

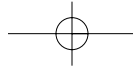
Chapter 2

XML OVERVIEW

When looking for a way to express the SOAP payload, the authors of the specification had a number of ways they could have gone. They could have invented their own protocol, declared that CORBA or DCOM would now be known as SOAP, or invented something new by combining existing technologies. In the end, they chose to minimize the amount of required invention by combining existing technologies. To express the content of a SOAP message, the authors chose the *eXtensible Markup Language*, XML.

XML contains a large number of features—far more than SOAP uses or needs. For example, the SOAP specification states, “A SOAP message **MUST NOT** contain a Document Type Declaration. A SOAP message **MUST NOT** contain Processing Instructions.”¹ Of the XML standards that SOAP has adopted, it specifies how that feature will be used. You will see this in Chapter 3 when looking at SOAP serialization. As we will see later, this decision makes it fairly easy to implement solutions using SOAP because developers do not need to have a full-fledged XML parser to use SOAP. To understand SOAP, we need to understand the following items first:

¹ SOAP 1.1 specification, section 3, “Relation to XML.”



24 Chapter 2 XML Overview

- Uniform Resource Identifiers (URIs)
- XML basics
- XML schemas
- XML namespaces
- XML attributes

Uniform Resource Identifiers

To access a unique item over the Internet, you need to know how to identify that one object among everything else out there. URIs provide a way of uniquely identifying those many different items. Described in detail by Request for Comments (RFC) 1630, this specification spells out the rules used to use many different protocols within the URI framework. A URI has the form

```
<scheme>:<scheme-specific-part>
```

When the `scheme-specific-part` contains slashes (/), those slashes indicate some hierarchical structure within the path.

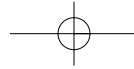
Uniform Resource Locators

The best-known type of URI is the Uniform Resource Locator, or URL. Like all URIs, a URL follows the `<scheme>:<scheme-specific-part>` method of addressing. Table 2-1 identifies the schemes named by RFC 1738 and RFC 1808.² Using these schemes, we can connect to various places on the Web using nothing but a URL translator such as Microsoft Internet Explorer or Netscape Navigator. URLs define this layout for the `scheme-specific-part`:

```
//<user>:<password>@<host>:<port>/<url-path>
```

If you are at all familiar with URLs, you know that a good number of the items in this layout are optional. More often than not, you type in URLs such as:

² You can obtain these and other RFCs from <ftp://ftp.ietf.org/rfc>.



| |
|---|
| Table 2–1 Currently available URL Schemes. |
|---|

| <i>Scheme Name</i> | <i>Description</i> |
|--------------------|-----------------------------------|
| ftp | File Transfer Protocol |
| http | Hypertext Transfer Protocol |
| gopher | The Gopher Protocol |
| mailto | Electronic mail address |
| news | USENET news |
| nntp | USENET news using NNTP access |
| telnet | Reference to interactive sessions |
| wais | Wide Area Information Servers |
| prospero | Prospero Directory Service |

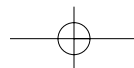
http://www.scottseely.com (my Web site) or
ftp://ftp.scottseely.com (my FTP site)

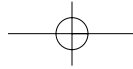
The various parts of the scheme syntax identify the following elements:

- **user:** User name at the target location (optional).
- **password:** The password assigned to user (optional).
- **host:** The Internet Protocol (IP) address or fully qualified domain name of a network host (required).
- **port:** Identifies the port to use when establishing a connection. Most protocols identify a default port number. For example, HTTP uses port 80 by default (optional).
- **url-path:** Contains details of how to access the specified resource. The / immediately after the host or port is not a part of the url-path.

Uniform Resource Names

Uniform Resource Names (URNs) are much less familiar to the average Web user than the ubiquitous URL. Unlike a URL, a URN does not resolve to a unique, physical location. URNs serve as persistent resource identifiers.





26 Chapter 2 XML Overview

They allow other collections of identifiers from one namespace to be mapped into URN space. Because of this requirement, the URN syntax provides the ability to pass and encode character data using existing protocols. RFC 2141 defines how to create and use a URN. The production for a URN follows the general rules for a URI. In general, it looks like this:

```
<URN> ::= "urn:" <NID> ":" <NSS>
```

A URN uses the string "urn:" to identify the scheme. NID specifies the Namespace ID and NSS specifies the Namespace-Specific String. When interpreting URNs we look to the NID to tell us how to interpret the NSS. When reading or creating a URN, the initial construct "urn:" <NID> is case-insensitive.

URLs and URNs represent two common uses for a URI. In the next section we see yet another use of URIs: XML Namespaces.

XML Basics

When XML first hit and the trade press began reviewing it in the 1996–1997 time frame, I dug around looking for examples of what XML looked like. I was surprised at how many industry wonks were saying that it was the next big thing but then would not (or could not) show what this markup language looked like. Given the hype and lack of examples, I imagined it to be a fairly complex, ornery beast. After a few months of hype, developers began writing articles on the topic, giving out the details I wanted. Some of these articles described it as a descendent of Standard Generalized Markup Language (SGML), only better suited for development. How was it made to work better in the program development area? SGML offers extraordinary levels of flexibility but makes it very difficult to implement a full-featured SGML parser. XML more or less defines a concrete set of rules that readers and writers of XML data must follow. Because the language definition for XML is more rigid, it is easier to create conforming documents and parsers. Do not get the wrong idea—XML is a subset of SGML. Anyhow, after the digerati calmed down and the developers got their chance to speak up I got really excited. Why? I finally saw some practical applications of XML. It works as a data language that both machines and people can easily understand. If you have ever read or written Hypertext Markup Language (HTML), you will find XML fairly easy to understand and use. Like HTML, it contains begin tags and end

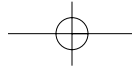
tags. Unlike HTML, every begin tag must have a matching end tag. End tags look like their matching begin tag with a leading /. Let's jump in and take a look at what XML can look like.

The following XML shows one way of encoding the contents of a library:

```
<?xml version="1.0">
<Library>
  <Book>
    <Title>Green Eggs and Ham</Title>
    <Author>Dr. Seuss</Author>
  </Book>
  <Book>
    <Title>Windows Shell Programming</Title>
    <Author>Scott Seely</Author>
  </Book>
  <Picture>
    <Title>American Gothic</Title>
    <Artist>Grant Wood</Artist>
  </Picture>
</Library>
```

Even if you have never read XML in your life, this example makes a fair amount of sense. The document demonstrates a number of the rules found in an XML document. The first line in the sample is a processing instruction declaring the version of XML used by the document. Documents do not have to include this element, but normally you should include it. All XML documents must have one enclosing element (the version information does not count as an enclosing element). The `Library` element wraps the entire document in this case. It contains three subelements: two books and a picture. As you may guess, not one word in the XML document shown is an XML keyword. If you want to be a free-wheeling XML author, all you need to do is watch the spelling in your tag names and make sure that every begin tag has an end tag. Writing XML documents this way can cause problems. For example, you could accidentally write this:

```
<Library>
  <Book>
    <Title>Green Eggs and Ham</Title>
    <Author>Dr. Seuss</Author>
  </Book>
  <Bokk>
    <Title>Windows Shell Programming</Title>
    <Author>Scott Seely</Author>
  </Bokk>
  <Picture>
```



28 Chapter 2 XML Overview

```
<Title>American Gothic</Title>
<Artist>Grant Wood</Artist>
</Picture>
</Library>
```

As a human reader, you recognize that the author of the document misspelled “Book” for the book *Windows Shell Programming*. Likewise, the parser will accept the document but it will not realize that you have two books in the library list. Instead, it will think you have one Book, one Bokk, and one Picture. If you want the XML parser to do some checking for you and only read valid constructs, you can use something called a Document Type Declaration (DTD) or an XML Schema. DTDs are not covered in this book because Section 3 of the SOAP specification specifies that a SOAP message “MUST NOT contain a Document Type Declaration.” If you really must know how to use DTDs, see the recommended reading list at the end of the chapter. With a few exceptions (i.e., publishing, document management, etc.), you should always use an XML Schema to describe data.

XML Schemas

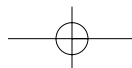
An XML Schema provides a superset of the capabilities found in a DTD. They both provide a method for specifying the structure of an XML element. Whereas both schemas and DTDs allow for element definitions, only schemas allow you to specify type information. All XML data is character based. It will specify a four as the character 4, rarely as the binary representation 0100.³ We can enhance the library example to demonstrate the benefits of schemas over DTDs, adding copyright date to the book information.

A simple DTD has elements that contain other elements or character data. The simplest element declaration would declare the element name and the contents as character data:

```
<!ELEMENT element-name (#PCDATA)>
```

An element may also consist of other elements. If an element contains exactly one instance of a given element, we would have the following DTD:

³ Yes, XML does allow for encoding binary data within the message. This method allows us to send things such as image data inside of an XML message.



```
<!ELEMENT parentElement (childElement)>
<!ELEMENT childElement (#PCDATA)>
```

Alternatively, the `parentElement` might contain zero or more `childElements`. We indicate this using an asterisk, `*`:

```
<!ELEMENT parentElement (childElement*)>
<!ELEMENT childElement (#PCDATA)>
```

Finally, you can also indicate composition of elements in a DTD. For example, `parentElement` might contain two different pieces of data:

```
<!ELEMENT parentElement (childElem1, childElem2)>
<!ELEMENT childElem1 (#PCDATA)>
<!ELEMENT childElem2 (#PCDATA)>
```

If we wanted to generate a DTD for a library of books, it might look like this:

```
<!ELEMENT Library (Book*)>
<!ELEMENT Book ( Title, Author*, Copyright )>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Copyright (#PCDATA)>
```

The `Library` consists of zero or more elements of type `Book`. Each `Book` has a `Title`, zero or more elements of type `Author`, and a `Copyright`. The `Title`, `Author`, and `Copyright` elements all contain character data. Rewriting the library example to use the DTD, we have the following XML document:

```
<?xml version="1.0" ?>
<!DOCTYPE Library PUBLIC "." "Library.dtd" >
<Library>
  <Book>
    <Title>Green Eggs and Ham</Title>
    <Author>Dr. Seuss</Author>
    <Copyright>1957</Copyright>
  </Book>
  <Book>
    <Title>Windows Shell Programming</Title>
    <Author>Scott Seely</Author>
    <Copyright>2000</Copyright>
  </Book>
</Library>
```

30 Chapter 2 XML Overview

A validating parser will load `Library.dtd` and use it to validate the contents of the document. This is all well and good, but wouldn't it be nice if we could specify more information than "this element contains character data"? You see, DTDs come from SGML. SGML primarily concerned itself with document publishing. As such, the print industry has been using it for years, because SGML provided ways to reproduce the same document in many different forms. Now that computing has embraced XML, programmer types (i.e., you and me) wanted a way to express the characteristics of the data. A DTD can specify the number of instances of a piece of data and what a particular structure looks like. By extending an SGML dialect, I could even specify the characteristics of the data. The problem here is that every developer may come up with a different naming system. I also have a gripe with DTDs—they do not look like XML. For these and other reasons, the World Wide Web Consortium (W3C) (www.w3c.org) eventually published the XML Schema recommendation. Here is the `Library DTD` defined as a schema:

```
<Schema xmlns:xsd
  "http://www.w3.org/2001/XMLSchema"
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance">
  <complexType name="Book" content="mixed">
    <element type="Title"></element>
    <element type="Author"></element>
    <element type="Copyright"></element>
  </complexType>
  <simpleType name="Title" content="textOnly"
    xsi:type="string">
  </simpleType>
  <simpleType name="Author" content="textOnly"
    xsi:type="string">
  </simpleType>
  <simpleType name="Copyright"
    content="textOnly" xsi:type="integer">
  </simpleType>
</Schema>
```

You would save this as an XML file. To use the schema, simply reference it in your document like so:

```
<myLibrary:Library xmlns:myLibrary=
  "x-schema:http://www.scottseely.com/LibrarySchema.xml">
  <myLibrary:Book>
```

```
<myLibrary:Title>Green Eggs and Ham
</myLibrary:Title>
<myLibrary:Author>Dr. Seuss
</myLibrary:Author>
<myLibrary:Copyright>1957
</myLibrary:Copyright>
</myLibrary:Book>
<myLibrary:Book>
  <myLibrary:Title>Windows Shell Programming
  </myLibrary:Title>
  <myLibrary:Author>Scott Seely
  </myLibrary:Author>
  <myLibrary:Copyright>2000
  </myLibrary:Copyright>
</myLibrary:Book>
</myLibrary:Library>
```

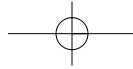
Both the schema and the document use the text `xmlns`. This string tells the parser to use the set of names specified by the namespace identified by the indicated URI. If the scheme of the URI is `x-schema` then the parser must load the schema at the specified address. All elements inside the tag using the `xmlns` declaration are part of the enclosing namespace unless otherwise specified.

Facets

To aid with the definition and validation of data, an XML Schema uses facets to define characteristics of a specific datatype. A *facet* defines an aspect of a value space. A *value space* is the set of all valid values for a given datatype. You use a facet to distinguish what makes one datatype different from another. The XML schema document specifies two types of facets: fundamental and nonfundamental facets.

A fundamental facet is an abstract property that characterizes the values of a value space. These include the following facets:

- **Equal:** Defines the notion of two values of the same datatype being equal. The following rules apply to this concept:
 1. For any two values (*a*, *b*), *a* is equal to *b* (denoted *a=b*) or *a* is not equal to *b* (*a!=b*).
 2. No pair of values (*a*, *b*) exists such that *a=b* and *a!=b*.

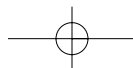


32 Chapter 2 XML Overview

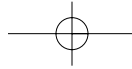
3. For every valid value a , $a=a$.
 4. For any two values (a, b) in the value space, $a=b$ if and only if $b=a$.
 5. For any three valid values (a, b, c) , if $a=b$ and $b=c$ then $a=c$.
- **Order:** This specifies a mathematical relation to set the total order of members in the value space. For every pair of values (a, b) , their relationship is either $a < b$, $b < a$, or $a = b$. For every triple (a, b, c) , if $a < b$ and $b < c$ then $a < c$.
 - **Bounds:** This simply states that a given value space may be *bounded above* or *bounded below*. If a value U exists such that for all values v in the value space the statement $v \leq U$ is true, U represents the upper bound of the value space (bounded above). If a value L exists such that for all values v in the value space the statement $v \geq L$ is true, L represents the lower bound of the value space (bounded below). If the datatype has both an upper and lower bound, then that datatype is bounded.
 - **Cardinality:** Some value spaces have a finite set of values. Others have an unlimited set of values. A datatype has the cardinality of the value space, which is either “finite” or “countable infinite.”
 - **Numeric:** If the values of the datatype are quantities in any mathematical number system, then the datatype is numeric. Everything else is nonnumeric.

The nonfundamental or constraining facets are optional properties that you can apply to a datatype to constrain its value space. The following facets do this for you:

- **length:** This facet has a different meaning depending on the base type. If the type derives from string, length measures units of Unicode code points (i.e., characters). For binary datatypes this facet is measured in octets (8 bits) of binary data. List datatypes (e.g., NMTOKENS, IDREFS, etc.) use this facet to indicate the number of list items.



- **minLength:** Sets the minimum number of units of length. The value of the facet must be a `nonNegativeInteger`.
- **maxLength:** Sets the maximum number of units of length. The value of the facet must be a `nonNegativeInteger`.
- **pattern:** This constrains the value space to values that match a regular expression defined by the `pattern` facet.
- **enumeration:** Specifies a value space by setting a set of values. This does not impose order on the created value space. Order is imposed on the enumeration's base type.
- **maxInclusive:** States the upper bound for an ordered datatype. Inclusive means that the upper bound is also in the value space. For an upper bound U , all values v must be $v \leq U$.
- **maxExclusive:** States the upper bound for an ordered datatype. Exclusive means that the upper bound is not in the value space. For an upper bound U , all values v must be $v < U$.
- **minInclusive:** States the lower bound for an ordered datatype. Inclusive means that the lower bound is also in the value space. For a lower bound L , all values v must be $v \geq L$.
- **minExclusive:** States the lower bound for an ordered datatype. Exclusive means that the lower bound is not in the value space. For a lower bound L , all values v must be $v > L$.
- **precision:** Used for value types derived from `decimal`, this facet defines the maximum number of decimal digits. Its value must be a `positiveInteger`.
- **scale:** Used for value types derived from `decimal`, this facet defines the maximum number of decimal digits in the fractional part of the value. Its value must be a `positiveInteger`.
- **encoding:** Used to form the lexical space for datatypes derived from binary. Its value must be either `hex` or `base64`. If the value is `hex`, the value consists of the two hexadecimal digits needed to represent the octet code. For example, "20" is the hex value for the US-ASCII space character. If the value is `base64`, the binary stream must use the Base64 Content-TransferEncoding defined in Section 6.8 of RFC 2045.
- **duration:** Set of values for datatypes derived from `recurringDuration`. Its value must be a `timeDuration`.



34 Chapter 2 XML Overview

- **period:** Set of values for datatypes used to define the period for datatypes derived from `recurringDuration`. Its value must be a `timeDuration`.

Using all of these facets you can constrain existing datatypes. This helps perform tasks such as data validation and verifying the overall “correctness” of an XML document.

Datatypes

Combined with facets, the XML Schema datatypes can help you give meaning to the items contained by your schema. For a complete listing of the datatypes specified by <http://www.w3.org/2001/XMLSchema>, go to <http://www.w3.org/TR/XMLSchema-2>.

XML Namespaces

We already saw these in use in the last section on XML Schemas. Simply put, namespaces define a set of unique names within a given context. A namespace can use any URN as long as that URN is unique. For example, the preceding schema defined the namespace `myLibrary`. The schema contained in the file `LibrarySchema.xml` is in the same directory as the source page and uniquely identifies the namespace.

What does a namespace *do* for us? It allows us to create multiple elements with the same name (e.g., `postOffice:address` and `memory:address`). Putting these similar structures into unique namespaces helps prevent the concepts from clashing with each other and allows the computer to unequivocally determine which structure is being referenced. This same practice exists in C++, Java, C#, and a number of other languages. A number of arguments exist both for and against namespaces. Many of the arguments against namespaces boil down to the idea that namespaces are a solution in search of a problem. The arguments for them state that developers are better off when they do not have to rearchitect an application because someone else used a function with the same name. With regards to the C language and pre-standardized C++, people avoided collisions with things such as standard library functions all the time. For better or worse, these same people also had to avoid collisions with names of functions supplied by various vendors.

Often, the code supplied by these vendors would clash with functions written by the developer. In Java, the location of the package often defines the namespace. For example, you can create two classes named Foo and differentiate them by putting them in different packages (`com.scottseely.foo` is different from `com.prenticehall.foo`). This is a bit off the topic, but here is an example of how namespaces work in C++.

Developer code:

```
#include "someVendorHeader.h"

void someFunc()
{
    // Code to do something
}
```

Inside of the vendor's header file, they have a function with the same signature as `someFunc()`, which means that the code will not compile. To fix this, the programmer can write this:

```
#include "someVendorHeader.h"

namespace myFuncs
{
    void someFunc()
    {
        // Code to do something, even call the
        // vendor's function!- :: says to use the
        // function in the global namespace.
        ::someFunc();
    }
}
```

Problem solved! With the various DTDs and schemas being created, the creators of XML namespaces figured that they could learn from others and include similar functionality. Let's look back at the schema example and the lines that define the namespace:

```
<Library xmlns:myLibrary=
    "x-schema:LibrarySchema.xml">
```

Regardless of the name of the schema in `LibrarySchema.xml`, the enclosing namespace is named `myLibrary`. The namespace could have easily been called `x`, `Bob`, or `yth443`. By using a namespace, we make it possible

36 Chapter 2 XML Overview

to use many different schemas that define `Book`. Imagine that you are an online bookseller. All of your vendors ship you their catalogs via XML. Each vendor defines the `Book` element slightly differently. Because there is no standardization, you have to read in the various catalogs and normalize the data for your database. Namespaces can help you do this by putting each `Book` definition into a uniquely identified namespace.⁴

Namespaces also come in handy for creating self-documenting XML. If you are using schema from many different sources, using namespaces will help the human reader know where the various bits of data came from. Within an XML document, a namespace remains active for the element declaring it and all elements contained by the declarer. Likewise, if an inner element declares a different namespace, then all of its inner elements use the new namespace. To see this, consider the following example. Elements in the outer namespace are displayed using regular characters and the inner namespace is in *italics*.

```
<outer:library xmlns:outer=
  "http://www.scottseely.com/library">
  <book>
    <title>The XML Handbook</title>
    <author>Charles F. Goldfarb</author>
    <author>Paul Prescod</author>
  </book>
  <inner:book
    xmlns:inner="http://www.phptr.com/book">
    <title>Windows Shell Programming</title>
    <writer>Scott Seely</writer>
  </inner:book>
</outer:library>
```

These scoping rules help by reducing the verbosity of the XML document. An XML document may also mix and match namespaces within a single element. The scoping rules just outlined still apply—they just seem to get a bit more complex. Consider this example that matches up a book with some Library of Congress information:

```
<lib:library xmlns:lib=
  "http://www.scottseely.com/lib"
  xmlns:LOC=
    "http://www.libraryofcongress.gov/book">
```

⁴Other examples abound. You could use namespaces to aggregate job databases, stock market data, or cooking recipes.

```
<book>
  <title>The XML Handbook</title>
  <author>Charles F. Goldfarb</author>
  <author>Paul Prescod</author>
  <LOC:ISBN>0-13-014714-1</LOC:ISBN>
</book>
</lib:library>
```

In the preceding example, the elements `book`, `author`, and `library` are all part of the `lib` namespace. `ISBN` exists as a part of the `LOC` namespace. The actual URNs used in the `xmlns` attribute declaration do not necessarily have to exist unless the URN specifies a schema:

```
<LIB:Library xmlns:LIB=
  "x-schema:www.scottseely.com/lib/lib.xml">
```

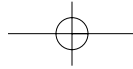
Namespaces combined with schemas provide some great opportunities for document validation and ease of readability. They can point to a schema definition if one exists, but they do not have to. A namespace can be used to simply make the named element unique within the XML document.

XML Attributes

All of the XML documents presented in this chapter have used elements to present data. XML also supports attributes. We saw these used as facets within the description of XML Schema. As stated earlier, elements require begin and end tags. Attributes do not. Instead, they are contained by the begin tag of an element. A given element can have one or more elements of the same type. It can only have one attribute of any given type. The following XML is legal:

```
<Library>
  <Book title="Windows Shell Programming">
    <Author>Scott Seely</Author>
  </Book>
</Library>
```

The `Book` element has an attribute, `title`, which gives the title of the book. The XML expresses `Author` as a subelement. This could have easily been expressed as another attribute and been 100% valid.



38 Chapter 2 XML Overview

```
<Library>
  <Book title="Windows Shell Programming"
        author="Scott Seely" />
</Library>
```

How would you express a book with more than one author? You could try this:

```
<Library>
  <Book title="The XML Handbook"
        author="Charles F. Goldfarb"
        author="Paul Prescod">
  </Book>
</Library>
```

As I mentioned already, this fragment is invalid. You cannot have two attributes with the same name. You could achieve a similar effect by writing this:

```
<Library>
  <Book title="The XML Handbook">
    <author name="Charles F. Goldfarb" />
    <author name="Paul Prescod" />
  </Book>
</Library>
```

The author element uses a name attribute to contain the names of any writers associated with the Book. Because these are empty elements, the fragment uses the empty element notation: “/”>. Attributes can be declared in three different ways:

1. Well-formed XML with no DTD or schema
2. Well-formed XML using a DTD
3. Well-formed XML using schema

The preceding examples use Option 1. This works well for learning but poorly for production environments. As mentioned earlier, SOAP forbids the use of DTDs, so we will not investigate that option. This leaves us with Option 3, schema. When creating attributes for an XML Schema, you use the `attributeType` and `attribute` keywords. These words only have meaning within the schema namespace. You use `attributeType` to define characteristics of the type. `attribute` includes the item within an element type definition. To create the book example using attributes, the schema would look something like this:

```

<Schema xmlns:xsd
  "http://www.w3.org/2001/XMLSchema"
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance">
  <attributeType name="title"
    xsi:type="string" />
  <attributeType name="name" xsi:type="string" />
  <complexType name="Author" content="empty">
    <attribute type="name" />
  </complexType>
  <complexType name="Book" content="eltOnly">
    <attribute type="title" />
    <element type="Author" />
  </complexType>
</Schema>

```

Looking at the `title` `attributeType` definition, we see that it specifies the datatype (`string`) and the name of the element. Fairly easy, right? The full syntax for an `attributeType` is:

```

<attributeType
  default="default value"
  xsi:type="type"
  xsi:values="enumerated values"
  name="idref"
  required="{yes | no }" >

```

- **default:** The default value for the attribute. This value must be legal. For example, enumerations can only use elements in the enumeration as the default value.
- **xsi:type:** The data type for the attribute. If enumeration is selected, `xsi:values` must be filled in.
- **name:** Identifies the attribute type. The `attributeType` must have a name to be valid.
- **required:** Indicates if the attribute must be present for any elements including the `attributeType`.

For an example using all the fields, let's add a new attribute to the `myBook` schema, `format`.

```

<Schema xmlns:xsd
  "http://www.w3.org/2001/XMLSchema"
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance">

```

40 Chapter 2 XML Overview

```
<attributeType name="title"
  xsi:type="string" />
<attributeType name="name" xsi:type="string" />
<attributeType name="format"
  default="soft-cover" nullable="0">
  <enumeration value="soft-cover" />
  <enumeration value="hard-cover" />
</attributeType>
<complexType name="Author" content="empty">
  <attribute type="name" />
</complexType>
<complexType name="Book" content="eltOnly">
  <attribute type="title" />
  <attribute type="format" />
  <element type="Author" />
</complexType>
</Schema>
```

Using this schema for one title, we would have the following XML:

```
<lib:Book xmlns:lib=
  "x-schema:www.scottseely.com/BookSchema.xml"
  title="The XML Handbook">
  <author name="Charles F. Goldfarb" />
  <author name="Paul Prescod" />
</lib:Book>
```

If a program requested the `format` attribute from the `Book` element, it should get back the value `soft-cover`. Viewing this XML document in Internet Explorer (IE) 5.5 yields the data shown in Figure 2-1. IE does not

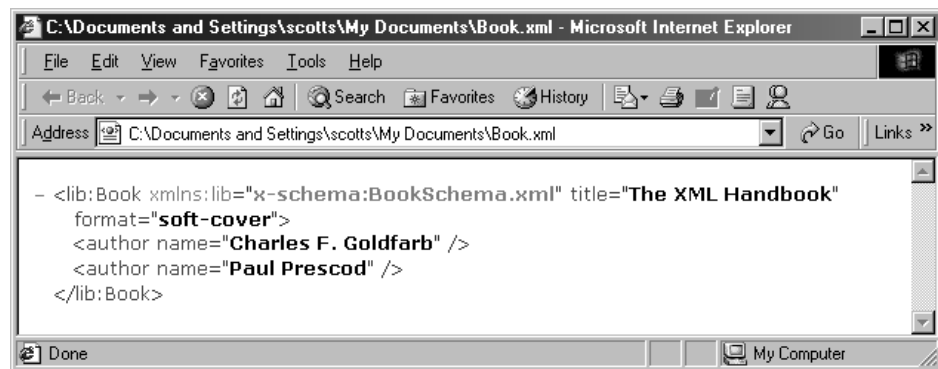
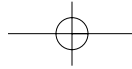


Figure 2-1 Using Microsoft Internet Explorer to view XML documents.



flag invalid data, but it does flag properly (and improperly) structured data. For example, you could set the format attribute to “stone-tablet” and IE would still display the document.

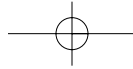
Summary

This chapter presented just enough information to make SOAP accessible to you. Many Internet technologies use URIs to express locations and other concepts. You must understand how these are formed and what they mean to appreciate their usefulness when used by other markup languages and protocols. After discussing the basics, we took a quick look at XML. Since this language came onto the scene in late 1997, many new ideas have been layered on top of it. Besides XML Schemas and Namespaces we have also seen other technologies layered on top of XML. Among the proposals winding their way through the W3C approval process are the following:

- **XML Style Language (XSL):** Specifies a way of converting documents from one format to another. XSL lets you convert documents between various schemas, generate text files, or create an HTML view of the data.
- **XML Schema:** An improvement over DTDs that allows the author to specify data types, maximum and minimum values, enumerations, and other items.
- **XPointer:** An extension and customization of XPath.⁵
- **XLink:** Allows XML authors the ability to establish relations within documents as well as between them. For example, XPath is used within XSL to specify which element to transform.
- **XML Query Language:** Allows an external entity to query an XML document for specific data.

At the request of my reviewers, I have to point out that these synopses are very limited descriptions of all you can do with the various W3C recommendations and their related implementations. Many of the specifications are

⁵ XPath is already a W3C recommendation. In W3C jargon, “recommendation” refers to the accepted, ratified standard.



42 Chapter 2 XML Overview

fairly long. I would recommend visiting www.w3c.org to read the current overviews of the various technologies if one looks interesting to you.

Of course, there are many other ideas related to XML winding their way through the standards process, such as SOAP. While working with SOAP, you will find it handy to have XML reference material handy. I went through much effort to make sure this book stands on its own. Still, it is hard to cover all of XML in a chapter. Fortunately, a lot of good books exist. The best all-around book on the market that I have found is *The XML Handbook* by Charles F. Goldfarb and Paul Prescod. Goldfarb has been involved with SGML (and consequently XML) since its inception. If you have the financial resources you should also purchase the *XML Developers Toolkit*, which contains three books at a reduced price. These truly are the best books I own regarding XML and I went through several before I found these.

At this point you should understand enough about XML to make the SOAP specification readable. Let's get moving and cover the specification!

