THIRD EDITION

Steven Holzner

SAMS Teach Yourself

XML

SAMS

in 21 Days

Steven Holzner

SAMS Teach Yourself

XML

in 21 Days

THIRD EDITION

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself XML in 21 Days, Third Edition

Copyright © 2004 by Sams Publishing

International Standard Book Number: 0-672-32576-4

Library of Congress Catalog Card Number: 2003110401

Printed in the United States of America

First Printing: October 2003

06 05 04 03 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

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
1-317-428-3341
international@pearsontechgroup.com

Associate Publisher
Michael Stephens

Acquisitions Editor
Todd Green

Development Editor
Songlin Qiu

Managing Editor
Charlotte Clapp

Project Editor
Matthew Purcell

Indexer
Mandie Frank

Proofreader
Paula Lowell

Technical Editor
Chris Kenyeres

Team Coordinator
Cindy Teeters

Interior Designer
Gary Adair

Cover Designer
Gary Adair

Page Layout
Michelle Mitchell

Contents at a Glance

Table of Contents

About the Author

Steven Holzner is an award-winning author who has written 80 computing books. He has been writing about XML since it first appeared and is one of the foremost XML experts in the United States, having written several XML bestsellers and being a much-requested speaker on the topic. He's also been a contributing editor at *PC Magazine*, has been on the faculty of Cornell University and MIT, and teaches corporate programming classes around the United States.

Dedication

To Nancy, as always and forever—for all the reasons she already knows!

Acknowledgments

A book like the one you're reading is the product of many people's hard work. I'd especially like to thank Todd Green, the acquisitions editor; Songlin Qiu, the development editor; Matt Purcell, the project editor; and Christian Kenyeres, the tech editor.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you would like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

For more information about this book or another Sams Publishing title, visit our Web site at `http://www.samspublishing.com`. Type the ISBN (0672325764) or the title of a book in the Search field to find the page you're looking for.

This page intentionally left blank

Introduction

Welcome to Extensible Markup Language (XML), the most influential innovation the Internet has seen in years. XML is a powerful, very dynamic topic, spanning dozens of fields, from the simple to the very complex. This book opens up that world, going after XML with dozens of topics—and hundreds of examples.

Unlike other XML books, this book makes it a point to show how XML actually works, making sure that you see everything demonstrated with examples. The biggest problem with most XML books is that they discuss XML and its allied specifications in the abstract, which makes it very hard to understand what's going on. This book, however, illustrates every XML discussion with examples. It shows all that's in the other books and more besides, emphasizing seeing things at work to make it all clear.

Instead of abstract discussions, this book provides concrete working examples because that's the only way to really learn XML. You're going to see where to get a lot of free software on the Internet to run the examples you create—everything from XML browsers to XPath visualizers to XQuery processors to XForms handlers, which you don't find in other books. You'll create XML-based documents that display multimedia shows you can play in RealPlayer, use browser plug-ins to handle XML-based graphics in the popular Hypertext Markup Language (HTML) browsers, enable Web pages to load and handle XML, and much more. XML can get complicated, and seeing it at work is the best way to understand it.

What This Book Covers

This book covers XML as thoroughly as any book you'll find: It goes from the most basic up through the advanced. XML ranges over many disciplines, and this book tracks it down where it lives. Part I, "Creating XML Documents," shows how to use XML in both current Web browsers as well as specialized XML-only browsers. Part I works through every part of an XML document to show how to construct such documents. You'll see how to use online XML validators to check XML and where to find software that lets you check an XML document's schema to make sure the document works as it should. You'll see how to format XML by using cascading style sheets (CSS), Extensible Stylesheet Language Transformations (XSLT), and XML-based formatting objects.

You don't need any programming skills to work with XML in Part I of this book. However, there's no way to ignore the terrific amount of XML support in programming languages such as JavaScript, Java, and the .NET programming languages. Later in the

book, you'll see how to use those languages with XML, navigating through XML documents, extracting data, formatting data, and even creating your own simple XML browsers.

Here's an overview of some of the topics covered in this book:

- The basics of XML
- Displaying XML in browsers
- Writing XML
- Creating well-formed and valid XML documents
- Working with XML validators
- Finding XML resources on the Internet
- Creating Document Type Definitions (DTDs)
- Creating XML schema
- Using XML schema-generating tools
- Using CSS with XML documents
- Displaying images
- Using XSLT to transform XML in the server, in the client, and with standalone programs
- Creating XSLT stylesheets
- Working with XPath
- Using the XSL formatting language
- Introducing Extensible HTML (XHTML)
- Validating XHTML
- Drawing basic shapes in Scalable Vector Graphics (SVG)
- Using SVG hyperlinks, animation, scripting, and gradients
- Creating SMIL documents
- Using Synchronized Multimedia Integration Language (SMIL)
- Creating XLinks, XPointers, and XForms
- Separating data and presentations in XForms
- Handling XML with JavaScript
- Using Java and the XML Document Object Model (DOM)
- Using XML data islands
- Parsing XML documents

- Navigating through an XML document by using Java
- Creating graphical XML browsers by using Java
- Using Java and the Simple API for XML (SAX)
- Using Simple Object Access Protocol (SOAP) to communicate between Web applications
- Binding XML data to HTML controls
- Navigating through XML data
- Displaying XML data in tables
- Managing XML databases
- Working with XML database storage in .NET
- Using XQuery to query an XML document
- Editing XML documents and XML schemas in .NET
- Writing and reading XML documents from code
- Creating XML Web services

As you can see, this book covers many facets of XML.

Who This Book Is For

This book is for anyone who wants to learn XML and how it is used today. This book assumes that you've had some experience with HTML, but that's about all it assumes. In Part IV, "Programming and XML," knowledge of JavaScript and Java helps, although the chapters in Part IV discuss where you can find free online tutorials on these subjects. The .NET programming discussed on Day 21, "Handling XML in .NET," may be a little hard to follow unless you've worked with Visual Basic .NET before.

Note that this book is as platform-independent as possible. XML is not the province of any one particular operating system, so this book does not lean one way or another on that issue. This book aims to show you as much of XML as it can, in the greatest depth possible. However, it's a fact of life that a great deal of XML software these days is targeted at Windows. And among the standard browsers, Internet Explorer has many times more XML support than any other browser does. This book doesn't have any special pro- or anti-Microsoft bias, but in order for this book to cover what's available for XML these days, you're going to find yourself in Microsoft territory fairly often; there's no getting around it.

Conventions Used in This Book

The following conventions are used in this book:

- Code lines, commands, statements, and any other code-related terms appear in a `monospace` typeface. Placeholders (which stand for what you should actually type) appear in *`italic monospace`*. Text that you should type appears in **`bold`**.
- When a line of code is too long to fit on one line of this book, it is broken at a convenient place and continued to the next line. The continuation is preceded by a special code continuation character (➥).
- New lines of XML or programming code that are added and are being discussed appear shaded, and when there's more code to come, you see three vertical dots. Here's how these features look:

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

- Throughout the book are notes that are meant to give you something more. This is what a note looks like:

NOTE

A note presents interesting information related to the discussion—a little more insight or a pointer to some new technique.

- This book also contains tips. This is what a tip looks like:

TIP

A tip offers advice or shows you an easier way of doing something.

- This book also contains cautions. This is what a caution looks like:

CAUTION

A caution alerts you to a possible problem and gives you advice on how to avoid it.

- Each day's lesson ends with questions pertaining to that day's subject matter, with answers from the book's author. Each day's discussion also includes a quiz that is designed to test your knowledge of the day's concepts. The answers to these quiz questions are provided in Appendix A, "Answers to Quiz Questions." Many lessons conclude with exercises that give you an opportunity to practice what you've learned in the lesson.

Where to Download the Book's Code

You can download all the code examples used throughout this book from `http://www.samspublishing.com`. Simply enter this book's ISBN without the hyphens (**`0672325764`**) in the Search box and click Search. When the book's title is displayed, click it to go to a page where you can download the code.

This page intentionally left blank

PART I

DAY 3

Creating Well-Formed XML Documents

Yesterday, you took a look at the various parts of XML documents—prologs, elements and attributes, processing instructions, and so forth. Today, you're going to start putting those items to work as you create well-formed documents.

Why is it so important to make an XML document well-formed? For one thing, W3C doesn't consider an XML document to be XML unless it's well-formed. For another, XML processors won't read XML documents unless those documents are well-formed. All of which is to say that making your XML well-formed is integral to creating XML documents—software isn't even going to be able to read your documents unless they are. Here's an overview of today's topics:

- Well-formed XML documents
- The W3C Well-formedness constraints
- Nesting constraints
- Element and attribute constraints

- Namespaces
- Local and default namespaces
- XML Infosets
- Canonical XML

To some extent, the current loose state of HTML documents is responsible for the great emphasis W3C puts on making sure XML documents are well-formed. HTML browsers have become more and more friendly to HTML pages as time has gone on, which means a Web page can have dozens of errors and still be displayed by a browser. That's not such a problem when it comes to simply displaying a Web page, but when it comes to handling what might be crucial data, it's a different story.

So W3C changed the rules from HTML to XML—unlike an HTML browser, an XML processor is *never* supposed to guess when it reads an XML document. If it finds an error (if the document is not well-formed, or if it uses a DTD or XML schema and it's not valid), the XML processor is supposed to inform you of the error, but then it can quit immediately. Ideally, according to W3C, a validating XML processor should list all the errors in an XML document and then quit; a non-validating one doesn't even have to do that—it can quit the first time it sees an error.

This enforced precision has two sides to it—there's no doubt that your data is transferred more faithfully using XML, but because XML processors make no guesses as to what you're trying to do, XML and XML processors can come across as non-user friendly, and not as generous or as easy to work with as HTML. On the other hand, you don't end up with the many possible errors that can creep into HTML, and that's important. XML authors have to be aware of the constraints on what they write, which is why we spend time in this book on document well-formedness and validity. In fact, in the XML 1.0 specification, W3C says that you can't even call a data object an XML document unless it's well-formed:

> A data object is an XML document if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.

What Makes an XML Document Well-Formed?

The W3C, which is responsible for the term *well-formedness*, defines it this way in the XML 1.0 recommendation:

A textual object is a well-formed XML document if:

- Taken as a whole, it matches the production labeled *document*.
- It meets all the well-formedness constraints given in this specification (that is, the XML 1.0 specification, `http://www.w3.org/TR/REC-xml`).
- Each of the parsed entities, which is referenced directly or indirectly within the document, is well-formed.

Because the major differences between XML 1.0 and XML 1.1 have to do with what characters are legal, you probably won't be surprised to learn that a well-formed XML 1.0 document is also a well-formed XML 1.1 document, as long as it avoids certain characters. From the XML 1.1 specification:

> If a document is well-formed or valid XML 1.0, and provided it does not contain any characters in the range [#x7F-#x9F] other than as character escapes, it may be made well-formed or valid XML 1.1 respectively simply by changing the version number.

Let's get into three conditions that make an XML document well-formed, starting with the requirement that the document must match the production named `document`.

Matching the Production Labeled `document`

W3C calls the individual specifications within a working draft or recommendation *productions*. In this case, to be well-formed, a document must follow the `document` production, which means that the document itself must have three parts:

- a prolog (which can be empty)
- a root element (which can contain other elements)
- a miscellaneous part (unlike the preceding two parts, this part is optional)

You've seen XML prologs yesterday; they can contain an XML declaration (such as `<?xml version = "1.0"?>`), as well as comments, processing instructions, and doctype declarations (that is, DTDs).

You've also seen root elements; the root element is the XML element that contains all the other elements in your document. Each well-formed XML document must have one, and only one, root element.

The optional miscellaneous part can be made up of XML comments, processing instructions, and whitespace, all items you saw yesterday.

In other words, this first requirement says that an XML document must be made up of the parts you saw yesterday. So far, so good.

Meeting the Well-Formedness Constraints

The next requirement is a little more difficult to track down, because it says that to be well-formed, XML documents must also satisfy the well-formedness constraints in the XML 1.0 specification. This means that your XML documents should adhere to the syntax rules specified in the XML 1.0 recommendation. You'll discuss those rules, which are sprinkled throughout the XML 1.0 specification, in a few pages.

Making Parsed Entity Must Be Well-Formed

The final requirement is that each parsed entity in a well-formed document must itself be well-formed. When an XML document is parsed by an XML processor, entity references (such as `&#x03C0;`) are replaced by the entities they stand for (such as Π in this case). The requirement that all parsed entities must be well-formed simply means that when you replace entity references with the entities they stand for, the result must be well-formed.

That's the W3C's definition of a well-formed document, but you still need more information. What are the well-formedness constraints given throughout the XML specification? You're going to go over these constraints today; to start, you'll create an XML document that you'll use as we discuss what it means for a document to be well-formed.

Creating an Example XML Document

The sample document you'll use today, and which you'll also see tomorrow when working with DTDs, will store data about a set of employees, such as their names, projects they're working on, and so on. This document will start, as all XML documents should, with an XML declaration:

```
<?xml version = "1.0"?>
```

Because all the documents you'll see today are self-contained (they don't refer to or include any external entities), you'll also add the `standalone` attribute, setting it to `"yes"`, and specify that we're using UTF-8 encoding:

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

And you'll also add a root element, called `<document>` in this case, although you can use any legal name:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    .
    .
    .
</document>
```

The root element will contain all the other elements in the document. In this case, that will be three <employee> elements:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
    .
    .
    .
    </employee>
    <employee>
    .
    .
    .
    </employee>
    <employee>
    .
    .
    .
    </employee>
</document>
```

For each employee, we can store a name in a <name> element, which itself encloses a <lastname> and <firstname> element:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
        <name>
            <lastname>Kelly</lastname>
            <firstname>Grace</firstname>
        </name>
    .
    .
    .
    </employee>
    .
    .
    .
</document>
```

We'll also store each employee's hire date, as well as the projects they're working on. For each project, we can store the product name, ID, and price:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
        <name>
            <lastname>Kelly</lastname>
            <firstname>Grace</firstname>
        </name>
```

```
        <hiredate>October 15, 2005</hiredate>
        <projects>
            <project>
                <product>Printer</product>
                <id>111</id>
                <price>$111.00</price>
            </project>
            <project>
                <product>Laptop</product>
                <id>222</id>
                <price>$989.00</price>
            </project>
        </projects>
    </employee>
    .
    .
    .
</document>
```

That's what the data looks like for one employee; you can see the full document, `ch03_01.xml`, in Listing 3.1. Documents like this one can grow very long, but that presents no problem to XML processors—as long as the document is well-formed.

LISTING 3.1 Sample Well-Formed XML Document (`ch03_01.xml`)

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
        <name>
            <lastname>Kelly</lastname>
            <firstname>Grace</firstname>
        </name>
        <hiredate>October 15, 2005</hiredate>
        <projects>
            <project>
                <product>Printer</product>
                <id>111</id>
                <price>$111.00</price>
            </project>
            <project>
                <product>Laptop</product>
                <id>222</id>
                <price>$989.00</price>
            </project>
        </projects>
    </employee>
    <employee>
        <name>
            <lastname>Grant</lastname>
```

LISTING 3.1 continued

```
            <firstname>Cary</firstname>
        </name>
        <hiredate>October 20, 2005</hiredate>
        <projects>
            <project>
                <product>Desktop</product>
                <id>333</id>
                <price>$2995.00</price>
            </project>
            <project>
                <product>Scanner</product>
                <id>444</id>
                <price>$200.00</price>
            </project>
        </projects>
    </employee>
    <employee>
        <name>
            <lastname>Gable</lastname>
            <firstname>Clark</firstname>
        </name>
        <hiredate>October 25, 2005</hiredate>
        <projects>
            <project>
                <product>Keyboard</product>
                <id>555</id>
                <price>$129.00</price>
            </project>
            <project>
                <product>Mouse</product>
                <id>666</id>
                <price>$25.00</price>
            </project>
        </projects>
    </employee>
</document>
```


Today's work gets us into the structure of XML documents, and there's some terminology we should get to know at this point having to do with the relative position of elements in an XML document. As an example, take a look at an employee element in `ch03_01.xml`.

Elements on the same level, such as `<name>`, `<hiredate>`, and `<projects>` in an `<employee>` element, are all called *siblings*. Similarly, the two `<project>` elements in each `<projects>` element are siblings.

This family-type relationship is also continued with *child* and *parent* relationships. For example, the parent of the two `<project>` elements is the `<projects>` element. And the two `<project>` elements are children of the `<projects>` element.

You can always count on every non-root element to have exactly one, and only one, parent element. And a parent element can enclose an indefinite number of child elements (which can also mean zero child elements). You can continue the analogy to multiple generations as well; for example, the two `<project>` elements in this case are also *grandchildren* of the `<employee>` element.

That gives us the example document and terminology we'll need; now let's take a look at the well-formedness constraints you'll find in XML.

Understanding the Well-Formedness Constraints

The well-formedness constraints in the XML 1.0 specification are sprinkled throughout the document, and some of them are hard to dig out because they're not clearly marked. You'll get a look at the well-formedness constraints here, although note that some of them have to do with DTDs and entity references, and those will appear in Day 4, "Creating Valid XML Documents: Document Type Definitions," and Day 5, "Handling Attributes and Entities in DTDs."

Beginning the Document with an XML Declaration

The first well-formedness structure constraint is to start the document with an XML declaration. Even though some XML processors won't insist on it, W3C says you should always include this declaration first thing:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
    .
    .
    .
```

TIP

Although the XML 1.0 specification says that only the `version` attribute is required here, some software—notably including W3C's own Amaya testbed browser—will consider XML documents as not well-formed if you don't also include the `encoding` attribute.

Using Only Legal Character References

Another well-formedness constraint is that character references, which are character codes enclosed in `&` and `;`, and which are replaced by the characters that code stands for, must only refer to characters supported by the XML specification.

This constraint is more or less obvious—it simply means that you have to stick to the established character set for the version of XML you're using. Note that, as you saw yesterday, the characters that are legal in XML 1.0 differ somewhat from what's legal in XML 1.1.

Including at Least One Element

To be a well-formed document, a document must include *one or more* elements. The first element, of course, is the root element, so to be well-formed, a document must contain at least a root element. In other words, an XML document must contain more than just a prolog. Of course, your documents will usually contain many elements, as in our example document:


```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
        <name>
            <lastname>Kelly</lastname>
            <firstname>Grace</firstname>
        </name>
        <hiredate>October 15, 2005</hiredate>
        <projects>
            <project>
            .
            .
            .
            </project>
        </projects>
    </employee>
        .
        .
        .
</document>
```

Structuring Elements Correctly

HTML browsers are pretty easygoing about how you structure HTML elements in a Web page as long as they can understand what you're doing. For example, you can often omit closing tags in elements—you might use a `<p>` tag and then follow it with another `<p>` tag—without using a `</p>` tag—and the browser will have no problem.

That's not the way things work in XML. In XML, every non-empty element must have both a start tag and an end tag, as in our example document:

```
<employee>
    <name>
        <lastname>Gable</lastname>
        <firstname>Clark</firstname>
    </name>
    <hiredate>October 25, 2005</hiredate>
    <projects>
        <project>
            <product>Keyboard</product>
            <id>555</id>
            <price>$129.00</price>
        </project>
        <project>
            <product>Mouse</product>
            <id>666</id>
            <price>$25.00</price>
        </project>
    </projects>
</employee>
```

Besides making sure that every non-empty element has an opening tag and a closing tag, another well-formedness constraint says that end tags must match start tags, and both must use the same name.

Some elements—empty elements—don't have closing tags. These tags have no content of any kind (although they can have attributes), which means that they do not enclose any character data or markup. Instead, these elements are made up entirely of one tag like this:

```
<?xml version = "1.0" standalone="yes"?>
<document>
    <heading text = "Hello From XML"/>
</document>
```

In XML, empty elements must always end with `/>`.

TIP

HTML elements can also be ended with `/>`, such as `<BR/>`, and HTML browsers will not have a problem with them. That's good, because the alternative is to write `<BR></BR>`, which some browsers, such as Netscape Navigator, interpret as two `<BR>` elements.

Using the Root Element to Contain All Other Elements

Another well-formedness constraint is that the root element must contain all the other elements in the document, as in our sample XML document, where we have three `<employee>` elements, which themselves contain other elements, in the document element:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <employee>
        .
        .
        .
    </employee>
    <employee>
        .
        .
        .
    </employee>
    <employee>
        .
        .
        .
    </employee>
</document>
```

That's how a well-formed XML document works—you start with a prolog, followed by the root element, which contains all the other the elements, if there are any. Among other things, containing all elements in a root element makes it easier for an XML processor to understand the structure of an XML document—starting at the single root element, it can navigate the entire document.

Nesting Elements Properly

Nesting elements correctly is a big part of well-formedness; the requirement here is that if an element contains a start tag for a non-empty tag, it must also contain that element's end tag. In other words, you cannot spread an element over other elements at the same level. For example, this XML is nested properly:

```
<employee>
    <name>
        <lastname>Kelly</lastname>
        <firstname>Grace</firstname>
    </name>
    <hiredate>October 15, 2005</hiredate>
    <projects>
        <project>
            <product>Printer</product>
            <id>111</id>
```

```
            <price>$111.00</price>
        </project>
        <project>
            <product>Laptop</product>
            <id>222</id>
            <price>$989.00</price>
        </project>
    </projects>
</employee>
```

But as you can see, there's a nesting problem in this next element, because an XML processor will encounter a new `<project>` tag before finding the closing `</project>` tag it's looking for at the end of the current `<project>` element:

```
<employee>
    <name>
        <lastname>Kelly</lastname>
        <firstname>Grace</firstname>
    </name>
    <hiredate>October 15, 2005</hiredate>
    <projects>
        <project>
            <product>Printer</product>
            <id>111</id>
            <price>$111.00</price>
        <project>
        </project>
            <product>Laptop</product>
            <id>222</id>
            <price>$989.00</price>
        </project>
    </projects>
</employee>
```

In fact, this nesting requirement is where the whole term *well-formed* comes from—the original idea was that a document where the elements were not garbled and mixed up with each other was well-formed.

There are other well-formedness constraints that have nothing to do with elements, however—for example, the next two concern attributes.

Making Attribute Names Unique

Another well-formedness constraint is that you can't use the same attribute more than once in one start-tag or empty-element tag. This is another well-formedness constraint that seems more or less obvious, and it's hard to see how you might violate this one except by mistake, as in this case:

```
<message text="Hi there!" text="Hello!">
```

XML is case sensitive, so you could theoretically do something like this:

```
<message Text="Hi there!" text="Hello!">
```

Obviously, that's not a very good idea, however; attribute names that differ only in capitalization are bound to be confusing.

Enclose Attribute Values in Quotation Marks

One well-formedness constraint that trips up most XML novices sooner or later is that you must quote every value you assign to an attribute, using either single quotation marks or double quotation marks. This trips many people up because you don't have to quote attribute values in HTML, as in this HTML example (which also doesn't have a closing tag):

```
<img src=mountains.jpg>
```


An XML processor would have problems with this element, however. Here's what it would look like properly constructed:

```
<img src="mountains.jpg" />
```

If you prefer, you could use single quotation marks:

```
<img src=mountains.jpg' />
```

As you've seen, using single quotation marks helps when an attribute's value contains quoted text:

```
<message text='I said, "No, no, no!"' />
```

And as you've also seen, in worst-case scenarios, where an attribute value contains both single and double quotation marks, you can escape `"` as `&quot;` and `'` as `&apos;`—as here, where you're reporting the height of a tree as 50' 6" :

```
<tree type="Maple" height="50&apos;6&quot;" />
```

Avoiding Entity References and < in Attribute Values

Also, W3C makes it an explicit well-formedness constraint that you should avoid references to external entities (this means XML-style references—general entity references or parameter entity references, not just, for example, using an image file's name) in attribute values. This means that an XML processor doesn't have to replace an attribute value with the contents of an external entity.

In addition, another constraint says that you are not supposed to use < in attribute values, because an XML processor might mistake it for markup. If you really have to use the text <, use `<` instead, which will be turned into < when parsed. For example, this XML:

```
<project note="This is a <project> element.">
```

should be written as this, where you're escaping both < and >:

```
<project note="This is a &lt;project&gt; element.">
```

In fact, < is a particularly sensitive character to use anywhere in an XML document, except as markup, and that's another well-formedness constraint concerning <, coming up next.

Avoiding Overuse of < and &

XML processors assume that < starts a tag and & starts an entity reference, so you should avoid using those characters for anything else. Sometimes, this is a problem, as in the JavaScript example you saw yesterday, which uses the JavaScript < operator that enclosed in a CDATA section:

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/tr/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>
            Checking the temperature
        </title>
    </head>

    <body>
        <script language="javascript">
            <![CDATA[
                var temperature
                temperature = 234.77
                if (temperature < 32) {
                    document.writeln("Below freezing!")
                }
            ]]>
        </script>

        <center>
            <h1>
                Checking the temperature
            </h1>
        </center>
    </body>
</html>
```

However, because modern Web browsers don't understand CDATA sections, this solution (which was suggested by W3C) doesn't really work. And if you escape the > operator as `<`, very few browsers will understand what you're doing.

There are two main ways of handling the < JavaScript operator in XML with today's browsers. You can reverse the logical sense of the test—for example, in this case, instead of checking whether the temperature is below 32, you would check to make sure it isn't above or equal to 32, which lets you use > instead of < (note that the JavaScript `!` operator, the Not operator, reverses the logical sense of an expression) :

```
<script language="javascript">
    var temperature
    temperature = 234.77
    if (!(temperature >= 32)) {
        document.writeln("Below freezing!")
    }
</script>
```


Practically speaking, the best way is usually to remove the whole problem by placing the script code in an external file, which you'll name `script.js` here, so the browser won't parse it as XML in the first place. You can do that like this in JavaScript (more on JavaScript and how to use it in XML is coming up in Day 15, "Using JavaScript and XML"):

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/tr/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>
            Checking the temperature
        </title>
    </head>

    <body>
        <script language="javascript" src="script.js">
        </script>
        <center>
            <h1>
                Checking the temperature
            </h1>
        </center>
    </body>
</html>
```

That completes today's discussion of well-formedness, although you'll see more in the next two days as we discuss the well-formedness constraints that have to do with DTDs.

As your XML documents evolve and become more complex, it's also going to be increasingly important to understand namespaces, which are the second major topic for today.

Using XML Namespaces

There's a lot of freedom in XML, because you get to create your own markup. As time went on, however, XML authors started noticing a problem that the original creators of XML hadn't really anticipated—conflicting tag names.

For example, you've already seen that two popular XML applications are XHTML, which is the derivation of HTML in XML, and MathML, which lets you format and display math equations. Suppose that you want to display an equation in an XHTML Web page. That could be a problem, because because the tag set in XHTML and MathML overlap—in particular, each XML application defines a `<var>` and `<select>` element.

The way to solve this problem is to use *namespaces*. Namespaces give you a way to make sure that one set of tags will not conflict with another. You prefix a name to tag and attribute names. Changing the resulting names won't conflict with others that have a different prefix.

XML namespaces are one of those XML companion recommendations that keep being added to the XML specification. You can find the specification for namespaces at `http://www.w3.org/TR/REC-xml-names/`. There's still a lot of debate about this one (mostly because namespaces can make writing DTDs difficult), but it's an official W3C recommendation now.

Creating Namespaces

An example will make namespaces and why they're important clearer. For example, suppose you're the boss of one of the employees in our sample document, `ch03_01.xml`:

```
<employee>
    <name>
        <lastname>Kelly</lastname>
        <firstname>Grace</firstname>
    </name>
    <hiredate>October 15, 2005</hiredate>
    <projects>
        <project>
            <product>Printer</product>
            <id>111</id>
            <price>$111.00</price>
        </project>
        <project>
```

```
            <product>Laptop</product>
            <id>222</id>
            <price>$989.00</price>
        </project>
    </projects>
</employee>
```

Now suppose that you want to add your own comments to this employee's data in a `<comment>` element. The problem with that is that the XML data on this employee comes from the Human Resources department, and they haven't created an element named `<comment>`. You can indeed create your own `<comment>` element, but first you should confine the human resource's department's XML data to its own namespace to indicate that your comments are not part of the Human Resource Department's set of XML tags.

To define a new namespace, use the `xmlns:`*`prefix`* attribute, where *`prefix`* is the prefix you want to use for the namespace. In this case, you'll define a new namespace called `hr` for the Human Resources department:

```
<employee>
   xmlns:hr="http://www.superduperbigco.com/human_resources">
    <name>
        <lastname>Kelly</lastname>
        <firstname>Grace</firstname>
    </name>
    <hiredate>October 15, 2005</hiredate>
    <projects>
        <project>
            <product>Printer</product>
            <id>111</id>
            <price>$111.00</price>
        </project>
        <project>
            <product>Laptop</product>
            <id>222</id>
            <price>$989.00</price>
        </project>
    </projects>
</employee>
```

To define a namespace, you assign the `xmlns:`*`prefix`* attribute to a unique identifier, which in XML is usually a URI that might direct the XML processor to a DTD for the namespace (but doesn't have to). So what's a URI?

Defining Namespaces with URIs

The XML specification expands the idea of standard URLs (Uniform Resource Locators) into *URIs (Uniform Resource Identifiers)*. In HTML and on the Web, you use URLs; in

XML, you use URIs. URIs are supposed to be more general than URLs, as we'll see when we discuss XLinks and XPointers in Day 14, "Handling XLinks, XPointers, and XForms."

For example, in theory, a URI can point not just to a single resource, but to a cluster of resources, or to *arcs* of resources along a path. The truth is that the whole idea of URIs as the next step after URLs is still being developed, and in practice, URLs are almost invariably used in XML—but you still call them URIs. Some software accepts more general forms of URIs, letting you, for example, access only a specific section of an XML document, but such usage and the associated syntax is far from standardized yet.

TIP

You might want to look up the current formal definition of URIs, which you can find in its entirety at `http://www.ics.uci.edu/pub/ietf/uri/rfc2396.txt`.

When you define a namespace with the `xmlns:`*`prefix`* attribute, you usually assign a URI to that attribute (in practice, this URI is always a URL today). The document that URI points to can describe more about the namespace you're creating; an example of this is the XHTML namespace, which uses the namespace `http://www.w3.org/1999/xhtml`:

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/tr/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:xhtml="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
        .
        .
        .
```

A namespace's URI can also hold a DTD or XML schema that defines the syntax for the XML elements you can use in that namespace (then it's up to the XML processor to use that DTD or XML schema, if it's been written to be smart enough to interpret namespaces in this way—most aren't). All that's really necessary, however, is that you assign a unique identifier, which can be any text, to the `xmlns:`*`prefix`* attribute.

After defining the `hr` namespace in our example, you can preface every tag and attribute name in this namespace with `hr:` like this:

```
<hr:employee
   xmlns:hr="http://www.superduperbigco.com/human_resources">
    <hr:name>
        <hr:lastname>Kelly</hr:lastname>
        <hr:firstname>Grace</hr:firstname>
    </hr:name>
```

```
    <hr:hiredate>October 15, 2005</hr:hiredate>
    <hr:projects>
        <hr:project>
            <hr:product>Printer</hr:product>
            <hr:id>111</hr:id>
            <hr:price>$111.00</hr:price>
        </hr:project>
        <hr:project>
            <hr:product>Laptop</hr:product>
            <hr:id>222</hr:id>
            <hr:price>$989.00</hr:price>
        </hr:project>
    </hr:projects>
</hr:employee>
```


Now you've made it clear that all these tags come from the Human Resources department. Note how this works—the actual tag names themselves have been changed, because a colon is a legal character to use in tag names. (Now you know why you shouldn't use colons in tag names, although they're legal—they can make it look like you're using namespaces when you're not.) For example, the `<product>` tag has now become the `<hr:product>` tag. In other words, using namespaces keeps elements separate by actually changing tag and attribute names. This was a clever solution to the problem of tag and attribute name conflicts, because this way, even XML processors that have never heard of namespaces can still "support" them.

At this point, all tag and attribute names from the `hr` namespace are in their own namespace, so you can add your own namespace to the document, allowing you to use your own elements without fear of conflict. Since you're the boss, you might start by defining a new namespace named `boss`:

```
<hr:employee
   xmlns:hr="http://www.superduperbigco.com/human_resources"
   xmlns:boss="http://www.superduperbigco.com/big_boss">
    <hr:name>
        <hr:lastname>Kelly</hr:lastname>
        <hr:firstname>Grace</hr:firstname>
    </hr:name>
    <hr:hiredate>October 15, 2005</hr:hiredate>
    <hr:projects>
        <hr:project>
            <hr:product>Printer</hr:product>
            <hr:id>111</hr:id>
            <hr:price>$111.00</hr:price>
        </hr:project>
        <hr:project>
            <hr:product>Laptop</hr:product>
            <hr:id>222</hr:id>
            <hr:price>$989.00</hr:price>
```

```
        </hr:project>
    </hr:projects>
</hr:employee>
```

Now you can use the new `boss` namespace to add your own markup to the document, as you see in Listing 3.2.

LISTING 3.2 XML Document with Namespaces (`ch03_02.xml`)

```
<hr:employee
   xmlns:hr="http://www.superduperbigco.com/human_resources"
   xmlns:boss="http://www.superduperbigco.com/big_boss">
    <hr:name>
        <hr:lastname>Kelly</hr:lastname>
        <hr:firstname>Grace</hr:firstname>
    </hr:name>
    <hr:hiredate>October 15, 2005</hr:hiredate>
    <boss:comment>Needs much supervision.</boss:comment>
    <hr:projects>
        <hr:project>
            <hr:product>Printer</hr:product>
            <hr:id>111</hr:id>
            <hr:price>$111.00</hr:price>
        </hr:project>
        <hr:project>
            <hr:product>Laptop</hr:product>
            <hr:id>222</hr:id>
            <hr:price>$989.00</hr:price>
        </hr:project>
    </hr:projects>
</hr:employee>
```

You can also add your own attributes in the `boss` namespace as long as you prefix them with `boss:` this way:

```
<hr:employee>
   xmlns:hr="http://www.superduperbigco.com/human_resources"
   xmlns:boss="http://www.superduperbigco.com/big_boss">
    <hr:name>
        <hr:lastname>Kelly</hr:lastname>
        <hr:firstname>Grace</hr:firstname>
    </hr:name>
    <hr:hiredate>October 15, 2005</hr:hiredate>
    <boss:comment boss:date="10/15/2006">
        Needs much supervision.
    </boss:comment>
    <hr:projects>
        <hr:project>
            <hr:product>Printer</hr:product>
            <hr:id>111</hr:id>
```

LISTING 3.2 continued

```
            <hr:price>$111.00</hr:price>
        </hr:project>
        <hr:project>
            <hr:product>Laptop</hr:product>
            <hr:id>222</hr:id>
            <hr:price>$989.00</hr:price>
        </hr:project>
    </hr:projects>
</hr:employee>
```

And that's how namespaces work—you can use them to separate tags, even tags with the same name, so there's no conflict. As you can see, using multiple namespaces in the same document is no problem at all—just use the `xmlns:`*`prefix`* attribute in the enclosing element to define the appropriate namespace. In fact, you can use this attribute attribute in child elements to redefine an enclosing namespace, if you want to.

Namespace prefixes are really just text prefixed to (*prepended* is the offical term) tag and attribute names. They follow the same rules for naming tags and attributes. For example, in XML 1.0, a namespace name can start with a letter or an underscore. The following characters can include underscores, letters, digits, hyphens, and periods. Note also that although colons are legal in tag names, you can't use a colon in a namespace name, for obvious reasons. Also, there are two namespace names that are reserved: `xml` and `xmlns`.

Creating Local Namespaces

The `xmlns:`*`prefix`* attribute can be used in any element, not just the document element. Just bear in mind that this attribute defines a namespace for the current element and any enclosed element, which means you shouldn't use the namespace prefix until you've defined the namespace with an attribute like `xmlns:`*`prefix`*.

For example, you can create the `boss:` namespace prefix and use it in the same element, as you see in Listing 3.3.

LISTING 3.3 XML Document with a Local Namespaces (`ch03_03.xml`)

```
<hr:employee
   xmlns:hr="http://www.superduperbigco.com/human_resources">
    <hr:name>
        <hr:lastname>Kelly</hr:lastname>
        <hr:firstname>Grace</hr:firstname>
    </hr:name>
    <hr:hiredate>October 15, 2005</hr:hiredate>
    <boss:comment
```

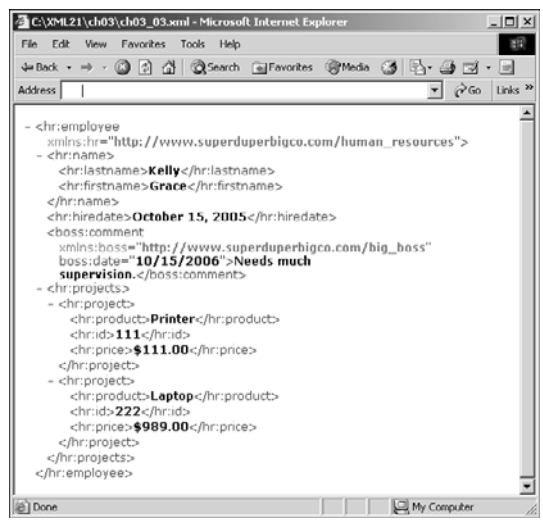
LISTING 3.3 continued

```
        xmlns:boss="http://www.superduperbigco.com/big_boss"
        boss:date="10/15/2006">
        Needs much supervision.
    </boss:comment>
    <hr:projects>
        <hr:project>
            <hr:product>Printer</hr:product>
            <hr:id>111</hr:id>
            <hr:price>$111.00</hr:price>
        </hr:project>
        <hr:project>
            <hr:product>Laptop</hr:product>
            <hr:id>222</hr:id>
            <hr:price>$989.00</hr:price>
        </hr:project>
    </hr:projects>
</hr:employee>
```

You can see `ch03_03.xml` in the Internet Explorer, complete with namespaces, in Figure 3.1.

FIGURE 3.1

Viewing an XML document with local namespaces.



Creating Default Namespaces

You can use the `xmlns:`*`prefix`* attribute to define a namespace, or you can use the `xmlns` attribute by itself to define a *default* namespace. When you define a default namespace, elements and attributes without a namespace prefix are in that default namespace.

To see how this works, we'll come full circle and put to work the example that introduced our discussion of namespaces in the first place—mixing XHTML with MathML. We'll start with some XHTML (all the details on XHTML are coming up in Day 11, "Extending HTML with XHTML," and Day 12, "Putting XHTML to Work"), like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/tr/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>
            Using XHTML and MathML Together
        </title>
    </head>

    <body>
        <center>
            <h1>
                Using XHTML and MathML Together
            </h1>
        </center>
        <br/>
        Consider the equation
        .
        .
        .
    </body>
</html>
```

You'll see what you need to create XHTML documents like this, such as the `<!DOCTYPE>` element, in Day 11. Note in particular here that in the `<html>` element, the `xmlns` attribute defines a default namespace for the `<html>` and all enclosed elements. (This namespace is the XHTML namespace, which W3C defines as `"http://www.w3.org/1999/xhtml"`.) When you use the `xmlns` attribute alone this way, without specifying any prefix, you are defining a default namespace. The current element and all child elements are assumed to belong to that namespace. Making use of a default namespace in this way, you can use the standard XHTML tag names without any prefix, as you see here.

However, we also want to use MathML markup in this document, and to do that, we add a new namespace, named `m` to this document, using the namespace W3C has specified for MathML, `"http://www.w3.org/1998/Math/MathML"`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/tr/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
    xmlns:m="http://www.w3.org/1998/Math/MathML">
    <head>
```

```
        <title>
            Using XHTML and MathML Together
        </title>
    </head>

    <body>
        <center>
            <h1>
                Using XHTML and MathML Together
            </h1>
        </center>
        <br/>
        Consider the equation
        .
        .
        .
    </body>
</html>
```

Now you can use MathML as you like, as long as you prefix it with the `m` namespace. You can see this at work in `ch03_04.html` (XHTML documents use the extension `.html`), shown in Listing 3.4, where we're using the MathML we developed in Day 1 to display an equation.

LISTING 3.4 An XML Document Combining XHTML and MathML (`ch03_04.html`)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/tr/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
    xmlns:m="http://www.w3.org/1998/Math/MathML">
    <head>
        <title>
            Using XHTML and MathML Together
        </title>
    </head>

    <body>
        <center>
            <h1>
                Using XHTML and MathML Together
            </h1>
        </center>
        <br/>
        Consider the equation
        <m:math>
            <m:mrow>
                <m:mrow>
                <m:mn>4</m:mn>
```

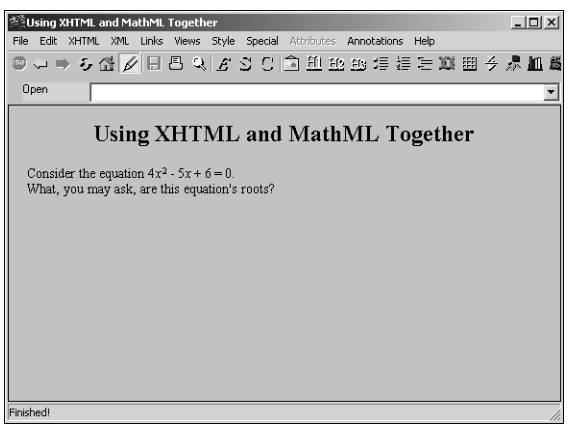
LISTING 3.4 continued

```
                    <m:mo>&InvisibleTimes;</m:mo>
                    <m:msup>
                        <m:mi>x</m:mi>
                        <m:mn>2</m:mn>
                    </m:msup>
                    <m:mo>-</m:mo>
                    <m:mrow>
                        <m:mn>5</m:mn>
                        <m:mo>&InvisibleTimes;</m:mo>
                        <m:mi>x</m:mi>
                    </m:mrow>
                    <m:mo>+</m:mo>
                    <m:mn>6</m:mn>
                </m:mrow>
                <m:mo>=</m:mo>
                <m:mn>0.</m:mn>
            </m:mrow>
        </m:math>
        <br/>
        What, you may ask, are this equation's roots?
    </body>
</html>
```


Thanks to namespaces, this XHTML/MathML document works just as it should, as you can see in the W3C Amaya browser in Figure 3.2.

FIGURE 3.2

Viewing an XML document with local namespaces.



You'll be seeing XML namespaces throughout this book, especially when we use the popular XML applications available, such as XHTML.

That finishes the main topics for today's discussion—well-formed documents and namespaces. Before getting into validation in tomorrow's work, however, we'll round off our discussion of XML documents by taking a look at XML infosets and canonical XML. These two topics are worth discussing before we start talking about validation, because they're terms you'll run across as you work with XML, but we're going to consider them optional topics—if you want to skip them and get directly to DTDs, just turn to Day 4.

Understanding XML Infosets

The inspiration behind both *XML infosets* (formally named *XML information sets*) and canonical XML is to make handling the data in XML documents easier. Reducing an XML document down to its infoset is intended to make comparisons between all kinds of XML documents easier by presenting the data in those documents in a standard way. You can find the official XML Information Set specification at `http://www.w3.org/TR/xml-infoset`.

To understand what infosets are and what they're used for, imagine searching for data on the World Wide Web. You might want to search for a particular topic, such as XML, and you would turn up millions of matches. How could you possibly write software to compare those documents? The data in those documents isn't stored in any way that's directly comparable.

That's where infosets come in, because the idea is to regularize how data is stored in an XML document that, ultimately, is designed to let you work with thousands of such documents. The idea behind infosets is to set up an abstract way of looking at an XML document that allows it to be compared to others. (Note that documents need to be well-formed to have an infoset.)

An XML infoset can contain fifteen different types of information items:

- A document information item
- Element information items
- Attribute information items
- Processing instruction information items
- Reference to skipped entity information items
- Character information items
- Comment information items
- A document type declaration information item
- Entity information items

- Notation information items
- Entity start marker information items
- Entity end marker information items
- CDATA start marker information items
- CDATA end marker information items
- Namespace declaration information items

So what software works with infosets? None, really—infosets are primarily theoretical constructs, and the infoset specification is mostly designed to provide a set of definitions that other XML specifications can use when they need to refer to the information in an XML document. Although the term *infoset* has entered common usage as a way to refer to the information in an XML document, it's not a specific enough specification to allow any real implementation. The closest you can come these days to truly regularizing the data in XML documents to make it easy to compare them is to use canonical XML, coming up next.

Understanding Canonical XML

Infosets are only abstract formulations of the information in an XML document. So without reducing an XML document to its infoset, how can you actually approach the goal of being able to actually compare XML documents character by character? You can write your documents in canonical XML.

TIP

You can find a canonical XML tutorial at `www.xfront.com/canonical/CanonicalXML.html`.

Canonical XML is a companion specification to XML, and you can read all about it at `http://www.w3.org/TR/xml-c14n`. Canonical XML is a very strict XML syntax, which lets documents in canonical XML be compared directly.

Using this strict syntax makes it easier to see whether two XML documents are the same. For example, a section of text in one document might read `Black & White`, whereas the same section of text might read `Black &amp; White` in another document, and even `<![CDATA[Black & White]]>` in another. If you compare those three documents byte by byte, they'll be different. But if you write them all in canonical XML, which specifies every aspect of the syntax you can use, these three documents would all have the same version of this text (which would be `Black &amp; White`) and could be compared without problem.

As you might imagine, the canonical XML syntax is very strict; for example, canonical XML uses UTF-8 character encoding only, carriage-return linefeed pairs are replaced with linefeeds (that is, `
`), tabs in CDATA sections are replaced by spaces, all entity references must be expanded, and much more, as specified in `http://www.w3.org/TR/xml-c14n`.

TIP

In their canonical form, documents can be compared directly, and any differences will be readily apparent. Because canonical XML is intended to be byte-by-byte correct, it's often a good idea to use software to convert your XML documents to that form. One such package that will convert valid XML documents to canonical form comes with the XML for Java software that you can get free from IBM's AlphaWorks (`http://www.alphaworks.ibm.com/tech/xml4j`). The actual program is named DOMWriter, and it's part of the XML for Java package.

That completes today's discussion on constructing XML documents. We've covered everything we need to know before we start discussing how to create valid XML documents—and we're going to start doing that tomorrow.

Summary

Today, you took a look at how to create well-formed XML documents. W3C doesn't even consider an XML document to be XML unless it's well-formed. W3C considers an XML document well-formed if it meets three criteria:

- Taken as a whole, it matches the production labeled *document*.
- It meets all the well-formedness constraints given in this specification (that is, the XML 1.0 specification, `http://www.w3.org/TR/REC-xml`).
- Each of the parsed entities, which is referenced directly or indirectly within the document, is well-formed.

The most general of these items says that an XML document must meet the well-formedness constraints in the XML specification, and you took a look today at what that meant.

Those constraints include beginning a document with an XML declaration, using only legal character references, the document must include at least one element, elements must be structured and nested correctly, the root element must contain all other elements, attribute names must be unique, attribute values must be quoted, and so on.

You also took a look at creating namespaces, and how namespaces help you avoid conflicts in XML. To define a namespace, you can assign the `xmlns:`*`prefix`* attribute to a unique identifier (usually a URI), or you can use the `xmlns` attribute to define a default namespace.

Q&A

Q Can I use an XML validator to test an XML document's well-formedness?

A Yes, if you have a DTD or XML schema for the document—an XML validator will also report whether the document is well-formed or not. However, you do need a DTD or XML schema if you want to use a validator—very few will check a document without one. One program that will check an XML document's well-formedness without a DTD or XML schema is Internet Explorer. If the document is not well-formed, you'll see the message `"The XML page cannot be displayed"`, and Internet Explorer will tell you the exact problem with the document.

Q Do I need to use namespaces if there's no chance of tag name conflicts with other XML applications?

A Often, yes. Namespaces aren't used solely to avoid tag (and attribute) name conflicts—using a namespace also indicates to an XML processor what XML application you're using. For example, if you're using MathML, you must use the current MathML namespace or most MathML-enabled XML processors will complain.

Workshop

This workshop tests whether you understand the concepts you saw today. It's a good idea to make sure you can answer these questions before pressing on to tomorrow's work.

Quiz

1. To be well-formed, what's the least number of elements an XML document can contain?
2. Why is the following XML document not well-formed?

```
<?xml version = "1.0" standalone="yes"?>
<employee>
    <name>Frank</name>
    <position>Chef</position>
</employee>
<employee>
    <name>Ronnie</name>
    <position>Chef</position>
</employee>
```

3. Why is the following XML document not well-formed?

```
<?xml version = "1.0" standalone="yes"?>
<employee>
    <kitchen_staff/>
    <name language=en>Frank</name>
    <new_hire />
    <position language=en>Chef</position>
</employee>
```

4. How can you create a namespace named `service` whose URI is `http://www.superduperbigco.com/customer_service`?

5. How could you set the default namespace in a set of XML elements to the URI `http://www.superduperbigco.com/customer_returns`?

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K-L

M

N

O

P

Q-R

S

T

U

V

W

X-Z