The Definitive Guide to Berkeley DB XML

Danny Brian

Apress®

The Definitive Guide to Berkeley DB XML

ISBN-13: 978-1-59059-666-1

ISBN-10: 1-59059-666-8

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matt Wade
Technical Reviewer: George Feinberg
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Nancy Sixsmith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Molly Sharp
Proofreader: Linda Seifert
Indexer: John Collin
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `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 Source Code section.

APPENDIX A

XML Essentials

Most XML tutorials focus on the XML itself—its syntax and rules. These subjects will be covered here, of course, but beginners can best understand XML with a focus on the data. This chapter will not only explain the "why" and "how" of XML (assuming that you have no previous XML experience) but also provide sufficient information for you to begin working with BDB XML.

It's About the Data

A common misunderstanding about XML is that it is a markup language. Although this is a true statement, the classification implies that XML is similar to HTML or is somehow an outgrowth of it. Although XML can be used like HTML to mark up documents, indicate formatting rules, and so on, these usages are just a few of many. The real purpose of XML is to *describe data*. In this way, it isn't so different from other data formats. Consider the case of the typical comma-delimited list (they are often called *comma separated values [CSV]*):

```
Brown, Jim, 24, 612-323-0091, Minneapolis, MN, male
Thompson, Sarah, 51,,,, female
Jackson, Jeremy, 31,, Salt Lake City, UT, male
Jones, Sue, 19, 313-555-1123, San Jose, CA, female
Carter, Frank,, 800-555-1123,,, male
```

Other examples include tab-delimited and space-delimited data (for instance, the Wordnet example in Chapter 2, "The Power of an Embedded XML Database"). Data formats such as these are usually used when dumping data from actual databases. With read-only databases such as Wordnet, however, they are sometimes used in production.

Note XHTML—the strict specification for HTML—is a dialect of XML. XML can express blocks and styles of text as well as define its own organization.

Humans can gather the meaning of fields from their contents: the first field is most likely a surname; the second is probably a given name; following fields might contain age, phone number, city, state, and gender. Of course, any program that has to interpret this data needs identifiers for each field, which can sometimes occur at the top of the file in the field of table field names:

```
Last name, First name, Age, Phone, City, State, Sex
```

A script or application can now parse this file, understand what each field means, and then do something with the data. Blank values are left empty, indicating that the information is not available.

If you have used a comma- or tab-delimited format extensively, you know that certain text must be *escaped*. Values with commas break the formatting, which can lead to conditions that require special rules. Values with commas are required to be contained in double quotes. Picture a field that lists hobbies, like so:

```
Brown, Jim, 24, 612-323-0091, Minneapolis, MN, male, "sports, boats, carpentry"
```

One solution is to create a field for each hobby; you embedded a CSV row into a value instead. What if one of the values contains double quotes? You have to replace the quotes in the value with two consecutive double quotes and include the value itself in double quotes:

```
Brown, """Big"" Jim", 24, 612-323-0091, Minneapolis, MN, male, "sports, boats, carpentry"
```

And because your CSV file uses line breaks to separate the records themselves, adding a field for a street address and a value with a line break also requires double quotes:

```
Brown, """Big"" Jim", 24, 612-323-0091, "Attn: Jim Brown
Pleax Systems, Inc.
18520 25th Ave
",Minneapolis, MN, male, "sports, boats, carpentry"
```

It's easy to see how such a format can grow unwieldy, but the rules are necessary to keep the format machine-interpretable.

Going back to the original CSV example, a novice translation of this data to an XML format might look something like this (abbreviated to only the first record):

```
<list>
    <entry>
        <field>
            <name>Last name</name>
            <value>Brown</value>
        </field>
        <field>
            <name>First name</name>
            <value>Jim</value>
        </field>
        <field>
            <name>Age</name>
            <value>24</value>
        </field>
        <field>
            <name>Phone</name>
            <value>612-323-0091</value>
        </field>
        <field>
            <name>City</name>
            <value>Minneapolis</value>
        </field>
        <field>
            <name>State</name>
            <value>MN</value>
        </field>
        <field>
            <name>Sex</name>
            <value>male</value>
        </field>
    </entry>
</list>
```

XML documents typically have the extension .xml, so this document can be called list.xml. In the example, a person has attempted to describe the CSV data in XML. The bold text indicates XML *values*. A novice might look at this example and comment on how long or redundant the XML is when compared with the CSV: "What a waste of space!" Of course, there is a better way to use XML.

The phrase *semantically rich* is frequently used in connection with XML. XML is a self-defining markup language, so you are essentially free to invent your own format. The best *tags*—the names in the brackets (< >)—are those that describe the data itself.

Recall that the novice document example was given the name list.xml. This seems a poor choice for a filename because a person seeing the file would have no idea about the nature of its contents; a better name might be people.xml.

This problem also occurs with the tag names. Consider the tags to take the place of the field names at the top of the CSV document. This would give you an XML translation closer to the following (note that the XML values are still bold):

```
<people>
    <person>
        <lastname>Brown</lastname>
        <firstname>Jim</firstname>
        <age>24</age>
        <phone>612-323-0091</phone>
        <city>Minneapolis</city>
        <state>MN</state>
        <sex>male</sex>
    </person>
</people>
```

The field names have moved into tags. The outermost tag is called `people` because this document supposedly contains a list of people. You can tell what any given value means, even if the value itself is ambiguous, without looking at the top of the file for field names or counting commas in a CSV file. Tag names, called *elements* in XML, are similar in this example to relational database (RDB) field names in tables. XML is a semantically rich data format because each value has complete identification—its meaning is clear without reference to tables or remote sources. Editing this record manually is simply a matter of editing a file and then replacing or adding the elements and values that you want. Semantically rich formats provide readability for humans, editability for humans and programs, and self-contained meaning throughout the document. Although this format requires a longer file than others (the CSV example, for instance), it brings a dramatic increase in usability. The extra bytes are a small price to pay for context: with XML, data has "inline meaning."

A single XML file can contain an entire database, or individual records can be stored in many XML files. (This is where BDB XML will come into play.) What might be considered a single record or row in an RDB can be treated as a single file. Individual records can contain hierarchical data all their own. Suppose that you want to add a list of hobbies to your record, permit multiple phone numbers, and enable the addition of a middle name or name prefix. You could change the XML to read as follows:

```
<person>
    <name>
        <last>Brown</last>
        <first>Jim</first>
        <middle>Austin</middle>
    </name>
    <age>24</age>
    <phone>
        <office>612-323-0091</office>
        <home/>
    </phone>
```

```
    <city>Minneapolis</city>
    <state>MN</state>
    <sex>male</sex>
    <hobby>sports</hobby>
    <hobby>boats</hobby>
    <hobby>carpentry</hobby>
</person>
```

Notice that the outermost element is now called `person` because this file contains only one. New elements have been added to the `<name/>` tag: `<last/>`, `<first/>`, and `<middle/>`.

■ **Note** In referring to the elements, I am using their empty counterparts, although they are not empty in the document. In the context of describing them, however, they have no content. Chalk it up to excessive logic.

The new elements make sense conceptually because the full name comprises a first, middle, and last name. Options for multiple phone numbers are also available under the `<phone/>` tag. In this example, there is no home phone, so its element is empty. It is a shortcut for (but carries the same meaning as) the following:

```
<home></home>
```

There is now a `<hobby/>` tag for each of the hobbies. (You could instead add a tag named `<hobbies/>` and then add three `<hobby/>` tags inside of it.)

Finally, add the quoted nickname and street address containing line breaks:

```
<person>
    <name>
        <last>Brown</last>
        <first>"Big" Jim</first>
        <middle>Austin</middle>
        <nick>Big</nick>
    </name>
    <age>24</age>
    <phone>
        <office>612-323-0091</office>
        <home/>
    </phone>
    <street> Attn: Jim Brown
                Pleax Systems, Inc.
                18520 25th Ave
    </street>
    <city>Minneapolis</city>
    <state>MN</state>
    <sex>male</sex>
    <hobby>sports</hobby>
    <hobby>boats</hobby>
    <hobby>carpentry</hobby>
</person>
```

For illustration, I added the nickname to both the `<first/>` tag and a new `<nick/>` tag. This would be a matter of personal preference. For the new `<street>` tag, notice that because XML is not based on line breaks for record delimitation, no escaping is needed with this value. Of course, you

might guess that greater-than and less-than brackets in values break the XML format. You'll learn about that soon enough.

Note that we invented the markup shown here: `<person/>`, `<name/>`, `<age/>`. All the tags used here have no meaning outside of the document. This is what makes XML flexible: you can make up your own formats, use them in files and applications, and retain compatibility with any XML tools or applications. Of course, there are many standardized XML formats—they include address book formats (vCard), content syndication formats (Really Simple Syndication [RSS] and Atom), and HTML itself (XHTML, specifically).

XML can be used for markup as with HTML. It does not tend to make XML easier to learn for beginners, so I will continue to describe XML in the pure context of data. We will return to this topic at the end of the chapter.

XML Building Blocks

XML consists primarily of two basic pieces: elements and attributes.

Elements

An *element* is what people commonly describe as a *tag*:

```
<city>Minneapolis</city>
```

This tag has a name and a value. Its name is `city` and its value is `Minneapolis`. An XML document always has one or more elements. All but one of these elements are required to be inside other elements, and the outermost element is referred to as the *root* element. In this example, `<city/>` is the root element because it is the only element.

Element names can contain letters, numbers, and other characters (including non-English and non-ASCII characters), but they must start with a letter. Nonetheless, it's a good idea to avoid using characters that might cause confusion or look out of place. For example, a reader—or software—might interpret a dash to be a minus sign or interpret a period to be an object method. The software you use (or write) to process XML usually imposes restrictions on the characters you use before XML does. Element names cannot contain spaces or start with the letters *xml*. They should be short and concise. Most XML dialects stick to lowercase letters (names are case-sensitive), but this is a matter of preference.

Element values can include other elements, text (`Minneapolis`, in this case), or a combination of the two. The content of an XML element is everything from its opening tag to its closing tag, including white space. Elements can be empty, as shown earlier.

Attributes

The second main piece of an XML document (and a reason why element names cannot contain spaces) is the *attribute*. An attribute is a name/value pair just like an element, but with stricter usage rules. The attribute occurs inside of the element tag, with a name, equal sign, and a value in quotes. Here, two are added:

```
<city latitude="44°57'N" longitude="93°16'W">Minneapolis</city>
```

Attributes are usually used to "qualify" the element. More than one attribute in the same tag cannot use the same name. The naming restrictions are the same as elements, and attribute values are typically of the short variety.

Well-Formedness

Well-formedness describes the compliance of a format to its rules. For XML, these rules are rather straightforward. An element can contain other elements and each element can contain one or many attributes.

Tag names are case-sensitive, so opening and closing tag names must be spelled identically. Elements must always have an opening and a closing tag unless they're empty (in that case, the `<tag/>` format can be used). Elements must never *cross*—elements must be closed at the same level in which they are opened. The following is not legal:

```
<person>
    <name>
        <first>Mike</first>
        <last>Stevens</name>
    </last>
</person>
```

It's helpful to imagine XML elements as file directories that can contain other directories: you can't jump up to a parent directory and remain inside of the subdirectory. Following this analogy, the files within a directory would be an element's text content, and the directory name and permissions could be considered attributes.

Unquoted attribute values are illegal:

```
<city latitude=44°57'N>
```

The proper syntax is as follows:

```
<city latitude="44°57'N">
```

You will notice that most XML documents—especially those created by a program—start with an XML *declaration*. This provides the XML version and character encoding of the document, which is discussed later.

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

For now, note that most of the examples in this book omit this declaration—and most XML parsers do not enforce it.

White space in the content of elements is preserved with XML unless you specify a different behavior to your XML parser. This is unlike HTML, which consolidates consecutive white space (spaces, tabs, and so on) to a single space. Also note that a new line is always stored as line feed (LF) in XML documents. Windows applications usually store a new line as a pair of characters (a carriage return plus an LF); in Macintosh Classic applications, a carriage return (CR) is typically used. But XML applications on all platforms (should) know to read and store lines in an XML file terminated with a single LF.

Because the greater-than and less-than signs are used for tags, when they occur in the content of an element they must be replaced with an equivalent *entity*, which is already familiar to HTML users. Assume that the text is the following:

```
$x > $y
```

This text can be placed into an element as follows:

```
<statement>$x &gt; $y</statement
```

You can define your own entities to be used (discussed later); the default XML entities are those characters used in the tags (see Table A-1).

Table A-1. *Default XML Entities*

Entity	Meaning	Character
lt	Less-than sign	<
gt	Greater-than sign	>
amp	Ampersand	&
apos	Single quote	'
quote	Double quote	"

Finally, XML files can contain comments in the following form:

```
<!—comment -->
```

■ **Tip** XML editors check for compliance with the syntax rules as you type. Alternatively, shell tools such as the libxml2 `xmllint` utility (`http://www.xmlsoft.org`) can be used to check syntax:

```
$ xmllint --noout people.xml
```

This code outputs contextual errors in the file or outputs nothing if the file is well-formed XML.

CDATA

An XML parser parses the text content inside elements; it must do this to determine where the closing tag occurs. This is why illegal characters—such as the greater-than and less-than characters—must be escaped (expressed as entities).

XML does provide for the storage of data without using entities. For example, you might want to store a math equation or a script in the content of an XML element (this will be familiar to JavaScript users), and escaping every greater-than and less-than symbol could be excessive. To force the XML parser to ignore the content of an element, you can use a CDATA section. It begins with the text `<![CDATA[` and ends with `]]>`, as shown in this example:

```
<script>
<![CDATA[
    function decide(x,y) {
        if  (x > y && y > 0) then {
            return x;
        }
    }
]]>
</script>
```

The only string the CDATA section cannot contain is `]]>`, making nested CDATA sections impossible.

You should use CDATA sparingly; it is not intended as means of "getting around" strict XML formatting. XML Stylesheet Language Transformations (XSLT) beginners sometimes put HTML

fragments in these sections to recombine them later. This practice usually leads to unintended results and lessens the benefits of using XML and/or XSLT in the first place. When using CDATA, keep in mind that to the XML processor, the following are equivalent:

```
<example><![CDATA[ x > y ]]></example>

<example>x &gt; y</example>
```

The effect of CDATA sections is to have the processor treat element content that is not escaped as if it were. After these examples have been parsed, they are essentially the same to the program.

That's really all there is to the syntax rules of XML. There are a few more content types, but we'll get to those later. You can probably tell that XML itself has little to do with its usefulness as applied to real-world formats.

Relationships

Many XML technologies use the relationship between elements and attributes to process the XML. The outermost element is the root element, and elements inside of it are child elements. All elements in an XML document have one (and only one) parent element—with the exception of the root element. In this example, the `<name/>` element has a parent and children.

```
<person>
    <name>
        <last>Brown</last>
        <first>"Big" Jim</first>
        <middle>Austin</middle>
    </name>
    <age>24</age>
</person>
```

For element `<name/>`, its parent is `<person/>`; its children are `<last/>`, `<first/>`, and `<middle/>`. The element `<age/>` has no child elements; its parent is also `<person/>`. Because elements `<name/>` and `<age/>` share the same parent, they are *siblings*.

All elements that occur inside of another are *descendants* of that element. The descendants of `<person/>` are `<name/>`, `<last/>`, `<first/>`, `<middle/>`, and `<age/>` (every element in the document other than itself). Similarly, all elements that occur as a parent or parents-of-parents are *ancestors*. In this example, `<middle/>` has the ancestors `<name/>` and `<person/>`.

Attributes that occur within an element tag are technically children of that element and have the same relationships just described.

Namespaces

Whether on the web, in programming languages, or elsewhere, a *namespace* qualifies some piece of data to make it unique. The HTTP path index.html depends on the URL before it to be located and differentiated from all other index.html paths. XML also provides namespaces to make element names unique. Take the case of this XML document:

```
<img>
    <path>/images/stephan.jpg</path>
    <size>122K</size>
</img>
```

The <img/> element, which is reminiscent of the HTML image element, could lead to conflicts by processors or people. By using namespaces, element and attribute names can be qualified with a prefix:

```
<myapp:img>
    <path>/images/stephan.jpg</path>
    <size>122K</size>
</myapp:img>
```

■ **Note** You might wonder why we would want to use a namespace if a document contains only one kind of <img/> element, and any application that uses this document is unlikely to know about the HTML version. In practice, most XML documents do not need to use namespaces. If you intend your XML to be used by several people or multiple applications, it's probably a good idea to use them. It can also save headaches if you later decide to distribute your XML or use it in different environments, but it is by no means required.

XML namespaces are still local to the file in which they occur. To be unique outside of your file, the prefix gets associated with a unique Uniform Resource Identifier (URI). Here again is the previous example:

```
<myapp:img xmlns:myapp="http://www.apress.com/myapp">
    <myapp:path>/images/stephan.jpg</myapp:path>
    <myapp:size>122K</myapp:size>
</myapp:img>
```

The xmlns attribute prefix is special; it tells the reading application to associate the namespace prefix myapp with the specified URI. Thus, the element namespace myapp is arbitrary: the XML parser relies on the URI to determine uniqueness. Putting this declaration in the <myapp:img/> element causes all child elements with the same namespace (notice that these were added) to also be associated with that URI. The parser sees this URI (in this case, a URL) as simply a string, used to give the namespace a unique name. The URI is not accessed or used by the parser.

To avoid repeating a namespace for every child or an element, a default namespace can be declared:

```
<img xmlns="http://www.apress.com/myapp">
    <path>/images/stephan.jpg</ path>
    <size>122K</size>
</img>
```

This is equivalent to the previous example, but shorter and more readable. When namespaces are used, they are often used to permit the mixing of elements that do not share namespace. In that case, this shortcut obviously cannot be used, and each element requires a namespace prefix or it will be assumed to not have a namespace. XSLT is one such case (it will be demonstrated shortly).

Validation

Given so much flexibility to invent your own markup (XML "dialect"), how can you enforce a format on your XML documents? You're probably familiar with the notion of *validation*: web forms get validated to ensure that they have all the required information, merchants validate credit card numbers to be sure that they are acceptable, and so on. To validate XML means to not just confirm well-formedness but also to confirm that it conforms to additional rules. These rules can include the names of elements and attributes, the number of children an element is allowed to have, or even the kind of content an attribute or element can contain.

Consider the previous XML example. This document could be fed to an XML parser and pass with flying colors—it is well-formed XML. However, no address book application could make heads or tails of it. It might make for a cute message to a friend, equivalent to saying this:

```
Name: Jim Brown
City: Minneapolis
```

But again, this information would not be particularly useful to an application because it doesn't know for certain what is being described. To provide data in a format that can be understood by an application, you need to know which *standards* it supports. For example, a prevalent standard for address book data is called vCard and is supported (or used natively) by most contact-management applications. vCard (and its XML version, RDF vCard) documents can be imported into an address book, and vCard would then know exactly what each field represented. The preceding example might look thus in the vCard format:

```
BEGIN:VCARD
VERSION:3.0
CLASS:PUBLIC
REV:2005-09-28 13:35:45
FN:Jim Brown
N:Brown;Jim;;;
ADR;TYPE=work:;;;Minneapolis;;;
END:VCARD
```

Address book applications understand this format because they have been written to process it. Similarly, address books by different developers running on different operating systems could export and import address books without loss of information. This is another benefit of a text-based format: no specific operating system software is necessary to read the file, as is true for binary data formats.

Credit card numbers are often validated before they are charged. For example, many online orders are emailed to the merchant, who then charges your card manually (instead of charging the card in real time as you place an order). To save the merchant the hassle of getting an erroneous credit card number (and to save you the hassle of getting a phone call to confirm it), credit card numbers are assigned to allow a simple computation to know whether the number is *valid*. Similarly, software that reads an XML document can quickly confirm whether it conforms to the expected format. If it doesn't, processing stops and an error is reported. This saves the software the expense of working on data that might be incomplete. In the case of software that generates the XML file, validation can be performed before the document is even saved.

Thus, validation is a process that a person or software must perform, and both require some formal description of the rules for this to happen. There are two popular ways to describe these rules: by using Document Type Definition (DTD) and XML schema. DTDs are older and less concise; schemas are concise, flexible, and are written with XML (making them easy to parse for XML software). If you are familiar with the idea of RDB schemas, the idea isn't too different: an RDB schema describes what the database (its tables and rows) looks like and what it is allowed to contain.

XML schemas do more than enforce the format of an XML file; they also explain to developers what a given XML document should look like in the form of documentation. It is important to understand that without some formal description of an XML format, even well-formed XML is useless for the storage or exchange of data. Sure, a human might be able to figure out what the document contains, but software has no point of reference. For this reason, XML is frequently referred to as "not very useful," and this assessment is correct. XML is simply some rules for formatting tags and their contents; it's the *schema* for the XML that makes the document intelligible to software and its *application* that makes it useful.

XML Schemas

I won't describe DTD here because it is a somewhat dated and unpopular format for describing rules. XML schema is the first schema language to be recommended by the World Wide Web Consortium (W3C) and is written in XML. Schema documents typically have the extension .xsd. The XML schema is a complicated subject all by itself, so I only introduce the topic here. Note that XML schemas are not completely necessary for using and understanding either XML or BDB XML.

XML schemas enable you to describe what constitutes a "valid" XML format for your data, with varying degrees in strictness. As was already discussed, well-formed XML is not necessarily intelligible to software. Consider that an XML element that you expect to contain a price for a product instead contains text or contains two prices instead of the expected single element. This is an error, even if the XML document is well-formed. Schemas give you a way to describe and enforce these rules.

Here again is the XML file person.xml:

```
<person>
    <name>
        <last>Brown</last>
        <first>Jim</first>
    </name>
    <age>24</age>
</person>
```

XML elements are *optionally ordered*, which means that some applications might depend on the order of tags, and that order must be retained to reproduce or write out the XML file. In the example, the `<last/>` tag comes before the `<first/>` tag. Although this placement is purely coincidental, you might want to enforce an order on these elements. Notice that these same two elements contain string content, whereas the `<age/>` element contains a number.

■ **Note** Of course, having an age field in any application makes little sense because you would want it incremented; given a birth date to do so, you could compute a person's age. Unless, of course, the list shows deceased persons. I digress.

Here is an XML schema document to accompany person.xml; it is called person.xsd:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns="http://www.w3schools.com">
<xs:element name="person">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name">
                <xs:complexType>
                    <xs:all>
                        <xs:element name="first" type="xs:string"/>
                        <xs:element name="last" type="xs:string"/>
                    </xs:all>
                </xs:complexType>
            </xs:element>
            <xs:element name="age" type="xs:integer"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

The document is valid XML and uses namespaces. The root element is <xs:schema/>, indicating that this it a schema document. The same element defines the xs namespace and defines a default namespace. So far, the document is not saying anything about the "person" XML.

The bold sections in this schema example are the opening <xs:element/> tags that describe the "person" XML file. The first child element is <xs:element/> with an attribute "name" with value "person". This element describes the root element of this XML file—if the XML file did not begin with the <person/> element (which it does), an XML parser would fail right away with a validation error. Other <xs:element/> elements describe the <name/>, <first/>, <last/>, and <age/> elements.

Without delving too deeply, note that the <xs:element/> tags for the <first/> and <last/> elements have a type of "xs:string", meaning that any string value is valid for those tags. By contrast, the <age/> tag's content is to an integer with the type "xs:integer". The <xs:complexType/> elements occur beneath elements that are allowed to contain other elements. In this case, there are the <person/> and <name/> elements; no others are allowed to have child elements. Finally, the <xs:sequence/> element requires that the children of <person/> occur in the order specified (having the <age/> tag before the <name/> tag would result in a validation error), and the <xs:all/> element tells the validator that <first/> and <last/> are required—but not necessarily in the order shown here. You can see that schemas give you a lot of control over what XML format you will consider legal.

■ **Note** Most XML editors enable you to associate an XML document with a schema, performing validation as you type. Alternatively, shell tools such as the libxml2 `xmllint` enable validation, but they don't always do this by default. You can also pass it a path to an XSD for validation:

```
$ xmllint --schema people.xsd people.xml
```

This code outputs errors if the XML file does not conform to the specified schema.

Although an XML schema must be enforced, the XML document typically contains a reference to the XSD document. In the person.xml file, the xsi namespace is defined (the schema namespace), and the location of the schema document is specified:

```
<person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="person.xsd">
    <name>
        <last>Brown</last>
        <first>Jim</first>
    </name>
    <age>24</age>
</person>
```

Not all XML processors recognize this declaration or enforce it by default. Consult documentation for your XML parser or application to determine whether it is supported.

When should you use XML schemas? XML is pretty flexible, and schemas offer a lot of control. Even when you don't intend to validate all the XML you use, schemas can help you create well-designed XML and avoid some of the bloating and confusion that XML might otherwise cause when allowed to freely "evolve."

XPath: the Gist

As if this weren't enough fun already, it gets even better. Much of the power of XML becomes apparent when developers begin the process of searching and querying documents. In fact, XML querying

technologies are the very reason why many shops opt to use XML in the first place: the ease of use, intuitive syntax, and standardized usage. This is certainly the case with BDB XML.

XPath is the original and most widely supported language for querying XML documents. It is now in version 2.0, which is fully supported by BDB XML. This section describes only XPath 1.0. But never fear—version 1.0 is a subset of 2.0 and is completely compatible with even the newest implementations.

Paths

I earlier compared an XML document to a file system. File directories have children in the form of other directories, attributes in the form of names and timestamps and permissions, and content in the form of files. Regardless of your choice of operating system, the syntax of a typical file path is familiar:

```
/home/garron/documents/resume.pdf
```

This example refers to a `resume.pdf` file, inside a `documents` directory, inside a `garron` directory, inside a `home` directory. The `home` directory could be considered the root (Unix mount points and nomenclature notwithstanding), the `garron` directory could be considered a child, and so on.

XPath uses a similar syntax to refer to nodes in an XML document. For these purposes, a node is any piece of data in the XML file: an element, an attribute, or even the content of an element can be considered a node. Using our person XML example, the following path refers to the node for the person's last name in that file:

```
/person/name/last
```

Executed against the XML document with an XPath interpreter, this query would match one node of our document: the element `<last/>`. It would return the value `<last>Brown</last>`, in this case. You can test this with several XPath command-line tools.

Nodes

It is important to understand why this example refers to the `<last/>` element instead of just giving the value `Brown`. Forget any notion you might have of a node as a tag. XPath (and other technologies, including the Document Object Model [DOM]—discussed next) view every part of an XML document as a node, any place in the document it can return some value—whether an element, an attribute, or text content. Each is a kind of node, and an XPath query sees it as such. Nodes can contain other nodes, as is the case with this example. The `<last/>` element contains a child text node with value `Brown`. If elements and attributes were object classes in a given programming language (as is the case with the XML DOM), they would be subclasses of a parent node class and would inherit its properties. Thus, a node is the most primitive piece of an XML document, and everything in an XML document is a node. XPath (and XQuery) operate on this data model view of a document instead of the text it contains. The differentiation is critical to understanding how technologies such as XPath view a document and is explored in more detail in the next section.

There are several ways to get the text value of an element. The most universal is to append the `text()` "node test" to the end of your XPath query:

```
/person/name/last/text()
```

In truth, every word in this expression is a node test. The string `person` is used by the processor to test the name of the top-level elements, as is `name` and `last`. The `text()` example is also a test, but in the form of an instruction to match text nodes (this is explained in the next section). Here, the XPath processor is told to return the text node children of the `<last/>` element, and you would get back the value `Brown`.

Some XPath processors enable you to query for a text value directly. For example, the Perl `XML::LibXML` module gives you a `findvalue()` method on node objects; this returns the literal value of the node, which means all child text it contains and no child elements (or text children of child elements).

Understanding how XPath and DOM processors (and everything that uses them) treat XML nodes will save you future headaches.

Document Object Model (DOM)

A discussion about the DOM is required to understand how XPath and other XML technologies "see" an XML document. The XML DOM is a W3C specification to give programming languages a consistent means of processing XML. When a program parses an XML file, it must give the file some internal representation to enable a program to browse it. It's this internal representation that an XPath processor uses to find requested nodes in the document.

The DOM defines a set of node types, which are typically defined as an object class in programming languages. (You don't need a solid understanding of all these node types to understand or use the DOM or XPath.) Each type might contain children nodes of the types listed in Table A-2. (This table is an abbreviated list of node types; I have omitted `DocumentType`, `DocumentFragment`, `EntityReference`, `ProcessingInstruction`, `Entity`, and `Notation` for clarity.)

Table A-2. *Abbreviated List of DOM Node Types*

Node Type	Description	Children
`Document`	The entire document, root node	`Element` (only one), `Comment`
`Element`	Element	`Element`, `Text`, `Comment`, `CDATASection`
`Attr`	Attribute	`Text`
`Comment`	Comment	None
`Text`	Text content (character data)	None
`CDATASection`	Block of CDATA text	None

Assume that you're working with this XML document:

```
<person>
    <!-- Record needs revision -->
    <name lang="en">
        Mr.
        <last>Brown</last>
        <first>Jim</first>
    </name>
    <age>24</age>
    <note><![CDATA[ 24 > 20 ]]></note>
</person>
```

Parsing it into a DOM tree will result in a code language similar to Table A-3, regardless of the language or DOM implementation you are using. Indents identify children, and parentheses contain the node name (if any) and value (if any). Compare this tree closely with the preceding XML; you will probably find that it does not match your expectations.

Table A-3. *Parsed DOM Node Tree and XPath for Each Node*

Node Tree	Corresponding XPath
`Document ("#document", none)`	`/`
`Element ("person", "")`	`/person`
`Comment ("#comment", " Record needs revision ")`	`/person/comment()`
`Element ("name", "")`	`/person/name`
`Attr("lang", "en")`	`/person/name/@lang`
`Text("#text", "     Mr.     ")`	`/person/name/text()`
`Element ("last", "")`	`/person/name/last`
`Text("#text", "Brown")`	`/person/name/last/text()`
`Element ("first", "")`	`/person/name/first`
`Text("#text", "Brown")`	`/person/name/first/text()`
`Element ("age", "")`	`/person/age`
`Text ("#text", "24")`	`/person/age/text()`
`Element ("note", "")`	`/person/note`
`CDATASection (none, "24 > 20")`	`(none)`

Note the following:

- The top or root node is `/`, not `/person`. This node has no name or value; it has only children nodes. (Some processors report a name `#document` or no name.) This node can have only one element child because more than one would not be well-formed XML (in which every node needs a parent but one).
- The `/person` element has five child nodes: a text node, a comment node, two element nodes, and a CDATA section.
- The `/person` element has no obvious value, but `/person/text()` has a white space value. All the white space inside of `<person/>` but not contained in its children elements is counted as its value.
- The `/person/comment()` node is addressed with the `comment()` function, much as text nodes need `text()`. Comment nodes have no given name. (Some processors report a name `#comment` or no name.)
- The attribute node `/person/name/@lang` has a name and value. Getting values out of attributes is inexpensive.
- The element `/person/name/text()` has a value that includes the white space in `<name/>`, including line breaks (not shown here). Text nodes have no given name. (Some processors report a name `text` or no name.)
- White space throughout the document has been ignored by setting an option at parse time. Otherwise, the previous node tree would have many text nodes containing only white space, including each newline and leading white space. This often leads to confusion when querying and is expensive to both parse and store. (For this reason, prettily printed XML that retains all formatting does not store efficiently.)

- The CDATA node is not addressable with XPath. These sections are treated as text blocks that occur as text within their parent element. In this case, even though `<note/>` contains no text other than the CDATA section, the processor expands the CDATA to text content. Calling `/person/note/text()` gives a value of `24 > 20`, even though the DOM tree contains no text child of `<note/>`. CDATA is simply a node type of convenience for enclosing character data that is not trusted to be well-formed XML. (Some processors report a name of `#cdata-section`, `cdata-section`, or no name.)

We will return to the DOM shortly to discuss the classes and their methods in more depth.

ELEMENTS OR ATTRIBUTES?

On multiple occasions I have been asked, "When should I use elements instead of attributes to store information?" This is really a matter of personal preference, with some obvious constraints. First, only one attribute of a given name can exist per element. If you require more than one piece of data by the same name, you need to use elements. Second, storing anything but relatively short text values in attributes makes for poor readability. If the value is anything else, an element is the better option.

In other cases, I do have a personal preference. The way you like to code can influence where you prefer to put simple values. When writing XPath predicates, I find it easier to query off of attributes than I do elements, mostly for the sake of XPath readability, so I tend to put oft-queried values in attributes. Also, in most DOM implementations, attribute lookups tend to be slightly faster than element values because element text value queries have to look at the text child nodes of that element to compare.

On the other hand, when working with DOM objects, I find it slightly easier to add and manipulate elements than to do so with attributes, partly because I do it more often. So I put data that changes frequently into elements.

XPath: the Details

With an understanding of how the XPath sees nodes in a tree of XML, XPath is easier to learn. Many developers opt to use XPath over the DOM because of its terseness and ease of use. To keep this introduction simple, I will cover only the most common operators, axes, and functions.

Contexts

All XPath expressions have *contexts*. If you were using a shell to browse a file system, you might find yourself several directories deep from the root directory. From there, you could `cd` to a parent directory, to a child directory, to a sibling directory, and so on. XPath is similar in that a statement carries varied meaning depending on the node in a node tree that is referenced. The XML analog of a *current working directory* on an operating system is a *current node*.

Path Operators

XPath provides the expressions shown in Table A-4 to select or navigate nodes. Each expression is actually a shortcut for a longer formal operation.

Table A-4. *Common Path Operators*

Expression	Description
`/`	Selects from the root node at the start of an expression; otherwise delimits parents and children
`//`	Unrestrained select operation from the current node that selects nodes anywhere in the document
`.`	Current node
`..`	Parent of the current node
`@`	Selects attributes
`*`	Element node wildcard; matches any element node
`@*`	Attribute node wildcard; matches any attribute node
`node()`	Matches any node of any kind

The period operators are used primarily in predicates, discussed in the next section. The others are fairly straightforward in their usage.

Table A-5 lists some example expressions.

Table A-5. *Example Paths*

Expression	Description
`/`	Selects the root document node
`//age`	Selects all nodes named `age` regardless of where they occur in the document
`/person/name/*`	Selects all child elements of `/person/name`
`/person//last`	Selects all elements named `last` that descend from `/person`
`name/@lang`	From the current node (this is the context), selects the `lang` attribute of any child nodes named `name`
`//@*`	Selects all attribute nodes in the document

Clearly, the use of the `//` operator incurs substantial overhead because the XPath processor must traverse the entire XML file (or all descendants if it comes after the current node) to determine all matches.

Predicates

A *predicate* is a conditional that gets placed in your query, enabling you to select nodes more specifically and use more criteria than a simple path. Predicates always occur inside of straight brackets within an XPath expression, after a node for which the query needs to be qualified. You'll examine the most common operators in a moment. For now, consider the examples of XPath queries that use simple predicates shown in Table A-6.

Table A-6. *Example Expressions Using Predicates*

Expression	Description
`/person/age[1]`	Selects the first element node named `age` that is a child of `/person`
`//name[@lang]`	Selects all nodes named `name` that have an attribute `lang` from anywhere in the document
`/person[name/@lang = "en"]`	Selects a node named `person` if it has a child `name` that in turn has an attribute `lang` with value `"en"`
`/person[age > 20]`	Selects a node `person` if it has a child element `age` with a value greater than 19
`/person/name[last()]`	Selects the last element `name` that is a child of `/person`

XPath statements are used to pull data out of a document and to determine a document match. Predicates are often used as `if` statements against an XML document. Take the following XPath query, for example:

```
/person[name/last = "Johnson"]
```

Knowing that you will be querying large collections of XML with XPath should make the intent of this statement obvious. You probably don't want to select the matching `/person` node as much as you want to find all documents about a person with the last name `"Johnson"`. This expression can be used for both purposes.

■ **Note** Many examples use the `text()` function appended to a path, but it does not do what many XPath programmers think it does. Instead of returning the string value of a query, it is a node test that returns all child text nodes. The function `string()` is more often what is intended; it returns the string result of a query, the path provided as argument to `string()`. Of course, this works only when the interface used to make the query accepts a return value as text instead of nodes.

Predicates supply only criteria for the processor to apply to the node selection. They don't select anything themselves; they provide tests that the select expression must accommodate. Each is evaluated in context, given the current node at that point in the expression. Consider this predicate-free expression:

```
string(/person/age)
```

This is the select statement that grabs the text content of the `/person/age` element. Adding a predicate might result in this expression:

```
string(/person[name/first = "Billy"]/age)
```

The predicate causes the processor to narrow the selection criteria. A natural language version of the original expression would read, "Give me the age of any person with the first name of Billy." Notice that the predicate starts with `name`; at the point in which the predicate is evaluated, the current node is `/person`. After the predicate, the selection path picks right up with a child `age` of `/person`. The predicate doesn't interrupt the selection; it merely qualifies the portion of the select statement that precedes it.

Note that predicates can be chained and even nested:

```
/person[name[@lang = "en"]/first = "Jim"]/age
```

This code selects the age element of a /person/name that has an attribute lang with value "en", which in turn has a /person/name/first element with text value "Jim".

Operators

The most common XPath operators are already familiar to you. Table A-7 shows an abbreviated listing.

Table A-7. *Common XPath Operators*

Operator	Description	Example
+	Addition	20 + 4
-	Subtraction	20 - 4
*	Multiplication	5 * 4
div	Division	20 div 4
=	Equal (test, *not* assign)	age = 24
!=	Not equal	age != 23
<	Less than	age < 25
>	Greater than	age > 23
<=	Less than or equal to	age <= 30
>=	Greater than or equal to	age >= 24
or	Logical or	name/first = "Sue" or name/first = "Jim"
and	Logical and	age < 25 and age > 19
|	Union of node sets	/person/name/first | /person/name/last

Operators used in XPath predicates do not make variable assignments; they stick around only long enough to determine selection criteria. XQuery greatly broadens the functional potential of XML queries, but we're not there yet.

Axes

Things get a bit more complicated with XPath *axes*. (Do not worry if the concepts here don't register immediately; they will after you use them in real examples.)

An XPath axis defines directions that a query can take through an XML document. Axes have been in all the XPath examples thus far, but you have been using shortcuts. For example, each element and attribute name in the slash-delimited queries implies a child selection from the current node. More formally, each slash-delimited section of an XPath expression is referred to as a *step*. This is important for understanding the way a query is processed—starting at the left, evaluating each step in turn, taking the results from that step, and applying the next step to those results. In this way, an XPath expression is really a set of selection instructions given to the processor in the order of evaluation.

Recalling the file system analogy, simply typing **cd name** at a shell prompt will succeed only if the current working directory (whatever it might be) contains a directory named name. XPath also treats a single name in isolation as a child element. The explicit way to declare the child axis is to put the axis name in front of the element name by using double colons:

```
child::name
```

The name portion of this expression is referred to as the node test, with child as the axis. (Remember that the XPath processor uses the node test to test the node names and values as it crawls the node tree.)

When would you want to use such verbosity? The answer concerns contexts. When working with XML and related technologies such as XSLT and XQuery, you might want to know, for example, whether a given node descended from another node and how you can gain access to those nodes. Remember that nodes are represented within programs as class objects in the DOM model. Programs pass these objects around all by themselves, without the benefit (or overhead) of passing the entire XML tree. Having a way to know something about that node's context is useful in such cases.

The available axes used in XPath are listed in Table A-8.

Table A-8. *XPath Axes*

Axis	Description
ancestor	Ancestors of the current node
ancestor-or-self	Ancestors of the current node and the current node
attribute	Attributes of current node; the abbreviation is @
child	Children of current node; this is the default in the absence of an axis
descendant	Descendants of the current node
descendant-or-self	Descendants of the current node and the current node
following	Everything in the document after the current node (excluding descendants)
following-sibling	All siblings after the current node
namespace	All namespaces currently open at the current node
parent	Parent of the current node
preceding	All elements before the current node (excluding ancestors)
preceding-sibling	All sibling elements before the current node
self	Current node

■ **Note** Seeing that @ is an abbreviation for the axis attribute, you might be tempted to declare . as an abbreviation for the axis self. In actuality, the proper basis expression for . is self::node() because an axis is not a selection itself; it is a direction for the selection to take place. Similarly, .. is an abbreviation for parent::node(), and // is an abbreviation for descendant-or-self::node().

With an axis placed before a node test (as with child::name), the expression is often referred to as a basis. A predicate that follows a basis further qualifies it (as with child::name[1]). Together they comprise a step, and a list of steps forms the location path.

■ **Caution** Axes used with predicates can deliver unexpected results—unless, of course, your expectations are accurate. Although most basis expressions return matching nodes in order, some axes change that order. For example, the axis ancestor returns nodes in reverse order, which makes sense because you're looking "up" from the current node. This is important when using predicates that use ordering to apply select criteria. For example, ancestor::name[1] returns the closest ancestor element name, not the first ancestor name in the document.

Axes are a potentially confusing element of XPath, but you won't run in to them very often. When you do, they will make sense given their context.

Functions

The last aspect of XPath to be discussed is the XPath function. *Functions* introduce the missing functionality to XPath expressions. There are many standard functions in XPath 1.0 and many more in XPath 2.0, and most XPath implementations make it easy to write your own functions in a given programming language. Functions take on a life all their own with XQuery. Put simply, functions are where the interesting stuff happens.

■ **Note** You have repeatedly seen the use of `comment()`, `node()`, and `text()` where a node name would normally be in a node test. They look like functions; they *are* functions because they return values to the processor. But where a node test of `age` in the path `/person/age` is used as a literal, comparing `age` with the names of each child element of `person` (and returning any nodes that match the node test), these function node tests tell the processor to return certain kinds of nodes. Because comment and text nodes do not have names, this is the only way to address them.

Functions in XPath statements occur anywhere—they can be in predicates or they can contain the entirety of the path expression. As an example of the latter, the following demonstrates the function `contains()` used in a predicate:

```
/person[contains(name/first, "Dan")]
```

This function takes two arguments and returns `true` if the second string is contained within the first. Thus, this query would select `/person` if the text value of `/person/name/first` contained the string `"Dan"`. "Why isn't the first argument to the function `name/first` instead of `name/first/text()`,"you ask? It could be either, in fact. But because the `contains()` function expects text arguments, the XPath processor is smart enough to know that handing the node `first` to the function instead of its literal text value wouldn't achieve the desired result. Most XPath functions reflect a similar behavior, and you can cause your own custom functions to do the same.

Here is an example of a math function:

```
count(/person/name/*)
```

The function `count()` simply counts nodes passed to it. This is an example of an expression that doesn't select nodes. It returns a string instead.

Arguments to XPath functions can be fully qualified paths or they can use context:

```
/person[age > count(/person/shoesize)]/age
```

This expression selects the `age` element node, but only if that `age` value is greater than the person's shoe size. The expression could have been written this way:

```
/person[age > count(shoesize)]/age
```

The lack of a leading forward slash on the expression as argument to `count()` tells the processor that the note test is relative and not absolute.

More XPath functions are shown in Table A-9.

Table A-9. *Some XPath Functions*

Function	Description
`ceiling()`	Rounds a passed number to the smallest integer not smaller than the number
`concat()`	Concatenates all string arguments into one string
`contains()`	Returns `true` if the first string argument contains the second string argument
`last()`	Returns the index of the last node in the context note set
`name()`	Returns the name of the passed node
`local-name()`	Returns the `local` part of the name of the first node in the passed node set, minus a namespace
`normalize-space()`	Returns a white space–normalized copy of the passed string (leading and duplicate spaces removed)
`not()`	Returns the inverse of the passed value
`position()`	Returns the position of the current node in the context node set
`starts-with()`	Returns `true` if the first string argument starts with the second string argument
`sum()`	Sums the text values of all nodes in the passed node set

You will most often use functions within predicates to place conditions on your select. Here, too, you have already seen shortcuts for some of the functions:

```
/person/age[1]
```

This index syntax is a shortcut for the `position()` function:

```
/person/age[position() = 1]
```

The implication is that the node test before the predicate is an argument to the function itself. This is what is meant by the *context node set* in the function list descriptions. The node selection before the predicate comprises a node set, and functions such as `position()` and `last()` address this set as an array. This example selects the second-to-the-last `age` element that is a child of `person`:

```
/person/age[position() = last() - 1]
```

Finally, consider this multipredicate example:

```
/person/name[@lang = "en"]/*[position() = last()]
```

Here, everything before the `position() = last()` predicate determines the node set to be used to determine the set positions. This example selects the last child element of `/person/name` where `name` has an attribute `lang` with value `"en"`.

In XML applications, custom XPath functions are frequently written and used to introduce external data into the evaluations. We won't look at the writing of custom XPath functions in this book, but most XPath implementations make it a relatively painless thing. You will likely find it unnecessary using BDB XML and XQuery because they provide better ways to move data between query processors and the application. A major exception is if you want to use BDB XML databases with XSLT.

This section has described XPath 1.0, which is more lightweight than XPath 2.0. Note that BDB XML uses XPath 2.0 and XQuery, which adopt XPath 1.0 as a subset (and therefore will understand examples in this section). The differences are outlined in Chapter 7, "XQuery with BDB XML."

XML DOM, Continued

This chapter already discussed the DOM in some detail; this section will look briefly at the DOM methods.

The whirlwind of buzz around the technologies dubbed *Asynchronous JavaScript and XML (Ajax)*—Google Maps is the showcase example—is really just excitement over a new approach to using tools that have been around for some time. Within a web browser, the DOM is the interface to change the HTML dynamically using JavaScript. The XML DOM is a specification for programming languages to navigate and manipulate XML. It specifies classes and methods that have been adapted to interfaces in most programming languages and makes fairly easy work of using XML within programs, as well as moving between programming languages.

The node tree example earlier in this chapter illustrated how an XML processor views a parsed XML document. The XML processor can be a command-line utility, a programming API, or a web browser. The Firefox web browser has a DOM Inspector window accessible via the Tools menu that shows this internal document representation and highlights within the web page a DOM node selected in the Inspector window (see Figure A-1).

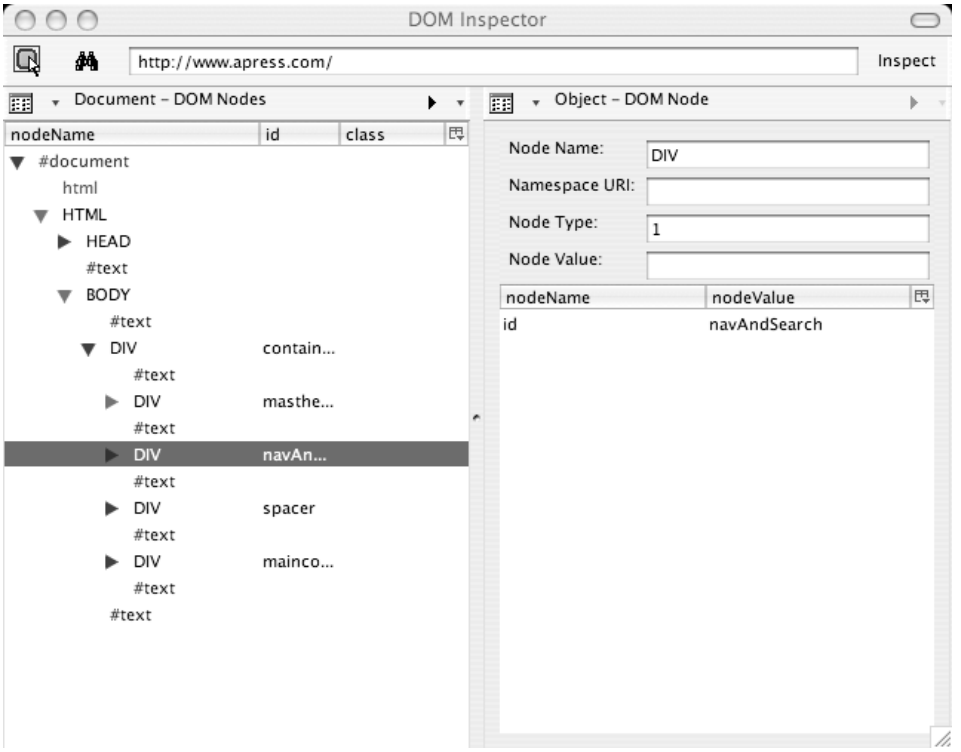

Figure A-1. *The Firefox DOM Inspector window*

DOM implementations do vary, but all share common (or at least similar) class and method organizations. The DOM defines the following interfaces that are implemented as classes. Table A-10 shows an abbreviated list.

Table A-10. *DOM Classes with Some Attribute and Method Examples*

Class	Attribute Example	Method Example
Node	nodeName, nodeValue, childNodes	replaceChild, appendChild, insertBefore
Document	doctype, documentElement	createElement, createComment
Attr	name, value	
Element	tagName	getAttribute, setAttribute
NodeList	length	item
Text		splitText
CharacterData	data, length	appendData, insertData, deleteData

The DOM interface enables both the reading and navigating of a parsed XML file, as well as the programmatic construction of an XML file. Writing XML using the DOM interface has the benefit that you never need to worry about properly formatting your elements, closing your tags, or escaping entities in text content because it is taken care of for you. Moreover, XML written with the DOM interface is assured to generate the same tree when parsed. Because XML is intended to be processed programmatically (not that you wouldn't want to read a book written in XML, of course), it makes the most sense to generate it programmatically, too.

■ **Note** There are in fact several versions of the DOM specification, each building upon the next: Level 1, Level 2, and Level 3. Level 1, which is referred to as *dynamic HTML*, defines the basic DOM structure, node types, and classes and methods. Level 2 adds support for Cascading Style Sheets (CSS) and events to the DOM, making the rich interaction of applications such as Google Maps easier. Level 3 adds a host of event modules that include loading and saving documents and an XPath module. Up to this point, XPath interfaces (not the query language, but the methods to execute them) have varied significantly between implementations. You won't deal with much that isn't defined in Level 1 in this book.

Implementation Considerations

DOM implementations are not always (or even usually) compatible—you cannot use two DOM libraries and pass documents or nodes between them. This isn't much of a problem unless you need to do very tight integration (for example, BDB XML with an XSLT processor). If so, your choice of programming language is more constrained. This is uncommon, however. XML makes it easy for incompatible programs to operate together with the same data, and not having compatible binary DOM objects is not much of a problem. In my own production environments, I use Python and Perl BDB XML interfaces to query and modify the database, but I use libxslt and Perl to perform XSLT transformations on an accompanying web site. Because BDB XML queries—whether returning lists of document matches or values—do not give you a DOM for that object without the overhead of rebuilding it, little is actually saved by operating on the same DOM object between applications.

In other words, don't worry about DOM compatibility when it comes to building applications that perform different operations with the same XML. You might choose to keep text copies of your XML files for use by processors other than BDB XML, treating them as authoritative and updating them to the database when they change, or treating the database as authoritative and getting XML source from your queries to parse.

Of course, if you are concerned with DOM compatibility, the Apache Xerces library should dictate your choice of processors and languages because BDB XML is built on it.

Reading and Writing XML

The uses of and differences between DOM implementations often reflect the styles of the programming language. We will explore a few parsing modules here, just to put the XML discussion into the context of actual code and provide a sense of XML processing with different languages.

Xerces C++

BDB XML uses (and includes) the Apache project's Xerces C++ XML parser. Xerces is a rather straightforward processor supporting the DOM specification, namespaces, and XML schema. It doesn't do XPath; it focuses instead on the core XML processing and lets related projects such as Apache Xalan tackle XPath. Xerces has parsers in Java as well, with Perl and COM bindings for the C++ libraries.

Listing A-1 omits declarations, includes, and error handling; it highlights the DOM method calls.

Listing A-1. *A C++ DOM Browse with Xerces*

```
static char* elementname = "Word";
static char* gXmlFile = "12.xml";

int main() {
    XMLPlatformUtils::Initialize();
    XercesDOMParser *parser = new XercesDOMParser;
    parser->parse(gXmlFile);

    // get the DOM representation
    DOMNode *doc     = parser->getDocument();
    DOMNode *element = doc->getFirstChild();
    DOMNodeList *nodelist = element->getChildNodes();

    for(int i=0; i < nodelist->getLength(); ++i) {
        DOMNode *child = nodelist->item(i);
        if (XMLString::compareString(child->getNodeName(),
                    XMLString::transcode(elementname)) == 0) {
            DOMNode *text = child->getFirstChild();
            printf ("%s: %s\n",
                        XMLString::transcode(child->getNodeName()),
                        XMLString::transcode(text->getNodeValue())
            );
        }
    }

    // clean up
    delete parser;
    XMLPlatformUtils::Terminate();
    return 0;
}
```

The file 12.xml has the following abbreviated content, which you might recognize from Chapter 2:

```
<Synset fileVersion="1.0" pos="n">
  <Id>12</Id>
  <WnOffset version="2.1" pos="n">00004576</WnOffset>
  <LexFileNum>03</LexFileNum>
  <SsType>n</SsType>
  <Word lexId="0">organism</Word>
  <Word lexId="0">being</Word>
</Synset>
```

Compiled (with the necessary header includes) and run, this listing outputs the following result:

```
Word: organism
Word: being
```

This is an example of a pure DOM parse without XPath. Of course, it can be cumbersome to navigate a document with the DOM if you don't know what it contains. It's made easier with Xerces filters that can be registered to act as node handlers.

Perl's XML::LibXML

The Perl XML::LibXML module, available from the Comprehensive Perl Archive Network (CPAN—`http://www.cpan.org`), embeds the libxml2 C libraries. Its DOM classes are XML::LibXML::Document, XML::LibXML::Element, and so on. Given Perl's strength at parsing, it makes a good choice for converting data to XML. Its sister, XML::LibXSLT (embedding libsxlt), enables XSLT processing on the same DOM objects.

Chapter 2 discussed a conversion of the Wordnet database to XML files. Recall that the desired format looked like this:

```
<Synset fileVersion="1.0" pos="n">
    <Id>14861</Id>
    <WnOffset version="2.1" pos="n">02772480</WnOffset>
    <LexFileNum>06</LexFileNum>
    <SsType>n</SsType>
    <Word lexId="0">baseball</Word>
    <Pointers>
        <Hypernym>14746</Hypernym>
        <Hypernym>14866</Hypernym>
    </Pointers>
    <Gloss>a ball used in playing baseball</Gloss>
</Synset>
```

In the truncated script example shown in Listing A-2, the parsing functions are removed, and the document operations are highlighted. This listing should be fairly straightforward, assuming that the Perl method call operator (`->`) is familiar to you.

Listing A-2. *Building XML with the Perl Module XML::LibXML*

```
#!/usr/bin/perl -w
use strict;
use XML::LibXML;

open my $file, "data.noun";
my $id = 1;

# iterate each line (synset) in the file
while (my $line = <$file>) {

    # create the DOM object for the synset
    my $document = XML::LibXML::Document->new("1.0", "UTF8");

    # create a new element, set its attribute
    my $element = $document->createElement("Synset");
    $element->setAttribute("fileVersion", "1.0");
```

```
    # read the offset and create an element for it
    my $offset = extract_offset($line);
    my $offset_element = $document->createElement("WnOffset");
    $offset_element->appendText($offset);

    # set attributes for version and pos ("part-of-speech")
    my $wn_version = "2.1";
    $offset_element->setAttribute("version", $wn_version);
    my $pos = extract_pos($line);
    $offset_element->setAttribute("pos", $pos);

    # make the offset element a child of our synset element
    $element->appendChild($offset_element);
    ...

    # set the root element for the document
    $document->setDocumentElement($element);

    # increment the id, and write the file
    open my $newfile, ">" . $id++ . ".xml";
    print $newfile $document->toString();
    close $newfile;
}
```

The result is a single XML file, starting with `1.xml`, for each line in the data file. In this case, the result is about 120,000 XML files, each complying with the format created in this script. You could write an XML schema to describe this format and ensure that the DOM usage is correct. If you ran this script, you'd have a decent collection of XML documents containing interesting data. I use this data in examples throughout the rest of the book.

Parsing the file with XML::LibXML is simple, and it provides a full-featured XPath (1.0) implementation (see Listing A-3).

Listing A-3. *Parsing a File in Perl with XML::LibXML*

```
#!/usr/bin/perl -w
use strict;
use XML::LibXML;

my $parser      = new XML::LibXML;
my $document = $parser->parse_file("12.xml");
foreach my $node ($document->findnodes("/Synset/Word")) {
    print "Word: "  . $node->to_literal . "\n";
}
```

This listing outputs the following result:

```
Word: organism
Word: being
```

Recall that the resulting XML files contain pointers to the other files. These numbers are the incremented `$id`; the script created an index of offset-to-ID mappings. Most XPath implementations provide a standard `document()` function, which enables the processor to dynamically load XML files. For example, this XPath queried against a node takes the first `/Synset/Pointers/Hyponym` element,

concatenates it with ".xml" to get the filename for the record, opens the file with the document() function, and selects the /Synset/Word node:

```
document(concat(/Synset/Pointers/Hyponym[1], ".xml"))/Synset/Word
```

This XPath expression placed in place of /Synset/Word in the previous code example causes the script to output this result:

```
Word: benthos
```

Thus, the first hyponym or "kind of" organism type listed in the lexicon is benthos, which is an organism at the bottom of the sea. Other hyponym pointers in 12.xml include plant, plankton, parasite, mutant, and animal (the lexicon section in which human is located).

Other XML Technologies

XML dialects have been developed (and often standardized) for nearly every conceivable field, including genealogy, astronomy research, and even music. Each dialect, although it is valid XML, requires that applications know and understand that dialect—usually as it is described via an XML schema.

XSLT

Translating from one XML dialect to another is useful in many situations, not least of all in transforming XML to HTML. XSLT is a very popular technology for doing this. XSLT is actually written in XML, meaning that it can process and output itself! XSLT templates are *declarative*. Instead of a procedural language that tells the program what to do and in what order, declarative languages create functions with conditions. If you imagine that your entire program consists of procedural case statements, you aren't far from declarative programming. Declarative programming can be especially useful when dealing with semantically rich data such as XML. For example, here is the baseball XML source, file 14861.xml:

```
<Synset fileVersion="1.0" pos="n">
    <Id>14861</Id>
    <WnOffset version="2.1" pos="n">02772480</WnOffset>
    <LexFileNum>06</LexFileNum>
    <SsType>n</SsType>
    <Word lexId="0">baseball</Word>
    <Pointers>
        <Hypernym>14746</Hypernym>
        <Hypernym>14866</Hypernym>
    </Pointers>
    <Gloss>a ball used in playing baseball</Gloss>
</Synset>
```

■ **Note** At this time. XSLT 2.0 is approaching a final recommendation. Note that this section describes XSLT 1.0, and the specification is changing significantly in the new version.

An XSLT document could transform this into another XML dialect, as well as a non-XML format. Rules get created for each piece of data that needs to be transformed. For example, suppose

that you want to output an HTML page to display this and every other `synset` XML file. Each piece gets its own XSLT template in the following form:

```
<xsl:template match="/Synset">
    <p>This is synset number <xsl:value-of select="Id"/>.</p>
</xsl:template>
```

Notice that the HTML elements have no namespace, but the XSLT elements all use the `xsl` namespace prefix. All templates get enclosed in a single stylesheet element:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:template match="/Synset">
        <p>This is synset number <xsl:value-of select="Id"/>.</p>
    </xsl:tempate>
</xsl:stylesheet>
```

If you save this stylesheet as `synset.xsl` and execute the transformation, it outputs this result:

```
<?xml version="1.0"?>
<p>This is synset number 14861.</p>
```

■ **Note** You can execute XSLT transformations from the command line with `libxslt` installed. The utility is called `xsltproc` and its syntax is as follows:

```
$ xsltproc synset.xsl 14861.xml
```

Notice the attributes on the `<xsl:template/>` and `<xsl:value-of/>` elements in the template. One has a `match` attribute, and the other has `select`—both with an XPath expression for value. The `match` attribute registers the template with the XSLT engine and says, "As you parse the XML file, when you encounter a node that matches XPath `/Synset`, use this template." The `select` attribute on `<xsl:value-of/>` tells the engine to output the value of the selected node, using the current node as context. The current node within this template is `/Synset`, so the selected `Id` is its child element.

You could achieve the same output with two templates instead of one:

```
<xsl:template match="/Synset">
    <xsl:apply-templates select="Id"/>
</xsl:template>

<xsl:template match="Id">
    <p>This is synset number <xsl:value-of select="."/>.</p>
</xsl:tempate>
```

The `<xsl:apply-templates/>` element tells the processor to keep going—to continue to apply templates to the children of the current node (or just the child indicated by its `select` attribute). All the `Id` template does here is output the value of the current node, which is the `Id` element in the second template.

Without explaining every XSLT element, Listing A-4 shows the file `synset.xsl` filled out with several templates. It also shows some of the power of XPath used within a stylesheet.

Listing A-4. *XSLT Stylesheet* `synset.xsl`

```
<xsl:template match="Gloss">
    <p><b>Gloss:</b> <xsl:value-of select="."/></p>
</xsl:template>
```

```
<xsl:template match="Pointers">
    <p>Synset has the following pointers:</p>
    <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="Hypernym">
    <p>This is a kind of
        <a href="{concat(., '.xml')}">
            <xsl:value-of select="document(concat(., '.xml'))/Synset/Word"/
>.
        </a>
    </p>
</xsl:template>

<xsl:template match="*">
    <!-- catch the rest of the elements and put them in comments -->
    <xsl:comment><xsl:value-of select="."/></xsl:comment>
</xsl:template>

</xsl:stylesheet>
```

Applied to `14861.xml`, the output is the following:

```
<html><body>
  <p>This is synset number 14861.</p>
  <!--02772480-->
  <!--06-->
  <!--n-->
  <p><b>Word:</b>baseball</p>
  <p>Synset has the following pointers:</p>
    <p>This is a kind of
        <a href="14746.xml">ball.
        </a></p>
    <p>This is a kind of
        <a href="14866.xml">baseball equipment.
        </a></p>
  </p>
  <p><b>Gloss:</b>a ball used in playing baseball  </p>
</body></html>
```

Viewed in a web browser, this listing displays details on the `synset` and also enables the user to navigate to the hypernym `synset` files. Setting up a handler in your web server to apply stylesheets to an associated XSLT stylesheet is trivial, and some web browsers can do this transformation all on their own.

A Note on Current Node

Within applications such as XSLT and where processing is context-aware (for example, inside a predicate), it is often necessary to refer explicitly to the current node. The solution is to use the XSLT XPath function `current()`. In most cases, `current()` and `.` are equivalent:

```
<xsl:value-of select="current()"/>
<xsl:value-of select="."/>
```

However, the current node is usually different from the context node within a predicate. Consider the case when, inside a template, you want to select a node using a value from your context. For example, let's say that you are inside a template processing a `Word` element and you want to know whether the `Gloss` for this record contains that word:

```
<xsl:template match="Word">
    <xsl:value-of select="../Gloss[contains(., .)]"/>
</xsl:template>
```

Using `.` twice is obviously incorrect. In this case, the `contains()` function is getting `Gloss` as both arguments, meaning it will always be a true test (`Gloss` contains itself). You want `Gloss` to be the first argument, but the current node, `Word`, to be the second. The `current()` function does this, which always returns the current node:

```
<xsl:template match="Word">
    <xsl:value-of select="../Gloss[contains(., current())]"/>
</xsl:template>
```

Because `current()` returns a node set (albeit with only one member), it can precede a path (as with `current()/Word/text()`) or be used alone to get just the current node.

SAX

Simple API for XML (SAX) is a standard for parsing XML. A SAX processor does not have to parse the entire file or store it in memory. SAX is event-driven, so a parser triggers functions each time it encounters a new element, attribute, and so on. As it happens, most DOM implementations use a SAX parser to generate a node tree. Thus, SAX can be considered a lower-level parse interface, which is the reason why it is faster and more efficient than DOM.

RPC-XML and SOAP

Remote procedure calls (RPCs) enable software to request information and perform operations non-locally. If you aren't familiar with RPCs and have the need to allow a client's website to submit orders to your website without human intervention, for example, you might write a script to run on the client's website that accepts a posted order form, connects to your website, fills in the form, and submits the form. Consider the problems with this process. If the web page or form input names ever change (which they are likely to do), your client's orders will no longer work and might get lost. And using a user interface from a program just to move data from one place to another is horribly inelegant.

A better way to move data to—and request actions from—another system is to use RPCs. Most programming languages have RPC-XML or Simple Object Access Protocol (SOAP) modules. If you will be providing the functionality, you write a response function to run on the server. If you need to access the server, you write a requestor function. Using one of these modules, you don't need to write any code to connect to the server, perform the request, or parse the response because it is all handled automatically. In fact, a developer working with an interface that uses RPC-XML or SOAP might have no idea that a given method call is occurring remotely at all. Functionality that is available via an RPC server is often referred to as a *Web Service* because it is accessed over HTTP and provides information or function services to other software programs.

RPC-XML is an example of a technology in which the XML is invisible to the developer. You might not even know that the function calls are exchanging XML. The point of RPCs is that you don't *need* to know how data is being moved between systems, enabling you to focus instead on the functionality. XML as a standard is the perfect format for transmitting RPC data because the methods are often operating between programs in different programming languages or operating systems.

Because all can process XML, they all understand the data. You will find this to be true with much of XML: after programs are enabled to process the data autonomously, developers can worry less about minutiae such as data formats. XML serves no end in and of itself; it simply makes the formatting and parsing of data transparent to software built on top of it.

Conclusion

From a simple foundation of tags built from greater-than and less-than signs, XML has standardized data formats, enabling dozens or hundreds of technologies to be designed and standardized. In many ways, Berkeley DB XML represents a major culmination of these technologies, providing everyone the capability to manage huge collections of XML files. The power of this capability won't be clear until you delve into BDB XML itself.