It is actually easier to say this in EBNF (Extended Backus-Naur Form) than in English. EBNF is a formal notation for describing the syntax of programming languages, and it is also used in many XML specifications. In *XML 1.0*, it is used to define the well-formedness and validity rules of XML documents. The entire *XML 1.0* boils down to 89 EBNF rules, or "productions" as they are called. It is useful to be able to read EBNF productions because they pack a lot of information in very few lines of text. Here is a small sample, numbered as in *XML 1.0*, with brief comments. As you read the rules, remember that the characters ?, +, and * indicate the number of repetitions: ? stands for 0 or 1, + stands for 1 or more, and * stands for 0 or more.

Production 1 specifies the structure of a document:

```
[1]  document ::=  prolog element Misc*
```

This says that a document is composed of a prolog, followed by a single element (the root of the element tree), followed by optional miscellaneous material.

Prolog is composed of optional elements, as follows: an optional XML declaration, followed by Misc*, followed by an optional DTD, again with Misc* thrown in:

```
[22]  prolog ::=  XMLDecl? Misc* (doctypedecl Misc*)?
```

Misc* is any sequence of whitespace, comments, and PIs. Here's the Misc production (S stands for whitespace, and the vertical bar means "OR"):

```
[27]  Misc ::=  Comment | PI |  S
[3]    S   ::=  (#x20 | #x9 | #xD | #xA)+
```

We will show more EBNF in the namespace section later in the chapter.

## A "Kitchen Sink" Example

Listing 2-3 shows everything you can find in a DTD-less document (ksink.xml). Note that, to display escape sequences in the browser, we have to escape the escapes: for instance, to display `<` we have to enter `&amp;lt;`. Because CDATA sections cannot be nested, we have to enter its closing character sequence outside the section itself, using entity references. The document is followed by a modest CSS stylesheet (ksink.css, Listing 2-4) and a screenshot (Figure 2-5).

*Listing 2-3. All You Can Find in an XML Document Without a DTD*

```
<?xml version='1.0' encoding='utf-8'?>
<!-- The line above is the XML declaration. Nothing can precede it,
       not even comments or whitespace.
      The line below is a PI. It associates a stylesheet with the document.
      In XML, PIs are rarely used for any other purpose.
-->
<?xml-stylesheet href="ksink.css" type="text/css"?>
<root_elt>
<h1>Document without a DTD</h1>
<non_empty_element>
This is element content, parsed character data (PCDATA). <br />
To insert a markup character here, such as <, you have to use a reference
```

```
to a <em>pre-declared general entity</em>, &amp;amp;lt;
or a <em>character entity</em>, &amp;amp;#60;. <br />
If you have many such characters, you can put them all into a
<![CDATA[CDATA section: <[CDATA[ (a<b && b>c) ]]>. ]]&gt;.
CDATA sections cannot be nested. <br />
You can also use character entities to insert characters that are not easy
to enter from the keyboard, such as the copyright character &#xA9;.
(In this case, we used the character's hexadecimal code, &#38;#38;#xA9;.) <br />
</non_empty_element>
</root_elt>
```

*Listing 2-4. A Minimalist Stylesheet*

```
rootElement, h1, non_empty_element, br {
  display: block; margin-bottom: .6em;
}
h1 {font-weight:bold;font-size:large;text-align:center;}
em {font-weight:bold;font-style:italic}
```

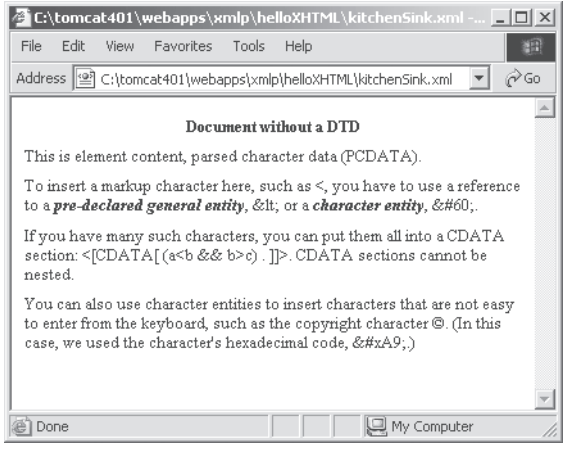


*Figure 2-5. XML document without a DTD*

## Names and Namespaces

A markup language is a vocabulary of names: the names of elements and attributes. The names of attributes must be unique within an element, but different elements can have attributes of the same name. We can describe this by saying that attributes of different elements belong in different namespaces. The notion of a namespace is very familiar from programming: an object in C++ or Java is a namespace for its variables and methods; a package in Java is a namespace for

its class names; a database is a namespace for the names of its tables; and a database table is a namespace for the names of its fields. The ability to partition the names in your program or database into different namespaces is essential for preventing name conflicts.

The initial *XML 1.0* specification has no provisions for partitioning element names within a document into different namespaces. This could potentially result in name conflicts when two different vocabularies are merged in a single document. The danger is real for XML languages—such as XSLT, SVG, XML Schema, or JSP—that are designed for use with a great many other languages. It would be reckless to leave the vocabularies of such languages unprotected. *XML Namespaces* (1998) was primarily developed to protect the vocabularies of such widely used XML languages.

## Namespaces and Prefixes

A common way to create a globally unique name is by forming a pair that consists of a namespace prefix and a local name. The namespace prefix uniquely identifies the namespace, the local name must be unique within that namespace, and the combination of the two creates a globally unique name. This is how Java classes and packages operate: the name of a package is globally unique (or at least has a good chance of being so), class names are unique within a package, and a fully qualified class name consists of the package name as a prefix, followed by a period and the class name. Reversed URLs are often used as package names: a good deal of Sun's software is in the com.sun package or its subpackages, for instance:

```
com.sun.xml.parser.Resolver res = new com.sun.xml.parser.Resolver();
```

Here, `Resolver` is a local name, `com.sun.xml.parser` is both the name of the package and a unique namespace prefix, and the combination of the two is a fully qualified, globally unique name of Sun's Resolver class.

The designers of *XML Namespaces* had a well-known source of globally unique names ready at hand: the URL, or its generalization, the URI. It was a natural decision to make it a source of unique namespace prefixes, so that a unique element name would consist of a URI prefix to identify the namespace and a local name that is unique within that namespace. Conceptually, if our company URL is `http://www.n-topus.com`, and we want to put our Address element in a protected namespace, we would say that its fully qualified globally unique name is something like `{http://www.n-topus.com/elemnames}Address`.

The problem is that this name, as written, is not a legal XML name, and it's also extremely long. The solution of *XML Namespaces* is to use a two-step procedure for establishing a namespace. In the first step, a unique namespace URI is

declared and mapped to a prefix that contains only legal characters. In the scope of that declaration, the prefix serves as a proxy for the namespace URI. Here is an example that you have already seen as Listing 1-15:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/">
<!-- The rest of the program goes here -->
</xsl:template>
</xsl:stylesheet>
```

The second line of this code contains a namespace declaration that maps the URI to a prefix. (The prefix for this particular namespace is usually `xsl`, but any prefix would do as long as it is associated with the right URI.) Syntactically, the declaration is an attribute. The name of the attribute consists of a reserved sequence of characters, `xmlns`, followed by a colon and the prefix to which the namespace URI is mapped; the value of the attribute is the namespace URI. The scope of the declaration includes the element whose start tag contains that declaration and all the descendants of that element (unless there is another declaration with more local scope, as discussed shortly).

### *The* XML 1.0 *Perspective vs. the* XML Namespaces *Perspective*

A namespace declaration is an attribute only from the "naive" perspective of the pre-namespace *XML 1.0* recommendation. From the perspective of namespace-aware specifications (such as XPath, DOM, or the infoset), this is not an attribute at all but a namespace declaration: it is not an attribute node in the DOM or XPath tree, and it is not an "attribute information item" in the document's infoset.

A related fact of some importance is that the following two documents are the same from the *XML Namespaces* perspective but different from the *XML 1.0* perspective:

```
<a:doc xmlns:a="http://a.b.c.d.com"><a:p>some text</a:p></a:doc>
<b:doc xmlns:b="http://a.b.c.d.com"><b:p>some text</b:p></b:doc>
```

Because DTDs originate with *XML 1.0*, they are namespace unaware and use the *XML 1.0* perspective. This creates validation problems discussed in the next chapter.

### The Syntax of Names and EBNF Productions

The same colon character (:) that separates `xmlns` from the prefix being declared is used to separate the prefix from the local name. Here again, there was a change from *XML 1.0* to *XML Namespaces*. In *XML 1.0*, a colon could appear anywhere in the name except as the first character, any number of times. In the *XML Namespaces* recommendation, a colon can appear at most once, as a separator between the namespace prefix and the local name. In the following EBNF productions, NCName is a "No-Colon Name" that does not contain a colon.

```
[6]  QName ::=  (Prefix ':')? LocalPart
[7]  Prefix ::=  NCName
[8]  LocalPart ::=  NCName
```

Most of today's parsers will reject as non-well-formed any documents with names containing more than one colon.

## Scope of Declarations

Namespace declarations are inherited: the scope of a namespace declaration is the element to which it is attached, together with all its descendants, except those that declare their own namespaces. Everywhere within that scope, the names qualified by the prefix belong to the declared namespace. (In the previous XSLT example, these tag names are `stylesheet`, `output`, and `template`.) Conversely, if an element's name has a prefix but no namespace declaration, the parser will go up its line of ancestors until a declaration for that prefix is found. If no such declaration is found, the parser must return an error.

Because namespace declarations are inherited, it is possible (and recommended) to declare all namespaces on the root element, as in the example shown in Listing 2-5.

*Listing 2-5. The Root Element of a Document with Three Namespaces*

```
<citeDB
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xlink="http://www.w3.org/1999/xlink"
>
```

The three namespaces declared on this element are all well known. One of them is for XLink elements and attributes that will be the subject of much discussion and an extended example later in this chapter. The other two are for

Resource Description Framework (RDF) and Dublin Core Metadata that are discussed in detail in Chapter 7.

The same prefix can be mapped to different namespaces within the same document. It is therefore possible to shadow one declaration with another one in an embedded element. We have never seen a convincing case for using this functionality, but here is a contrived example (the quotes, however, are real):

```
<prfx:bk xmlns:prfx="http://chairman.mao.sayings.org.red">
  <prfx:saying>
The world is progressing, the future is bright
 and no one can change this general trend of history.
  </prfx:saying>
  <prfx:bk xmlns:prfx="http://chairman.greenspan.sayings.org.green">
    <prfx:saying>
History provides excellent lessons for banking institutions
with regard to appropriate pricing, underwriting, and diversification.
    </prfx:saying>
  </prfx:bk>
</prfx:bk>
```

The first `prfx:saying` element in this example is in the `mao.sayings.org.red` namespace, but the second such element is in the more local `greenspan.sayings.org.green` namespace. We can prove this by writing an XSLT that will extract the inherited namespace URI from both of those elements and color the text content red or green accordingly. In the process, we will see how namespace URIs are accessed in the XPath tree (not as attributes).

## *Namespaces in XPath and XSLT*

Listing 2-6 is the stylesheet that colors Mao's sayings red and Greenspan's green. It is done in the so-called "push" style of multiple independent templates. We will discuss and use it extensively in Chapter 5. Our interest here is in the XPath functions that have to do with namespaces. (Both XPath and DOM have functions that extract the local name, the qualified name (with the prefix), and the namespace URI from a given element or attribute node in the tree.) Some of those functions are used in the highlighted part of the second template. Outside the second template, there may be details that have not yet been explained: please suspend your curiosity until Chapter 5.

*Listing 2-6. Namespace Handling in XSLT and XPath*

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0"
>
<xsl:output method="html"/>
<xsl:template match="/">
  <html><head><title>Two Chairmen's Wisdom</title></head><body>
    <xsl:apply-templates />
  </body></html>
</xsl:template>
<xsl:template match="*[local-name()='saying']">
  <xsl:variable name="color">
    <xsl:choose>
      <xsl:when test="contains(namespace-uri(),'red')">red</xsl:when>
      <xsl:when test="contains(namespace-uri(),'green')">green</xsl:when>
      <xsl:otherwise>blue</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <div style="{concat('color:',$color)}">
    <xsl:apply-templates/>
  </div>
</xsl:template>
</xsl:stylesheet>
```

The match attribute of that template matches all element nodes (that's what the asterisk stands for) such that their local name is `saying`. As you can see, XPath has a `local-name()` function and (a few lines farther down) a `namespace-uri()` function. Both return strings that you can work with using XPath string functions, such as `contains()`. To compute the color name, we use `xsl:choose` (the XSLT equivalent of the `switch` statement in C and derived languages).

If all namespace URIs contained the color name right after the substring `sayings.org`, we could replace the `xsl:choose` expression with code that extracts the color name from the namespace URI:

```
<xsl:variable name="color"
  select="substring-after(namespace-uri(),'.sayings.org.')"
 >
```

This would generate different colors from different data without any changes in the stylesheet.

## Default Namespaces

You can save yourself a little typing by creating a default namespace that is not mapped to a prefix. The syntax is as follows.

```
<doc xmlns="http://a.b.c.d.com"><p>some text</p></doc>
```

From the namespace perspective, this element is equivalent to

```
<a:doc xmlns:a="http://a.b.c.d.com"><a:p>some text</a:p></a:doc>
```

Put differently, prefix-less element names within the scope of a default namespace declaration belong to the declared namespace. This is very different from prefix-less names that belong to no namespace at all. For any namespace-aware program, the two preceding single-line documents are completely different from

```
<doc><p>some text</p></doc>
```

Note that prefix-less attribute names remain in no namespace. The only way for an attribute to be in a namespace is by having a prefix that is mapped to that namespace. (See the next section.)

Default namespaces are useful when you have some XML data that you want to cut and paste into a new XML context. If name conflicts are a possible concern, you can create a default namespace for the data to be pasted in:

```
<insertFromData xmlns="http://www.n-topus.com/ns/temp">
  <!-- inserted prefixless data goes here -->
</insertFromData>
```

To override a default namespace declaration (that is, put an element in its scope into no namespace), you have to use a special form of the namespace declaration:

```
<doc xmlns="http://a.b.c.d.com">
<p>this element is in the "http://a.b.c.d.com" namespace</p>
<nons xmlns="">this element is in no namespace</nons>
</doc>
```

This facility may also be useful in a cut-and-paste situation.

## Namespaces and Attributes

Attributes and elements are treated differently by *XML Namespaces* because attributes have a natural namespace (their owner element) and don't need extra protection. It makes no sense to put attributes in the same namespace as their owner element instead of simply leaving them in no namespace at all. Attributes are like local variables in a procedure that don't need fully qualified names to prevent name conflicts. In XSLT, the attributes of XSLT elements (such as `match` or `select`) remain local:

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

It does make sense to put attributes in a namespace of their own that is different from the containing element's namespace, especially if the attributes come from an XML language of wide application. Later in this chapter, you will see examples of XLink attributes that are used to describe links within XML data. Their (reserved) names are quite common: `type`, `title`, `href`, and so on. To protect them from conflict with unrelated attributes of the same name, they are placed into a namespace and are always used with a prefix. (It can be any prefix, but traditionally `xlink:` is used.) For examples, see the XLink section that is coming up shortly.

Attributes that are placed in a separate namespace become, in effect, "global attributes" that can be added to any element in any document to provide specific functionality. Two important groups of global attributes are those with the fixed `xml:` prefix and XLink attributes that usually are mapped to the `xlink:` prefix. We introduce them in the remainder of this section.

### Attributes with the `xml:` Prefix

The `xml:` prefix is reserved by W3C and cannot be used by anybody else. It is declared in the `XML Namespaces` recommendation and bound to `www.w3.org/XML/1998/namespace`. Several attributes always appear with the `xml:` prefix and have a fixed meaning. We mention two: `xml:lang` and `xml:base`.

The `xml:lang` attribute can be added to any element to specify the language of that element's content. The value of the attribute is either a two-letter language code as defined in ISO 639, *Codes for the Representation of Names of Languages*, or a language identifier registered with IANA (Internet Assigned Numbers Authority), or user defined. The two-letter codes cover the most familiar languages, and some of them can be extended to indicate a regional variant: fr-ca, fr-be, and fr-ch stand for the French dialects of Canada, Belgium, and Switzerland, respectively. IANA-registered names start with "i-": i-navajo. User-defined names start with "x-": x-esperanto. For an in-depth treatment of

language identifiers in XML, see Robin Cover's Web page at `http://xml.coverpages.org/languageIdentifiers.html`.

A recent newcomer to the `xml:` family is `xml:base`. Its purpose is to define a base URI for resolving relative URIs in parts of XML documents. The value of an `xml:base` attribute must be an absolute URI; its scope is the element on which it is defined and its descendants, unless a descendant defines its own. The most common use for `xml:base` will be within `xlink:href` attributes, to resolve relative links to images, applets, form-processing programs, style sheets, and other external resources.

*XLink (XML Linking Language)* is a recent (June 2001) W3C recommendation, released together with *XBase*. It defines several global attributes in the XLink namespace. The namespace is usually mapped to the `xlink:` prefix, which we will use throughout the rest of the book. Although small, the XML Linking Language is quite intricate and conceptually complex because it overlays a graph structure over a collection of XML and non-XML resources. It is also a very important member of the XML family of specifications.

## XLink Attributes and XLink Graphs

The purpose of XLink is to establish connections between and among resources. A resource, as usual, is anything that can be addressed with a URI. It doesn't have to be an XML resource; if it is, it does not have to be a complete document because the URI can be extended with a fragment identifier to select a document part. A very common kind of fragment identifier is an XPath expression, as you will see in a moment.

A structure that consists of nodes connected by arcs is called a *graph*. XLink is about directed graphs of resources. A graph is called *directed* if its arcs have a direction from source to target. In the case of XLink graphs, both source and target are resources. The nodes and arcs of an XLink graph can also have labels attached to them.

The most important XLink attribute is `xlink:type`. It can have several possible values, including `simple` and `extended`. An element that has an `xlink:type` attribute with one of those two values is called a `link element`. A link element can have any tag name whatsoever: it's the `xlink:type` attribute that defines it as a link element.

Link elements can be simple or extended, depending on the value of `xlink:type`. It is important to understand from the start that a link element describes an entire graph of resources, not just the arcs. The word *link* is used in its general meaning, meaning a connection or as a synonym for *arc*, but a link element is an XML element that has an `xlink:type` attribute whose value is `simple` or `extended`. From the XLink perspective, it describes a graph.

Let's take a look at a simple link element, in comparison with the HTML <a> element.

## A Simple XLink "Link Element" and an HTML Hyperlink

If you think about it, the HTML <a> element describes a directed labeled arc between two resources: the source of the arc is the <a> element itself, the target of the arc is specified in the href attribute, and the label is the content of the <a> element:

```
<a href="cs303/classList.htm">cs303 class list</a>
```

The closest XLink analog would look like this:

```
<somePrefix:someElement
  xlink:type="simple"
  xlink:href="classList.xml"
  xlink:title="cs303 class list"
  xlink:actuate="onRequest">
  <!-- wait for user request to traverse the arc -->
 xlink:show="replace"
 <!-- upon traversal, replace the current document with the target -->
>
  Any text with any non-xlink markup.
</somePrefix:someElement>
```

The two links show the usual contrast between the fixed appearance and behavior of HTML and the flexibility of XML. The name of the HTML link element is fixed. Links created by that element are for human users. Their labels are automatically highlighted; even if they are not blue and underlined, which is the most common way to style them in Web pages, they have to be visible to perform their function, which is to provide a hypertext jump from the source to the target. (Such links are called *hypertext links*.) By default, the jump replaces the source document with the target in the browser window.

By contrast, an XML simple link element can have any tag name; its content doesn't have to be highlighted; it may be intended for human users or programs; and the behavior of the link is up to the application that processes it. XLink provides "behavioral attributes" that supply hints about the intended behavior, as indicated by code comments, but these are just hints and they can be ignored. Even if they are followed, the hints say nothing about blue underline, the raised cursor finger, or a single click.

For all the differences between the HTML hypertext link and the XLink simple link element, they have two important similarities:

- They describe a graph that involves just two resources and a single arc.
- The arc is outbound: its source is local (the link element itself, not specified by a URI) and its target is remote (specified by a URI).

XLink extended link elements do not have either of these restrictions. An extended link element can

- describe a complex graph containing multiple sources, targets, and arcs.
- describe arcs that are inbound (local target, remote source) or third-party (both source and target are remote).

In other words, you can create links from and between resources for which you don't have a write permission.

The natural question is: "What does one do with those links?" Hypertext links are very intuitive: you click on them, and they take you there. If there are multiple targets, though, where does the click take you? The answer, as always, is that it's up to the application to decide what to do with different kinds of links, and as of today nobody has yet come up with a killer app for multiple-target or third-party links. All we have is a flexible and powerful language to describe them.

## *Extended Link Element and Its XLink Graph*

To specify a graph, an extended link needs to specify resources and arcs between them. These are defined by children elements of the extended link element. The names and namespaces of those elements are completely unconstrained, but they must all have an `xlink:type` attribute to indicate their role in the graph. The possible values of `xlink:type` for children elements of an extended link element are as follows:

- `locator` for "locator elements" that specify remote resources
- `resource` for "resource elements" that specify local resources
- `arc` for "arc elements" that specify arcs

The notion of a local resource is a tricky one. A local resource is an XML element that satisfies three conditions:

- It is a child of an extended link element or the link element itself.
- It has an `xlink:type` attribute whose value is `resource`.
- It does not have an `xlink:href` attribute.

A remote resource, by contrast, is a resource specified by a URI that is the value of an `xlink:href` attribute of a locator element. Put differently, for locator elements an `xlink:href` attribute is required, whereas for resource elements an `xlink:href` attribute is not allowed. The same exact XML element can be either a local or a remote resource depending on whether it is specified by a URI reference or by its position with respect to the link element.

Both locator elements and resource elements must have an `xlink:label` attribute that contains an identifying label. The label has to be an NCName; that is, it cannot have a prefix and a colon. Arc elements (the children of an extended link element that have `xlink:type="arc"`) refer to their source and target resources using those labels. In summary, the XML structure of an extended link element looks as shown below. In the outline, `elt` stands for an arbitrary tag name, `uri` for an arbitrary URI, and `lbl` for an arbitrary label. There have to be two or more resources and at least one arc. The order of children is not constrained by XLink itself but can be constrained by a DTD or some other type of schema or grammar.

```
<elt xlink:type="extended" . . . ><!-- possibly other attributes -->
  <!-- external resources / locator elements -->
  <elt xlink:type="locator"
       xlink:href="uri"
       xlink:label="lbl"/>
  <!-- local resources / resource elements -->
  <elt xlink:type="resource"
       xlink:label="lbl"/>
  <!-- arcs / arc element -->
  <elt xlink:type="arc" xlink:from="lbl" xlink:to="lbl" />
</elt>
```

This outline contains only structural markup that defines the graph. XLink also has behavioral and semantic markup.

### *Behavioral and Semantic Markup*

You have already seen behavioral attributes, `xlink:show` and `xlink:actuate`. Their values are largely self-explanatory. The specification spells out in some detail

what the conformant applications should do in the presence of behavioral markup. There are no required actions, only recommended ones:

```
show: "new", "replace", "embed", "other", and "none"
actuate: "onLoad", "onRequest", "other", and "none"
```

Semantic attributes are `xlink:title`, `xlink:role`, and `xlink:arcrole`. The `title` attribute can appear on any element to provide a brief description. The `role` attribute can appear on extended, simple, locator, and resource elements. The `arcrole` attribute can appear on simple and arc elements. Both `xlink:role` and `xlink:arcrole` must be absolute URI, perhaps with a fragment identifier attached. Their intended use is rather vaguely defined, but see the following RDDL section for an example.

In addition to the `xlink:title` attribute, extended link-, locator-, and arc-elements can have any number of children elements of type `title`, that is, elements with `xlink:type="title"`. Their purpose is to provide a more structured and extensive annotation or a series of annotations, perhaps in different languages. (The value of an attribute can only be CDATA.)

### *Summary of XLink Attributes*

Table 2-2, quoted from the XLink specification, shows all XLink attributes and how they coexist with different values of `xlink:type`, listed as column headers. Now that you have seen them all, this table may actually be useful. (*R* stands for *required*, *O* stands for *optional*, and the *N/A* stands for *not applicable*.)

*Table 2-2. Summary of XLink Attributes*

| ATTRIBUTE | SIMPLE | EXTENDED | LOCATOR | ARC | RESOURCE | TITLE |
|---|---|---|---|---|---|---|
| `type` | R | R | R | R | R | R |
| `href` | O | N/A | R | N/A | N/A | N/A |
| `role` | O | O | O | N/A | O | N/A |
| `arcrole` | O | N/A | N/A | O | N/A | N/A |
| `title` | O | O | O | O | O | N/A |
| `show` | O | N/A | N/A | O | N/A | N/A |
| `actuate` | O | N/A | N/A | O | N/A | N/A |
| `label` | N/A | N/A | O | N/A | O | N/A |
| `from` | N/A | N/A | N/A | O | N/A | N/A |
| `to` | N/A | N/A | N/A | O | N/A | N/A |

## An XLink Example

The following document shows an extended link element with third-party arcs. For our examples, we chose a Bible commentary. The Bible itself, as you know, contains many more-or-less obvious cross-references within itself, from later to earlier books. It also contains elaborate rhetorical and narrative patterns that take time and study to discover, and Bible commentaries point out those cross-references and patterns. These commentaries also contain cross-references to themselves and other commentaries, making it all a very tangled graph indeed.

We will do a simple example of a narrative pattern within the Bible itself, primarily because we couldn't find any XML-marked Bible commentaries. As XML source, we use ot.xml, the King James Version marked up by Jon Bosak and distributed as part of his Religious Works package (`www.ibiblio.org/bosak/`). The package contains tstmt.dtd, which defines the markup structure. Both the XML source and the DTD can be downloaded from this book's Web site, but the following XPath expressions (within XLinks) should be self-explanatory without the DTD: the root element is `tstmt`, which contains `bookcoll` elements (book collections), which contain `book` elements, which contain `chapter` elements, which contain `v` elements (verses). To select verses 5 through 11, we say `[position()>4 and position()<12]`. In an XML file, we encode ">" and "<" as `>` and `<`.

We assume two namespaces, one for XLink mapped to `xlink:` and the other for the text of the commentary, mapped to `c:`. We do a single extended link element, `c:comm`, but you should think of it as a child of the root element, `c:commentary`, that contains an arbitrary number of `c:comm` extended link elements. Such collections of third-party links are called *linkbases*, and we can say that we show (and process) a minimal linkbase.

### *An Extended Link*

Our example (see Listing 2-7) comes from the story of Joseph (Genesis 37-49) that contains three dream sequences, each consisting of two dreams. We will describe this by an extended link element that has three locator elements (one for each dream sequence) and six arcs, connecting each dream sequence to the other two.

Because the code is quite repetitious, we show only one locator element and one arc element. All namespaces are declared on the root element. The value of the `xlink:href` attribute is a very long string that, on the printed page, has to be broken into three lines.

*Listing 2-7. An Extended Link Example from dreams.xml*

```
<c:comm xlink:type="extended" xlink:title="Dreams in the story of Joseph">
  <c:txt>There are three dream sequences in the story of Joseph,
               Joseph's dreams,  Prisoners' dreams and Pharaoh's dreams.</c:txt>
<c:node type="narrativePattern"
  xlink:type="locator"
  xlink:title="Joseph's dreams"
  xlink:label="Jdreams"
  xlink:href=
    "http://localhost:8080/xmlp/dat/jb/ot.xml
      #xpointer(/tstmt/bookcoll/book[bktshort='Genesis']
                /chapter[37]/v[position()&gt;4 and position()&lt;12])">
Joseph tells his brothers about his dreams. The dreams predict that the brothers
will bow to Joseph and become subservient to him. The brothers are not happy.
</c:node>
<!-- two more nodes like this -->
<c:crossRef xlink:type="arc" from="Jdreams" to="Pdreams"  </crossref>
<!-- five more arcs like this -->
</c:comm>
```

The new material in this example is the fragment identifier that follows the URI in the multiline `xlink:href` attribute. It is a single string that is again broken into three lines on the printed page:

```
"http://localhost:8080/xmlp/dat/jb/ot.xml
  #xpointer(/tstmt/bookcoll/book[bktshort='Genesis']
    /chapter[37]/v[position()>4 and position()<12])"
```

The fragment identifier consists of two parts: a URI and an XPointer expression. (XPointers are defined in `www.w3c.org/tr/xptr`.) The XPointer expression is the function `xpointer()` whose argument is an XPath expression. Together, the URI and the XPath expression uniquely identify a node set on the Internet. In our example, the expression says

> *at the top-level, pick the "tstmt" element; within that look for a "bookcoll" which contains a "book" whose "bktshort" (book-title-short) subelement is "Genesis"; from this book take the 37th "chapter" element; within this chapter take every verse whose position is > 4 and < 12.*

Note that predicates that constrain the set of nodes selected by an XPath or XPointer expression appear in square brackets after the tag name, as in

`[bktshort='Genesis']` or `v[position()&gt;4 and position()&lt;12]`. Because this is an XML document, angle brackets are encoded as `&gt;` and `&lt;`.

### *Note on XPointers*

XPointer expressions are mostly XPath expressions, with two additions:

- Expressions that refer to a specific point between two characters in the text content of the document, and
- Expressions that refer to character ranges within the text content of the document.

All of the XPointers in this chapter are also XPaths.

It is anticipated that XPointers will typically be used in XLink elements to indicate link endpoints. When used that way, the XPointer expression is given as an argument to the `xpointer()` function, as in our example.

## *An XLink Application*

Our first XLink application does not do much; it does not even output any blue underlined links to click on. However, it does process an extended link in a general way, extracting all the information it contains, including XML data referenced by XPointers. It is a Web application that uses Java Server Pages (JSPs) and XSLTs and runs in Tomcat. The XSLTs it uses introduce some useful general-purpose techniques.

The application assumes that there is a source of XML data that is not subject to change (the King James Bible). A separate "linkbase" file contains extended links that are cross-references within the source. The entry page to the application is the familiar xx.jsp that expects, as you recall, two arguments: an XML file and an XSLT to apply to it. In this application, the XML file is our linkbase and the XSLT is a data-specific program, dreams.xsl. It incorporates, by inclusion, a data independent, general purpose XLink application called linkTransform.xsl. This is a fairly complex program that is partially discussed in the next section; a complete explanation will have to wait until Chapter 5. We use it in this chapter to provide an interesting example of XLink processing. You can experiment with dreams.xsl and the linkbase even if you skip the next section altogether.

Schematically, the application consist of components shown in Figure 2-6. The components above the broken line are completely explained in this chapter and can be experimented with. The components underneath the broken line are briefly explained in the next section and completely explained in Chapter 5.

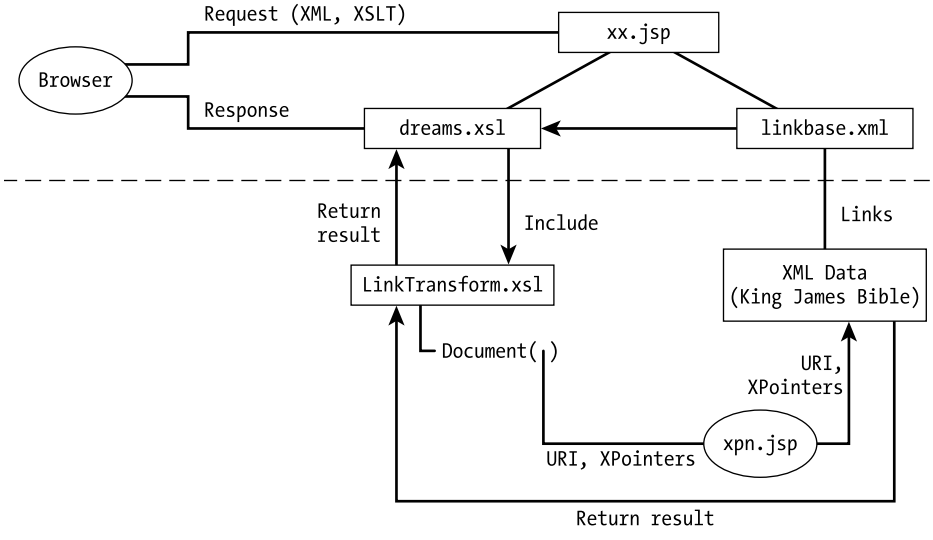


*Figure 2-6. An XLink application with extended links*

The data-specific dreams.xsl is quite readable:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="html"/>
<xsl:include href="linkTransform.xsl"/><!-- include another file -->
<xsl:template match="v">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Remember that our task is to understand the syntax and semantics of XLink and the general outlines of the XSLT that processes it. The complete details of XSLT are quite involved, but the general outlines are clear. In dreams.xsl, just note that the code expects to receive some material from linkTransform.xsl that contains v elements in it. (This is the import of <xsl:template match="v">) All it does is outputs those elements as XHTML <p> elements.

## *linkTransform.xsl*

All the real work is done in linkTransform.xsl, which knows nothing about the data source. In outline, this is what's happening. The linkTransform.xsl file extracts information from the linkbase, including the `xlink:href` attributes. These attributes, as we just discussed, contain the data source URI and an

XPointer. After some fairly elaborate footwork (which is discussed in detail in the next section), the XSLT sends both the URI and the XPointer to another JSP application. That other JSP, xpn.jsp, returns the data referenced by the XPointers. The data consists of v elements that contain Biblical verses. Eventually, that data ends up in dreams.xsl that converts verses to paragraphs.

For example, with Tomcat running and all the files in the right places, this URL (shown broken over two lines)

```
http://localhost:8080/xmlp/xx.jsp?
     xmlUri=helloXLink/dreams.xml&xslUri=helloXLink//dreams.xsl
```

results in the screenshot shown in Figure 2-7.

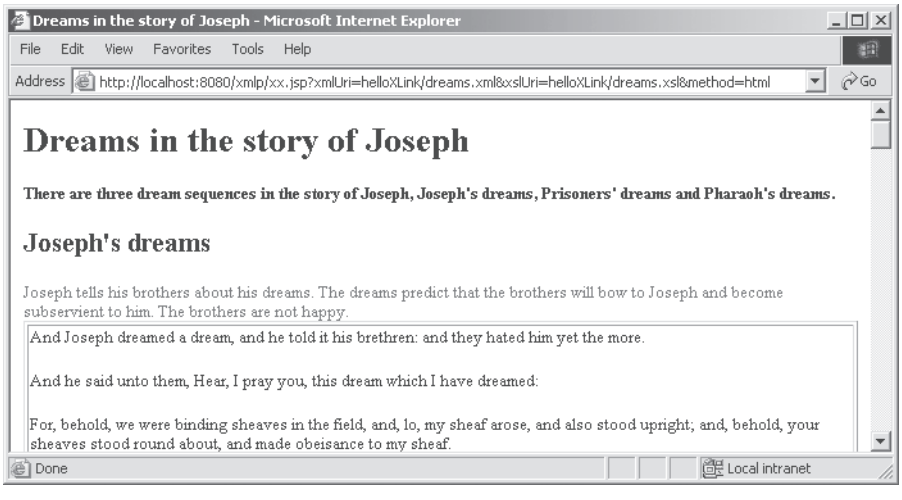


*Figure 2-7. XLink processed with XSLT (Joseph's Dreams)*

## *The Code of linkTransform.xsl*

This XSLT can be classified as advanced: it goes beyond what we have so far covered and can be skipped on first reading. We don't discuss all of it but we do address two details that concern calling xpn.jsp and sending parameters to it. First, we introduce the XPath `document()` function. Its main use is to include another XML document for processing. So, for instance, if you have an XML document additionalData.xml and you want to store its XPath tree in a variable in your XSLT program, you would say

```
<xsl:variable name="moreData" select="document('additionalData.xml')" />
```

The `document()` function takes any URI as argument, including URIs that connect to JSPs and are followed by a query string. In other words, the `document()` function is perfectly happy to connect to a JSP application and send it some arguments. For instance, at some point in linkTransform.xsl, we say

```
<table border="1"><tr><td>
 <xsl:apply-templates select="document(concat($uri,$qstring))/*" />
  </td></tr></table>
```

This results in the following sequence of events.

- The values of two variables, `uri` and `qstring`, are concatenated.
- The result is an argument to a document() call that in turn is a call on xpn.jsp.
- The call returns some material—a single element called `<nodeset>`—from the XML data source using an XPath expression. (Both are specified in the `qstring`.)

All children elements of the extracted element are processed by whatever templates will match them in the containing stylesheet that knows about the structure of the data. In our case, it's dreams.xsl that outputs the Bible text in the bordered box in the screenshot.

The two concatenated variables are declared as follows.

```
<xsl:variable name="uri" select="'http://localhost:8080/xmlp/xpn.jsp?'"/>
<xsl:variable name="qstring" select="concat('x=',$x-enc,'&amp;p=',$p-enc)"/>
```

The first declaration is straightforward, but note the single quotes within double quotes: the double quotes enclose the XML attribute, and the single quotes enclose the constant string that becomes the value of the variable. Without single quotes, the XSLT processor would try to evaluate the string as an XPath expression.

The `qstring` declaration concatenates some constant strings and variable values. Its purpose is to produce the following query string, shown broken over two lines**.**

```
x=ot.xml&p=/tstmt/bookcoll/book[bktshort='Genesis']/chapter[37]
            /v[position()>4 and position()<12]
```

Before this string can be sent to the server, it has to be appropriately encoded. What does *appropriately encoded* mean? As you know, one can add

parameters to a URI, separated by a question mark, as you saw in Chapter 1 with xx.jsp (the URI is divided into two lines):

```
http://localhost:8080/tryXSL/xx.jsp?
    xmlUri=helloXSL/hello.xml&xslUri=helloXSL/hello.xsl
```

The technical name for the part of the URI that follows the question mark is *query string*, because it is intended for sending queries to Web applications. The problem with the query string is that it can contain only characters that are allowed in a URI. The forbidden characters (including, for instance, square brackets) have to be URL-encoded. The URL encoding of a character consists of a "%" followed by two hexadecimal digits showing the UTF-8 code for the character. Because these are single-byte characters, UTF-8 codes are the same as ASCII codes: space comes out as `%20`, left bracket as `%5B`, and right bracket as `%5D`.

### *XSLT/XPath Extension Functions*

Instead of doing URL encoding by hand, we call a function to do that. XPath itself does not have such a function but most XSLT processors have a facility for adding your own extension functions. This facility will be standardized in the next release of XSLT, but even now it works pretty much the same way in different implementations. Follow these two steps if you want to call a static method of a Java class:

1. Declare a namespace which, for Xalan, is `xmlns:java="http://xml.apache.org/xslt/java"`. Note that this is a specialized use of namespaces, completely unrelated to their use in general-purpose XML documents, as opposed to XSLT programs.

2. Within XPath, put the namespace prefix before your function call:

   ```
   java:java.net.URLEncoder.encode().
   ```

Because we use a built-in Java function, we don't have to write any code. The function is a public static method of the `java.net.URLEncoder` class.

This is how we extract the XPointers from dreams.xml and encode them:

```
<xsl:variable name="h" select="@xlink:href"/>
<xsl:variable name="x" select="substring-before($h,'#xpointer')"/>
<xsl:variable name="p" select="substring-after($h,'#xpointer')"/>
<xsl:variable name="x-enc" select="java:java.net.URLEncoder.encode($x)"/>
<xsl:variable name="p-enc" select="java:java.net.URLEncoder.encode($p)"/>
```

The preceding section shows how `x-enc` and `p-enc` are used to construct `qstring`. The entire linkTransform.xsl file is shown in Listing 2-8, with the part we have discussed highlighted. We've also highlighted the beginning of every template, of which there are several. The stylesheet uses the `xsl:apply-templates` construct extensively. We will discuss the construct and the programming style based on it in Chapter 5. In the meantime, you can experiment by simply changing the XML file that contains your XLinks.

*Listing 2-8. XSLT Stylesheet to Process XLinks*

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:c="http://n-topus.com/ns/jdreams"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                xmlns:java="http://xml.apache.org/xslt/java"
                version="1.0">
<xsl:output method="html"/>
<xsl:template match="/"><-- match root, apply templates to its children -->
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="c:comm">
  <xsl:variable name="title" select="@xlink:title"/>
  <html> <head> <title>  <xsl:value-of select="$title"/> </title>
  </head>
    <body >
      <h1><xsl:value-of select="$title"/></h1>
        <xsl:apply-templates/>
      </body>
    </html>
 </xsl:template>
<xsl:template match="c:txt">
  <p><strong><xsl:apply-templates/></strong></p>
</xsl:template>
<xsl:template match="c:node">
  <h2><xsl:value-of select="@xlink:title"/></h2>
  <p><span style="color:green">
       <xsl:apply-templates/>
    </span>
  <xsl:variable name="h" select="@xlink:href"/>
  <xsl:variable name="x" select="substring-before($h,'#xpointer')"/>
  <xsl:variable name="p" select="substring-after($h,'#xpointer')"/>
  <xsl:variable name="x-enc" select="java:java.net.URLEncoder.encode($x)"/>
  <xsl:variable name="p-enc" select="java:java.net.URLEncoder.encode($p)"/>
  <xsl:variable name="qstring" select="concat('x=',$x-enc,'&amp;p=',$p-enc)"/>
  <xsl:variable name="uri" select="'http://localhost:8080/xmlp/xpn.jsp?'"/>
```

```
  <table border="1"><tr><td>
   <xsl:apply-templates select="document(concat($uri,$qstring))/*" />
    </td></tr></table></p>
</xsl:template>
</xsl:stylesheet>
```

The last three lines before the closing tag basically say "get the XPointer-referenced stuff out of the data source, apply whatever templates apply to it in the containing XSLT, and put the result in a box on screen." In our minimal stylesheet, there is only one simple template, but the document-specific processing can be as complex as it needs be. This is the signature feature of the `apply-templates` processing model.

### *The JSP Page*

The remaining software module is the JSP that uses XPath to extract a node-set. Its operation consists of these steps:

1. Parse the data source.

2. Obtain a DOM object.

3. Call a `selectNodeList()` method to extract the list of nodes that satisfies the XPath/XPointer condition. The method takes two arguments: the root of the subtree to apply XPath to and the XPath expression to use.

Because the data source remains unchanged, an obvious optimization is to parse it once and cache the resulting DOM tree in an object that persists from one application call to another. Listing 2-9, xpn.jsp, illustrates this idea. In JSP, a simple way to obtain a persistent object is to make it "application scope." Similar functionality is available in ASP.

*Listing 2-9. JSP Page with XPath*

```
%@ page errorPage="error.jsp"
import="org.apache.xalan.xslt.*,org.apache.xerces.parsers.*,
org.w3c.dom.*,org.apache.xml.serialize.*"
%><jsp:useBean id="cache" class="java.util.Hashtable" scope="application"/><%
  String xPath=request.getParameter("p");
  String xml=request.getParameter("x");
  if(0>xml.indexOf(":"))  xml="file:///"+application.getRealPath(xml);
  Document doc=(Document) cache.get(xml);
  if(doc==null){ // has not been parsed yet
```

```
    DOMParser parser=new DOMParser();
    parser.parse(xml);
    doc=parser.getDocument();
    cache.put(xml,doc);
  }
  NodeList ndList=org.apache.xpath.XPathAPI.
            selectNodeList(doc.getDocumentElement(),xPath);
  if(ndList==null){ // format and display an error message, see the code file
  } else {
  OutputFormat outputFormat=new OutputFormat();
  outputFormat.setOmitXMLDeclaration(true);
  // this XML data is a fragment, no declaration,
  // can be spliced into another document
  XMLSerializer ser= new XMLSerializer(out,outputFormat);
%><nodeList><%
  for(int i=0;i<ndList.getLength();i++)
     ser.asDOMSerializer().serialize((Element)ndList.item(i));
%></nodeList><%
}
%>
```

A similar Web application can be made using ASP: both JScript and VBScript have all the relevant functionality, including querying DOM trees with XPath expressions. This feature will become standard in *DOM Level 3*, currently under development.

## Namespace Controversies and RDDL

Unlike other early XML recommendations (*XML 1.0*, *DOM Level 1*, *XSLT*, and *XPath*), *XML Namespaces* provoked a lot of discussion and argument. Until *XML Schema* came along, it was easily the most controversial of XML specifications. It also provoked a good deal of confusion because XML namespaces were very different from the familiar programming language namespaces, sometimes in counterintuitive ways.

### *Namespace Explanations*

Fortunately, in response to confusions and controversy, many illustrious people wrote perceptively about namespaces, including Tim Bray, David Megginson, and James Clark. All their contributions are online, together with an excellent

FAQ by Ronald Bourret, who also wrote a separate piece exploding namespace myths. Here is a list of resources from which our own summary is synthesized.

- *XML Namespaces* by James Clark (`www.jclark.com/xml/xmlns.htm`)
- *XML Namespaces by Example* by Tim Bray (`www.xml.com/pub/1999/01/namespaces.html`)
- *19 Short Questions about Namespaces (with Answers)* by David Megginson (`www.megginson.com/docs/namespaces/namespace-questions.html`)
- *Namespace Myths Exploded* by Ronald Bourret (`www.xml.com/pub/2000/03/08/namespaces/index.html`)
- *Namespaces FAQ* by Ronald Bourret (`www.rpbourret.com/xml/NamespacesFAQ.htm`)

### *A Brief List of Confusions and Controversies*

Of the many items that have come up in multiple discussions, we have chosen three that we think are the most important.

- Unlike in programming languages, the names of XML namespaces are different from the prefixes that qualify local names. Those prefixes (with the exception of `xml:`) are arbitrary. The same prefix can map to different namespaces within the same document, and the same namespace can map to different prefixes.
- *XML 1.0* pre-dates *XML Namespaces*. It treats namespace declarations as any other attribute; it treats qualified names as monolithic strings with no internal structure, and it knows nothing about the relationship between namespace prefixes and namespace URIs. One consequence of this is that DTDs and later specifications such as *DOM* or *XPath* have different ideas about element names and document identity.
- Programming language namespaces are real and contain information about their names. A Java or C++ class contains declarations and definitions of the names that its own name qualifies. If your class is called Address, and you have a variable named postalIndex, then the qualified name of this variable is Address.postalIndex; but, also, the definition of the Address class declares that variable and specifies some of its properties

(such as data type and initial value). By contrast, XML namespace URI, contrary to its name (Resource Identifier) does not identify any resource. There is no commitment whatsoever that de-referencing that URI will get you to any place reasonable, or any place at all. According to the namespace recommendation, the namespace URI has no intended meaning: it's just a unique string of characters that protects against name conflicts.

This last feature of XML namespaces has been especially difficult to accept. A number of people argued that a namespace URI should point to a DTD, or an XML schema, or some such resource that would provide information about the syntax of the names that belong to that namespace, and perhaps also about their intended meaning. However, it proved impossible to build a consensus on what such an authoritative resource would be, and many argued that XML's unique decentralized strength is in having its intended interpretation left unconstrained by anything authoritative.

### *RDDL to the Rescue*

In the end of 2000, Jonathan Borden and Tim Bray developed a compromise proposal that seems to be gaining acceptance. Instead of a specific resource, they proposed that the namespace URI should point to a resource-description document that would describe standard resources (such as stylesheets, schemas, and so on) in a standard format. They called the format *RDDL* (*Resource Directory Description Language*) (For more information, visit `www.rddl.org`.)

By design, an RDDL document is human-readable and machine-processable. For humans, RDDL looks just like XHMTL. To give machines something to do, RDDL has a single additional element called `resource`. The resource element of RDDL is also a simple link element in the XLink sense: it has a required `xlink:type` attribute whose value, in the current version of the specification, can only be `simple`. (However, when extended link processors are widely available, it will probably be common to find an RDDL resource element that is an extended link element and has XLink resource elements as its children.)

Here is an RDDL example, a document describing resources for the pdata.xml example of Listing 1-7. We repeat here the beginning of that example, with a default namespace declaration added.

```
<?xml version="1.0"?>
<!-- personal data for people and other kinds of personalities -->
<pdata xmlns="http://csproj.colgate.edu/xmlp/ns/pdata/">
...
</pdata>
```

The namespace URI points to an existing directory, `http://csproj.colgate.edu/xmlp/ns/pdata/`, within which the RDDL file, index.html, is the default. The RDDL file, shown in Listing 2-10, follows this outline:

- root element with namespace declarations
- introductory prose
- validating resources (DTD, XML Schema, RELAX NG)
- display resources (CSS)

*Listing 2-10. An RDDL Example*

```
<!DOCTYPE html PUBLIC "-//XML-DEV//DTD XHTML RDDL 1.0//EN"
                       "http://www.rddl.org/rddl-xhtml.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
           xmlns:xlink="http://www.w3.org/1999/xlink"
           xmlns:rddl="http://www.rddl.org/">
<head><title>PData Resources</title></head><body>
<h1>PData Resources</h1>
<p>
  This is a resource file for the <tt>pdata</tt> example of chapters 1 and 2 of
  <cite>XML for Programmers</cite> by Alexander D. Nakhimovsky and Tom Myers.
  The standard namespace for this example is
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/">
    http://csproj.colgate.edu/xmlp/ns/pdata/</a>
  and that directory's default,
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/index.html">
    http://csproj.colgate.edu/xmlp/ns/pdata/index.html</a>,
should  be the URL of the current version of this file.
</p>
<h2>Validation</h2>
<p>You can validate a <tt>pdata</tt> document with a dtd,
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd">
    http://csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd</a>.
</p>
<rddl:resource
  xlink:title="DTD for pdata validation"
  xlink:href="http://www.csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd"
  xlink:role="http://www.isi.edu/in-notes/iana/assignments/media-types/text/xml-
dtd"
  xlink:arcrole="http://www.rddl.org/purposes#validation"
>
```

```
  <p>A sample DOCTYPE would be</p>
<pre><tt>&lt;!DOCTYPE pdata  PUBLIC "-//Nakhimovsky-Myers//DTD pdata 0.05//EN"
          "http://www.csproj.colgate.edu/xmlp/ns/pdata/pdata.dtd"&gt;
</tt></pre>
</rddl:resource>
<p>You can also validate a <tt>pdata</tt> document with a RELAX NG grammar.</p>
<rddl:resource
  xlink:title="RELAX NG grammar for pdata validation"
  xlink:href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.rng"
  xlink:role="http://www.rddl.org/#resource"
  xlink:arcrole="http://www.rddl.org/purposes#validation"
>
<p>Such a grammar is available at
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.rng">
    http://csproj.colgate.edu/xmlp/ns/pdata/pdata.rng</a>
 </p>
</rddl:resource>
<h2>Display</h2>
<rddl:resource
  xlink:title="CSS Style Sheet for pdata display"
  xlink:href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.css"
  xlink:role=
 "http://www.isi.edu/in-notes/iana/assignments/media-types/text/css"
>
 <p>
   There is a simple CSS1 style sheet for pdata documents at
   <a href="http://csproj.colgate.edu/xmlp/ns/pdata/pdata.css">
      http://csproj.colgate.edu/xmlp/ns/pdata/pdata.css</a>
 </p>
</rddl:resource>
</body>
</html>
```

As you can see, each resource element is a simple link element with `xlink:role` and `xlink:arcrole` attributes on it (in addition to the required `xlink:href`). Guidelines on how to use these two attributes (to specify the nature and purpose of the resource, respectively) can be found at `www.rddl.org`. Some of the common natures and purposes are also listed there. The intent is to provide enough information to make RDDL documents processable by machines.

Some of the required processing is fairly straightforward. Consider DTD validation. The "RDDL standard" values for the nature and purpose attributes are as follows.

```
xlink:role=
    " http://www.isi.edu/in-notes/iana/assignments/media-types/text/xml-dtd"
xlink:arcrole="http://www.rddl.org/purposes#validation"
```

Given a namespace URI, we extract the document at the end of it (with the `document()` function in XSLT) and select an RDDL resource element with the "validation" `arcrole`. The XPath expression would be

```
//rddl:resource[@xlink:arcrole='http://www.rddl.org/purposes#validation']
```

If we can confirm that this element has the DTD nature as well, then `xlink:href` must be a link to a DTD that can be used for validation in ordinary ways, as presented in the next chapter.

Similarly, we can look for other standard resources with other standard roles. It seems plausible that systems of such roles will be developed: with XML used to describe commercial objects, we can expect a "purchase" role for a resource that helps you link to software for buying one of whatever it is, and a "complaints-department" role that helps you link to software to say that what you bought wasn't what you thought you were buying.

## Conclusion

We've covered a lot of ground in this chapter. The most important notion is a well-formed document, and the second most important notion is a namespace. We have seen several XML languages, including XHTML, XLink, and RDDL. What we have not done in this chapter is pose a question such as "How do we check that a particular XHTML document is not only well formed but also contains only the markup that is expected in XHTML?" This is the question of validity or conformance to a specific grammar, and we take it up in the next chapter.