Building Web Services with Java

MAKING SENSE OF XML, SOAP, WSDL, AND UDDI

Second Edition

Steve Graham
Doug Davis
Simeon Simeonov
Glen Daniels
Peter Brittenham
Yuichi Nakamura
Paul Fremantle
Dieter König
Claudia Zentner

DEVELOPER'S LIBRARY

Sams Publishing, 800 East 96th Street, Indianapolis, Indiana 46240

Building Web Services with Java, Second Edition

Copyright © 2005 by Sams Publishing

International Standard Book Number: 0-672-32641-8

Library of Congress Catalog Card Number: 2004091343

Printed in the United States of America

First Printing: July 2004

07 06 05 4

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

Associate Publisher
Michael Stephens

Acquisitions Editor
Todd Green

Development Editor
Tiffany Taylor

Managing Editor
Charlotte Clapp

Senior Project Editor
Matthew Purcell

Indexer
Larry Sweazy

Proofreader
Eileen Dennie

Technical Editors
Alan Moffet
Alan Wexelblat
Marc Goldford
Kunal Mittal

Publishing Coordinator
Cindy Teeters

Designer
Gary Adair

2
XML Primer

SINCE ITS INTRODUCTION IN 1998, Extensible Markup Language (XML) has revolutionized how we think about structuring, describing, and exchanging information. The ways in which XML is used in the software industry are many and growing. Certainly for Web services the importance of XML is paramount; all key Web service technologies are based on it.

One great thing about XML is that it's constantly changing and evolving. However, this can also be its downside. New problems require new approaches and uses of XML that drive aggressive technological innovation. The net result is a maelstrom of invention—a pace of change so rapid that it leaves most people confused. To say that you're using XML is meaningless. Are you using DTDs or XML Schema and, if so, whose? How about XML Namespaces, XML Encryption, XML Signature, XPointer, XLink, XPath, XSLT, XQuery, XKMS, RDF, SOAP, WSDL, UDDI, XAML, BPEL, WSIA, WSRP, or WS-Whatever? Does your software use SAX, DOM, JAXB, JAXP, JAXM, JAXR, or JAX-RPC? It's easy to get lost, to drown in the acronym soup. You're interested in Web services (you bought this book, remember?). How much do you really need to know about XML?

The truth is pleasantly surprising. First, many XML technologies you might have heard about aren't relevant to Web services. You can safely forget half the acronyms you wish you knew more about. Second, even with relevant technologies, you need to know only a few core concepts. (The 80/20 rule doesn't disappoint.) Third, this chapter is all you need to read and understand to be able to handle the rest of the book and make the most of it.

This chapter will develop a set of examples around SkatesTown's processes for submitting POs and generating invoices. The examples cover all the technologies we've listed here.

If you're an old hand at XML who understands the XML namespace mechanism and feels at home with schema extensibility and the use of `xsi:type`, you should go straight to Chapter 3, "The SOAP Protocol," and dive into Web services. If you can parse and process a significant portion of the previous sentence, you should skim this chapter to get a quick refresher of some core XML technologies. And if you're someone with more limited XML experience, don't worry—by the end of this chapter, you'll be able to hold your own.

XML is here to stay. The XML industry is experiencing a boom. XML has become the de facto standard for representing structured and semistructured information in textual form. Many specifications are built on top of XML to extend its capabilities and enable its use in a broader range of scenarios. One of the most exciting areas of use for XML is Web services. The rest of this chapter will introduce the set of XML technologies and standards that are the foundation of Web services:

- *XML instances*—The rules for creating syntactically correct XML documents
- *XML Schema*—A standard that enables detailed validation of XML documents as well as the specification of XML datatypes
- *XML Namespaces*—Definitions of the mechanisms for combining XML from multiple sources in a single document
- *XML processing*—The core architecture and mechanisms for creating, parsing, and manipulating XML documents from programming languages as well as mapping Java data structures to XML

Document- Versus Data-Centric XML

Generally speaking, there are two broad application areas of XML technologies. The first relates to document-centric applications, and the second to data-centric applications. Because XML can be used in so many different ways, it's important to understand the difference between these two categories.

Document-Centric XML

Because of its SGML origins, in the early days of its existence, XML gained rapid adoption within publishing systems as a mechanism for representing semistructured documents such as technical manuals, legal documents, and product catalogs. The content in these documents is typically meant for human consumption, although it could be processed by any number of applications before it's presented to humans. The key element of these documents is semistructured marked-up text.

The following markup is a perfect example of XML used in a document-centric manner. The content is directed toward human consumption—it's part of the FastGlide skateboard user guide. The content is semistructured. The usage rules for tags such as `<B>`, `<I>`, and `<LINK>` are loosely defined; they could appear just about anywhere in the document:

```
<H1>Skateboard Usage Requirements</H1>
<P>In order to use the <B>FastGlide</B> skateboard you have to
have:</P>
<LIST>
```

```
<ITEM>A strong pair of legs.</ITEM>
<ITEM>A reasonably long stretch of smooth road surface.</ITEM>
<ITEM>The impulse to impress others.</ITEM>
</LIST>
<P>If you have all of the above, you can proceed to <LINK
HREF="Chapter2.xml">Getting on the Board</LINK>.</P>
```

Data-Centric XML

By contrast, data-centric XML is used to mark up highly structured information such as the textual representation of relational data from databases, financial transaction information, and programming language data structures. Data-centric XML is typically generated by machines and is meant for machine consumption. XML's natural ability to nest and repeat markup makes it the perfect choice for representing these types of data.

Consider the example in Listing 2.1. It's a purchase order (PO) from the Skateboard Warehouse, a retailer of skateboards to SkatesTown. The order is for 5 backpacks, 12 skateboards, and 1,000 SkatesTown promotional stickers (this is what the stock-keeping unit [SKU] 008-PR stands for).

Listing 2.1 **Purchase Order in XML**

```
<po id="43871" submitted="2004-01-05" customerId="73852">
   <billTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </billTo>
   <shipTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </shipTo>
   <order>
      <item sku="318-BP" quantity="5">
         <description>Skateboard backpack; five pockets</description>
      </item>
      <item sku="947-TI" quantity="12">
         <description>Street-style titanium skateboard.</description>
      </item>
      <item sku="008-PR" quantity="1000">
```

Listing 2.1 **Continued**

```
      </item>
   </order>
</po>
```

The use of XML is very different from the previous user guide example:

- The ratio of markup to content is high. The XML includes many types of tags. There is no long-running text.
- The XML includes machine-generated information; for example, the PO's submission date uses a date-time format of *year-month-day*. A human authoring an XML document is unlikely to enter a date-time value in this format.
- The tags are organized in a highly structured manner. Order and positioning matter, relative to other tags. For example, `<description>` must be under `<item>`, which must be under `<order>`, which must be under `<po>`. The `<order>` tag can be used only once in the document.
- Markup is used to describe what a piece of information *means* rather than how it should be presented to a human.

In short, if you can easily imagine the XML as a data structure in your favorite programming language, you're probably looking at a data-centric use of XML. An example Java class that could, with a bit more work, be used to represent the PO data is shown here:

```
class PO
{
    int id;
    Date submitted;
    int customerId;
    Address billTo;
    Address shipTo;
    Item order[];
}
```

Document Lifetime

Document- and data-centric uses of XML can differ in one other significant aspect: the lifetime of the XML document. Typically, XML documents for human consumption (such as technical manuals and research papers) live a long time because the information they contain can be used for a long time. On the other hand, some data-centric XML may live for only a few milliseconds. Consider the example of a database that is returning the results of a query in XML format. The whole operation takes several milliseconds. After the query is used, the data is discarded. Further, no real XML document exists—the XML is just bits on a wire or bits in an application's data structure. Still, for convenience purposes, we'll use the term *XML document* to refer to any whole piece of XML. To identify parts of a whole XML document, this book uses the highly technical term *chunk*.

Web services are about data-centric uses of XML. Through the rest of this chapter and the rest of this book, we'll purposefully ignore discussing document-centric XML.

XML Instances

The structure and formatting of XML in an XML document must follow the rules of the XML instance syntax. The term *instance* is used to explicitly distinguish the difference between the use of some particular type of XML and its specification. This usage parallels the difference in object-oriented terminology between an object instance and an object type.

Document Prolog

XML documents contain an optional *prolog* followed by a *root element* that holds the contents of the document. Typically the prolog serves up to three roles:

- Identifies the document as an XML document
- Includes any comments about the document
- Includes any meta-information about the content of the document

A document is identified as an XML document through the use of a *processing instruction*. Processing instructions (PIs) are special directives to the application that will process the XML document. They have the following syntax:

```
<?PITarget ...?>
```

PIs are enclosed in `<? ... ?>`. The PI target is a keyword meaningful to the processing application. Everything between the PI target and the `?>` marker is considered the contents of the PI.

In general, data-oriented XML applications don't use application-specific processing instructions. Instead, they tend to put all information in elements and attributes. However, you should use one standard processing instruction—the *XML declaration*—in the XML document prolog to determine two important pieces of information: the version of XML in the document and the character encoding. Here's an example:

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

The `version` parameter of the `xml` PI tells the processing application the version of the XML specification to which the document conforms. (W3C released an updated XML specification, XML 1.1, in early 2004; but all examples in this book use the 1.0 version of XML, which came in 1998.) The `encoding` parameter is optional. It identifies the character set of the document. The default value is `"UTF-8"`.

Note

UTF-8 (RFC 2279) stands for Unicode Transformation Format-8. It's an octet (8-bit) lossless encoding of characters from the Universal Character Set (UCS), aka Unicode (ISO 10646). UTF-8 is an efficient representation of English because it preserves the full US-ASCII character range. One ASCII character is encoded in 8 bits, whereas some Unicode characters can take up to 48 bits. UTF-8 encoding makes it easy to move XML on the Internet using standard communication protocols such as HTTP, SMTP, and FTP. XML is internationalized by design and can support other character encodings such as Unicode and ISO/IEC 10646. However, for simplicity and readability purposes, this book will use UTF-8 encoding for all samples.

If you omit the XML declaration, the XML version is assumed to be 1.0, and the processing application will try to guess the encoding of the document based on clues such as the raw byte order of the data stream. This approach has problems, and whenever interoperability is of high importance—such as for Web services—applications should provide an explicit XML declaration and use UTF-8 encoding.

XML document prologs can also include comments that pertain to the whole document. Comments use the following syntax:

```
<!-- Sample comment and more ... -->
```

Comments can span multiple lines but can't be nested (comments can't enclose other comments). The processing application will ignore everything inside the comment markers. Some of the XML samples in this book use comments to provide you with useful context about the examples in question.

With what we've discussed so far, we can extend the PO example from Listing 2.1 to include an XML declaration and a comment about the document (see Listing 2.2).

Listing 2.2 **XML Declaration and Comment for the Purchase Order**

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by Bob Dister, approved by Mary Jones -->
<po id="43871" submitted="2004-01-05" customerId="73852">
   <!-- The rest of the purchase order will be the same as before -->
   ...
</po>
```

In this case, `po` is the root element of the XML document.

Elements

The term *element* 📖 is a technical name for the pairing of a start tag and an end tag in an XML document. In the previous example, the `po` element has the start tag `<po>` and the end tag `</po>`. Every start tag must have a matching end tag and vice versa. Everything between these two tags is the *content* 📖 of the element. This includes any nested elements, text, comments, and so on.

Element names can include all standard programming language identifier characters (`[0-9A-Za-z]`) as well as the underscore (_), hyphen (-), and colon (`:`), but they must begin with a letter. `customer-name` is a valid XML element name. However, because XML is case-sensitive, `customer-name` isn't the same element as `Customer-Name`.

According to the XML Specification, elements can have three different *content types* 📖: *element-only* content 📖, *mixed* content 📖, or *empty* content 📖. Element-only content consists entirely of nested elements. Any whitespace separating elements isn't considered significant in this case. Mixed content refers to any combination of nested elements and text. All elements in the purchase order example, with the exception of `description`, have element content. Most elements in the skateboard user guide example earlier in the chapter had mixed content.

Note that the XML Specification doesn't define a text-only content model. Outside the letter of the specification, an element that contains only text is often referred to as having *data* content; but, technically speaking, it has mixed content. This awkwardness comes as a result of XML's roots in SGML and document-oriented applications. However, in most data-oriented applications, you'll never see elements whose contents are both nested elements and text. The content will typically be one or the other, because limiting it to be either elements or text makes processing XML much easier.

The syntax for elements with empty content is a start tag immediately followed by an end tag, as in `<emptyElement></emptyElement>`. This is too much text, so the XML Specification also allows the shorthand form `<emptyElement/>`. For example, because the last item in our PO doesn't have a nested `description` element, it has empty content. Therefore, we could have written it as follows:

```
    <item sku="008-PR" quantity="1000"/>
```

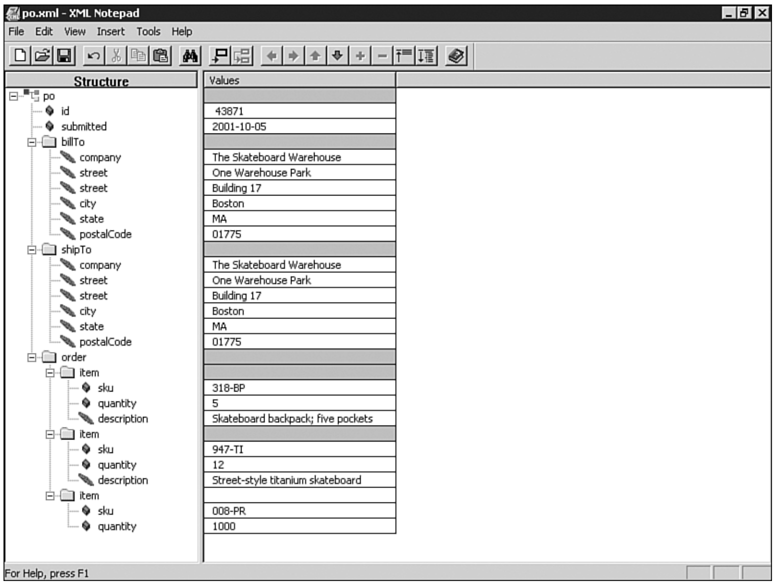
XML elements must be strictly nested. They can't overlap, as shown here:

```
<!-- This is correct nesting -->
<P><B><I>Bold, italicized text in a paragraph</I></B></P>

<!--Bad syntax: overlapping I and B tags -->
<P><I><B>Bold, italicized text in a paragraph</I></B></P>

<!-- Bad syntax: overlapping P and B tags -->
<B><P><I>Bold, italicized text in a paragraph</I></B></P>
```

The notion of an XML document root implies that there is only one element at the very top level of a document. For example, the following wouldn't be a valid XML document:

```
<first>I am the first element</first>
<second>I am the second element</second>
```

It's easy to think of nested XML elements as a hierarchy. For example, Figure 2.1 shows a hierarchical tree representation of the XML elements in the purchase order example together with the data (text) associated with them.



Structure	Values
po	
id	43871
submitted	2001-10-05
billTo	
company	The Skateboard Warehouse
street	One Warehouse Park
street	Building 17
city	Boston
state	MA
postalCode	01775
shipTo	
company	The Skateboard Warehouse
street	One Warehouse Park
street	Building 17
city	Boston
state	MA
postalCode	01775
order	
item	
sku	318-BP
quantity	5
description	Skateboard backpack; five pockets
item	
sku	947-TI
quantity	12
description	Street-style titanium skateboard
item	
sku	008-PR
quantity	1000

Figure 2.1 Tree representation of XML elements in a purchase order.

Unfortunately, it's often difficult to identify XML elements precisely in the hierarchy. To aid this task, the XML community has taken to using genealogy terms such as *parent*, *child*, *sibling*, *ancestor*, and *descendant*. Figure 2.2 illustrates the terminology as it applies to the `order` element of the PO:

- Its parent (the element immediately above it in the hierarchy) is `po`.
- Its ancestor is `po`. Ancestors are all the elements above a given element in the hierarchy.
- Its siblings (elements on the same level of the hierarchy and that have the same parent) are `billTo` and `shipTo`.
- Its children (elements that have this element as a parent) are three `item` elements.
- Its descendants (elements that have this element as an ancestor) are three `item` elements and two `description` elements.

Attributes

The start tags for XML elements can have zero or more *attributes*. An attribute is a name-value pair. The syntax for an attribute is a name (which uses the same character set as an XML element name) followed by an equal sign (=), followed by a quoted value. The XML Specification requires the quoting of values; you can use both single and double quotes, provided they're correctly matched. For example, the `po` element of our PO has three attributes, `id`, `submitted`, and `customerId`:

```
<po id="43871" submitted="2004-01-05" customerId="73852"> ... </po>
```

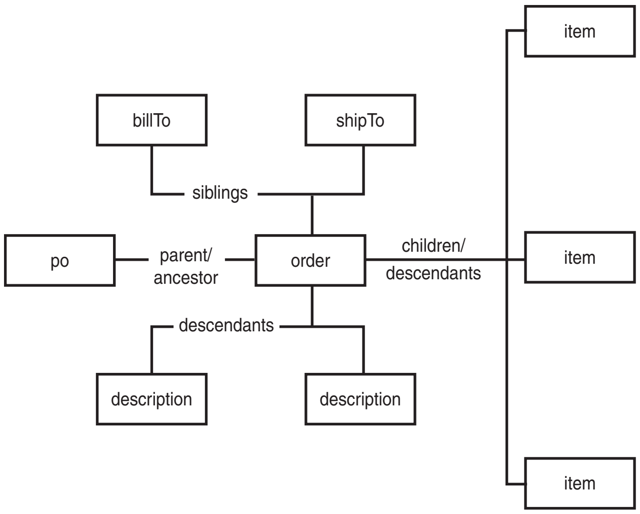


Figure 2.2 Common terminology for XML element relationships

A family of attributes whose names begin with `xml:` is reserved for use by the XML Specification. Probably the best example is `xml:lang`, which identifies the language of the text used in the content of the element that has the `xml:lang` attribute. For example, we could have written the `description` elements in our purchase order example to identify the description text as English:

```
<description xml:lang="en">Skateboard backpack; five pockets</description>
```

Note that applications processing XML aren't required to recognize, process, and act based on the values of these attributes. The key reason the XML Specification identified these attributes is that they address common use-cases; standardizing them aided interoperability between applications.

Without any meta-information about an XML document, attribute values are considered to be pieces of text. In the previous example, the ID might look like a number and the submission date might look like a date, but to an XML processor, they will both be strings. This behavior causes headaches when processing data-oriented XML, and it's one of the primary reasons most data-oriented XML documents have associated meta-information described in XML schemas (introduced later in this chapter).

XML applications are free to attach any semantics they choose to XML markup. A common use-case leverages attributes to create a basic linking mechanism within an XML document. The typical scenario involves a document that has duplicate information in multiple locations. The goal is to eliminate information duplication. The process has three steps:

1. Put the information in the document only once.
2. Mark the information with a unique identifier.
3. Refer to this identifier every time you need to refer to the information.

The purchase order example offers the opportunity to try this technique (see Listing 2.3). As shown in the example, in most cases, the bill-to and ship-to addresses will be the same.

Listing 2.3 **Duplicate Address Information in a Purchase Order**

```
<po id="43871" submitted="2004-01-05" customerId="73852">
   <billTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </billTo>
   <shipTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </shipTo>
   ...
</po>
```

There is no reason to duplicate this information. Instead, we can use the markup shown in Listing 2.4.

Listing 2.4 **Using ID/IDREF Attributes to Eliminate Redundancy**

```
<po id="43871" submitted="2004-01-05" customerId="73852">
   <billTo id="addr-1">
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </billTo>
```

Listing 2.4 **Continued**

```
   <shipTo href="addr-1"/>
   ...
</po>
```

We followed the three steps described previously:

1. We put the address information in the document only once, under the `billTo` element.
2. We uniquely identified the address as `"addr-1"` and stored that information in the `id` attribute of the `billTo` element. We only need to worry about the uniqueness of the identifier within the XML document.
3. To refer to the address from the `shipTo` element, we use another attribute, `href`, whose value is the unique address identifier `"addr-1"`.

The attribute names `id` and `href` aren't required but nevertheless are commonly used by convention.

You might have noticed that now both the `po` and `billTo` elements have an attribute called `id`. This is fine, because the attribute names are unique within the context of the two elements.

Elements Versus Attributes

Given that information can be stored in both element content and attribute values, sooner or later the question of whether to use an element or an attribute arises. This debate has erupted a few times in the XML community and has claimed many casualties.

One common rule is to represent structured information using markup. For example, you should use an `address` element with nested `company`, `street`, `city`, `state`, `postalCode`, and `country` elements instead of including a whole address as a chunk of text.

Even this simple rule is subject to interpretation and the choice of application domain. For example, the choice between

```
<work number="617.219.2000">
```

and

```
<work>
   <area>617</area>
   <number>219.2000</number>
   <ext/>
</work>
```

depends on whether your application needs to have phone number information in granular form (for example, to perform searches based on the area code only).

In other cases, only personal preference and stylistic choice apply. We might ask if SkatesTown should have used

```
<po>
   <id>43871</id>
```

```
      <submitted>2004-01-05</submitted>
      <customerId>73852</customerId>
      ...
   </po>
```

instead of

```
   <po id="43871" submitted="2004-01-05" customerId="73852">
      ...
   </po>
```

There's no good way to answer this question without adding stretchy assumptions about extensibility needs and so on.

In general, whenever humans design XML documents, you'll see more frequent use of attributes. This is true even in data-oriented applications. On the other hand, when XML documents are automatically "designed" and generated by applications, you may see more prevalent use of elements. The reasons are somewhat complex; Chapter 3 will address some of them.

Character Data

Attribute values as well as the text and whitespace between tags must follow precisely a small but strict set of rules. Most XML developers think of this character data as mapping to the string data type in their programming language of choice. Unfortunately, things aren't that simple.

Encoding

First, and most important, all character data in an XML document must comply with the document's encoding. Any characters outside the range of characters that can be included in the document must be escaped and identified as *character references* 📖. The escape sequence used throughout XML uses the ampersand (`&`) as its start and a semicolon (`;`) as its end. The syntax for character references is an ampersand, followed by a pound/hash sign (`#`), followed by either a decimal character code or lowercase *x* followed by a hexadecimal character code, followed by a semicolon. Therefore, the 8-bit character code 128 is encoded in a UTF-8 XML document as `&#x80;`.

Unfortunately, for obscure document-oriented reasons, there is no way to include character codes 0 through 7, 9, 11, 12, or 14 through 31 (typically known as *non-whitespace control characters* 📖 in ASCII) in XML documents. Even a correctly escaped character reference won't do. This situation can cause unexpected problems for programmers whose string data types sometimes end up with these values.

Whitespace

The rules for whitespace handling are also a legacy from the document-centric world XML came from. It isn't important to completely define these rules here, but a couple of them are worth mentioning:

- An XML processor is required to convert any carriage return (CR) character it sees in the XML document, as well as the sequence of a carriage return and a line feed (LF) character, into a single LF character.
- Whitespace can be treated as either significant or insignificant. The set of rules for how applications are notified about either of these has caused more than one debate in the XML community.

Luckily, most data-oriented XML applications care little about whitespace.

Entities

In addition to character references, XML documents can define *entities* 📖 as well as references to them (*entity references* 📖). Entities typically aren't important for data-oriented applications, and we won't discuss them in detail. However, all XML processors must recognize several predefined entities that map to characters that can be confused with markup delimiters. These characters are less than (`<`); greater than (`>`); ampersand (`&`); apostrophe, aka single quote (`'`); and quote, aka double quote (`"`). Table 2.1 shows the syntax for escaping these characters.

Table 2.1 **Predefined XML Character Escape Sequences**

Character	Escape Sequence
`<`	`&lt;`
`>`	`&gt;`
`&`	`&amp;`
`'`	`&apos;`
`"`	`&quot;`

For example, to include a chunk of XML as text rather than markup inside an XML document, you should escape all special characters:

```
<example-to-show>
   &lt;?xml version=&quot;1.0&quot;?&gt;
   &lt;rootElement&gt;
      &lt;childElement id=&quot;1&quot;&gt;
         The man said: &quot;Hello, there!&quot;.
      &lt;/childElement&gt;
   &lt;/rootElement&gt;
</example-to-show>
```

The result is not only reduced readability but also a significant increase in the size of the document, because single characters are mapped to character escape sequences whose length is at least four characters.

To address this problem, the XML Specification has a special multi-character escape construct. The name of the construct, the *CDATA section* 📖, refers to the section holding character data. The syntax is `<![CDATA[`, followed by any sequences of characters

allowed by the document encoding that don't include `]]>`, followed by `]]>`. Therefore, you can write the previous example much more simply as follows:

```
<example-to-show><![CDATA[
   <?xml version="1.0"?>
   <rootElement>
      <childElement id="1">
         The man said: "Hello, there!".
      </childElement>
   </rootElement>
]]></example-to-show>
```

A Simpler Purchase Order

Based on the information in this section, we can re-write the PO document as shown in Listing 2.5.

Listing 2.5 **Improved Purchase Order Document**

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by Bob Dister, approved by Mary Jones -->
<po id="43871" submitted="2004-01-05" customerId="73852">
   <billTo id="addr-1">
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </billTo>
   <shipTo href="addr-1"/>
   <order>
      <item sku="318-BP" quantity="5">
         <description>Skateboard backpack; five pockets</description>
      </item>
      <item sku="947-TI" quantity="12">
         <description>Street-style titanium skateboard.</description>
      </item>
      <item sku="008-PR" quantity="1000"/>
   </order>
</po>
```

XML Namespaces

An important property of XML documents is that they can be composed to create new documents. This is the most basic mechanism for reusing XML. Unfortunately, simple composition creates the problems of recognition and collision.

To illustrate these problems, consider a scenario where SkatesTown wants to receive its POs via the XML messaging system of XCommerce Messaging, Inc. The format of the messages is simple:

```
<message from="..." to="..." sent="...">
   <text>
      This is the text of the message.
   </text>
   <!-- A message can have attachments -->
   <attachment>
      <description>Brief description of the attachment.</description>
      <item>
         <!-- XML of attachment goes here -->
      </item>
   </attachment>
</message>
```

Listing 2.6 shows a complete message with a PO attachment.

Listing 2.6 **Message with Purchase Order Attachment**

```
<message from="bj@bjskates.com" to="orders@skatestown.com"
   sent="2004-01-05">
   <text>
      Hi, here is what I need this time. Thx, BJ.
   </text>
   <attachment>
      <description>The PO</description>
      <item>
         <po id="43871" submitted="2004-01-05" customerId="73852">
            <billTo id="addr-1">
               <company>The Skateboard Warehouse</company>
               <street>One Warehouse Park</street>
               <street>Building 17</street>
               <city>Boston</city>
               <state>MA</state>
               <postalCode>01775</postalCode>
            </billTo>
            <shipTo href="addr-1"/>
            <order>
               <item sku="318-BP" quantity="5">
                  <description>
                     Skateboard backpack; five pockets
                  </description>
               </item>
               <item sku="947-TI" quantity="12">
                  <description>
```

Listing 2.6 **Continued**

```
                     Street-style titanium skateboard.
                  </description>
               </item>
               <item sku="008-PR" quantity="1000"/>
            </order>
         </po>
      </item>
   </attachment>
</message>
```

It's relatively easy to identify the two problems mentioned earlier in the composed document:

- *Recognition*—How does an XML processing application distinguish between the XML elements that describe the message and the XML elements that are part of the PO?
- *Collision*—Does the element `description` refer to attachment descriptions in messages or order item descriptions? Does the `item` element refer to an item of attachment or an order item?

Very simple applications might not be bothered by these problems. After all, the knowledge of what an element means can reside in the application logic. However, as application complexity increases and the number of applications that need to work with a particular composed document type grows, the need to clearly distinguish between the XML elements becomes paramount. The XML Namespaces specification brings order to the chaos.

Namespace Mechanism

The problem of collision in composed XML documents arises because of the likelihood that elements with common names (description, item, and so on) will be reused in different document types. This problem can be addressed by *qualifying* an XML element name with an additional *identifier* that's much more likely to be unique within the composed document. In other words:

Qualified name (aka QName) = Namespace identifier + Local name

This approach is similar to the way namespaces are used in languages such as C++ and C# and to the way package names are used in the Java programming language.

The problem of recognition in composed XML documents arises because no good mechanism exists to identify all elements belonging to the same document type. Given namespace qualifiers, the problem is addressed in a simple way—all elements that have the same namespace identifier are considered together.

For identifiers, XML Namespaces uses *Uniform Resource Identifiers* 📖 (URIs), which are described in RFC 2396. URIs are nothing fancy, but they're very useful. They can be locators, names, or both. URI locators are known as *Uniform Resource Locators* 📖 (URLs), a term familiar to anyone using the Web. URLs are strings such as `http://www.skatestown.com/services/POSubmission` and `mailto:orders@skatestown.com`.

Uniform Resource Names 📖 (URNs) are URIs that are globally unique and persistent. *Universally Unique Identifiers* 📖 (UUIDs) are perfect for use as URNs. UUIDs are 128-bit identifiers that are designed to be globally unique. Typically, they combine network card (Ethernet) addresses with a high-precision timestamp and an increment counter. An example URN using a UUID is `urn:uuid:2FAC1234-31F8-11B4-A222-08002B34C003`. UUIDs are used as unique identifiers in Universal Description Discovery and Integration (UDDI) as detailed in Chapter 6, "Web Services Registries."

Namespace Syntax

Because URIs can be long and typically contain characters that aren't allowed in XML element names, the syntax of including namespaces in XML documents involves two steps:

1. A namespace identifier is associated with a *prefix*, a name that contains only legal XML element name characters with the exception of the colon (:).
2. Qualified names are obtained as a combination of the prefix, the colon character, and the local element name, as in `myPrefix:myElementName`.

Listing 2.7 shows an example of the composed XML document using namespaces.

Listing 2.7 **Message with Namespaces**

```
<msg:message from="bj@bjskates.com" to="orders@skatestown.com"
   sent="2004-01-05" xmlns:msg="http://www.xcommercemsg.com/ns/message"
   xmlns:po="http://www.skatestown.com/ns/po">
   <msg:text>
      Hi, here is what I need this time. Thx, BJ.
   </msg:text>
   <msg:attachment>
      <msg:description>The PO</msg:description>
      <msg:item>
         <po:po id="43871" submitted="2004-01-05" customerId="73852">
            <po:billTo id="addr-1">
               <po:company>The Skateboard Warehouse</po:company>
               <po:street>One Warehouse Park</po:street>
               <po:street>Building 17</po:street>
               <po:city>Boston</po:city>
               <po:state>MA</po:state>
               <po:postalCode>01775</po:postalCode>
            </po:billTo>
            <po:shipTo href="addr-1"/>
```

Listing 2.7 **Continued**

```
            <po:order>
               <po:item sku="318-BP" quantity="5">
                  <po:description>
                     Skateboard backpack; five pockets
                  </po:description>
               </po:item>
               <po:item sku="947-TI" quantity="12">
                  <po:description>
                     Street-style titanium skateboard.
                  </po:description>
               </po:item>
               <po:item sku="008-PR" quantity="1000"/>
            </po:order>
         </po:po>
      </msg:item>
   </msg:attachment>
</msg:message>
```

In this example, the elements prefixed with `msg` are associated with a namespace whose identifier is `http://www.xcommercemsg.com/ns/message`, and those prefixed with `po` are associated with a namespace whose identifier is `http://www.skatestown.com/ns/po`. The prefixes are linked to the complete namespace identifiers by the attributes on the top `message` element beginning with `xmlns:` (`xmlns:msg` and `xmlns:po`). XML processing software has access to both the prefixed name and the mapping of prefixes to complete namespace identifiers.

Adding a prefix to every element in the document decreases readability and increases document size. Therefore, XML Namespaces lets you use a default namespace in a document. Elements belonging to the default namespace don't require prefixes. Listing 2.8 makes the `msg` namespace the default.

Listing 2.8 **Using Default Namespaces**

```
<message from="bj@bjskates.com" to="orders@skatestown.com"
   sent="2004-01-05" xmlns ="http://www.xcommercemsg.com/ns/message"
   xmlns:po="http://www.skatestown.com/ns/po">
   <text>
      Hi, here is what I need this time. Thx, BJ.
   </text>
   <attachment>
      <description>The PO</description>
      <item>
         <po:po id="43871" submitted="2004-01-05" customerId="73852">
            ...
         </po:po>
```

Listing 2.8 **Continued**

```
      </item>
   </attachment>
</message>
```

Default namespaces work because the content of any namespace-prefixed element is considered to belong to the namespace of its parent element—unless, of course, the element is explicitly defined to be in another namespace with its own `xmlns`-type attribute. We can use this to further clean up the composed XML document by moving the PO namespace declaration to the `po` element (see Listing 2.9).

Listing 2.9 **Using Nested Namespace Defaulting**

```
<message from="bj@bjskates.com" to="orders@skatestown.com"
   sent="2004-01-05" xmlns="http://www.xcommercemsg.com/ns/message">
   <text>
      Hi, here is what I need this time. Thx, BJ.
   </text>
   <attachment>
      <description>The PO</description>
      <item>
         <po:po id="43871" submitted="2004-01-05" customerId="73852"
            xmlns:po="http://www.skatestown.com/ns/po">
            <billTo id="addr-1">
               ...
            </billTo>
            <shipTo href="addr-1"/>
            <order>
               ...
            </order>
         </po:po>
      </item>
   </attachment>
</message>
```

This example shows an efficient, readable syntax that eliminates the recognition and collision problems. XML processors can identify the namespace of any element in the document.

Namespace-Prefixed Attributes

Attributes can also have namespaces associated with them. Initially, it might be hard to imagine why a capability like this would be useful for XML applications. The common use-case scenario is the desire to extend the information provided by an XML element without having to make changes directly to its document type.

A concrete example might involve SkatesTown wanting to have an indication of the priority of items in POs. High-priority items could be shipped immediately, without waiting for any back-ordered items to become available. SkatesTown's automatic order-processing software doesn't understand item priorities; they're just hints that tell the fulfillment system how it should react in case of back-ordered items.

A simple implementation could involve extending the `item` element with an optional `priority` attribute. However, doing so could cause a problem for the order-processing software, which doesn't expect to see such an attribute. A better solution is to attach priority information to items using a namespace-prefixed `priority` attribute. Because the attribute will be in a namespace different from that of the `item` element, the order-processing software will ignore it.

The example in Listing 2.10 uses this mechanism to make the backpacks high priority and the promotional materials low priority. By default, any items without a `priority` attribute, such as the skateboards, are presumed to be of medium priority.

Listing 2.10 **Adding Priority to Order Items**

```
<message from="bj@bjskates.com" to="orders@skatestown.com"
   sent="2004-01-05" xmlns="http://www.xcommercemsg.com/ns/message">
   <text>
      Hi, here is what I need this time. Thx, BJ.
   </text>
   <attachment>
      <description>The PO</description>
      <item>
         <po:po id="43871" submitted="2004-01-05" customerId="73852"
            xmlns:po="http://www.skatestown.com/ns/po">
            xmlns:p="http://www.skatestown.com/ns/priority">
            ...
            <po:order>
               <po:item sku="318-BP" quantity="5" p:priority="high">
                  <po:description>
                     Skateboard backpack; five pockets
                  </po:description>
               </po:item>
               <po:item sku="947-TI" quantity="12">
                  <po:description>
                     Street-style titanium skateboard.
                  </po:description>
               </po:item>
               <po:item sku="008-PR" quantity="1000" p:priority="low"/>
            </po:order>
         </po:po>
      </item>
   </attachment>
</message>
```

Dereferencing URIs

All the examples in this section use namespace URIs that are URLs. A natural question arises: What is the resource at that URL? The answer is that it doesn't matter. XML Namespaces doesn't require that a resource be there. The URI is used entirely for identification purposes.

This could cause problems for applications that see an unknown namespace in an XML document and have no way to obtain more information about the elements and attributes that belong to that namespace. In the next section, you'll see a mechanism that addresses this issue.

XML Schemas

XML provides a flexible set of structures that can represent many different types of document- and data-oriented information. XML offers an optional feature called Document Type Definitions (DTDs). A document associated with a DTD has a set of rules regarding the elements and attributes that can be part of the document and where they can appear. DTDs offer the basic mechanism for defining a *vocabulary* specifying the structure of XML documents in an attempt to establish a contract (how an XML document will be structured) between multiple parties working with the same type of XML.

DTDs came into existence because people and applications needed to be able to treat XML at a higher level than a collection of elements and attributes. Well-designed DTDs attach meaning to the XML syntax in documents. At the same time, DTDs fail to address the common needs of namespace integration, modular vocabulary design, flexible content models, and tight integration with data-oriented applications. This failure comes as a direct result of XML's SGML origins and the predominantly document-centric nature of SGML applications. To address these issues, the XML community, under the leadership of the W3C, took up the task of creating a meta-language for describing both the structure of XML document and the mapping of XML syntax to data types. After long deliberation, the effort produced the final version of the XML Schema specification in March 2001. All the Web services specifications use XML Schema for defining their vocabularies.

Well-Formedness and Validity

The presence of schema information allows us to distinguish the concepts of *well-formedness* and *validity*. If a document subscribes to the rules of XML syntax (as described in the section "XML Instances"), it's considered well-formed. Well-formedness implies that XML processing software can read the document without any basic errors associated with parsing, such as invalid character data, mismatched start and end tags, multiple attributes with the same name, and so on. The XML Specification mandates that if any well-formedness constraint isn't met, the XML parser must immediately generate a nonrecoverable error. This rigid mandate makes it easy to separate the doings of the software focused on the *logical structure* of an XML document (what the markup means) from the mundane details of the *physical structure* of the document (the markup syntax).

However, well-formedness isn't sufficient for most applications. Consider, for example, the SkatesTown order-processing application. When an XML document is submitted to the application, it doesn't care whether the document is well-formed XML but that the document is a PO in the specific XML format it requires. The notion of *format* applies to the set of rules describing SkatesTown's POs: "The document must begin with a `po` element that has three attributes (`id`, `submitted`, and `customerId`), which will be followed by a `billTo` element, …" and so on. In other words, before a submitted document is processed, it must be identified as a valid PO.

This is how the notion of validity comes in. Schemas offer an automated, declarative mechanism for validating the contents of XML documents as they're parsed. Therefore, XML applications can limit the amount of validation they need to perform. If the SkatesTown PO-processing application couldn't delegate validation to the XML processor, it would have to express all validation rules directly in code. Code is procedural in nature and much harder to maintain than schemas, which are declarative and have readable XML syntax.

To handle validity checks, schemas enable the following:

- Identification of the elements that can be in a document
- Identification of the order and relation between elements
- Identification of the attributes of every element and whether they're optional or required or have some other special properties
- Identification of the datatype of attribute content

Last but not least, schemas offer significant capabilities for modular vocabulary design that let you reuse and repurpose existing vocabularies.

XML Schema Basics

In a nutshell, XML Schema is both powerful and complex. It's powerful because it allows for much more expressive and precise specification of the content of XML documents. It's complex for the same reason. The specification is broken into three parts:

- *XML Schema Part 0: Primer* is a non-normative document that tries to make sense of XML Schema by parceling complexity into small chunks and using many examples.
- *XML Schema Part 1: Structures* focuses primarily on serving the needs of document-oriented applications by laying out the rules for defining the structure of XML documents.
- *XML Schema Part 2: Datatypes* builds on the structures specification with additional capabilities that address the needs of data-oriented applications such as defining reusable datatypes, associating XML syntax with schema datatypes, and mapping these to application-level data.

Part 0 is meant for general consumption, whereas Parts 1 and 2 are deeply technical and require a skilled and determined reader. The rest of this section will provide an introduction to XML Schema that is biased toward schema usage in data-oriented applications. You should gain sufficient understanding of structure and datatype specifications to comprehend and use common Web service schemas. Still, because XML Schema is fundamental to Web services, we recommend that you go through the primer document of the XML Schema specification.

One way to visualize the structure of a document is as a tree of possible element and attribute combinations. For example, Figure 2.3 shows the document structure for POs as expressed by a popular XML processing tool. The image uses some syntax from regular expressions to visualize the multiplicity of elements: question mark (?) stands for optional (zero or one), asterisk (*) stands for any (zero or more), and plus (+) stands for at least some (one or more).

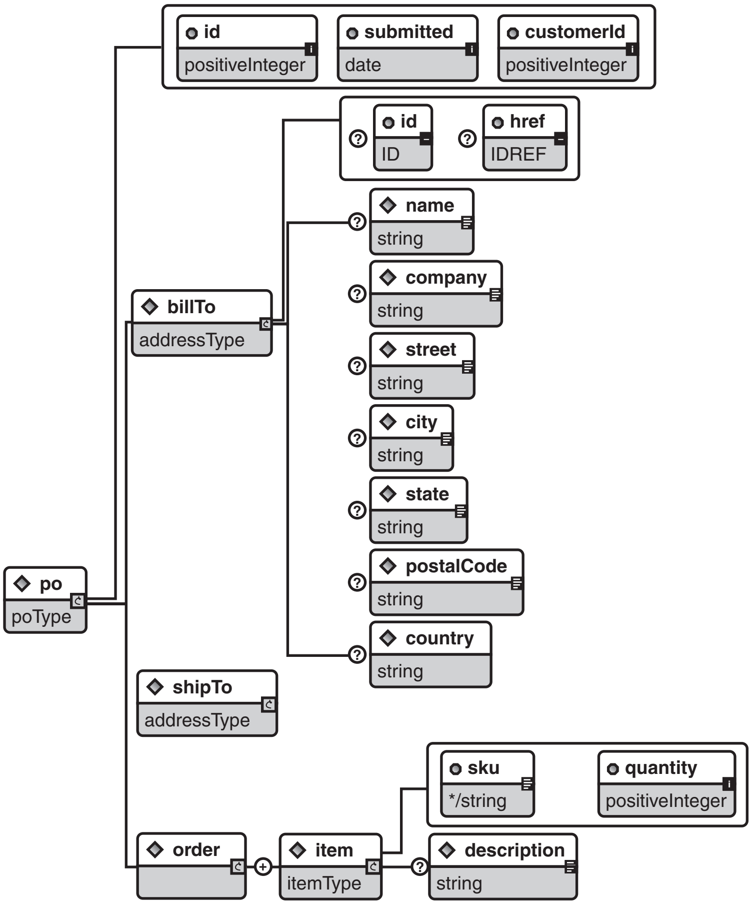


Figure 2.3 Document structure defined by purchase order schema

Listing 2.11 shows the basic structure of the SkatesTown PO schema.

Listing 2.11 **Basic XML Schema Structure**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.skatestown.com/ns/po">

   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Purchase order schema for SkatesTown.
      </xsd:documentation>
   </xsd:annotation>

   ...

</xsd:schema>
```

Schema are expressed in XML and designed with namespaces in mind from the ground up. In this particular schema document, all elements belonging to the schema specification are prefixed with `xsd:`. The prefix's name isn't important, but `xsd:` (which comes from XML Schema Definition) is the convention. The prefix is associated with the `http://www.w3.org/2001/XMLSchema` namespace, which identifies the W3C Recommendation of the XML Schema specification. The default namespace of the document is set to be `http://www.skatestown.com/ns/po`, the namespace of the SkatesTown PO. The schema document needs both namespaces to distinguish between XML elements that belong to the schema specification versus XML elements that belong to POs. Finally, the `targetNamespace` attribute of the `schema` element identifies the namespace of the documents that will conform to this schema. This is set to the PO schema namespace.

The schema is enclosed by the `xsd:schema` element. The content of this element is other schema elements that are used for element, attribute, and datatype definitions. The `annotation` and `documentation` elements can be used liberally to attach auxiliary information to the schema.

Associating Schemas with Documents

Schemas don't have to be associated with XML documents. For example, applications can be preconfigured to use a particular schema when processing documents. Alternatively, there is a powerful mechanism for associating schemas with documents. Listing 2.12 shows how to associate the previous schema with a PO document.

Listing 2.12 **Associating Schemas with Documents**

```
<?xml version="1.0" encoding="UTF-8"?>
<po:po xmlns:po="http://www.skatestown.com/ns/po"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.skatestown.com/ns/po
                          http://www.skatestown.com/schema/po.xsd"
       id="43871" submitted="2004-01-05" customerId="73852">

   ...

</po:po>
```

First, because the PO schema identifies a target namespace, PO documents are required to use namespaces to identify their elements. The PO document uses the `po` prefix for this task.

Next, the document uses another namespace—`http://www.w3.org/2001/XMLSchema-instance`—which has a special meaning. It defines a number of attributes that are part of the schema specification. These attributes can be applied to elements in instance documents to provide additional information to a schema-aware XML processor. By convention, most documents use the namespace prefix `xsi:` (for XML Schema: Instance).

The binding between the PO document and its schema is established via the `xsi:schemaLocation` attribute. This attribute contains a pair of values. The first value is the namespace identifier whose schema's location is identified by the second value. Typically, the second value is a URL, but specialized applications can use other types of values, such as an identifier in a schema repository or a well-known schema name. If the document used more than one namespace, the `xsi:schemaLocation` attribute would contain multiple pairs of values.

Simple Types

Prior to the arrival of schemas, one of the biggest problems with XML processing was that XML had no notion of datatypes, even for simple values such as the character data content of an element or an attribute value. Because of this limitation, XML applications included a large amount of validation code. For example, even a simple PO requires the following validation rules, which are outside the scope of the XML Specification:

- Attributes `id` and `customerId` of the `po` element must be positive integers.
- Attribute `submitted` of the `po` element must be a date in the format *yyyy-mm-dd*.
- Attribute `quantity` of the `item` element must be a positive integer.
- Attribute `sku` (stock keeping unit) of the `item` element must be a string with this format: three digits, followed by a dash, followed by two uppercase letters.

XML schemas address these issues in two ways. First, the specification comes with a large set of predefined basic datatypes such as `string`, `positiveInteger`, and `date`, which you can use directly. For custom data types, such as the values of the sku attribute, the specification defines a powerful mechanism for defining new types. Table 2.2 shows some of the commonly used predefined schema types with examples of their use.

Table 2.2 **Predefined XML Schema Simple Types**

Simple Type	Examples (Delimited by Commas)	Notes
`string`	Confirm this is electric	
`base64Binary`	GpM7	
`hexBinary`	0FB7	
`integer`	-126789, -1, 0, 1, 126789	
`positiveInteger`	1, 126789	
`negativeInteger`	-126789, -1	
`nonNegativeInteger`	0, 1, 126789	
`nonPositiveInteger`	-126789, -1, 0	
`decimal`	-1.23, 0, 123.4, 1000.00	
`boolean`	true, false	
	1, 0	
`time`	13:20:00.000	
	13:20:00.000-05:00	
`dateTime`	1999-05-31T13:20:00.000-05:00 (May 31, 1999 at 1.20pm Eastern Standard Time, which is 5 hours behind Coordinated Universal Time)	
`duration`	P1Y2M3DT10H30M12.3S (1 year, 2 months, 3 days, 10 hours, 30 minutes, and 12.3 seconds)	
`date`	1999-05-31	
`Name`	shipTo	XML Name type
`QName`	po:USAddress	XML Namespace QName
`anyURI`	http://www.example.com/, http://www.example.com/doc.html#ID5	
`ID`		XML ID attribute type
`IDREF`		XML IDREF attribute type

The information in this table comes from the XML Schema Primer.

A note on `ID`/`IDREF` attributes: An XML processor is required to generate an error if a document contains two `ID` attributes with the same value or an `IDREF` with a value that has no matching `ID` value. This makes `ID/IDREF` attributes perfect for handling attributes such as `id` and `href` in SkatesTown's PO `address` element.

Extending Simple Types

The process for creating new simple datatypes is straightforward. The new type must be derived from a *base type*: a predefined schema type or another already-defined simple type. The base type is *restricted* along a number of *facets* to obtain the new type. The facets identify various characteristics of the types, such as

- `length`, `minLength`, `maxLength`—The exact, minimum, and maximum character length of the value
- `pattern`—A regular expression pattern for the value
- `enumeration`—A list of all possible values
- `whiteSpace`—The rules for handling whitespace in the value
- `minExclusive`, `minInclusive`, `maxInclusive`, `maxExclusive`—The range of numeric values that are allowed
- `totalDigits`—The number of decimal digits in a numeric value
- `fractionDigits`—The number of decimal digits after the decimal point

Of course, not all facets apply to all types. For example, the notion of fraction digits makes no sense for a date or a name. Tables 2.3 and 2.4 cross-link the predefined types and the facets that are applicable for them.

Table 2.3 **XML Schema Facets for Simple Types**

Simple Type	**Facets**					
	`length`	`minLength`	`maxLength`	`pattern`	`enumeration`	`whiteSpace`
`string`	✔	✔	✔	✔	✔	✔
`base64Binary`	✔	✔	✔	✔	✔	✔
`hexBinary`	✔	✔	✔	✔	✔	✔
`integer`				✔	✔	✔
`positiveInteger`				✔	✔	✔
`negativeInteger`				✔	✔	✔
`nonNegativeInteger`				✔	✔	✔
`nonPositiveInteger`				✔	✔	✔
`decimal`				✔	✔	✔
`boolean`				✔		✔
`time`				✔	✔	✔
`dateTime`				✔	✔	✔
`duration`				✔	✔	✔
`date`				✔	✔	✔

Table 2.3 **Continued**

Simple Type	**Facets**					
	`length`	`minLength`	`maxLength`	`pattern`	`enumeration`	`whiteSpace`
`Name`	✔	✔	✔	✔	✔	✔
`QName`	✔	✔	✔	✔	✔	✔
`anyURI`	✔	✔	✔	✔	✔	✔
`ID`	✔	✔	✔	✔	✔	✔
`IDREF`	✔	✔	✔	✔	✔	✔

The information in this table comes from the XML Schema Primer.

The facets listed in Table 2.4 apply only to simple types that have an implicit order.

Table 2.4 **XML Schema Facets for Ordered Simple Types**

Simple Types	**Facets**					
	Max Inclusive	**Max Exclusive**	**Min Inclusive**	**Min Exclusive**	**Total Digits**	**Fraction Digits**
`integer`	✔	✔	✔	✔	✔	✔
`positiveInteger`	✔	✔	✔	✔	✔	✔
`negativeInteger`	✔	✔	✔	✔	✔	✔
`nonNegativeInteger`	✔	✔	✔	✔	✔	✔
`nonPositiveInteger`	✔	✔	✔	✔	✔	✔
`decimal`	✔	✔	✔	✔	✔	✔
`time`	✔	✔	✔	✔		
`dateTime`	✔	✔	✔	✔		
`duration`	✔	✔	✔	✔		
`date`	✔	✔	✔	✔		

The information in this table comes from the XML Schema Primer.

The syntax for creating new types is simple. For example, the schema snippet in Listing 2.13 defines a simple type for purchase order SKUs. The name of the type is `skuType`. It's based on a string, and it restricts the string to the following pattern: three digits, followed by a dash, followed by two uppercase letters.

Listing 2.13 **Using Patterns to Define a String's Format**

```
<xsd:simpleType name="skuType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Listing 2.14 shows how to force purchase order IDs to be greater than 10,000 but less than 100,000, and also how to define an enumeration of all U.S. states.

Listing 2.14 **Using Ranges and Enumerations**

```
<xsd:simpleType name="poIdType">
   <xsd:restriction base="xsd:integer">
      <xsd:minExclusive value="10000"/>
      <xsd:maxExclusive value="100000"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="stateType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="AK"/>
      <xsd:enumeration value="AL"/>
      <xsd:enumeration value="AR"/>
      ...
   </xsd:restriction>
</xsd:simpleType>
```

Complex Types

In XML Schema, simple types define the valid choices for character-based content such as attribute values and elements with character content. *Complex types* 📖, on the other hand, define complex content models, such as those of elements that can have attributes and nested children. Complex type definitions address both the sequencing and multiplicity of child elements as well as the names of associated attributes and whether they're required or optional.

The syntax for defining complex types is straightforward:

```
<xsd:complexType name="typeName">
   <xsd:someTopLevelModelGroup>
      <!-- Sequencing and multiplicity constraints for
           child elements defined using xsd:element -->
   </xsd:someTopLevelModelGroup>
   <!-- Attribute declarations using xsd:attribute -->
</xsd:complexType>
```

The element `xsd:complexType` identifies the type definition. There are many different ways to specify the model group of the complex type. The most commonly used top-level model group elements you'll see are

- `xsd:sequence`—A sequence of elements
- `xsd:choice`—Allows one out of a number of elements
- `xsd:all`—Allows a certain set of elements to appear once or not at all but in any order
- `xsd:group`—References a model group that is defined someplace else

These could be further nested to create more complex model groups. The `xsd:group` model group element is covered later in this chapter in the section "Content Model Groups."

Inside the model group specification, child elements are defined using `xsd:element`. The model group specification is followed by any number of attribute definitions using `xsd:attribute`.

For example, one possible way to define the content model of the PO address used in the `billTo` and `shipTo` elements is shown in Listing 2.15. The name of the complex type is `addressType`. Using `xsd:sequence` and `xsd:element`, it defines a sequence of the elements `name`, `company`, `street`, `city`, `state`, `postalCode`, and `country`.

Listing 2.15 **Schema Fragment for the Address Complex Type**

```
<xsd:complexType name="addressType">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="0"/>
      <xsd:element name="company" type="xsd:string" minOccurs="0"/>
      <xsd:element name="street" type="xsd:string"
                   maxOccurs="unbounded"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string" minOccurs="0"/>
      <xsd:element name="postalCode" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="country" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID"/>
   <xsd:attribute name="href" type="xsd:IDREF"/>
</xsd:complexType>
```

The multiplicities of these elements' occurrences are defined using the `minOccurs` and `maxOccurs` attributes of `xsd:element`. The value of zero for `minOccurs` renders an element's presence optional (`?` in the document structure diagrams). The default value for `minOccurs` is 1. The special `maxOccurs` value `"unbounded"` is used for the `street` element to indicate that at least one must be present (`+` in the document structure diagrams).

As we mentioned earlier, every element is associated with a type using the type attribute `xsd:element`. In this example, all elements have simple character content of type `string`, identified by the `xsd:string` type. It might seem unusual that the namespace prefix is used inside an attribute value. It's true, the XML Namespaces specification doesn't explicitly address this use of namespace prefixes. However, the idea is simple. A schema can define any number of types. Some of them are built into the specification, and others are user-defined. The only way to know for sure which type is being referred to is to associate the type name with the namespace from which it's coming. What better way to do this than to prefix all references to the type with a namespace prefix?

After the model group definition come the attribute definitions. In this example, `xsd:attribute` defines attributes `id` and `href` of types `ID` and `IDREF`, respectively. Both attributes are optional by default.

Now, consider a slightly more complex example of a complex type definition—the `po` element's type (see Listing 2.16).

Listing 2.16 **Schema Fragment for the Purchase Order Complex Type**

```
<xsd:complexType name="poType">
   <xsd:sequence>
      <xsd:element name="billTo" type="addressType"/>
      <xsd:element name="shipTo" type="addressType"/>
      <xsd:element name="order">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="item" type="itemType"
                            maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" use="required"
                  type="xsd:positiveInteger"/>
   <xsd:attribute name="submitted" use="required"
                  type="xsd:date"/>
   <xsd:attribute name="customerId" use="required"
                  type="xsd:positiveInteger"/>
</xsd:complexType>
```

The `poType` introduces three interesting aspects of schemas:

- It shows how easy it is to achieve basic reusability of types. Both the `billTo` and `shipTo` elements refer to the `addressType` defined previously. Note that because this is a user-defined complex type, a namespace prefix isn't necessary.
- The association between elements and their types can be implicit. The `order` element's type is defined inline as a sequence of one or more `item` elements of type

`itemType`. This is convenient because it keeps the schema more readable and prevents the need to define a global type that is used in only one place.

- The presence of attributes can be required through the `use="required"` attribute-value pair of the `xsd:attribute` element. To give default and fixed values to attributes, you can also use the aptly named `default` and `fixed` attributes of `xsd:attribute`.

The Purchase Order Schema

With the information gathered so far, we can completely define the SkatesTown purchase order schema (Listing 2.17).

Listing 2.17 **The Complete SkatesTown Purchase Order Schema (`po.xsd`)**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.skatestown.com/ns/po">

   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Purchase order schema for SkatesTown.
      </xsd:documentation>
   </xsd:annotation>

   <xsd:element name="po" type="poType"/>

   <xsd:complexType name="poType">
      <xsd:sequence>
         <xsd:element name="billTo" type="addressType"/>
         <xsd:element name="shipTo" type="addressType"/>
         <xsd:element name="order">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="item" type="itemType"
                               maxOccurs="unbounded"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" use="required"
                     type="xsd:positiveInteger"/>
      <xsd:attribute name="submitted" use="required"
                     type="xsd:date"/>
      <xsd:attribute name="customerId" use="required"
                     type="xsd:positiveInteger"/>
   </xsd:complexType>
```

Listing 2.17 **Continued**

```
   <xsd:complexType name="addressType">
      <xsd:sequence>
         <xsd:element name="name" type="xsd:string" minOccurs="0"/>
         <xsd:element name="company" type="xsd:string" minOccurs="0"/>
         <xsd:element name="street" type="xsd:string"
                      maxOccurs="unbounded"/>
         <xsd:element name="city" type="xsd:string"/>
         <xsd:element name="state" type="xsd:string" minOccurs="0"/>
         <xsd:element name="postalCode" type="xsd:string"
                      minOccurs="0"/>
         <xsd:element name="country" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
      <xsd:attribute name="href" type="xsd:IDREF"/>
   </xsd:complexType>

   <xsd:complexType name="itemType">
      <xsd:sequence>
         <xsd:element name="description" type="xsd:string"
                      minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="sku" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
               <xsd:pattern value="\d{3}-[A-Z]{2}"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="quantity" use="required"
                     type="xsd:positiveInteger"/>
   </xsd:complexType>
</xsd:schema>
```

Global Versus Local Elements and Attributes

Everything should look familiar except perhaps the standalone definition of the `po` element after the schema annotation. This brings us to the important topic of local versus global elements and attributes. Any element or attribute defined inside a complex type definition is considered local to that definition. Conversely, any element or attribute defined at the top level (as a child of `xsd:schema`) is considered global.

All global elements can be document roots. That is the main reason why most schemas define a single global element. In the case of the SkatesTown PO, the `po` element must be the root of the PO document and is hence defined as a global element.

The notion of global attributes might not make much sense at first, but these attributes are very convenient. You can use them (in namespace-prefixed form) on any element in a document that allows them. The item priority attribute discussed in the section "XML Namespaces" is defined with the short schema in Listing 2.18.

Listing 2.18 **Defining the Priority Global Attribute Using a Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/priority"
            targetNamespace="http://www.skatestown.com/ns/priority"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:attribute name="priority" use="optional" default="medium">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:enumeration value="low"/>
            <xsd:enumeration value="medium"/>
            <xsd:enumeration value="high"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:attribute>
</xsd:schema>
```

Basic Schema Reusability

The concept of reusability is important for XML Schema. Reusability deals with the question of how to best leverage existing assets in new projects. In schemas, the assets include element and attribute definitions, content model definitions, simple and complex datatypes, and whole schemas. We can roughly break down reusability mechanisms into two kinds: basic and advanced. The basic reusability mechanisms address the problems of using existing assets in multiple places. Advanced reusability mechanisms address the problems of modifying existing assets to serve needs that are different than those for which the assets were originally designed.

This section will address the following basic reusability mechanisms:

- Element references
- Content model groups
- Attribute groups
- Schema includes
- Schema imports

Element References

In XML Schema, you can define elements using a name and a type. Alternatively, element declarations can refer to preexisting elements using the `ref` attribute of `xsd:element` as follows, where a globally defined `comment` element is reused for both person and task complex types:

```
<xsd:element name="comment" type="xsd:string"/>

<xsd:complexType name="personType">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element ref="comment" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="taskType">
   <xsd:sequence>
      <xsd:element name="toDo" type="xsd:string"/>
      <xsd:element ref="comment" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
```

Content Model Groups

Element references are perfect for reusing the definition of a single element. However, if your goal is to reuse all or part of a content model, then element groups are the way to go. Element groups are defined using `xsd:group` and are referred to using the same mechanism used for elements. The following schema fragment illustrates the concept. It extends the previous example so that instead of a single `comment` element, public and private comment elements are reused as a group:

```
<xsd:group name="comments">
   <xsd:sequence>
      <xsd:element name="publicComment" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="privateComment" type="xsd:string"
                   minOccurs="0"/>
   </xsd:sequence>
</xsd:group>

<xsd:complexType name="personType">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:group ref="comments"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="taskType">
   <xsd:sequence>
```

```
      <xsd:element name="toDo" type="xsd:string"/>
      <xsd:group ref="comments"/>
   </xsd:sequence>
</xsd:complexType>
```

Attribute Groups

The same reusability mechanism can be applied to commonly used attribute groups. The following example defines the `ID/IDREF` combination of `id` and `href` attributes as a referenceable attribute group. It's then applied to both the person and the task type:

```
<xsd:attributeGroup name="referenceable">
   <xsd:attribute name="id" type="xsd:ID"/>
   <xsd:attribute name="href" type="xsd:IDREF"/>
</xsd:attributeGroup>

<xsd:complexType name="personType">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="referenceable"/>
</xsd:complexType>

<xsd:complexType name="taskType">
   <xsd:sequence>
      <xsd:element name="toDo" type="xsd:string"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="referenceable"/>
</xsd:complexType>
```

Schema Includes and Imports

Element references and groups as well as attribute groups provide reusability within the same schema document. However, when you're dealing with very complex schema or trying to achieve maximum reusability, you'll often need to split a schema into several documents. The schema *include* and *import* mechanisms allow these documents to reference one another.

Consider the scenario where SkatesTown is intent on reusing the schema definition for its address type for a mailing list schema. SkatesTown must solve three small problems:

- Put the address type definition in its own schema document
- Reference this schema document from the purchase order schema document
- Reference this schema document from the mailing list schema document

Pulling the address definition into its own schema is as easy as a cut-and-paste operation (see Listing 2.19). Even though this is a different document than the main purchase order schema, they both define portions of the SkatesTown PO namespace. The binding between schema documents and the namespaces they define isn't one-to-one. It's explicitly identified by the `targetNamespace` attribute of the `xsd:schema` element.

Listing 2.19 **Standalone Address Type Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.skatestown.com/ns/po">

   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Address type schema for SkatesTown.
      </xsd:documentation>
   </xsd:annotation>

   <xsd:complexType name="addressType">
      <xsd:sequence>
         <xsd:element name="name" type="xsd:string" minOccurs="0"/>
         <xsd:element name="company" type="xsd:string" minOccurs="0"/>
         <xsd:element name="street" type="xsd:string"
                      maxOccurs="unbounded"/>
         <xsd:element name="city" type="xsd:string"/>
         <xsd:element name="state" type="xsd:string" minOccurs="0"/>
         <xsd:element name="postalCode" type="xsd:string"
                      minOccurs="0"/>
         <xsd:element name="country" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
      <xsd:attribute name="href" type="xsd:IDREF"/>
   </xsd:complexType>

</xsd:schema>
```

Referring to this schema is also easy. Instead of having the address type definition inline, the PO schema needs to include the address schema using the `xsd:include` element. During the processing of the PO schema, the address schema will be retrieved and the address type definition will become available (see Listing 2.20).

Listing 2.20 **Referring to the Address Type Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.skatestown.com/ns/po">
```

Listing 2.20 **Continued**

```
   <xsd:include
      schemaLocation="http://www.skatestown.com/schema/address.xsd"/>

   ...
</xsd:schema>
```

The mailing list schema is very simple. It defines a single `mailingList` element that contains any number of contact elements whose type is `address`. Being an altogether different schema than that used for POs, the mailing list schema uses a new namespace: `http://www.skatestown.com/ns/mailingList`. Listing 2.21 shows one possible way to define this schema.

Listing 2.21 **Mailing List Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.skatestown.com/ns/mailingList">

   <xsd:include
      schemaLocation="http://www.skatestown.com/schema/address.xsd"/>

   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Mailing list schema for SkatesTown.
      </xsd:documentation>
   </xsd:annotation>

   <xsd:element name="mailingList">
      <xsd:sequence>
         <xsd:element name="contact" type="addressType"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:element>

</xsd:schema>
```

This example uses `xsd:include` to bring in the schema fragment defining the address type. There is no problem with that approach. However, there might be a problem with authoring mailing-list documents. The root of the problem is that the `mailingList` and `contact` elements are defined in one namespace (`http://www.skatestown.com/ns/mailingList`), whereas the elements belonging to the address type—`name`, `company`, `street`, `city`, `state`, `postalCode`, and `country`—are defined in another (`http://www.skatestown.com/ns/po`). Therefore, the mailing list document must reference both namespaces (see Listing 2.22).

Listing 2.22 **Mailing List That References Two Namespaces**

```
<?xml version="1.0" encoding="UTF-8"?>
<list:mailingList xmlns:list="http://www.skatestown.com/ns/mailingList"
   xmlns:addr="http://www.skatestown.com/ns/po"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.skatestown.com/ns/mailingList
                       http://www.skatestown.com/schema/mailingList.xsd
                       http://www.skatestown.com/ns/po
                       http://www.skatestown.com/schema/address.xsd">
   <contact>
      <addr:company>The Skateboard Warehouse</addr:company>
      <addr:street>One Warehouse Park</addr:street>
      <addr:street>Building 17</addr:street>
      <addr:city>Boston</addr:city>
      <addr:state>MA</addr:state>
      <addr:postalCode>01775</addr:postalCode>
   </contact>
</list:mailingList>
```

Ideally, when reusing the address type definition in the mailing list schema, we want to hide the fact that it originates from a different namespace and treat it as a true part of the mailing list schema. Therefore, the `xsd:include` mechanism isn't the right one to use, because it makes no namespace changes. The reuse mechanism that will allow the merging of schema fragments from multiple namespaces into a single schema is the import mechanism. Listing 2.23 shows the new mailing list schema.

Listing 2.23 **Importing Rather Than Including the Address Type Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:addr="http://www.skatestown.com/ns/po"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.skatestown.com/ns/po
               http://www.skatestown.com/schema/address.xsd"
            targetNamespace="http://www.skatestown.com/ns/mailingList">

   <xsd:import namespace="http://www.skatestown.com/ns/po"/>

   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Mailing list schema for SkatesTown.
      </xsd:documentation>
   </xsd:annotation>

   <xsd:element name="mailingList">
      <xsd:sequence>
```

Listing 2.23 **Continued**

```
         <xsd:element name="contact" type="addr:addressType"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:element>

</xsd:schema>
```

Although the mechanism is simple to describe, it takes several steps to execute:

1. Declare the namespace of the address type definition and assign it the prefix `addr`.
2. Use the standard `xsi:schemaLocation` mechanism to hint about the location of the address schema.
3. Use `xsd:import` instead of `xsd:include` to merge the contents of the PO namespace into the mailing list namespace.
4. When referring to the address type, use its fully qualified name: `addr:addressType`.

The net result is that the mailing list instance document has been simplified (see Listing 2.24).

Listing 2.24 **Simplified Instance Document That Requires a Single Namespace**

```
<?xml version="1.0" encoding="UTF-8"?>
<list:mailingList xmlns:list="http://www.skatestown.com/ns/mailingList"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.skatestown.com/ns/mailingList
                       http://www.skatestown.com/schema/mailingList.xsd">
   <contact>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </contact>
</list:mailingList>
```

Advanced Schema Reusability

The previous section demonstrated how you can reuse types and elements as is from the same or a different namespace. This capability can go a long way in some cases, but many real-world scenarios require more sophisticated reuse capabilities. Consider, for example, the format of the invoice that SkatesTown will send to the Skateboard Warehouse based on its PO (see Listing 2.25).

Listing 2.25 **SkatesTown Invoice Document**

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice:invoice xmlns:invoice="http://www.skatestown.com/ns/invoice"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.skatestown.com/ns/invoice
                       http://www.skatestown.com/schema/invoice.xsd"
   id="43871" submitted="2004-01-05" customerId="73852">
   <billTo id="addr-1">
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </billTo>
   <shipTo href="addr-1"/>
   <order>
      <item sku="318-BP" quantity="5" unitPrice="49.95">
         <description>Skateboard backpack; five pockets</description>
      </item>
      <item sku="947-TI" quantity="12" unitPrice="129.00">
         <description>Street-style titanium skateboard.</description>
      </item>
      <item sku="008-PR" quantity="1000" unitPrice="0.00">
         <description>Promotional: SkatesTown stickers</description>
      </item>
   </order>
   <tax>89.89</tax>
   <shippingAndHandling>200</shippingAndHandling>
   <totalCost>2087.64</totalCost>
</invoice:invoice>
```

The invoice document has many of the features of a PO document, with a few important changes:

- Invoices use a different namespace: `http://www.skatestown.com/ns/invoice`.
- The root element of the document is `invoice`, not `po`.
- The `invoice` element has three additional children: `tax`, `shippingAndHandling`, and `totalCost`.
- The `item` element has an additional attribute: `unitPrice`.

How can we leverage the work done to define the PO schema in defining the invoice schema? This section will introduce the advanced schema reusability mechanisms that make this possible.

Design Principles

Imagine that purchase orders, addresses, and items were represented as classes in an object-oriented programming language such as Java. We could create an invoice object by subclassing `item` to `invoiceItem` (which adds `unitPrice`) and `po` to `invoice` (which adds `tax`, `shippingAndHandling`, and `totalCost`). The benefit of this approach is that any changes to related classes such as `address` will be automatically picked up by both POs and invoices. Further, any changes in base types such as `item` will be automatically picked up by derived types such as `invoiceItem`.

The following pseudocode shows how this approach might work:

```
public class Address { ... }

public class Item
{
   public String sku;
   public int quantity;
}

public class InvoiceItem extends Item
{
   public double unitPrice;
}

public class PO
{
   public int id;
   public Calendar submitted;
   public int customerId;
   public Address billTo;
   public Address shipTo;
   public Item order[];
}

public class Invoice extends PO
{
   public double tax;
   public double shippingAndHandling;
   public double totalCost;
}
```

Everything looks good except for one important detail. You might have noticed that `Invoice` shouldn't subclass `PO`, because the `order` array inside an `invoice` object must hold `InvoiceItems` and not just `Item`. The subclassing relationship will force you to work with `Items` instead of `InvoiceItems`. Doing so will weaken static type-checking and require constant downcasting, which is generally a bad thing in well-designed object-oriented systems. A better design for the `Invoice` class, unfortunately, requires some duplication of `PO`'s data members:

```
public class Invoice
{
   public int id;
   public Calendar submitted;
   public int customerId;
   public Address billTo;
   public Address shipTo;
   public InvoiceItem order[];
   public double tax;
   public double shippingAndHandling;
   public double totalCost;
}
```

Note that subclassing `Item` to get `InvoiceItem` is a good decision because `InvoiceItem` is a pure extension of `Item`. It adds new data members; it doesn't require modifications to `Item`'s data members, nor does it change the way they're used.

Extensions and Restrictions

The analysis from object-oriented systems can be directly applied to the design of SkatesTown's invoice schema. The schema defines the `invoice` element in terms of pre-existing types such as `addressType`, and the invoice's `item` type reuses the already-defined purchase order item type via *extension* (see Listing 2.26).

Listing 2.26 **SkatesTown Invoice Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.skatestown.com/ns/invoice"
    targetNamespace="http://www.skatestown.com/ns/invoice"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:po="http://www.skatestown.com/ns/po">

   <xsd:import namespace="http://www.skatestown.com/ns/po"
      schemaLocation="http://www.skatestown.cm/schema/po.xsd"/>

   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Invoice schema for SkatesTown.
      </xsd:documentation>
   </xsd:annotation>

   <xsd:element name="invoice" type="invoiceType"/>

   <xsd:complexType name="invoiceType">
      <xsd:sequence>
         <xsd:element name="billTo" type="po:addressType"/>
         <xsd:element name="shipTo" type="po:addressType"/>
         <xsd:element name="order">
```

Listing 2.26 **Continued**

```
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="item" type="itemType"
                               maxOccurs="unbounded"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="tax" type="priceType"/>
         <xsd:element name="shippingAndHandling" type="priceType"/>
         <xsd:element name="totalCost" type="priceType"/>
      </xsd:sequence>
      <xsd:attribute name="id" use="required"
                     type="xsd:positiveInteger"/>
      <xsd:attribute name="submitted" use="required"
                     type="xsd:date"/>
      <xsd:attribute name="customerId" use="required"
                     type="xsd:positiveInteger"/>
   </xsd:complexType>

   <xsd:complexType name="itemType">
      <xsd:complexContent>
         <xsd:extension base="po:itemType">
            <xsd:attribute name="unitPrice" use="required"
                           type="priceType"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <xsd:simpleType name="priceType">
      <xsd:restriction base="xsd:decimal">
         <xsd:minInclusive value="0"/>
      </xsd:restriction>
   </xsd:simpleType>

</xsd:schema>
```

By now the schema mechanics should be familiar. The beginning of the schema declares the PO and invoice namespaces. The PO schema has to be imported because it doesn't reside in the same namespace as the invoice schema.

The `invoiceType` schema address type is defined in terms of `po:addressType`, but the `order` element's content is of type `itemType` and not `po:itemType`. That's because the invoice's `itemType` needs to extend `po:itemType` and add the `unitPrice` attribute. This happens at the next complex type definition. In general, the schema extension syntax, although somewhat verbose, is easy to use:

```
<xsd:complexType name="...">
   <xsd:complexContent>
      <xsd:extension base="...">
         <!-- Optional extension content model -->
         <!-- Optional extension attributes -->
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

The content model of extended types contains all the child elements of the base type plus any additional elements added by the extension. Any attributes in the extension are added to the attribute set of the base type.

Last but not least, the invoice schema defines a simple price type as a non-negative decimal number. The definition happens via restriction of the lower boundary of the decimal type using the same mechanism introduced in the section on simple types.

The restriction mechanism in schemas applies not only to simple types but also to complex types. The syntax is similar to that of extension:

```
<xsd:complexType name="...">
   <xsd:complexContent>
      <xsd:restriction base="...">
         <!-- Content model and attributes -->
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

The concept of restriction has a precise meaning in XML Schema. The declarations of the type derived by restriction are very close to those of the base type but more limited. There are several possible types of restrictions:

- Multiplicity restrictions
- Deletion of optional elements
- Tighter limits on occurrence constraints
- Provision of default values
- Provision of types where there were none, or narrowing of types

For example, we can extend the address type by restriction to create a corporate address that doesn't include a name:

```
<xsd:complexType name="corporateAddressType">
   <xsd:complexContent>
      <xsd:restriction base="addressType">
         <xsd:sequence>
            <!-- Add maxOccurs="0" to delete optional name element -->
            <xsd:element name="name" type="xsd:string"
                         minOccurs="0" maxOccurs="0"/>
            <!-- The rest is the same as in addressType -->
            <xsd:element name="company" type="xsd:string"
                         minOccurs="0"/>
            <xsd:element name="street" type="xsd:string"
                         maxOccurs="unbounded"/>
            <xsd:element name="city" type="xsd:string"/>
            <xsd:element name="state" type="xsd:string"
                         minOccurs="0"/>
            <xsd:element name="postalCode" type="xsd:string"
                         minOccurs="0"/>
            <xsd:element name="country" type="xsd:string"
                         minOccurs="0"/>
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
         <xsd:attribute name="href" type="xsd:IDREF"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

The Importance of `xsi:type`

The nature of restriction is such that an application that is prepared to deal with the base type can certainly accept the derived type. In other words, you can use a corporate address type directly inside the `billTo` and `shipTo` elements of POs and invoices without a problem. Sometimes, however, it might be convenient to identify the actual schema type used in an instance document. XML Schema allows you to do this through the use of the global `xsi:type` attribute. This attribute can be applied to any element to signal its actual schema type, as Listing 2.27 shows.

Listing 2.27 **Using `xsi:type`**

```
<?xml version="1.0" encoding="UTF-8"?>
<po:po xmlns:po="http://www.skatestown.com/ns/po"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.skatestown.com/ns/po
                           http://www.skatestown.com/schema/po.xsd"
       id="43871" submitted="2004-01-05" customerId="73852">
   <billTo xsi:type="po:corporateAddressType">
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </billTo>
   ...
</po:po>
```

Although derivation by restriction doesn't require the use of `xsi:type`, derivation by extension often does. The reason is that an application prepared for the base schema type is unlikely to be able to process the derived type (it adds information) without a hint. But why would such a scenario ever occur? Why would an instance document contain data from a type derived by extension in a place where the schema expects a base type?

XML Schema allows derivation by extension to be used in cases where it really shouldn't be used, as in the case of the invoice and PO datatypes. In these cases, you must use `xsi:type` in the instance document to ensure successful validation. Consider a scenario where the invoice type was derived by extension from the PO type:

```
<xsd:complexType name="invoiceType">
   <xsd:complexContent>
      <xsd:extension base="po:poType">
         <xsd:element name="tax" type="priceType"/>
         <xsd:element name="shippingAndHandling" type="priceType"/>
         <xsd:element name="totalCost" type="priceType"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Remember, extension doesn't change the content model of the base type; it can only add to the content model. Therefore, this definition will make the `item` element inside invoices of type `po:itemType`, not `invoice:itemType`. The use of `xsi:type` (see Listing 2.28) is the only way to add unit prices to items without violating the validity constraints of the document imposed by the schema. An imperfect analogy from programming languages is that `xsi:type` provides the true type to downcast to when you're holding a reference to a base type.

Listing 2.28 **Using `xsi:type` to Correctly Identify Invoice Item Elements**

```
<order>
   <item sku="318-BP" quantity="5" unitPrice="49.95"
         xsi:type="invoice:itemType">
      <description>Skateboard backpack; five pockets</description>
   </item>
   <item sku="947-TI" quantity="12" unitPrice="129.00"
         xsi:type="invoice:itemType">
      <description>Street-style titanium skateboard.</description>
   </item>
   <item sku="008-PR" quantity="1000" unitPrice="0.00"
         xsi:type="invoice:itemType">
      <description>Promotional: SkatesTown stickers</description>
   </item>
</order>
```

This example shows a use of `xsi:type` that comes as a result of poor schema design. If, instead of extending PO, the invoice type is defined on its own, the need for `xsi:type` disappears. However, sometimes even good schema design doesn't prevent the need to identify actual types in instance documents.

Imagine that, due to constant typos in shipping and billing address postal codes, SkatesTown decides to become more restrictive in its document validation. The company defines three types of addresses that are part of POs and schema. The types have the following constraints:

- `Address`—Same as always
- `USAddress`—Country isn't allowed, and the ZIP Code pattern "`\d{5}(-\d{4})?`" is enforced
- `UKAddress`—Country is fixed to UK, and the postal code pattern "`[0-9A-Z]{3} [0-9A-Z]{3}`" is enforced

To get the best possible validation, SkatesTown's applications need to know the exact type of address that is being used in a document. Without using `xsi:type`, the PO and invoice schema will each have to define nine (three squared) possible combinations of `billTo` and `shipTo` elements: `billTo/shipTo`, `billTo/shipToUS`, `billTo/shipToUK`, `billToUS/shipTo`, and so on. It's better to stick with `billTo` and `shipTo` and use `xsi:type` to get exact schema type information.

There's More

This completes the whirlwind tour of XML Schema. Much material useful for data-oriented applications falls outside the scope of what is included in this chapter; we'll introduce some information throughout the book as needed.

Processing XML

So far, this chapter has introduced the key XML standards and explained how they're expressed in XML documents. The final section of the chapter focuses on processing XML, with a quick tour of the specifications and APIs you need to know to be able to generate, parse, and process XML documents in your Java applications.

Basic Operations

The basic XML processing architecture shown in Figure 2.4 consists of three key layers. At far left are the XML documents an application needs to work with. At far right is the application. In the middle is the infrastructure layer for working with XML documents, which is the topic of this section.

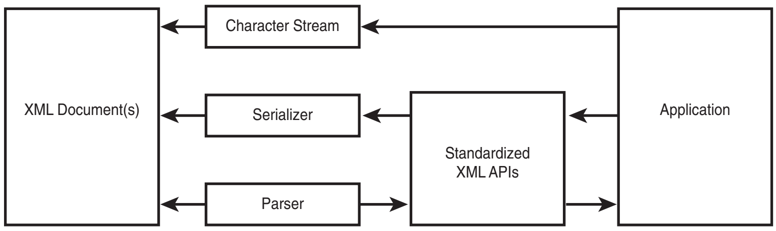


Figure 2.4 Basic XML processing architecture

For an application to be able to work with an XML document, it must first be able to parse the document. *Parsing* is a process that involves breaking the text of an XML document into small identifiable pieces (*nodes*). Parsers break documents into pieces such as start tags, end tags, attribute value pairs, chunks of text content, processing instructions, comments, and so on. These pieces are fed into the application using a well-defined API implementing a particular parsing model. Four parsing models are commonly used:

- *Pull parsing* 📖—The application always has to ask the parser to give it the next piece of information about the document. It's as if the application has to "pull" the information out of the parser (hence the name of the model). The XML community has not yet defined standard APIs for pull parsing. However, because pull parsing is becoming popular, this could happen soon.
- *Push parsing* 📖—The parser sends notifications to the application about the types of XML document pieces it encounters during the parsing process. The notifications are sent in reading order, as they appear in the text of the document. Notifications are typically implemented as event callbacks in the application code, and thus push parsing is also commonly known as *event-based parsing*. The XML community created a de facto standard for push parsing called *Simple API for XML (SAX)* 📖. SAX is currently released in version 2.0.
- *One-step parsing* 📖—The parser reads the whole XML document and generates a data structure (a *parse tree* 📖) describing its contents (elements, attributes, PIs, comments, and so on). The data structure is typically deeply nested; its hierarchy mimics the nesting of elements in the parsed XML document. The W3C has defined a *Document Object Model (DOM)* 📖 for XML. The XML DOM specifies the types of objects that are included in the parse tree, their properties, and their operations. The DOM is so popular that one-step parsing is typically referred to as *DOM parsing*. The DOM is a language- and platform-independent API. It offers many obvious benefits but also some hidden costs. The biggest problem with the DOM APIs is that they often don't map well to the native data structures of programming languages. To address this issue for Java, the Java community has started working on a Java DOM (JDOM) specification whose goal is to simplify the manipulation of document trees in Java by using object APIs tuned to the common patterns of Java programming.

- *Hybrid parsing* 📖—This approach combines characteristics of the other three parsing models to create efficient parsers for special scenarios. For example, one common pattern combines pull parsing with one-step parsing. In this model, the application thinks it's working with a one-step parser that has processed the whole XML document from start to end. In reality, the parsing process has just begun. As the application keeps accessing more objects on the DOM (or JDOM) tree, the parsing continues incrementally so that just enough of the document is parsed at any given point to give the application the objects it wants to see.

The reasons there are so many different models for parsing XML have to do with trade-offs between memory efficiency, computational efficiency, and ease of programming. Table 2.5 identifies some of the characteristics of the parsing models. In the table, *control of parsing* refers to who manages the step-by-step parsing process. Pull parsing requires that the application do that; in all other models, the parser takes care of this process. *Control of context* refers to who manages context information such as the level of nesting of elements and their location relative to one another. Both push and pull parsing delegate this control to the application; all other models build a tree of nodes that makes maintaining context much easier. This approach makes programming with DOM or JDOM generally easier than working with SAX. The price is memory and computational efficiency, because instantiating all these objects takes up time and memory. Hybrid parsers attempt to offer the best of both worlds by presenting a tree view of the document but doing incremental parsing behind the scenes.

Table 2.5 **XML Parsing Models and Their Trade-offs**

Model	Control of parsing	Control of context	Memory efficiency	Computational efficiency	Ease of programming
Pull	Application	Application	High	Highest	Low
Push (SAX)	Parser	Application	High	High	Low
One-step (DOM)	Parser	Parser	Lowest	Lowest	High
One-step (JDOM)	Parser	Parser	Low	Low	Highest
Hybrid (DOM)	Parser	Parser	Medium	Medium	High
Hybrid (JDOM)	Parser	Parser	Medium	Medium	Highest

In the Java world, a standardized API—*Java API for XML Processing (JAXP)* 📖—exists for instantiating XML parsers and parsing documents using either SAX or DOM. Without JAXP, Java applications weren't completely portable across XML parsers because different parsers, despite following SAX and DOM, had different APIs for creation, configuration, and parsing of documents. JAXP is currently released in version 1.2. It doesn't support JDOM yet because the JDOM specification isn't complete at this point.

Although XML parsing addresses the problem of feeding data from XML documents into applications, XML output addresses the reverse problem—applications generating XML documents. At the most basic level, an application can directly output XML markup. In Figure 2.4, this is indicated by the application working with a character stream. This isn't difficult to do, but handling the basic syntax rules (attributes quoting, special character escaping, and so on) can become cumbersome. In many cases, it might be easier for the application to construct a data structure (DOM or JDOM tree) describing the XML document that should be generated. Then, the application can use a *serialization* 📖 process to traverse the document tree and emit XML markup corresponding to its elements. This capability isn't directly defined in the DOM and JDOM APIs, but most XML toolkits make it very easy to do just that.

Data-Oriented XML Processing

When you're thinking about applications working with XML, it's important to note that all the mechanisms for parsing and generating XML described so far are *syntax-oriented*. They force the application to work with concepts such as elements, attributes, and pieces of text. This is similar to applications that use text files for storage being forced to work with characters, lines, carriage returns (CR), and line feeds (LF).

Typically, applications want a higher-level view of their data. They aren't concerned with the *physical structure* of the data, be it characters and lines in the case of text files or elements and attributes in the case of XML documents. They want to abstract this away and expose the *meaning* or semantics of the data. In other words, applications don't want to work with syntax-oriented APIs; they want to work with *data-oriented* APIs. Therefore, typical data-oriented XML applications introduce a data abstraction layer between the syntax-oriented parsing and output APIs and application logic (see Figure 2.5).

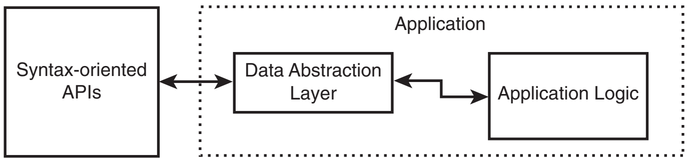


Figure 2.5 Data abstraction layer in XML applications

When working with XML in a data-oriented manner, you'll typically use one of two approaches: *operation-centric* and *data-centric*.

The Operation-Centric Approach

The operation-centric approach works in terms of custom-built APIs for certain operations on the XML document. The implementation of these APIs hides the details of XML processing. Only non-XML types are passed through the APIs.

Consider, for example, the task of SkatesTown trying to independently check the total amount on the invoices it's sending to its customers. From a Java application perspective, a good way to implement an operation like this would be through the interface shown in Listing 2.29.

Listing 2.29 `InvoiceChecker` **Interface**

```
package com.skatestown.invoice;

import java.io.InputStream;

/**
 * SkatesTown invoice checker
 */
public interface InvoiceChecker {
    /**
     * Check invoice totals.
     *
     * @param       invoiceXML Invoice XML document
     * @exception   Exception  Any exception returned during checking
     */
    void checkInvoice(InputStream invoiceXML) throws Exception;
}
```

The implementation of `checkInvoice()` must do the following:

1. Obtain an XML parser.
2. Parse the XML from the input stream.
3. Initialize a running total to zero.
4. Find all order items, and calculate item subtotals by multiplying quantities and unit prices. Add the item subtotals to the running total.
5. Add tax to the running total.
6. Add shipping and handling to the running total.
7. Compare the running total to the total on the invoice.
8. If there is a difference, throw an exception.
9. Otherwise, return.

The most important aspect of this approach is that any XML processing details are hidden from the application. It can happily deal with the `InvoiceChecker` interface, never knowing or caring about how `checkInvoice()` works.

The Data-Centric Approach

An alternative is the data-centric approach. Data-centric XML computing reduces the problem of working with XML documents to that of mapping the XML to and from application data and then working with the data independently of its XML origins. Application data covers the common datatypes developers work with every day: boolean values, numbers, strings, date-time values, arrays, associative arrays (dictionaries, maps, hash tables), database recordsets, and complex object types. Note that in this context, DOM tree objects aren't considered true application data because they're tied to XML syntax. The process of converting application data to XML is called *marshalling* 📖. The XML is a serialized representation of the application data. The process of generating application data from XML is called *unmarshalling* 📖.

For example, the XML invoice markup could be mapped to the set of Java classes introduced in the schema section (see Listing 2.30).

Listing 2.30 **Java Classes Representing Invoice Data**

```
class Address { ... }

class Item { ... }

class InvoiceItem extends Item { ... }

class Invoice
{
    int id;
    Date submitted;
    int customerId;
    Address billTo;
    Address shipTo;
    InvoiceItem order[];
    double tax;
    double shippingAndHandling;
    double totalCost;
}
```

Schema Compilers

The traditional approach for generating XML from application data has been to custom-code the way data values become elements, attributes, and element content. The traditional approach of working with XML to produce application data has been to parse it using a SAX or a DOM parser. Data structures are built from the SAX events or the DOM tree using custom code. However, there are better ways to map data to and from XML using technologies specifically built for marshalling and unmarshalling data to and from XML. Enter schema compilation tools.

Schema compilers are tools that analyze XML schema and code-generate marshalling and unmarshalling modules specific to the schema. These modules work with data structures tuned to the schema. Figure 2.6 shows the basic process for working with schema compilers. The schema compiler needs to be invoked only once; then the application can use the code-generated modules like any other API.

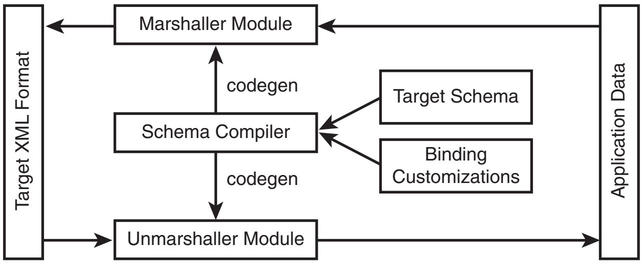


Figure 2.6 Using a schema compiler.

Binding Customization

In some cases, the object types generated by the schema compiler offer a good enough API for working with the types and elements described in the target schema. The application can use these classes directly. Other cases may require customization of the default binding of XML types to object types. That is where the binding customizations come in: They provide additional information to the schema compiler about how the binding between XML and application structures should happen.

There are two main reasons for applying customization:

- *To deal with predefined application data structures*—This reason applies in environments where the application already has defined object types to represent the concepts described in the schema. An example is a PO processing system that was designed to receive inputs from a human-facing UI and an EDI data feed. Now, the system must be extended to handle XML POs. The task is to map the XML of POs to the existing application data structures. There is zero chance that the default mapping defined by the schema compiler will do this in a satisfactory manner.
- *To simplify the API*—This reason for applying customization is driven by programming convenience. Sometimes the conventions of schema design don't map well to the conventions of object-oriented design. For example, localized text is often represented in schema as a subelement with an `xml:lang` attribute identifying the language. Most applications represent this construct as a string object property whose value is determined by the active internationalization locale. Further, there is often more than one way to express a schema type in a programming language. For example, should an `xsd:decimal` be mapped to a `BigDecimal`, `double`, or `float` in Java? The right answer depends on the application.

Common examples of customizations include the following:

- Changing the names of namespaces, object types, and object properties; for example, mapping the `customerID` attribute to the `_cid` object property.
- Defining the type mapping, especially for simple types, as the previous `xsd:decimal` mapping example suggested.

- Choosing which subelements to map to object properties and whether to map them as simple types or as properties that are objects themselves, as the localized text example suggested.
- Specifying how repeated types should be mapped to collection types in programming languages. For example, should the order items in POs be represented by a simple array type, a dynamic array type, or some other data structure such as a list?

The Java community has defined a standard set of tools and APIs for mapping schema types to Java data structures called *Java Architecture for XML Binding (JAXB)*. JAXB took a long time to develop because the problems it was trying to address were very complex. Initially, the work targeted DTD-to-Java mapping. This, and the fact that JAXB was JSR-31 in the Java Community Process (JCP), gives you an idea of how long JAXB has taken to evolve. Because of its long history, JAXB isn't yet fully aligned with the latest thinking about XML type mapping for Web services. The good news is that JAXB now supports a significant part of XML Schema and is ready for production use. JAXB 2.0 will synchronize JAXB with JAX-RPC (the Java APIs for remote procedure calls using Web services), which will make JAXB even better suited for use with Web services.

Chapters 3 and 5 ("Implementing Web Services with Apache Axis") introduce advanced data-mapping concepts specific to Web services as well as more sophisticated mechanisms for working with XML. The rest of this section will offer a taste of XML processing by implementing the `checkInvoice()` API described earlier using both a SAX and a DOM parser as well as JAXB.

SAX-Based `checkInvoice()`

The basic architecture of the JAXP SAX parsing APIs is shown in Figure 2.7. It uses the common abstract factory design pattern. First, you must create an instance of `SAXParserFactory` that is used to create an instance of `SAXParser`. Internally, the parser wraps a `SAXReader` object that is defined by the SAX API. JAXP developers typically don't have to work directly with `SAXReader`. When the parser's `parse()` method is invoked, the reader starts firing events to the application by invoking certain registered callbacks.

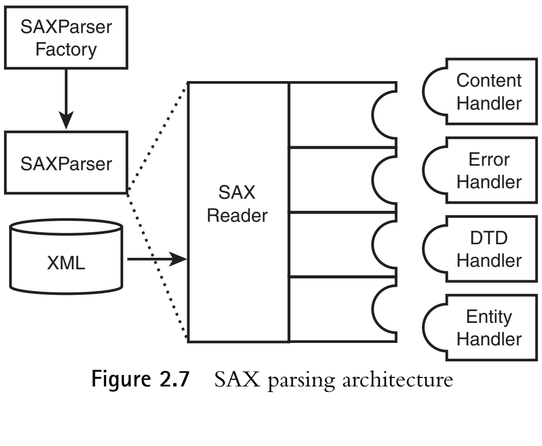


Figure 2.7 SAX parsing architecture

Working with JAXP and SAX involves four important Java packages:

- `org.xml.sax`—Defines the SAX interfaces
- `org.xml.sax.ext`—Defines advanced SAX extensions for DTD processing and detailed syntax information
- `org.xml.sax.helpers`—Defines helper classes such as `DefaultHandler`
- `javax.xml.parsers`—Defines the `SAXParserFactory` and `SAXParser` classes

Here is a summary of the key SAX-related objects:

- `SAXParserFactory`—A `SAXParserFactory` object creates an instance of the parser determined by the system property `javax.xml.parsers.SAXParserFactory`.
- `SAXParser`—The `SAXParser` interface defines several kinds of `parse()` methods. In general, you pass an XML data source and a `DefaultHandler` object to the parser, which processes the XML and invokes the appropriate methods in the handler object.
- `DefaultHandler`—Not shown in Figure 2.7, `DefaultHandler` implements all SAX callback interfaces with null methods. Custom handlers subclass `DefaultHandler` and override the methods they're interested in receiving.

The following list contains the callback interfaces and some of their important methods:

- `ContentHandler`—Contains methods for all basic XML parsing events:

  ```
  void startDocument()
  ```
  Receive notification of the beginning of a document.

  ```
  void endDocument()
  ```
  Receive notification of the end of a document.

  ```
  void startElement(String namespaceURI, String localName,
                    String qName, Attributes atts)
  ```
  Receive notification of the beginning of an element.

  ```
  void characters(char[] ch, int start, int length)
  ```
  Receive notification of character data.

- `ErrorHandler`—Contains methods for receiving error notification. The default implementation in `DefaultHandler` throws errors for fatal errors but does nothing for nonfatal errors, including validation errors:

  ```
  void error(SAXParseException exception)
  ```
  Receive notification of a recoverable error (for example, a validation error).

  ```
  void fatalError(SAXParseException exception)
  ```
  Receive notification of a nonrecoverable error (for example, a well-formedness error).

- `DTDHandler`—Contains methods for dealing with XML entities.
- `EntityResolver`—Contains methods for resolving the location of external entities.

SAX defines an event-based parsing model. A SAX parser invokes the callbacks from these interfaces as it's working through the document. Consider the following sample document:

```
<?xml version="1.0" encoding="UTF-8"?>
<sampleDoc>
   <greeting>Hello, world!</greeting>
</sampleDoc>
```

An event-based parser will make the series of callbacks to the application as follows:

```
start document
start element: sampleDoc
start element: greeting
characters: Hello, world!
end element: greeting
end element: sampleDoc
end document
```

Because of the simplicity of the parsing model, the parser doesn't need to keep much state information in memory. This is why SAX-based parsers are fast and highly efficient. The flip side to this benefit is that the application has to manage any context associated with the parsing process. For example, for the application to know that the string "Hello, world!" is associated with the `greeting` element, it needs to maintain a flag that is raised in the start element event for `greeting` and lowered in the end element event. More complex applications typically maintain a stack of elements that are in the process of being parsed. Here are the SAX events with an added context stack:

```
start document             ()
start element: sampleDoc   (sampleDoc)
start element: greeting    (sampleDoc, greeting)
characters: Hello, world!  (sampleDoc, greeting)
end element: greeting      (sampleDoc, greeting)
end element: sampleDoc     (sampleDoc)
end document               ()
```

With this information in mind, building a class to check invoice totals becomes relatively simple (see Listing 2.31).

Listing 2.31 **SAX-Based Invoice Checker (`InvoiceCheckerSAX.java`)**

```
package com.skatestown.invoice;

import java.io.InputStream;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.helpers.DefaultHandler;

/**
 * Check SkatesTown invoice totals using a SAX parser.
 */
public class InvoiceCheckerSAX
    extends DefaultHandler
    implements InvoiceChecker
{
    // Class-level data
    // invoice running total
    double runningTotal = 0.0;
    // invoice total
    double total = 0.0;

    // Utility data for extracting money amounts from content
    boolean isMoneyContent = false;
    double amount = 0.0;

    /**
     * Check invoice totals.
     * @param     invoiceXML    Invoice XML document
     * @exception Exception     Any exception returned during checking
     */
    public void checkInvoice(InputStream invoiceXML) throws Exception {
        // Use the default (non-validating) parser
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser saxParser = factory.newSAXParser();

        // Parse the input; we are the handler of SAX events
        saxParser.parse(invoiceXML, this);
    }

    // SAX DocumentHandler methods
    public void startDocument() throws SAXException {
        runningTotal = 0.0;
        total = 0.0;
        isMoneyContent = false;
    }
```

Listing 2.31 **Continued**

```
    public void endDocument() throws SAXException {
        // Use delta equality check to prevent cumulative
        // binary arithmetic errors. In this case, the delta
        // is one half of one cent
        if (Math.abs(runningTotal - total) >= 0.005) {
            throw new SAXException(
                "Invoice error: total is " + Double.toString(total) +
                " while our calculation shows a total of " +
                Double.toString(Math.round(runningTotal * 100) / 100.0));
        }
    }

    public void startElement(String namespaceURI,
                             String localName,
                             String qualifiedName,
                             Attributes attrs) throws SAXException {
        if (localName.equals("item")) {
            // Find item subtotal; add it to running total
            runningTotal +=
                Integer.valueOf(attrs.getValue(namespaceURI,
                    "quantity")).intValue() *
                Double.valueOf(attrs.getValue(namespaceURI,
                    "unitPrice")).doubleValue();
        } else if (localName.equals("tax") ||
                   localName.equals("shippingAndHandling") ||
                   localName.equals("totalCost")) {
            // Prepare to extract money amount
            isMoneyContent = true;
        }
    }

    public void endElement(String namespaceURI,
                           String localName,
                           String qualifiedName) throws SAXException {
        if (isMoneyContent) {
            if (localName.equals("totalCost")) {
                total = amount;
            } else {
                // It must be tax or shippingAndHandling
                runningTotal += amount;
            }
            isMoneyContent = false;
        }
    }

    public void characters(char buf[], int offset, int len)
```

Listing 2.31 **Continued**

```
        throws SAXException {
        if (isMoneyContent) {
            String value = new String(buf, offset, len);
            amount = Double.valueOf(value).doubleValue();
        }
    }
}
```

`InvoiceCheckerSAX` must implement the `InvoiceChecker` interface in order to provide the `checkInvoice()` functionality. It also subclasses `DefaultHandler` to obtain default implementations for all SAX callbacks. This way, the implementation can focus on overriding only the relevant callbacks.

The class members `runningTotal` and `total` maintain state information about the invoice during the parsing process. The class members `isMoneyContent` and `amount` are necessary in order to maintain parsing context. Because events about character data are independent of events about elements, we need the `isMoneyContent` flag to indicate whether we should attempt to parse character data as a dollar amount for the `tax`, `shippingAndHandling`, and `totalCost` elements. After we parse the text into a dollar figure, we save it into the `amount` member variable and wait until the `endElement()` callback to determine what to do with it.

The `checkInvoice()` method implementation shows how easy it is to use JAXP for XML parsing. Parsing an XML document with SAX only takes three lines of code.

At the beginning of the document, we have to initialize all member variables. At the end of the document, we check whether there is a difference between the running total and the total cost listed on the invoice. If there is a problem, we throw an exception with a descriptive message. Note that we can't use an equality check because no exact mapping exists between decimal numbers and their binary representation. During the many additions to `runningTotal`, a tiny error will be introduced in the calculation. So, instead of checking for equality, we need to check whether the difference between the listed and the calculated totals is significant. *Significant* in this case would be any amount greater than half a cent, because a half-cent difference can affect the rounding of a final value to a cent.

The parser pushes events about new elements to the `startElement()` method. If the element we get a notification about is an `item` element, we can immediately extract the values of the `quantity` and `unitPrice` attributes from its attributes collection. Multiplying them together creates an item subtotal, which we add to the running total. Alternatively, if the element is one of `tax`, `shippingAndHandling`, or `totalCost`, we prepare to extract a money amount from its text content. All other elements are ignored.

When we receive end element notifications, we only need to process the ones where we expect to extract a money amount from their content. Based on the name of the element, we decide whether to save the amount as the total cost of the invoice or whether to add it to the running total.

When we process character data and we're expecting a dollar value, we extract the element content, convert it to a double value, and save it in the `amount` class member for use by the `endElement()` callback.

Note that we could have skipped implementing `endElement()` if we had also stored the element name as a string member of the class or used an enumerated value. Then, we would have decided how to use the dollar amount inside `characters()`.

That's all there is to it. Of course, this is a simple example. A real application would have done at least two things differently:

- It would have used namespace information and prefixed element names instead of local names.
- It would have defined its own exception type to communicate invoice validation information. It would have also overridden the default callbacks for `error()` and `fatalError()` and used these to collect better exception information.

Unfortunately, these extensions fall outside the scope of this chapter. The rest of the book has several examples of building robust XML-processing software.

DOM-Based `checkInvoice()`

The basic architecture of the JAXP DOM parsing APIs is shown in Figure 2.8. This architecture uses the same factory design pattern as the SAX API. An application uses the `javax.xml.parsers.DocumentBuilderFactory` class to get a `DocumentBuilder` object instance, and uses that to produce a document that conforms to the DOM specification. The value of the system property `javax.xml.parsers.DocumentBuilderFactory` determines which factory implementation produces the builder. This is how JAXP enables applications to work with different DOM parsers.

The important packages for working with JAXP and DOM are as follows:

- `org.w3c.dom`—Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C
- `javax.xml.parsers`—Defines `DocumentBuilder` and `DocumentBuilderFactory` classes

The DOM defines APIs that allow applications to navigate XML documents and to manipulate their content and structure. The DOM defines interfaces, not a particular implementation. These interfaces are specified using the Interface Description Language (IDL) so that any language can define bindings for them. Separate Java bindings are provided to make working with the DOM in Java easy.

The DOM has several levels and various facets within a level. In the fall of 1998, DOM Level 1 was released. It provided the basic functionality to navigate and manipulate XML and HTML documents. DOM Level 2 builds upon Level 1 with more and better-segmented functionality:

- The DOM Level 2 Core APIs build on Level 1, fix some problem spots, and define additional ways to navigate and manipulate the content and structure of documents. These APIs also provide full support for namespaces.

- The DOM Level 2 Views API specifies interfaces that let programmers view alternate presentations of the XML or HTML document.
- The DOM Level 2 Style API specifies interfaces that let programmers dynamically access and manipulate style sheets.
- The DOM Level 2 Events API specifies interfaces that give programmers a generic event system.
- The DOM Level 2 Traversal-Range API specifies interfaces that let programmers traverse a representation of the XML document.
- The DOM Level 2 HTML API specifies interfaces that let programmers work with HTML documents.

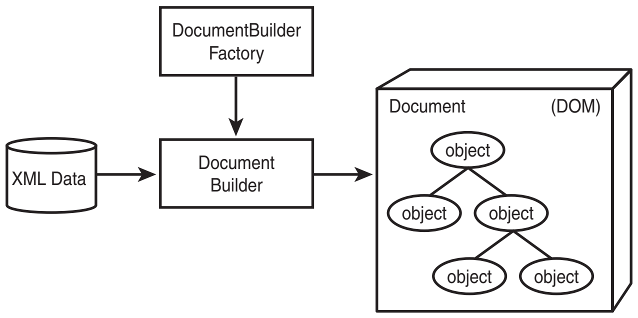


Figure 2.8 DOM parsing architecture

All interfaces (apart from the Core ones) are optional. This is the main reason most applications rely entirely on the DOM Core. You can expect parsers to support more of the DOM soon. In fact, the W3C is currently working on DOM Level 3.

The DOM originated as an API for XML processing at a time when the majority of XML applications were document-centric. As a result, the interfaces in the DOM describe low-level syntax constructs in XML documents. This makes working with the DOM for data-oriented applications somewhat cumbersome and is one of the reasons the Java community is working on the JDOM APIs.

To better understand the XML DOM, you need to be familiar with the core interfaces and their most significant methods. Figure 2.9 shows a Universal Modeling Language (UML) diagram describing some of them.

The root interface is `Node`. It contains methods for working with the node name (`getNodeName()`), type (`getNodeType()`), and attributes (`getNodeAttributes()`). `Node` types cover various XML syntax elements: document, element, attribute, character data, text node, comment, processing instruction, and so on. All of these are shown in subclass `Node`, but not all are shown in Figure 2.9. To traverse the document hierarchy, nodes can access their parent (`getParentNode()`) as well as their children (`getChildNodes()`). `Node` also has several convenience methods for retrieving the first and last child as well as the previous and following sibling.

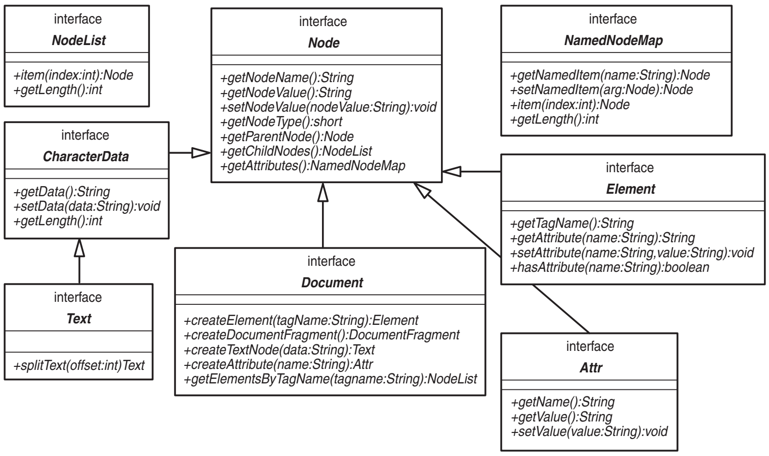

interface
NodeList
+item(index:int):Node
+getLength():int
interface
Node
+getNodeName():String
+getNodeValue():String
+setNodeValue(nodeValue:String):void
+getNodeType():short
+getParentNode():Node
+getChildNodes():NodeList
+getAttributes():NamedNodeMap
interface
NamedNodeMap
+getNamedItem(name:String):Node
+setNamedItem(arg:Node):Node
+item(index:int):Node
+getLength():int
interface
CharacterData
+getData():String
+setData(data:String):void
+getLength():int
interface
Element
+getTagName():String
+getAttribute(name:String):String
+setAttribute(name:String,value:String):void
+hasAttribute(name:String):boolean
interface
Text
+splitText(offset:int)Text
interface
Document
+createElement(tagName:String):Element
+createDocumentFragment():DocumentFragment
+createTextNode(data:String):Text
+createAttribute(name:String):Attr
+getElementsByTagName(tagname:String):NodeList
interface
Attr
+getName():String
+getValue():String
+setValue(value:String):void

Figure 2.9 Key DOM interfaces and operations

The most important operations in `Document` involve creating nodes (at least one for every node type); assembling these nodes into the tree (not shown); and locating elements by name, regardless of their location in the DOM (`getElementsByTagName()`). This last API is convenient because it can save you from having to traverse the tree to get to a particular node.

The rest of the interfaces in the figure are simple. Elements, attributes, and character data each offer a few methods for getting and setting their data members. `NodeList` and `NamedNodeMap` are convenience interfaces for dealing with collections of nodes and attributes, respectively.

What Figure 2.9 doesn't show is that DOM Level 2 is fully namespace-aware and that all DOM APIs have versions that take in namespace URIs. Typically, their name is the same as the name of the original API with *NS* appended, such as `Element`'s `getAttributeNS(String nsURI, String localName)`.

With this information in mind, building a class to check invoice totals becomes relatively simple. The DOM implementation of `InvoiceChecker` is shown in Listing 2.32.

Listing 2.32 **DOM-Based Invoice Checker (**`InvoiceCheckerDOM.java`**)**

```
package com.skatestown.invoice;

import java.io.InputStream;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.CharacterData;
import javax.xml.parsers.DocumentBuilder;
```

Listing 2.32 **Continued**

```
import javax.xml.parsers.DocumentBuilderFactory;

/**
 * Check SkatesTown invoice totals using a DOM parser.
 */
public class InvoiceCheckerDOM implements InvoiceChecker {
    /**
     * Check invoice totals.
     *
     * @param       invoiceXML Invoice XML document
     * @exception   Exception  Any exception returned during checking
     */
    public void checkInvoice(InputStream invoiceXML)
        throws Exception
    {
        // Invoice running total
        double runningTotal = 0.0;

        // Obtain parser instance and parse the document
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document doc = builder.parse(invoiceXML);

        // Calculate order subtotal
        NodeList itemList = doc.getElementsByTagName("item");
        for (int i = 0; i < itemList.getLength(); i++) {
            // Extract quantity and price
            Element item = (Element)itemList.item(i);
            Integer qty = Integer.valueOf(
                item.getAttribute("quantity"));
            Double price = Double.valueOf(
                item.getAttribute("unitPrice"));

            // Add subtotal to running total
            runningTotal += qty.intValue() * price.doubleValue();
        }

        // Add tax
        Node nodeTax = doc.getElementsByTagName("tax").item(0);
        runningTotal += doubleValue(nodeTax);

        // Add shipping and handling
        Node nodeShippingAndHandling =
            doc.getElementsByTagName("shippingAndHandling").item(0);
```

Listing 2.32 **Continued**

```
        runningTotal += doubleValue(nodeShippingAndHandling);

        // Get invoice total
        Node nodeTotalCost =
            doc.getElementsByTagName("totalCost").item(0);
        double total = doubleValue(nodeTotalCost);

        // Use delta equality check to prevent cumulative
        // binary arithmetic errors. In this case, the delta
        // is one half of one cent
        if (Math.abs(runningTotal - total) >= 0.005)
        {
            throw new Exception(
                "Invoice error: total is " + Double.toString(total) +
                " while our calculation shows a total of " +
                Double.toString(Math.round(runningTotal * 100) / 100.0));
        }
    }

    /**
     * Extract a double from the text content of a DOM node.
     *
     * @param       node A DOM node with character content.
     * @return      The double representation of the node's content.
     * @exception   Exception Could be the result of either a node
     *              that doesn't have text content being passed in
     *              or a node whose text content is not a number.
     */
    private double doubleValue(Node node) throws Exception {
        // Get the character data from the node and parse it
        String value = ((CharacterData)node.getFirstChild()).getData();
        return Double.valueOf(value).doubleValue();
    }
}
```

`InvoiceCheckerDOM` must implement the `InvoiceChecker` interface in order to provide the `checkInvoice()` functionality. Apart from this, it's a standalone class. Also, note that the class has no member data, because there is no need to maintain parsing context. The context is implicit in the hierarchy of the DOM tree that will be the result of the parsing process.

The factory pattern used here to parse the invoice is the same as the one from the SAX implementation; it just uses `DocumentBuilderFactory` and `DocumentBuilder`. Although the SAX parse method returns no data (it starts firing events instead), the DOM `parse()` method returns a `Document` object that holds the complete parse tree of the invoice document.

Within the parse tree, the call to `getElementsByTagName("item")` retrieves a node list of all order items. The loop iterates over the list, extracting the `quantity` and `unitPrice` attributes for every item, obtaining an item subtotal, and adding this to the running total.

The same `getElementsByTagName()` API combined with the utility function `doubleValue()` extracts the amounts of tax, the shipping and handling, and the invoice total cost.

Just as in the SAX example, the code has to use a difference check instead of a direct equality check to guard against inexact decimal-to-binary conversions.

The class also defines a convenient utility function that takes in a DOM node that should have only character content and returns the numeric representation of that content as a double. Any nontrivial DOM processing will typically require these types of utility functions. It goes to prove that the DOM is very syntax-oriented and not concerned about data.

That's all it takes to process the invoice using DOM. Of course, this is a simple example; just as in the SAX example, a real application would have done at least three things differently:

- It would have used namespace information and prefixed element names instead of using local names.
- It would have defined its own exception type to communicate invoice validation information. It would have implemented try-catch logic inside the `checkInvoice()` method in order to report more meaningful errors.
- It would have either explicitly turned on validation of the incoming XML document or traversed the DOM tree step by step from the document root to all the elements of interest. Using `getElementsByTagName()` presumes that the structure of the document (relative positions of elements) has already been validated. If this is the case, it's okay to ask for all item elements regardless of where they are in the document. The example implementation took this approach for code readability purposes.

These changes aren't complex, but they would have increased the size and complexity of the example beyond its goals as a basic introduction to DOM processing.

JAXB-Based `checkInvoice()`

The basic architecture of the JAXB implementation is shown in Figure 2.10. The various components are similar to those described in the general architecture of XML processing using schema compilers in Figure 2.7.

To use JAXB, you first have to invoke the schema compiler that comes with the distribution. The compiler that comes with the Sun distribution used in this example is called `xjc` (XML-to-Java Compiler). The compiler is easy to use: It can produce an initial mapping of a schema to Java classes by looking at an XML schema, without requiring any binding customization. To try this on the invoice schema, we have to execute the command `xjc invoice.xsd`. The output of this command is shown in Listing 2.33.

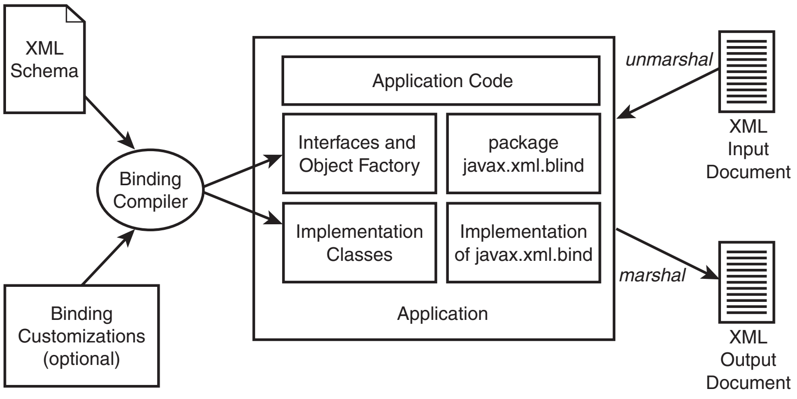


Figure 2.10 JAXB architecture

Listing 2.33 `xjc` **Output After Processing the Invoice Schema**

```
C:\dev\projects\jaxb>xjc invoice.xsd
parsing a schema...
compiling a schema...
com\skatestown\ns\invoice\impl\InvoiceImpl.java
com\skatestown\ns\invoice\impl\InvoiceTypeImpl.java
com\skatestown\ns\invoice\impl\ItemTypeImpl.java
com\skatestown\ns\invoice\impl\JAXBVersion.java
com\skatestown\ns\invoice\impl\runtime\XMLSerializer.java
com\skatestown\ns\invoice\impl\runtime\SAXUnmarshallerHandler.java
com\skatestown\ns\invoice\impl\runtime\MSVValidator.java
com\skatestown\ns\invoice\impl\runtime\PrefixCallback.java
com\skatestown\ns\invoice\impl\runtime\UnmarshallingEventHandlerAdaptor.java
com\skatestown\ns\invoice\impl\runtime\UnmarshallerImpl.java
com\skatestown\ns\invoice\impl\runtime\Discarder.java
com\skatestown\ns\invoice\impl\runtime\SAXUnmarshallerHandlerImpl.java
com\skatestown\ns\invoice\impl\runtime\ValidatorImpl.java
com\skatestown\ns\invoice\impl\runtime\GrammarInfo.java
com\skatestown\ns\invoice\impl\runtime\DefaultJAXBContextImpl.java
com\skatestown\ns\invoice\impl\runtime\ErrorHandlerAdaptor.java
com\skatestown\ns\invoice\impl\runtime\AbstractUnmarshallingEventHandlerImpl.java
com\skatestown\ns\invoice\impl\runtime\GrammarInfoFacade.java
com\skatestown\ns\invoice\impl\runtime\MarshallerImpl.java
com\skatestown\ns\invoice\impl\runtime\UnmarshallingContext.java
com\skatestown\ns\invoice\impl\runtime\UnmarshallableObject.java
com\skatestown\ns\invoice\impl\runtime\ContentHandlerAdaptor.java
com\skatestown\ns\invoice\impl\runtime\NamespaceContext2.java
com\skatestown\ns\invoice\impl\runtime\ValidatingUnmarshaller.java
com\skatestown\ns\invoice\impl\runtime\SAXMarshaller.java
```

Listing 2.33 **Continued**

```
com\skatestown\ns\invoice\impl\runtime\NamespaceContextImpl.java
com\skatestown\ns\invoice\impl\runtime\Util.java
com\skatestown\ns\invoice\impl\runtime\XMLSerializable.java
com\skatestown\ns\invoice\impl\runtime\ValidatableObject.java
com\skatestown\ns\invoice\impl\runtime\AbstractGrammarInfoImpl.java
com\skatestown\ns\invoice\impl\runtime\ValidationContext.java
com\skatestown\ns\invoice\impl\runtime\UnmarshallingEventHandler.java
com\skatestown\ns\po\impl\AddressTypeImpl.java
com\skatestown\ns\po\impl\ItemTypeImpl.java
com\skatestown\ns\po\impl\JAXBVersion.java
com\skatestown\ns\po\impl\PoImpl.java
com\skatestown\ns\po\impl\PoTypeImpl.java
com\skatestown\ns\invoice\Invoice.java
com\skatestown\ns\invoice\InvoiceType.java
com\skatestown\ns\invoice\ItemType.java
com\skatestown\ns\invoice\ObjectFactory.java
com\skatestown\ns\invoice\bgm.ser
com\skatestown\ns\invoice\jaxb.properties
com\skatestown\ns\po\AddressType.java
com\skatestown\ns\po\ItemType.java
com\skatestown\ns\po\ObjectFactory.java
com\skatestown\ns\po\Po.java
com\skatestown\ns\po\PoType.java
com\skatestown\ns\po\bgm.ser
com\skatestown\ns\po\jaxb.properties
```

After parsing, analyzing, and processing the invoice schema (and the PO schema on which it depends), the compiler outputs 50 (!) files that fall into three categories: interface files, implementation files, and supporting files.

Note that the namespace URIs for the invoice and PO schemas are mapped to the Java package names `com.skatestown.ns.po` and `com.skatestown.ns.invoice`. Inside these two packages are the interfaces generated for the schema types. A Java interface is generated for every type and element in the schema. For example, `invoiceType` in the schema is mapped to `InvoiceType.java`. The compiler also generates three supporting files:

- `ObjectFactory.java`—Contains factory methods for each generated Java interface. It allows you to programmatically construct new instances of Java objects representing XML content.
- `jaxb.properties`—Provides information about the specific JAXB implementation provider.
- `bgm.ser`—Contains a serialized representation of the schema information that can be used for efficient on-the-fly validation during XML-to-Java and Java-to-XML mapping.

All the files in the `...\impl` packages are specific to the Sun JAXB implementation and can be ignored.

Generated Interfaces

Let's look more closely at the interfaces generated by `xjc`, in particular the invoice and invoice item types we'll need to work with to check the invoice totals (Listing 2.34). For convenience purposes, we've reformatted the code and removed comments.

Listing 2.34 **Example Generated Interfaces**

```
package com.skatestown.ns.invoice;

public interface InvoiceType {
    java.math.BigDecimal getTotalCost();
    void setTotalCost(java.math.BigDecimal value);

    com.skatestown.ns.invoice.InvoiceType.OrderType getOrder();
    void setOrder(com.skatestown.ns.invoice.InvoiceType.OrderType value);

    java.math.BigInteger getCustomerId();
    void setCustomerId(java.math.BigInteger value);

    java.math.BigDecimal getShippingAndHandling();
    void setShippingAndHandling(java.math.BigDecimal value);

    com.skatestown.ns.po.AddressType getShipTo();
    void setShipTo(com.skatestown.ns.po.AddressType value);

    com.skatestown.ns.po.AddressType getBillTo();
    void setBillTo(com.skatestown.ns.po.AddressType value);

    java.util.Calendar getSubmitted();
    void setSubmitted(java.util.Calendar value);

    java.math.BigInteger getId();
    void setId(java.math.BigInteger value);

    java.math.BigDecimal getTax();
    void setTax(java.math.BigDecimal value);

    public interface OrderType {
        java.util.List getItem();
    }
}

public interface ItemType
    extends com.skatestown.ns.po.ItemType
```

Listing 2.34 **Continued**

```
{
    java.math.BigDecimal getUnitPrice();
    void setUnitPrice(java.math.BigDecimal value);
}


package com.skatestown.ns.po;

public interface ItemType {
    java.lang.String getSku();
    void setSku(java.lang.String value);

    java.lang.String getDescription();
    void setDescription(java.lang.String value);

    java.math.BigInteger getQuantity();
    void setQuantity(java.math.BigInteger value);
}
```

As you can see, the structure of the schema types is directly expressed in the Java classes with the appropriate type information. There is no sign of elements and attributes at the XML syntax level—they become properties of the Java classes. Working with repeated types maps well to Java programming patterns. For example, order items are accessed via `java.util.List`. We don't need to parse numbers; this is done by the JAXB implementation. It's easy to see why the JAXB implementation of `checkInvoice()` is likely to be the simplest and most resilient to potential future changes in the XML schema, compared to the SAX and DOM implementations.

JAXB Binding Customization

Only one thing about the default mapping generated by `xjc` doesn't seem quite right. All numeric values in the schema are mapped to `BigInteger` and `BigDecimal` types. The default rules of type mapping are meant to preserve as much information as possible. Therefore, schema types with unbounded precision such as `xsd:decimal` and `xsd:positiveInteger` are mapped to `BigDecimal` and `BigInteger`. The rules of JAXB name and type mapping are complex, and unfortunately we don't have space to discuss them here. However, we can address the number-mapping issue.

It would be nice to map to `int` and `double` in this example, because they're more convenient and efficient to use. To do so, we need to provide a binding customization to the schema compiler (Listing 2.35).

Listing 2.35 **Binding Customization for `xjc` (`binding.xjb`)**

```
<jxb:bindings version="1.0"
               xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Listing 2.35 **Continued**

```
    <jxb:bindings schemaLocation="po.xsd" node="/xsd:schema">
        <jxb:bindings node="//xsd:complexType[@name='poType']/
➥xsd:attribute[@name='id']">
            <jxb:javaType name="int"
                parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
                printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
        </jxb:bindings>
        <jxb:bindings node="//xsd:attribute[@name='customerId']">
            <jxb:javaType name="int"
                parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
                printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
        </jxb:bindings>
        <jxb:bindings node="//xsd:attribute[@name='quantity']">
            <jxb:javaType name="int"
                parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
                printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
        </jxb:bindings>
    </jxb:bindings> <!-- schemaLocation="po.xsd" node="/xsd:schema" -->

    <jxb:bindings schemaLocation="invoice.xsd" node="/xsd:schema">
        <jxb:bindings node="//xsd:complexType[@name='invoiceType']/
➥xsd:attribute[@name='id']">
            <jxb:javaType name="int"
                parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
                printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
        </jxb:bindings>
        <jxb:bindings node="//xsd:attribute[@name='customerId']">
            <jxb:javaType name="int"
                parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
                printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
        </jxb:bindings>
        <jxb:bindings node="//xsd:simpleType[@name='priceType']">
            <jxb:javaType name="double"
                parseMethod="javax.xml.bind.DatatypeConverter.parseDouble"
                printMethod="javax.xml.bind.DatatypeConverter.printDouble"/>
        </jxb:bindings>
    </jxb:bindings> <!-- schemaLocation="invoice.xsd" node="/xsd:schema" -->
</jxb:bindings>
```

In JAXB's case, you can insert binding customizations directly in the XML schema using schema extensibility mechanisms or provide them in a separate XML document. The latter is a better practice in that the XML schema is a programming-language-independent representation of data that shouldn't be encumbered by this information.

The basic mechanism of binding customizations involves two parts: identifying the part of the schema where the mapping should be modified and specifying the binding modification. All the customizations in this example are simple type mappings to `int` and `double` performed via the element `jxb:javaType`. The Java type to map to is specified via the `name` attribute. Two other attributes, `parseMethod` and `printMethod`, provide the unmarshalling and marshalling operations. JAXB provides convenience methods in the `javax.xml.bind.DatatypeConverter` class, which we use here.

Identifying the part of the schema to modify is more complicated. We need a mechanism to point to a part of the schema document. This mechanism is a language called *XPath*. XPath is one the XML standards developed by W3C. Think about the directory structure of your computer: The file path mechanism (for example, `C:\dev\projects\jaxb`) gives you a way to navigate that structure. Now think about the structure described by the DOM representation of XML documents: XPath lets you navigate that structure quickly and efficiently. We don't have the space to get into XPath, but here are a few examples taken from the binding customization file:

- `/xsd:schema`—The top-level element in the XML document called `xsd:schema`. The `/` syntax defines a level in the document element hierarchy, beginning from the current node. Initially, the current node is the root of the DOM.
- `//xsd:attribute[@name='quantity']`—An element called `xsd:attribute` (occurring anywhere in the document) whose `name` attribute value is `quantity`. The `//` syntax covers all descendants from the current node.
- `//xsd:complexType[@name='poType']/xsd:attribute[@name='id']`—An `xsd:complexType` element called `poType` (anywhere in the document) that has an `xsd:attribute` child with the name `id`. Note that we can't use the simpler XPath expression `//xsd:attribute[@name='id']` because `AddressType` in the schema also has an `id` attribute. The XPath expression would result in more than one DOM node, and the schema compiler would be unsure where to apply the binding customization.

In the binding customization, these XPath expressions are used within the context of nested `jxb:bindings` elements. The top-level element lets us change global binding rules. It has two children: one for the PO schema and one for the invoice schema. Within those, we modify each attribute with a numeric type. In the invoice schema, we modify the binding of `priceType`, and that modification automatically applies to all uses of that type in `unitPrice`, `tax`, and other attributes.

For `xjc` to take advantage of the binding customization, we need to modify the command line slightly to `xjc -b bindings.xjb invoice.xsd`. The same number of files are generated. This time, however, the `BigInteger` and `BigDecimal` types in Listing 2.34 are replaced with `int` and `double` types.

JAXB Processing Model

Now that the schema compiler is generating the correct Java classes, it's time to look at the JAXB processing model. Working with JAXB involves one main Java package: `javax.xml.bind`. This package provides abstract classes and interfaces for working with the JAXB framework's three operations: marshalling, unmarshalling, and validation. You access these operations via the `Marshaller`, `Unmarshaller`, and `Validator` classes in the package.

The `JAXBContext` class is the entry point into the JAXB framework. It provides support for multiple JAXB implementations, and it also manages the connection between XML elements and the Java classes that represent them. The package also has a rich set of exception classes for marshalling, unmarshalling, and validation events that make working with JAXB much easier and more natural than working with SAX or DOM.

With this information in mind, building a class to check invoice totals becomes relatively simple. The JAXB implementation of `InvoiceChecker` is shown in Listing 2.36.

Listing 2.36 **JAXB-Based Invoice Checker (`InvoiceCheckerJAXB.java`)**

```
package com.skatestown.invoice;

import com.skatestown.ns.invoice.InvoiceType;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import java.io.InputStream;
import java.util.Iterator;
import java.util.List;

/**
 * InvoiceChecker implementation using JAXB
 */
public class InvoiceCheckerJAXB implements InvoiceChecker
{
    /**
     * Check invoice totals.
     *
     * @param   invoiceXML  Invoice XML document
     * @exception   Exception  Any exception returned during checking
     */
    public void checkInvoice(InputStream invoiceXML)
    throws Exception
    {
        // Create JAXB context + point it to schema types
        JAXBContext jc = JAXBContext.newInstance(
                "com.skatestown.ns.po:com.skatestown.ns.invoice");

        // Create an unmarshaller
        Unmarshaller u = jc.createUnmarshaller();

        // Unmarshall the invoice document
        InvoiceType inv = (InvoiceType)u.unmarshal(invoiceXML);
```

Listing 2.36 **Continued**

```
        double runningTotal = 0.0;

        // Iterate over order items and update the running total
        List items = inv.getOrder().getItem();
        for( Iterator iter = items.iterator(); iter.hasNext(); ) {
            com.skatestown.ns.invoice.ItemType item =
                    (com.skatestown.ns.invoice.ItemType)iter.next();
            runningTotal += item.getQuantity() * item.getUnitPrice();
        }

        // Add tax and shipping and handling
        runningTotal += inv.getShippingAndHandling();
        runningTotal += inv.getTax();

        // Get invoice total
        double total = inv.getTotalCost();

        // Use delta equality check to prevent cumulative
        // binary arithmetic errors. In this case, the delta
        // is one half of one cent
        if (Math.abs(runningTotal - total) >= 0.005) {
            throw new Exception(
                "Invoice error: total is " + Double.toString(total) +
                " while our calculation shows a total of " +
                Double.toString(Math.round(runningTotal * 100) / 100.0));
        }
    }
}
```

`InvoiceCheckerJAXB` must implement the `InvoiceChecker` interface in order to provide the `checkInvoice()` functionality. Apart from this, it's a standalone class. As with DOM, note that the class has no member data, because there's no need to maintain parsing context. The context is implicit in the hierarchy of the Java object tree that will be generated during the unmarshalling process.

The JAXB context is initialized with the names of the Java packages we want to work with in the `JAXBContext.newInstance` factory call. This prepares the JAXB framework to deal with PO and invoice XML documents. The next factory pattern call creates an unmarshaller object. Parsing and unmarshalling the invoice document takes one call to `unmarshal()`. We have to cast to the top-level invoice type because the interface of the `Unmarshaller` class is generic to JAXB.

The code to recalculate the invoice total is simpler and more Java-friendly than in the SAX and DOM examples. There is no indication of any XML behind the scenes. This is what JAXB does best—it allows you to separate the Java data structures you want to work with from the XML representation of these structures. Just as in the SAX and

DOM examples, the code has to use a difference check instead of a direct equality check to guard against inexact decimal-to-binary conversions.

That completes the JAXB implementation of the invoice checker. Of course, JAXB is much more powerful (and complex), and we don't have space to dig into it here; but you've gotten an idea of why it's the preferred method of working with XML from Java.

Testing the Code

The code to test the three different invoice checker implementations is written using JavaServer Pages (JSP) (Listing 2.37). JSP allows Java code to be mixed with HTML for building Web applications. JSP builds on the Java Servlet standard for building Web components. Java application servers compile JSPs down to servlets.

Listing 2.37 **JSP Page for Checking Invoices**

```
<%@ page import="java.io.*,bws.BookUtil,com.skatestown.invoice.*" %>
<HTML>
<HEAD><TITLE>Invoice Checker</TITLE></HEAD>
<h1>Invoice Checker</h1>

<p>This example implements a web form driver for SkatesTown's invoice
checker. You can modify the invoice on the form if you wish (the
default one is from Chapter 2), select a DOM or SAX parser and perform
 a check on the invoice total.</p>

<FORM action="index.jsp" method="POST">
<%
    String xml = request.getParameter("xml");
    if (xml == null) {
        xml = BookUtil.readResource(application,
                         "/resources/sampleInvoice.xml");
    }
%>
    <TEXTAREA NAME="xml" ROWS="20" COLS="90"><%= xml%></TEXTAREA>
    <P></P>
    Select parser type:
    <INPUT type="RADIO" name="parserType" value="SAX" CHECKED> SAX
    <INPUT type="RADIO" name="parserType" value="DOM"> DOM
    <INPUT type="RADIO" name="parserType" value="JAXB"> JAXB
    <P></P>
    <INPUT type="SUBMIT" value=" Check Invoice ">
</FORM>

<%
    // Check for form submission
    if (request.getParameter("xml") != null) {
        out.println("<HR>");
```

Listing 2.37 **Continued**

```
        // Instantiate appropriate parser type
        InvoiceChecker ic;
        if (request.getParameter("parserType").equals("SAX")) {
            out.print("Using SAX parser...<br>");
            ic = new InvoiceCheckerSAX();
         } else if (request.getParameter("parserType").equals("DOM")) {
            out.print("Using DOM implementation...<br>");
            ic = new InvoiceCheckerDOM();
        } else {
            out.print("Using JAXB implementation...<br>");
            ic = new InvoiceCheckerJAXB();
        }

        // Check the invoice
        try {
            ic.checkInvoice(new StringBufferInputStream(xml));
            out.print("Invoice checks OK.");
        } catch(Exception e) {
            out.print(e.getMessage());
        }
    }
%>

</BODY>
</HTML>
```

JSP uses the `<%@ ... %>` syntax for compile-time directives. The `page import="..."` directive accomplishes the equivalent of a Java `import` statement.

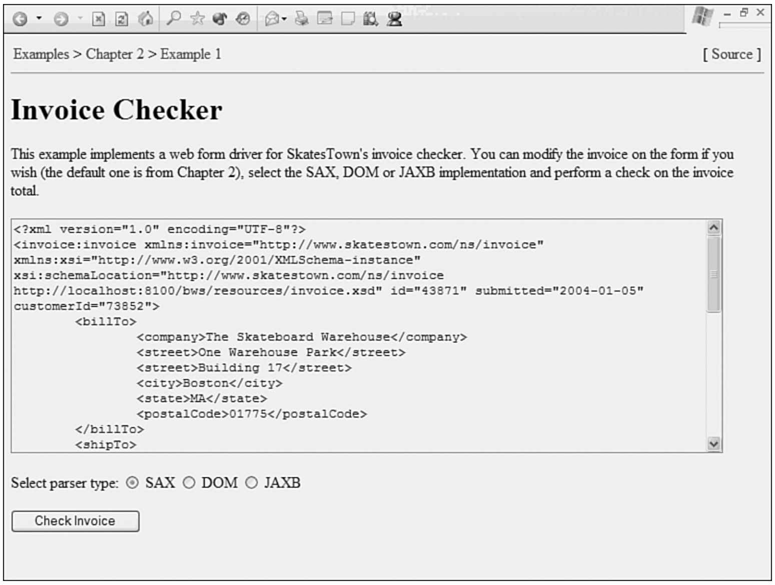
The HTML code sets up a Web form that posts back to the same page. The form contains a text area with the name `xml` that contains the XML of the invoice to be validated.

In JSP, you can use the construct `<% ... %>` to surround arbitrary Java code embedded in the JSP page. The request object is an implicit object on the page associated with the Web request. Implicit objects in JSP are set up by the JSP compiler. They can be used without requiring any type of declaration or setup. One of the most useful methods of the request object is `getParameter()`, which retrieves the value of a parameter passed from the Web such as a form field, or returns null if this parameter didn't come with the request. The code uses `getParameter("xml")` to check whether the form is being displayed (return is null) versus submitted (return is non-null). If the form is displayed for the first time, the page loads the invoice XML from a sample file in `/resources/sampleInvoice.xml`.

The rest of the Java code runs only if the form has been submitted. It uses the implicit `out` object to send output to the resulting Web page. It uses the value of the `parserType` field in the Web page to determine whether to instantiate a SAX, DOM, or

JAXB implementation. It then checks the invoice by passing the value of the `xml` text area on the page to the `checkInvoice()` method. If the call is successful, the invoice checks out okay, and an appropriate message is displayed. If `checkInvoice()` throws an exception, an invoice total discrepancy (or an XML processing error) has been detected, which is output to the browser.

Figure 2.11 shows the Web test client for the invoice checker, ready for submission.



Invoice Checker

This example implements a web form driver for SkatesTown's invoice checker. You can modify the invoice on the form if you wish (the default one is from Chapter 2), select the SAX, DOM or JAXB implementation and perform a check on the invoice total.

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice:invoice xmlns:invoice="http://www.skatestown.com/ns/invoice"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.skatestown.com/ns/invoice
http://localhost:8100/bws/resources/invoice.xsd" id="43871" submitted="2004-01-05"
customerId="73852">
        <billTo>
                <company>The Skateboard Warehouse</company>
                <street>One Warehouse Park</street>
                <street>Building 17</street>
                <city>Boston</city>
                <state>MA</state>
                <postalCode>01775</postalCode>
        </billTo>
        <shipTo>
```

Select parser type: [✓ SAX] DOM JAXB

Figure 2.11 Invoice checker Web page

Summary

This chapter has focused on core features of XML and related technologies. The goal was to prepare you for the Web service–related material in the rest of the book, which relies heavily on the concepts presented here. We covered the following topics:

- The origins of XML and the fundamental difference between document- and data-centric XML applications. Web services are an extreme example of data-centric XML use. The material in this chapter purposely ignored some aspects of XML that are more document-oriented.
- The syntax and rules governing the physical structure of XML documents: document prologs, elements, attributes, character content, CDATA sections, and so on. We omitted document-oriented features of XML such as entities and notations

due to their infrequent use in the context of Web services. The SkatesTown PO document format made its initial appearance.

- XML Namespaces, the key tool for resolving the problems of name recognition and name collision in XML applications. Namespaces are fundamental to mixing information from multiple schemas into a single document, and all core Web service technologies rely on them. SkatesTown's PO inside an XML message wrapper is an example of a common pattern for XML use; we'll explore it in depth in the next chapter. The namespace mechanism is simple; however, people often try to read more into it than is there, as demonstrated by the debate over whether namespace URIs should point to meaningful resources. One of the more complex aspects of the specification is that multiple namespace defaulting mechanisms simplify document markup while preserving namespace information.
- XML Schema, the de facto standard for describing document structure and XML datatypes for data-oriented applications. Although XML Schema is a recent standard, the XML community defined specifications based on draft versions for nearly two years. The flexible content models, the large number of predefined datatypes, and the powerful extensibility and reuse features make this one of the most important developments in the XML space since XML 1.0. All Web service specifications are described using schemas. Through the definition of SkatesTown's PO and invoice schemas, this chapter introduced enough of the key capabilities of the technology to prepare you for what is to come in the rest of the book.
- The key mechanisms for creating and processing XML with software. Starting with the basic syntax-oriented XML processing architecture, the chapter progressed to define a data-oriented XML processing architecture together with the key concepts of XML data mapping and XML parsing. In the context of SkatesTown's desire to independently validate invoice totals sent to its customers, we used the Java APIs for XML Processing (JAXP), the Simple APIs for XML (SAX), the XML Document Object Model (DOM), and the Java Architecture for XML Binding (JAXB) to build three separate implementations of an invoice checker. A Web-based front end served as the test bed for the code.

This chapter didn't focus on other less relevant XML technologies such as XPointer/XLink, Resource Definition Framework (RDF), XPath, Extensible Stylesheet Language Transformations (XSLT), or XQuery. They're important in their own domains but not commonly used in the context of Web services. Other more technical XML specifications, such as XML Digital Signatures, will be introduced later in the book as part of Web service usage scenarios.

You now know enough about XML to go deep into the world of Web services. Chapter 3 introduces the core Web service messaging technology: SOAP.

Resources

- *DOM Level 1*—"Document Object Model (DOM) Level 1 Specification" (W3C, October 1998), `http://www.w3.org/TR/REC-DOM-Level-1`
- *DOM Level 2 Core*—"Document Object Model (DOM) Level 2 Core Specification" (W3C, November 2000), `http://www.w3.org/TR/DOM-Level-2-Core/`
- *JAXB*—"Java Technology and XML Downloads—Java Architecture for XML Binding" (Sun Microsystems, Inc., January 2003), `http://java.sun.com/xml/downloads/jaxb.html`
- *JAXP 1.2*—"Java API for XML Processing (JAXP)" (Sun Microsystems, Inc., August 2003), `http://java.sun.com/xml/xml_jaxp.html`
- *JDOM*—`http://www.jdom.org/docs/apidocs`
- *JSP 1.2*—"JavaServer Pages Technology" (Sun Microsystems, Inc., April 2001), `http://java.sun.com/products/jsp`
- RFC 2396, "Uniform Resource Identifiers (URI): Generic Syntax" (IETF, August 1998), `http://www.ietf.org/rfc/rfc2396.txt`
- *SAX*—"Simple API for XML (SAX) 2.0.1" (January 2002), `http://www.saxproject.org/`
- *Unicode*—"Forms of Unicode" (Mark Davis, September 1999), `http://www-106.ibm.com/developerworks/library/utfencodingforms/`
- *XML*—"Extensible Markup Language (XML) 1.0 (Second Edition)" (W3C, October 2000), `http://www.w3.org/TR/REC-xml`
- *XML Namespaces*—"Namespaces in XML" (W3C, January 1999), `http://www.w3.org/TR/REC-xml-names/`
- *XML Schema*—"XML Schema Part 0: Primer," `http://www.w3.org/TR/xmlschema-0/`; "XML Schema Part 1: Structures," `http://www.w3.org/TR/xmlschema-1/`; "XML Schema Part 2: Datatypes," `http://www.w3.org/TR/xmlschema-2/` (all W3C, May 2001)