Business

- Advertising
- Branding
- Business Management
- Business Ethics
- Careers, Jobs & Employment
- Customer Service
- Marketing
- Networking
- Network Marketing
- Pay-Per-Click Advertising
- Presentation
- Public Relations
- Sales
- Sales Management
- Sales Telemarketing
- Sales Training
- Small Business
- Strategic Planning
- Entrepreneur
- Negotiation Tips
- Team Building
- Top Quick Tips

Internet & Businesses Online

- Affiliate Revenue
- Blogging, RSS & Feeds
- Domain Name
- E-Book
- E-commerce
- Email Marketing
- Ezine Marketing
- Ezine Publishing
- Forums & Boards
- Internet Marketing
- Online Auction
- Search Engine Optimization (SEO)
- Spam Blocking
- Streaming Audio & Online Music
- Traffic Building
- Video Streaming
- Web Design
- Web Development
- Web Hosting
- Web Site Promotion

Finance

- Credit
- Currency Trading
- Debt Consolidation
- Debt Relief
- Loan
- Insurance
- Investing
- Mortgage Refinance
- Personal Finance
- Real Estate
- Taxes
- Stocks & Mutual Fund
- Structured Settlements
- Leases & Leasing
- Wealth Building

Communications

- Broadband Internet
- Mobile & Cell Phone
- VOIP
- Video Conferencing
- Satellite TV

Reference & Education

- Book Reviews
- College & University
- Psychology
- Science Articles

Food & Drinks

- Coffee
- Cooking Tips
- Recipes & Food and Drink
- Wine & Spirits

Home & Family

- Crafts & Hobbies
- Elder Care
- Holiday
- Home Improvement
- Home Security
- Interior Design & Decorating
- Landscaping & Gardening
- Babies & Toddler
- Pets
- Parenting
- Pregnancy

News & Society

- Dating
- Divorce
- Marriage & Wedding
- Political
- Relationships
- Religion
- Sexuality

Computers & Technology

- Computer Hardware
- Data Recovery & Computer Backup
- Game
- Internet Security
- Personal Technology
- Software

Arts & Entertainment

- Casino & Gambling
- Humanities
- Humor & Entertainment
- Language
- Music & MP3
- Philosophy
- Photography
- Poetry

Shopping & Product Reviews

- Book Reviews
- Fashion & Style

Health & Fitness

- Acne
- Aerobics & Cardio
- Alternative Medicine
- Beauty Tips
- Depression
- Diabetes
- Exercise & Fitness
- Fitness Equipment
- Hair Loss
- Medicine
- Meditation
- Muscle Building & Bodybuilding
- Nutrition
- Nutritional Supplements
- Weight Loss
- Yoga

Recreation and Sport

- Fishing
- Golf
- Martial Arts
- Motorcycle

Self Improvement & Motivation

- Attraction
- Coaching
- Creativity
- Dealing with Grief & Loss
- Finding Happiness
- Get Organized - Organization
- Leadership
- Motivation
- Inspirational
- Positive Attitude Tips
- Goal Setting
- Innovation
- Spirituality
- Stress Management
- Success
- Time Management

Writing & Speaking

- Article Writing
- Book Marketing
- Copywriting
- Public Speaking
- Writing

Travel & Leisure

- Aviation & Flying
- Cruising & Sailing
- Outdoors
- Vacation Rental

Cancer

- Breast Cancer
- Mesothelioma & Asbestos Cancer

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Introduction

XML started as an obscure effort driven by a small group of dedicated SGML experts who were convinced that the world needed something more powerful than HTML. Although XML hasn't yet taken the world by storm, in its quiet way it is poised to revolutionize the Internet and usher in a new age of electronic commerce.

Until recently, the non-technical Internet user has largely written off XML as being more of a programmers' language than a technology that applies to us all. Nearly two years after XML's inception, there is still no real mainstream software support in the form of editors and viewers. However, just as with HTML, as the technology becomes adopted, the tools will start to arrive. Netscape and Microsoft have already given us a taste of what is to come.

Sams Teach Yourself XML in 21 Days teaches you about XML and its related standards (the XSL style language, XLink and XPointer hyperlinking, XML Data, and XSchema, to name just a few), but it doesn't stop there. As you follow the step-by-step explanations, you will also learn how to *use* XML. You will be introduced to a wide range of the available tools, from the newest to the tried and tested. By the time you finish this book, you'll know enough about XML and its use within the available tools to use it immediately.

How This Book Is Organized

Sams Teach Yourself XML in 21 Days covers the latest version of XML, its related standards, and a wide variety of tools. Some features of the tools will have been enhanced or expanded by the time you read this, and new tools will certainly have

become available. Keep this in mind when you're working with the early versions of some of the software packages. If something doesn't work as it should, or if you feel that there is something important missing, check the Web sites mentioned in Appendix B, "XML Resources," to see if a newer version of the package is available.

Sams Teach Yourself XML in 21 Days is organized into three separate weeks. Each week brings you to a certain milestone in your learning of XML and development of XML code.

In the first week, you'll learn a lot of the basics about XML itself:

- On Day 1, you'll get a basic introduction on what XML is and why it's so important. You will also see your first XML document.
- On Day 2, you will dissect an XML document to discover exactly what goes into making usable XML code. You will also create your first XML document.
- On Day 3, you'll go a little further into the basics of XML code. You'll learn about elements, comments, processing instructions, and using CDATA sections to hide XML code you don't want to be processed.
- On Day 4, you will learn more about markup and elements by exploring attributes. You'll also learn the basics of information modeling and some of the ground rules of Document Type Definition (DTD) development. You will learn how to work with DTDs without having to go as far as creating valid XML code, and you will discover how much you can already achieve by creating well-formed XML documents.
- On Day 5, you'll reach an important milestone. You will learn how to put together everything you have learned so far and produce well-formed XML documents. You will be introduced to some basic parsing tools and then learn how to check and correct your XML documents.
- On Day 6, you will learn all about DTDs, their subsets, and how they are used to check XML documents for validity.
- On Day 7, you'll delve even further into the treacherous waters of DTD development and learn some of the major tricks of the trade that open the doors to advanced XML document construction.

Week two takes you into the "power" side of XML authoring:

- On Day 8, you will learn about entities and notations, and how to import external objects such as binary code and graphics files into your XML documents.
- On Day 9, you'll arrive at the next major milestone. You will be introduced to a couple of the leading XML parsers, and you'll learn how to validate your XML documents and recognize and correct some of the most common errors.
- On Day 10, you will discover the power of XML's linking mechanisms. Using practical examples, you will learn how you can use XML links to go far beyond HTML's humble features.
- On Day 11, you will continue to explore XML's linking mechanisms. You will learn how you can link to ranges, groups, and indirect blocks of data inside both XML and non-XML data.
- On Day 12, with much of the theory already in your grasp, you will learn how you can actually display the XML code you've written in Microsoft's Internet Explorer 5.

- On Day 13, you will continue the hands-on work of Day 12 by learning how to display the XML code you've written in Mozilla, Netscape's Open Source testbed for the development of future versions of its Web browser software.
- On Day 14, you will learn the basics of XML document processing. You will be introduced to the principles of tree-based and event-driven processing and learn when and how to apply them.

Week three takes you beyond XML authoring and teaches you how to process XML and HTML code.

- On Day 15, you will learn more about event-driven processing. You will learn how to download, install, and use two of the leading tools: Omnimark and SAX.
- On Day 16, going several steps further, you will learn how to use the Document Object Model (DOM) to gain programmatic access to everything inside an XML document.
- On Day 17, you will temporarily turn your back on XML code as a means of coding documents and examine how it's used to code data. You will learn why a DTD sometimes isn't enough, and you'll be introduced to some of the most important XML schemas.
- On Day 18, you will return to using XML for documents and explore how the Cascading Style Sheet language (CSS), originally intended for use with HTML, can be used just as easily with XML code. With the aid of practical examples, you will learn how you can legitimately use CSS code to render XML code. If that doesn't work, you'll also learn a few tricks to fool the browser into doing what you want it to do.
- On Day 19, you will learn the basics of DSSSL, the style language for rendering and processing SGML code. You will learn how easy it can be to use DSSSL to transform not just SGML code, but also XML and HTML code. With the help of numerous examples, you will also learn how to convert XML code into HTML and RTF, and how to convert HTML into RTF or even FrameMaker MIF using jade.
- On Day 20, you will be briefly introduced to earlier versions of the XML style languages before concentrating on XSL. Using the very latest XSL tools, you will learn how to create your own XSL style code and display the results.
- On Day 21, you will learn the basics of MathML, the mathematics application of XML, as well as the various initiatives to describe graphics in XML. (No book on XML would be complete without some mention of its applications.) Using practical examples, you will be introduced to VML and see how you can already use it in Microsoft Internet Explorer, versions 4 and 5. Finally, you will take a peek at some of the new developments that are just around the corner, such as Office 2000, CSS behaviors, and Microsoft's Chrome.

The end of each chapter offers common questions and answers about that day's subject matter and some simple exercises for you to try yourself. At the end of the book, you will find a comprehensive glossary and an extensive appendix of XML resources containing pointers to most of the software packages available, whether mentioned in this book or not, and pointers to the most important sources of further information.

This Book's Special Features

This book contains some special features to help you on your way to mastering XML.

Tips provide useful shortcuts and techniques for working with XML. Notes provide special details that enhance the explanations of XML concepts or draw your attention to important points that are not immediately part of the subject being discussed. Warnings highlight points that will help you avoid potential problems.

Numerous sample XML, DSSSL, XSL, HTML, and CSS code fragments illustrate some of the features and concepts of XML so that you can apply them in your own document. Where possible, each code fragment's discussion is divided into three components: the code fragment itself, the output generated by it, and a line-by-line analysis of how the code fragment works. These components are indicated by special icons.

Each day ends with a Q&A section containing answers to common questions relating to that day's material. There is also a set of exercises at the end of each day. We recommend that you attempt each exercise. You will learn far more from doing yourself than just seeing what others have done. Most of the exercises do not have any one answer, and the answers would often be very long. As a result, most chapters don't actually provide answers, but the method for finding the best solution will have been covered in the chapter itself.

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

About the Author

Simon North originally hails from England, but thinks of himself as more of a European. Fluent in several European languages, Simon is a technical writer for Synopsys, the leading EDA software company, where he documents high-level IC design software. This puts him in the strange situation of working for a Silicon Valley company in Germany while living in The Netherlands.

Simon has been working with SGML and HyTime-based documentation systems for the past nine years, but was one of the first to adopt HTML. His writing credits include contributions on XML and SGML to the Sams.Net books *Presenting XML*, *Dynamic Web Publishing Unleashed*, and *HTML4 Unleashed*, Professional Reference Edition. Simon can be reached at north@synopsys.com (work) or sintac@xs4all.nl (or through his books Web page at http://www.xs4all.nl/~sintac/books.html).

Paul Hermans is founder and CEO of Pro Text, one of the leading SGML/XML consultant firms and implementation service providers in Belgium.

Since 1992 he has been involved in major Belgian SGML implementations. Previously he was head of the electronic publishing department of CED Samsom, part of the Wolters Kluwer group. He is also the chair of SGML BeLux, the Belgian-Luxembourgian chapter of the International SGML Users' Group.

Dedications

From Simon North:

To the thousands of givers in the online community without whose dedication, hard work, generosity, and selflessness the Internet would be just a poor, sad reflection of everyday life.

From Paul Hermans:

To Rika for bringing structure into my life and to my parents for caring.

Acknowledgements

From Simon North:

To all the folks at Sams for giving me the chance to write this book and for allowing me to make it the book I wanted it to be. To all my colleagues at Synopsys who made my working life so pleasant and gave me the enthusiasm and energy to survive the extra workload. Most of all, to my long-suffering wife Irma without whose willingness to spring into the breach and assume most of my parental responsibilities this book just wouldn't have been possible.

From Paul Hermans:

I would like to thank Simon North for giving me the opportunity to put some of my knowledge on paper. Furthermore I would like to acknowledge all the people at Sams Publishing who helped bring this book to completion.

Tell Us What You Think!

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'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As the Executive Editor for the Java team at Macmillan Computer Publishing, I welcome your comments. You can fax, 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 stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

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

Fax: 317-817-7070

Email: webdev@mcp.com

Mail: Mark Taber, Executive Editor

Web Development Team

Macmillan Computer Publishing

201 West 103rd Street

Indianapolis, IN 46290 USA

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Part I

Chapter 1
What Is XML and Why Should I Care?

Welcome to *Sams Teach Yourself XML in 21 Days*! This chapter starts you on the road to mastering the *Extensible Markup Language* (*XML*). Today you will learn

- The importance of XML in a maturing InternetN
- The weaknesses of HTML that make it unsuitable for Internet commerce
- What SGML, the Standard Generalized Markup Language is and XML's relation to it
- The weaknesses of other tag and markup languages
- What XML adds to both SGML and HTML

- The advantages of XML for non-programmers

The Web Grows Up

Love them or hate them, the Internet and the World Wide Web (WWW) are here to stay. No matter how much you try, you can't avoid the Web playing an increasingly important role in your life.

The Internet has gone from a small experiment carried out by a bunch of nuclear research scientists to one of the most phenomenal events in computing history. It sometimes feels like we have been experiencing the modern equivalent of the Industrial Revolution: the dawning of the Information Age.

In his original proposal to CERN (the European Laboratory for Particle Research) management in 1989, Tim Berners-Lee (the acknowledged inventor of the Web) described his vision of

> ...a universal linked information system, in which generality and portability are more important than fancy graphics and complex extra facilities.

The Web has certainly come a long way in the last ten years, and I sometimes wonder what Berners-Lee thinks of his invention in its present form.

The Web is still in its infancy, however. Use of the Web is slowly progressing beyond the stage of family Web pages, but the dawn of electronic commerce (e-commerce) via the Internet has not yet broken. By e-commerce, I do not mean being able to order things from a Web page, such as books, records, CDs, and software. This kind of commerce has been going on for several years, and some companies—most notably Amazon.com—have made a great success of it. My definition of e-commerce goes much deeper than this. Various new initiatives have appeared in recent years that are going to change the way a lot of companies look at the Web. These include

- Using the Internet to join the parts of distributed companies into one unit
- Using the Internet for the exchange of financial transaction information (credit card transactions, banking transactions, and so on)
- The exchange over the Internet of medical transaction data between patients, hospitals, physicians, and insurance agencies
- The distribution of software via the Web, including the possibility of creating zero-install software and of modularizing the massive suites of software in programs such as Microsoft Word so that you only load, use, and pay for the parts that you need

Every time you visit a Web site that supports Java, JavaScript, or some other scripting language, you are in fact running a program over the Web. After you've finished with it, all that's left in your Web browser's cache is possibly a few scraps of code. Several software companies—including Microsoft—want to distribute software in this way. They'd gain by constantly generating new income from their software, and you would benefit by only having to pay for the software you used at the time that you used it, and only for as long as you used it.

Whereas most of these applications are impossible using *Hypertext Markup Language* (*HTML*), XML can make all these applications (and many more) real possibilities. In a sense, XML is the enabling technology that heralds the appearance of a new form of Internet society. XML is probably the most important thing to happen to the Web since the arrival of Java.

So why can XML do what HTML can't? Read on for an explanation.

Where HTML Runs Out of Steam

Before we look at all the weaknesses of HTML, let's get one thing clear: HTML has been, and still is, a fantastic success.

Designed to be a simple tagging language for displaying text in a Web browser, HTML has done a wonderful job and will probably continue to do so for many years to come. It is no exaggeration to say that if there hadn't been HTML, there simply wouldn't have been a Web. Although Gopher, WAIS, and Hytelnet, among others, predated HTML, none of them offered the same trade-off of power for simplicity that HTML does.

Although HTML might still be considered the killer Internet application, there have been a lot of complaints leveled against it. Furthermore, people are now realizing that XML is superior to HTML. Following are some of the most frequently cited complaints against HTML (but many of them aren't really legitimate, as you will see from my comments):

- **HTML lacks syntactic checking: You cannot validate HTML code.**
  Yes and no. There are formal definitions of the structure of HTML documents—as you will learn later, HTML is an SGML application and there is a document type definition (DTD) for every version of HTML.

The *document type definition (DTD)* is an SGML or XML document that describes the elements and attributes allowed inside all the documents that can be said to conform to that DTD. You will learn all about XML DTDs in later chapters.

There are also some tools (and one or two Web sites) readily available for checking the syntax of HTML documents. This begs the question of why more people don't validate their HTML documents; the answer is that the validation is really a bit misleading. Web browsers are designed to accept almost anything that looks even slightly like HTML (which runs the risk that the display will look nothing like what you expected—but that's another story). Strangely enough, the only tag that is compulsory in an HTML document is the TITLE tag; equally strangely, this is one of the least common tags there is.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

- **HTML lacks structure.**

Not really. HTML has ordered heading tags (H1 to H6), and you can nest blocks of information inside DIV tags. Browsers don't care what order you use the headings in, and often the choice is simply based on the size of the font in which they are rendered. This isn't HTML's fault. The problem lies in how HTML code is used.

- **HTML is not content-aware.**

Yes and no. Searching the Web is complicated by the fact that HTML doesn't give you a way to describe the information content—the semantics—of your documents. In XML you can use any tags you like (such as <NAME> instead of <H3>), but using attributes in tags (such as <H3 CLASS="name">) can embed just as much semantic information as custom tags can. Without any agreement on tag names, the value of custom tags becomes a bit doubtful. To worsen matters, the same tag name in one context can mean something completely different in another. Furthermore, there are the complications of foreign languages—seeing <inkoopprijs> isn't going to help very much if you don't know that it's Dutch for "purchase price."

- **HTML is not international.**

Mostly true. There were a few proposals to internationalize HTML, and most particularly to give it a way of identifying the language used inside a tag.

- **HTML is not suitable for data interchange.**

Mostly true. HTML's tags do little to identify the information that a document contains.

- **HTML is not object-oriented.**

True. Modern programmers have been making a long and difficult transition to object-oriented techniques. They want to leverage these skills and have such

things as inheritance, and HTML has done very little to accommodate them.

- **HTML lacks a robust linking mechanism.**

Very true. If you've spent a few hours on the Web, you've probably encountered at least one broken link. Although broken links are the curse of Web managers the world over, there is little that can be done to prevent them. HTML's links are very much one-to-one, with the linking hard-coded in the source HTML files. If the location of one target file changes, a Webmaster may have to update dozens or even hundreds of other pages.

- **HTML is not reusable.**

True. Depending on how well-written they are, HTML pages and fragments of HTML code can be extremely difficult to reuse because they are so specifically tailored to their place in the web of associated pages.

- **HTML is not extensible.**

True but unfair. This is a bit like saying that an automobile makes a better motor vehicle than a bicycle. HTML was never *meant* to be extensible.

So what's really wrong with HTML? Not a lot, for everyday Web page use. However, looking at the future of electronic commerce on the Web, HTML is reaching its limits.

So What's Wrong with...?

All right, if HTML can't handle it, what's wrong with TeX, PDF, or RTF?

TeX is a computer typesetting language that still flourishes in scientific communities. In the early 1980's, there were online databases that returned data in TeX form that could be inserted straight into a TeX document. Adobe owns the *PDF* (Adobe Acrobat) standard, but it is fairly well documented. *RTF* is the property of Microsoft and, as many Windows Help authors will tell you, it is poorly documented and extremely unreliable. The RTF code created by Word 97 is not the same as the code created by Word 95, for example, and in some areas the two versions are completely incompatible.

All of these formats suffer from the same weaknesses: they are proprietary (owned by a commercial company or organization), they are not open, and they are not standardized. By using one of these formats, you risk being left out in the cold. Although the market represents a strong stabilizing force (as seen with RTF), when you place too much reliance on a format over which you have no control and into which you have little insight, you are leaving yourself open to a lot of problems if and when that format changes.

SGML

I'm going to try to avoid teaching you as much as I can about SGML. Although it can be helpful to know a little about it, in many ways you're probably better off not knowing anything about it at all. The problem with learning too much about SGML is that when you move to XML you'd have to spend most of your time forgetting a lot of the things you'd just learned. XML is different enough from SGML that you can become an expert in XML without knowing a thing about SGML.

That said, XML is very much a descendant of SGML, and knowing at least a little

about SGML will help put XML in context.

The *Standard Generalized Markup Language* (*SGML*), from which XML is derived, was born out of the basic need to make data storage independent of any one software package or software vendor. SGML is a *meta language*, or a language for describing markup languages. HTML is one such markup language and is therefore called an SGML application. There are dozens, maybe even hundreds, of markup languages defined using SGML. In XML, these applications are often called *markup languages*—such as the *hand-held device markup language* (*HDML*) and the *FAQ markup language* (*QML*).

In SGML, most of these markup languages haven't been given formal names; they are simply referred to by the name of their document type definition (DocBook), their purpose (LinuxDOC), their application (TEI), or even the standard they implement (J2008—automobile parts, Mil-M-38784—US Military).

By means of an SGML declaration (XML also has one), the SGML application specifies which characters are to be interpreted as data and which characters are to be interpreted as markup. (They do not have to include the familiar < and > characters; in SGML they could just as easily be { and } instead.)

Using the rules given in the SGML declaration and the results of the information analysis (which ultimately creates something that can easily be considered an information model), the SGML application developer identifies various types of documents—such as reports, brochures, technical manuals, and so on—and develops a DTD for each one. Using the chosen characters, the DTD identifies information objects (elements) and their properties (attributes).

The DTD is the very core of an SGML application; how well it is made largely determines the success or failure of the whole activity. Using the information elements defined in the DTD, the actual information is then marked up using the tags identified for it in the application. If the development of the DTD has been rushed, it might need continual improvement, modification, or correction. Each time the DTD is changed, the information that has been marked up with it might also need to be modified because it may be incorrect. Very quickly, the quantity of data that needs modification (now called *legacy data*) can become a far more serious problem—one that is more costly and time-consuming than the problem that SGML was originally introduced to solve.

You are already getting a feel for the magnitude of an SGML application. There are good reasons for this magnitude: SGML was built to last. At the back of the developers' minds were ideas about longevity and durability, as were thoughts of protecting data from changes in computer software and hardware in the future.

SGML is the industrial-strength solution: expensive and complicated, but also extremely powerful.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Why Not SGML?

The SGML on the Web initiative existed a long time before XML was even considered. Somehow, though, it never really succeeded. Basically, SGML is just too expensive and complicated for Web use on a large scale. It isn't that it *can't* be used—it's that it *won't* be used. Using SGML requires too much of an investment in time, tools, and training.

Why XML?

XML uses the features of SGML that it needs and tries to incorporate the lessons learned from HTML.

> One of the most important links between XML and SGML is XML's use of a DTD. On Day 17, "Using Meta-Data to Describe XML Data," you will learn more about the developments that are underway to cut this major link to SGML and replace the DTD with something more in keeping with the data-processing requirements of XML applications.

When the designers of XML sat down to write its specifications, they had a set of design goals in mind (detailed in the recommendation document). These goals and the degree to which they have already been met are why XML is considered better than SGML:

- XML can be used with existing Web protocols (such as HTTP and MIME) and mechanisms (such as URLs), and it does not impose any additional requirements. XML has been developed with the Web in mind—features of SGML that were too difficult to use on the Web were left out, and features that are needed for Web use either have been added or are inherited from applications that already work.
- XML supports a wide variety of applications. It is difficult to support a lot of applications with just HTML; hence, the growth of scripting languages. HTML is simply too specific. XML adopts the generic nature of SGML, but adds flexibility to make it truly extensible.
- XML is compatible with SGML, and most SGML applications can be converted into XML. In the foreseeable future, the SGML standard will be amended to make XML applications fully backward-compatible.
- It is easy to write programs that process XML documents. One of the major strengths of HTML is that it's easy for even a non-programmer to throw together a few lines of scripting code that enable you to do basic processing (and there's an amazing variety of scripting languages available). HTML even includes some features of its own that enable you to carry out some basic processing (such as forms and CGI query strings). XML has learned a lesson from HTML's success and has tried to stay as simple as possible by throwing out a lot of SGML's more complex features. XML processing applications are already appearing in Java, SmallTalk, C, C++, JavaScript, Tcl, Perl, and Python, to name just a few.
- The number of optional features in XML has been kept to an absolute minimum. SGML has many optional features, so SGML software has to support all of them. It can be argued that there isn't actually a single software package that supports all of SGML's features (and it's difficult to imagine an application that actually needs all of them). This degree of power immediately implies complexity, which also means size, cost, and sluggishness. The speed of the Web is already becoming a major concern; it's bad enough to wait for a document to download, but if you had to wait ages for it to be processed as well, XML would be doomed from the start.
- XML documents are reasonably clear to the layperson. Although it is becoming increasingly rare, and even difficult, for HTML documents to be typed in manually, and XML documents weren't intended to be created by human beings, this remains a worthy goal. Machine encoding is limited in longevity and portability, often being tied to the system on which it was created. XML's markup is reasonably self-explanatory.

Given the time, you can print out any XML document and work out its meaning—but it goes further than this. A valid XML document

  - Describes the structural rules that the markup attempts to follow
  - Lists any external resources (external entities) that are part of the document
  - Declares any internal resources (internal entities) that are used within the document
  - Lists the types of non-XML resources (notations) used and identifies any helper applications that might be needed
  - Lists any non-XML resources (binaries) that are used within the document and identifies any helper applications that might be needed

- The design of XML is formal and concise. The Extended Backus-Naur Format (EBNF) was used as the basis of the XML specification (a method well

understood by the majority of programmers). Information marked up in XML can be easily processed by computer programs. Better still, by using a system that is familiar to computer programmers and is almost completely unambiguous, it is reasonably easy for programmers to develop programs that work with XML.

- XML documents are easy to create. HTML is almost famous for its ease of use, and XML capitalizes on this strength. In fact, it is actually even easier to create an XML document than an HTML document. After all, you don't have to learn any markup tags—you can create your own!

What XML Adds to SGML and HTML

XML takes the best of SGML and combines it with some of the best features of HTML, and adds a few features drawn from some of the more successful applications of both. XML takes its major framework from SGML, leaving out everything that isn't absolutely necessary. Each facility and feature was examined, and if a good case couldn't be made for its retention, it was scrapped. XML is commonly called a *subset* of SGML, but in technical terms it's an *application profile* of SGML; whereas HTML uses SGML and is an application of SGML, XML is just SGML on a smaller scale.

From HTML, XML inherits the use of Web addresses (URLs) to point to other objects. From *HyTime* (a very sophisticated application of SGML, officially called *ISO/IEC 10744 Hypermedia/Time-based Structuring Language*) and an academic application of SGML called the *Text Encoding Initiative* (*TEI*), XML inherits some other extremely powerful addressing mechanisms that allow you to point to parts and ranges of other documents rather than simple single-point targets, for example.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

XML also adds a list of features that make it far more suitable than either SGML or HTML for use on an increasingly complex and diverse Web:

- **Modularity**—Although HTML appears to have no DTD, there is an implied DTD hard-wired into Web browsers. SGML has a limitless number of DTDs, on the other hand, but there's only one for each type of document. XML enables you to leave out the DTD altogether or, using sophisticated resolution mechanisms, combine multiple fragments of either XML instances or separate DTDs into one compound instance.
- **Extensibility**—XML's powerful linking mechanisms allow you to link to material without requiring the link target to be physically present in the object. This opens up exciting possibilities for linking together things like material to which you do not have write access, CD-ROMs, library catalogs, the results of database queries, or even non-document media such as sound fragments or parts of videos. Furthermore, it allows you to store the links separately from the objects they link (perhaps even in a database, so that the link lists can be automatically generated according to the dynamic contents of the collection of documents). This makes long-term link maintenance a real possibility.
- **Distribution**—In addition to linking, XML introduces a far more sophisticated method of including link targets in the current instance. This opens the doors to a new world of *composite documents*—documents composed of fragments of other documents that are automatically (and transparently) assembled to form what is displayed at that particular moment. The content can be instantly tailored to the moment, to the media, and to the reader, and might

have only a fleeting existence: a virtual information reality composed of virtual documents.

- **Internationality**—Both HTML and SGML rely heavily on ASCII, which makes using foreign characters very difficult. XML is based on Unicode and requires all XML software to support Unicode as well. Unicode enables XML to handle not just Western-accented characters, but also Asian languages. (On Day 8, “XML Objects: Exploiting Entities,” you will learn all about character sets and character encoding.)
- **Data orientation**—XML operates on data orientation rather than readability by humans. Although being humanly readable is one of XML’s design goals, electronic commerce requires the data format to be readable by machines as well. XML makes this possible by defining a form of XML that can be more easily created by a machine, but it also adds tighter data control through the more recent XML schema initiatives.

Is XML Just for Programmers?

Having read this far, you might think that XML is only for programmers and that you can quite happily go back to using HTML. In many ways you’d be right, except for one important point: If programmers can do more with XML than they can with HTML, eventually this will filter down to you in the form of application software that you can use with your XML data. To take full advantage of these tools, however, you will need to make your data available in XML. As of yet, support for XML in Web browsers is incomplete and unreliable (you will learn how to display XML code in Mozilla and Internet Explorer 5 later on), but full support will not take long.

In the meantime, is XML just for programmers? Definitely not! One of the problems with HTML is that all the tags are optional, so you have to be somewhat familiar with all of them in order to make the best choice. Worse, your choice will be affected by the way the code looks in a particular browser. But XML is extensible, and extensibility works both ways—it also means you can use less rather than more. Instead of having to learn more than 40 HTML tags, you can mark up your text in a way that makes a lot more sense to you and then use a style sheet to handle the visible appearance. Listing 1.1 shows a typical XML document that marks up a basic sales contact entry.

Listing 1.1 A Simple XML Document

```
1: <?xml version=“1.0”?>
2:   <contacts>
3:     <contact>
4:       <name>
5:         <first>John</first>
6:         <last>Belcher</last>
7:       </name>
8:       <address>
9:         <street>Pennington 13322</street>
10:        <city>Washington</city>
11:        <state>DC</state>
12:        <zip>66522</zip>
```

```
13:       </address>
14:       <tel>555 1276</tel>
15:       <fax>555 9983</fax>
16:       <mobile>887 8887 7777</mobile>
17:       <email>jb@southside.com</email>
18:     </contact>
19:  </contacts>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

As Listing 1.1 suggests, you can make your markup very rich in information (semantic content). The great thing about XML is that you can adapt it to your needs. When you need less you can use less, as demonstrated by Listing 1.2. (It would hardly be in keeping with all the other computer language-oriented books in the world if we didn't include some kind of "Hello World" example.)

Listing 1.2 "Hello World" in XML

```
1: <?xml version="1.0"?>
2:   <greeting>
3:     <salutation>Hello</salutation>
4:     <target>World!</target>
5:   </greeting>
```

XML is already becoming the preferred language for interfacing between databases and the Web, and it is becoming an important method for interchanging data between computers and computer applications. However, at the level of "ordinary" Web document authoring, XML still has a lot to offer. The wonderful thing about XML is that it can actually be even simpler than HTML! You can decide what tags you'll need and how many, and you can choose names that either mean something sensible to you

or to your readers. Instead of producing documents containing meaningless jumbles of H1, H2, P, LI, UL, and EM tags, you can say what you really mean and use CHAPTER, SECTION, PARAGRAPH, LIST.ITEM, UNNUMBERED.LIST, and IMPORTANT. This doesn't just make your documents more meaningful, it makes them more accessible to other people. Tools (such as search engines) will be able to make more intelligent inquiries about the content and structure of your documents and make meaningful inferences about your documents that could far exceed what you originally intended.

Summary

On this first day, you were introduced to XML as a markup language in abstract terms. You saw why XML is needed by the rapidly maturing Internet and its commercial applications. You were also given a very brief overview of why XML is seen as the solution to publishing text and data through the Internet, rather than SGML or HTML.

Just as medical students start their education by dissecting corpses, tomorrow you will dissect the anatomy of an XML document to determine what it is made of.

Q&A

Q Is XML a standard, and can I rely on it? A
A XML is recommended by a group of vendors, including Microsoft and Sun, called the World Wide Web Consortium (W3C). This is about as close to a standard as anything on the Web. The W3C has committed itself to supporting XML in all its other initiatives. Also, in the regular standardization circles, the SGML standard is being updated so that XML can rely on the support and formality of SGML.
Q Do I need to learn SGML to understand XML?
A No. It might help to know a little about SGML if you're going to get involved in highly technical XML developments, but no knowledge of SGML is needed for most XML applications.
Q I know SGML; how difficult will it be for me to learn XML?
A If you already have some experience with SGML, it will take less than a day to convert your knowledge to XML and learn anything extra you'll need to know. However, you'll need the discipline to unlearn some of the things you were doing with SGML.
Q I know HTML; how difficult will it be for me to learn XML?
A This depends on how deep your knowledge of HTML is and what you intend to do with XML. If all you want to do with XML is create Web pages, you can probably master the basics in a day or two.
Q Will XML replace SGML?
A No. SGML will continue to be used in the large-scale applications where its features are most needed. XML will take over some of the work from SGML but will never replace it.
Q Will XML replace HTML?
A Eventually, yes. HTML has done a wonderful job so far, and there is every reason to believe it will continue to do so for a long time to come. Eventually, though, HTML will be rewritten as an XML application instead of being an SGML application—but you are unlikely to notice the difference.
Q I have a lot of HTML code; should I convert it to XML? If so, how?

A No. Existing HTML code can be expressed very easily in XML syntax. It will also be possible to include HTML code in XML documents, and vice versa. However, it is not quite so simple to convert an HTML authoring environment into an XML one. Currently there are no XML DTDs for HTML. Until there are, it's easier to create the HTML code using HTML (or SGML) tools and then convert the finished code.

Exercise

1. You've already seen what a basic XML document looks like. Mark up a document that you'd like to use on the Web (something personal, like a home page or the tracks on a CD).

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 2
Anatomy of an xml document

Just as student doctors begin their medical training by dissecting a human body and learning the nature and relation of the parts before they learn how to treat them, this exploration of XML begins with an examination of a small XML document with all its parts identified. Today you will

- Learn about a short XML document and its components
- Examine the difference between markup and character data
- Explore logical and physical structures

Markup

Before cutting into the skin of XML, as it were, let's quickly take a small step back and review one of the most basic concepts—markup. Yesterday, you learned about a couple of markup languages and some of the detailed features of a few implementations, such as TeX, but what exactly is markup?

At its most simple, *markup* involves adding characters to a piece of information that can be used to process that information in a particular way. At one end of the scale, it could be something as basic as adding commas between pieces of data that can be interpreted as field separators when the information is imported into a database program. At its most complex, it can be an extremely rich meta-language such as the Text Encoding Initiative (TEI) SGML DTD. The TEI DTD makes it possible to mark up transcriptions of historical document manuscripts to identify the particular version, the translation, the interpretation, comments about the content, and even a whole library of additional information that could be of use to anyone carrying out academic work related to the manuscript.

What actually constitutes the markup and what doesn't is a matter that has to be resolved by the software—the *application.* Compare the WordPerfect code for a single sentence shown in Listing 2.1 with the same sentence in Microsoft's RTF format shown in Listing 2.2.

Listing 2.1 WordPerfect Code

```
ÿWPC)_  _  °ÿ_ 2 x_  _ _   B   _ (   J   _ ··   r    Z   _  _ | x
Times New Roman  (TT) Courier New  (TT)  _ X___Ù_C Ù_\__  _ Pé “6Q___
_ _ Ù___Ù_d ?_6_X_   __@... DèQ___
°?_ 2 C_  _ <   ™_  _ [   æ_  ??    A_  _ _   A_
ÿ__ ?Ùé?phoenix _   _ _ÿ—_   Ù_C Ù_\__           _ Pé “6Q___
__  _— At its most simple, markup is simply ad
Simon North Simon North    °?_ 2     __¥   u_
Default Paragraph Fo _Default Paragraph Font  _.   .
—_   X_P Ù_\__  _ Pé “6Q___ __  _—”__  __ _””__ _
_ _”—_   Ù_C Ù_\__  _ Pè “6Q___ __
_——_ 8_8_⇐_⇐_ _--__ _ _ _-”__
__ _”At its most simple, markup is simply adding
characters to a piece of information.—_
Ù_d ?_6_X_   __@... DèQ_____  _—
```

Listing 2.2 Microsoft Word RTF Code

```
{\rtf1\ansi\ansicpg1252\uc1 \deff0\deflang1033
\deflangfe1033{\fonttbl{\f0\froman\fcharset0\fprq2
{\*\panose 02020603050405020304}Times New Roman;}
{\f2\fmodern\fcharset0
\fprq1{\*\panose 02070309020205020404}Courier New;}}
{\stylesheet{
\widctlpar\adjustright \fs20\cgrid \snext0 Normal;}
{\*\cs10 \additive Default Paragraph Font;}}{\info{\title
simple}{\edmins1}{\nofpages1}{\nofwords0}{\nofchars0}{\nofcharsws0}
{\vern89}}
\widowctrl\ftnbj\aenddoc\formshade\viewkind4\viewscale100
\pgbrdrhead\pgbrdrfoot \fet0\sectd \linex0\endnhere\sectdefaultcl
{\*\pnseclvl9\pnlcrm\pnstart1\pnindent720\pnhang{\pntxtb (}
{\pntxta )}}
\pard\plain
\widctlpar\adjustright
\fs20\cgrid {At its most simple, markup is simply adding chara}
```

```
{cters to a piece of information.}
{\f2 \par }}
```

You could claim that WordPerfect and RTF codes aren't really markup as such, but they are. They certainly aren't what most people would think of as markup, and they aren't as readable as the markup you will encounter in the rest of this book, but they are just as much markup as any of the XML element tags you will encounter. These codes are in fact a form of *procedural* markup, which is used to drive processing by a particular application.

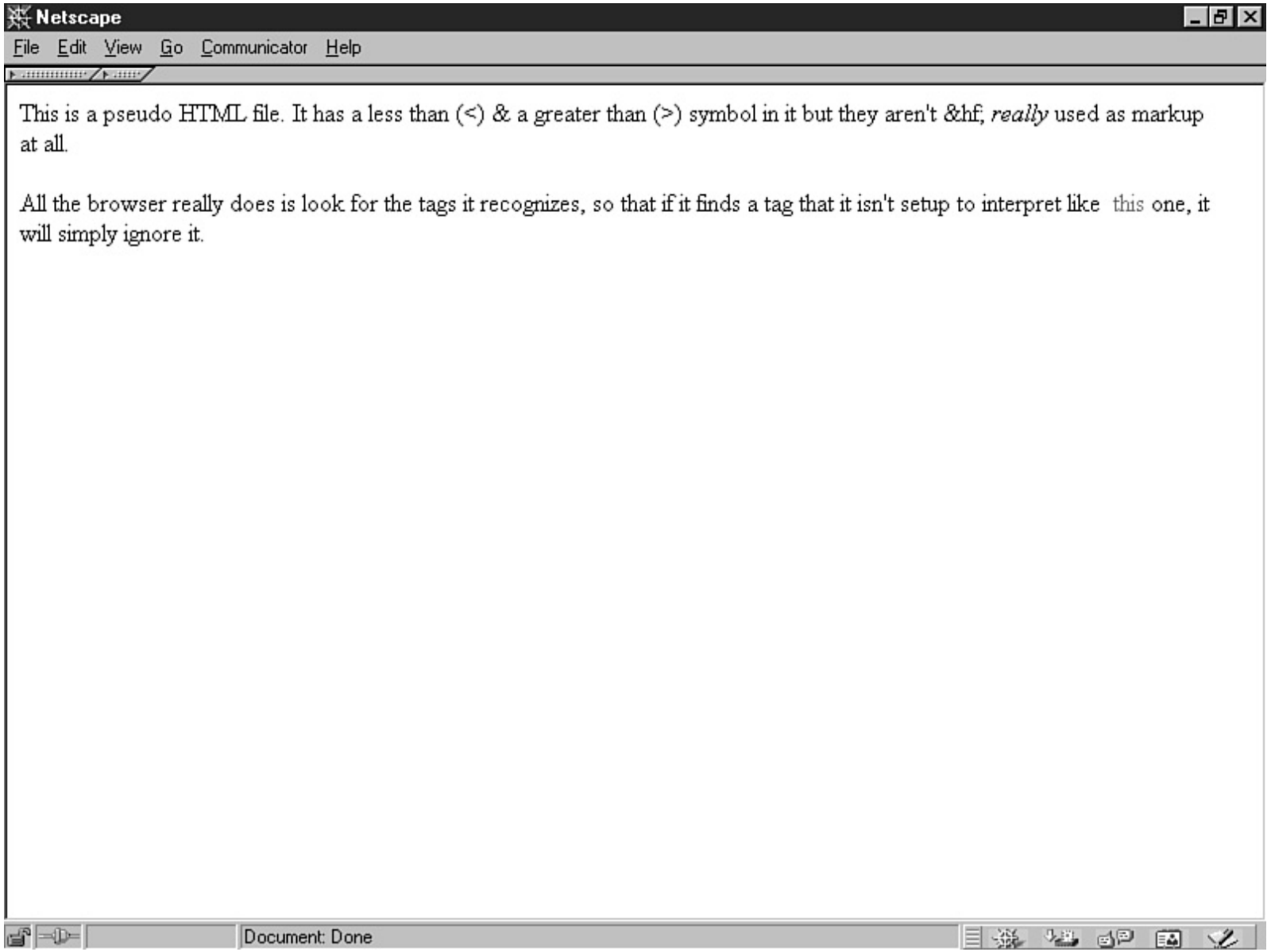
Obviously, there's no point in expecting WordPerfect code to be usable in Microsoft Word, and it would be just as unreasonable to expect WordPerfect to work with Microsoft RTF code (even though they can import each other's documents). These two examples of markup are proprietary, and any portability between applications should be considered a bonus rather than a requirement.

SGML is intended to be absolutely independent of any application. As pure markup, it often is independent, and the SGML code that you produce in one SGML package is directly portable to any other SGML application. (You might not be able to do much with it until you've added some local application code, but that's another story.) Life isn't quite that simple, though. Within the context of SGML, the word *application* has taken on a meaning of its own.

An SGML application consists of an SGML declaration and an SGML DTD. The SGML declaration establishes the basic rules for which characters are considered to be markup characters and which aren't. For example, the SGML declaration could specify that elements are marked up using asterisks instead of the familiar angle brackets (*book* instead of <book>). The DTD can then introduce all sorts of additional rules, such as *minimization* rules that allow markup to be deduced from the context, for example.

Going one step further, you could use the markup minimization rules and the element models defined in the DTD to create a document that contained only normal English words. When processed (parsed) by the SGML software, the beginnings and ends of the elements the document contained would be implied and treated as though they were explicitly identified. Compare Listing 2.3, which uses all the minimization techniques that SGML offers in as single document (it is highly unlikely that such extreme minimization would ever be used for real!) with Listing 2.4, which shows the same code without any minimization.

The code shown in these two listings is available, without line numbers, on this book's file download Web page. Go to http://www.mcp.com/ and click the Product Support link. On the Product Support page, enter this book's ISBN (1-57521-396-6) in the space provided under the heading Book Information and click the Search button.



Listing 2.3 Tag Minimization

```
1:  <p>SGML uses markup to identify the
2:  <em/logical/ structure of a document
3:  rather than its <em/physical/ appearance.
4:  Tags can be minimized using
5:  <it/tag omission/,
```

```
6:  <it/short tags/,
7:  <it/ranked elements/ and,
6:  <it/data tags/.
7:  <p>and these can <b<em/all/</> be
8:  used at the same time.</p>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

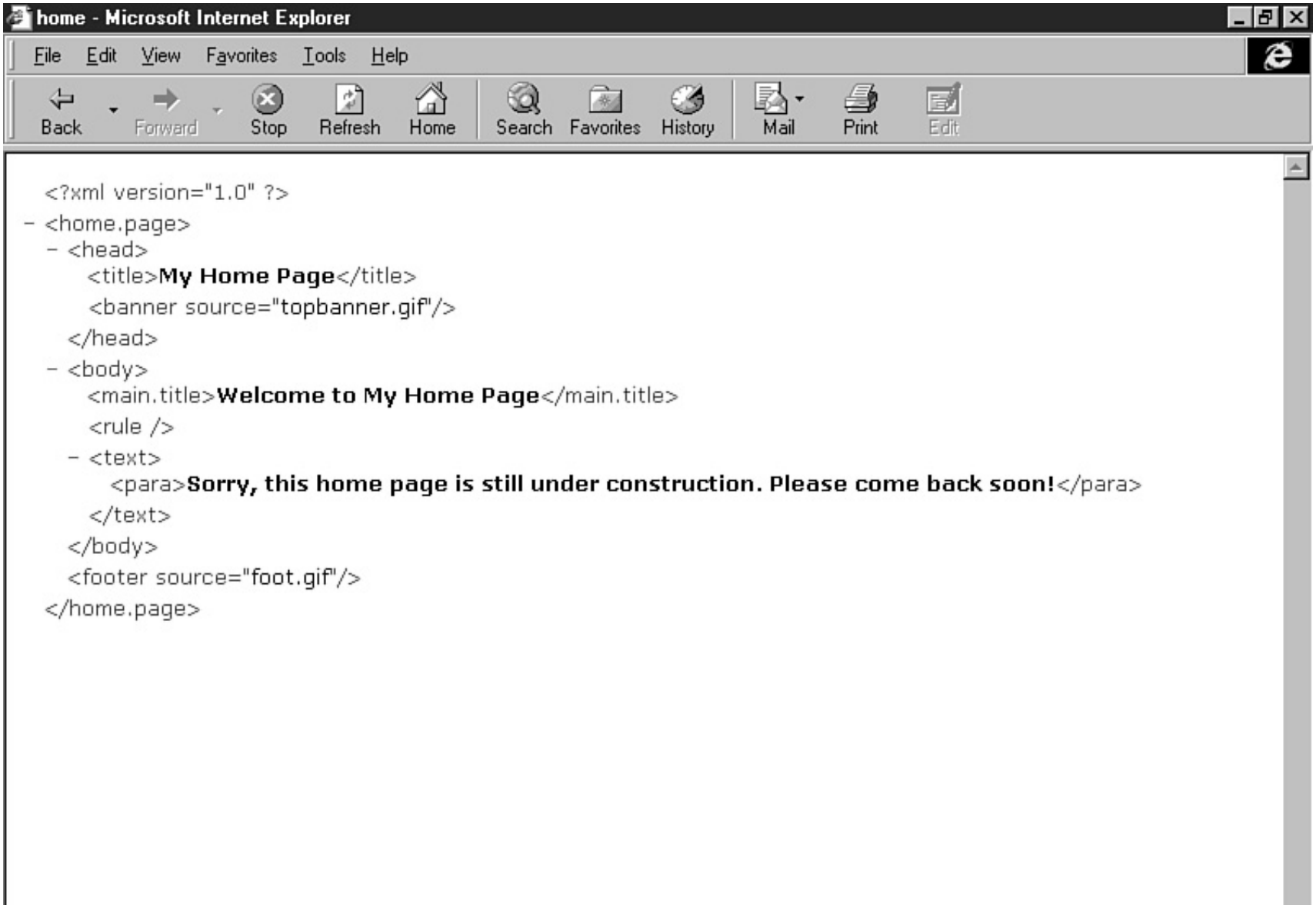
Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Listing 2.4 The Unminimized Version

```
1:  <section number="4"><p>SGML uses markup to identify the
2:  <em>logical</em> structure of a document
3:  rather than its <em>physical</em> appearance.
4:  Tags can be minimized using</p>
5:  <list type="unordered"><li><it>tag omission</it>,</li>
6:  <li><it>short tags</it>,</li>
7:  <li><it>ranked elements</it>, and</li>
6:  <li><it>data tags</it>.</li></list>
7:  <p>and these can <bold><em>all</em></bold> be
8:  used at the same time.</p></section>
```

Obviously, you need some pretty high-powered software to take advantage of these advanced features, not to mention the time and effort it takes to learn to write code like this. Also, considering the amount of divergence between SGML applications that this represents, even discounting the different elements that would be found in the separate applications, it's unlikely that SGML code from one application could be used in another application. It would be similar to converting from WordPerfect to Microsoft RTF, except that you'd be completely on your own.

In the face of this complexity, HTML takes a much simpler, more practical approach. Take a look at Listing 2.5, which shows some HTML that is intentionally full of "mistakes."



Listing 2.5 Bad HTML Code


```
1: This is a pseudo HTML file. It has a less than (<) &amp;
2: a greater than (>) symbol in it but they aren’t &hf;
3: <em>really</em> used as markup at all.
4: <p>
5: <P>All the browser really does is look for the tags
6: it recognizes, so that if it finds a tag that it isn’t
7: setup to interpret like <point><font color=”red”>
8: this</font></point> one, it will simply ignore it.
```

If you enter this text and view it in a Web browser, you might see something like the display shown in Figure 2.1. (There is really no guarantee that your particular browser will show something that looks anything like this!)

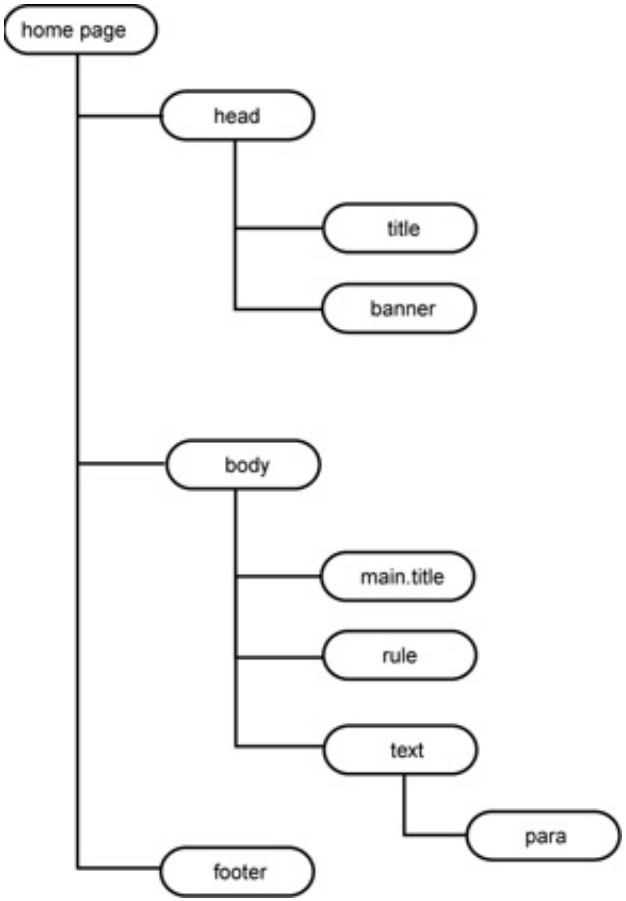


Figure 2.1 *This is how Netscape displays the bad code from Listing 2.5.*

As you can see from Figure 2.1, the Web browser tries to make sense of the markup it sees by comparing it to the tags that it *expects* to see. If there is a match, you get the prescribed appearance. If there is no match but the markup was meant to be interpreted as a special character or as a range of text that required special treatment (such as the <point> tag), the browser ignores it. If the markup was possibly not markup at all (the freestanding < and > symbols), the browser treats them as character data and displays them as if they were normal characters.

SGML and HTML almost represent two extremes. There is a very vague continuum between content-oriented markup and presentation-based markup. (We are getting a little ahead of ourselves here because this delves into DTD development, which is covered much later in the book.) SGML's markup is generic. It is primarily intended to be *content-oriented*—it says nothing about how something should be displayed, but rather identifies the nature or purpose of pieces of text. It *can* be used for either or both.

There are a lot of names for the various types of markup, and little agreement about how to use those names correctly. The most technically accurate term for content-oriented markup is *semantic markup*, in which the names of elements describe what the elements are rather than what they do, what they are used for, or how they are to be processed. Digging deeper, however, formatting instructions can be considered a form of semantics, and so I shall keep to the possibly less accurate but more neutral term *content-oriented.*

HTML's markup, however, is a use of SGML that is purely presentation-based and really has little to do with the information contained in its documents. This isn't really HTML's fault, but rather a result of the loose way it is used. For example, if a rigid hierarchy was applied to the order in which levels of headings were used, so that an H2 always came after an H1 and they were all properly nested within each other, HTML could be quite content-oriented. If you went as far as adding classes and types to all the HTML elements in a file, you could actually turn it into a very rich piece of content-oriented markup. However, the real world is not so perfect, and any possible semantic value to be gained from organizing heading levels would be negated by their willful and random use solely on the grounds of their font size on a browser. If you doubt that HTML is purely presentational, however, ask yourself what kind of semantic information the <HR> (horizontal rule) element conveys.

So where does XML fit in? Well, somewhere in the middle. Remembering that XML is meant to be unambiguous and free of options, XML's rules for distinguishing between markup and content are very simple:

1. The start of markup is identified by either the less-than symbol (<) or the ampersand character (&).
2. Three other characters are also treated as markup characters: the greater-than symbol (>), the apostrophe or single quote, (‘), and the (double) quotation mark (“).
3. If you want to use any of the preceding special characters as normal characters, you must “escape” them by using the general entities that represent them. The replacements you should use are shown in Table 2.1.

To *escape* a character means to conceal it from a subsequent software package or process. It is often used in computing terms to refer to prefixing certain characters in programming languages with a special character string to prevent them from being interpreted as special characters. Originally the ESC (escape) character string was used to prefix commands sent to the printer itself to control such things as the font or page size and distinguish the command strings from printable characters.

4. Everything that is not markup is content (character data).

Table 2.1 The Predefined General Entities

Character	***Replacement***
&	&amp;
‘	&apos;
>	&gt;
<	&lt;
“	&quot;

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

The rules for interpretation are somewhat closer to HTML's than SGML's, but unlike HTML, markup characters aren't treated as normal characters if they can't be interpreted sensibly as markup. When these rules are combined with the other rules about the structure of XML documents, which you will learn later, the results are always predictable—even if they aren't always what you want.

What kind of markup do you use XML for? The answer is really quite logical. XML is *extensible*, and it is also a generic markup language, so it can be content-oriented, presentation-oriented, or both. But there's one major difference between how XML and HTML approach physical appearance: The final appearance of the XML code is not determined by the Web browser, but is specified by the attached style.

- Like SGML (and unlike HTML), XML is not a markup language but a language for defining markup languages. Therefore, you can extend any XML markup language as the need arises, rather than having to always use the same fixed elements over and over again as you do with HTML. XML is truly *extensible:* It has no predefined list of elements, and you can name and use the elements according to the needs of the application.

A Sample XML Document

Listing 2.6 shows the XML code for a Web home page. This is a very simple example, but it contains all the important parts that you will find in nearly all XML documents.

As with all other listings in this book, the lines are numbered for ease of identification. The numbers are not part of the XML code, and you should leave them out if you type in this code yourself.

Listing 2.6 XML Code for a Simple WWW Home Page

```
1:  <?xml version=”1.0”?>
2:   <home.page>
3:     <head>
4:       <title>
5:         My Home Page
6:       </title>
7:       <banner source=”topbanner.gif”/>
8:     </head>
9:     <body>
10:       <main.title>
11:         Welcome to My Home Page
12:       </main.title>
13:       <rule/>
14:       <text>
15:         <para>
16:           Sorry, this home page is still
17:           under construction. Please come
18:           back soon!
19:         </para>
20:       </text>
21:     </body>
22:     <footer source=”foot.gif”/>
23:  </home.page>
```

Figure 2.2 shows what the code from Listing 2.6 looks like in Internet Explorer 5.

Figure 2.2 *The XML code for a simple home page (in Internet Explorer 5).*

The current versions of the leading Web browsers (the beta preview, release 2, of Internet Explorer 5 and the August 1998 build of the Mozilla source code) cannot display the XML code shown in Listing 2.6 without it being radically changed. The display in Figure 2.2 is less than optimal because IE5 needs either an XSL or a CSS style sheet to give the XML something other than the default rendering shown here. An explanation of how to do this is given on Day 12, “Viewing XML in Internet Explorer.”

The XML Declaration (Line 1)

The XML declaration identifies what follows it as XML code. It states which version of the XML standard the code complies with, and it specifies whether the document

can be treated as a standalone document (it can) or whether a DTD must also be retrieved in order to make full sense of the contents.

The XML declaration is in fact a *processing instruction* (identified by the ? at its start and end), but for now it's enough to just treat it as a standard declaration.

The XML declaration is not strictly compulsory (the fact that the document is XML code can also be announced by the Web server in the same way that is often done for HTML documents). However, you should get into the habit of including such a declaration because it increases the portability of your code.

The Root Element (Lines 2 through 23)

Each XML document must have only one root element, and all the other elements must be completely enclosed in that element. Line 2 identifies the start of the <home.page> element (the start tag), and line 23 identifies the end of the element (the end tag).

Note that unlike HTML, in which a <P> tag might often be used as a sort of formatting instruction to insert a blank line between paragraphs of text, in XML an element normally consists of three things: a start tag, content (either text or other elements), and an end tag.

An XML element doesn't always have content. Empty elements, such as the IMG element in HTML that simply points to an external graphics file through its SRC attribute, obviously have no content. An empty element might have an end tag, but it can have a special form of start tag that allows an explicit end tag to be omitted.

Also note that the name you use in the element start tag must exactly match the name you use in the end tag. If you want to use an odd combination of cases to increase the legibility of long names (for example, ThisIsAnIntelligibleName), you must be very careful to exactly match the case usage in both the opening and the closing tag.

XML is case sensitive, recognizing the difference between uppercase letters (A–Z) and lowercase letters (a–z). In applications that aren't case sensitive, mixed-case characters are usually converted—*folded* into one case or the other. The ASCII character set usually folds to uppercase characters. Unicode usually folds to lowercase characters. XML has to account for this, and for the fact that it might have to deal with languages in which the case folding is uncertain. Therefore, XML defaults to lowercase (and the XML declaration also has to be in lowercase). It is a good idea to keep all your markup in lowercase too, even though this can reduce the readability of your code.

An Empty Element (Line 13)

Empty elements are a special case in XML. In SGML and HTML, it is obvious from the DTD's definition of an empty element that it is empty and has no comment. XML, in keeping with its developers' design goals, requires you to be much more explicit. Indeed, you might not use a DTD at all, so it could be hard to decide whether an element is or should be empty. Therefore, empty elements have to be very clearly identified as such. To do so, there is a special empty tag close delimiter, />, as in the

following:

```
<empty_element/>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

To maintain a certain degree of backward-compatibility with SGML (until such time as the SGML standard is updated to allow the use of empty-tag close delimiters), and to make the conversion of existing SGML and HTML code into XML a little easier (a process called *normalization*, which adds end tags to all elements and is supported by a lot of SGML tools), you can use an end tag instead of the special empty tag close delimiter. The element declaration

```
<graphic source="file.gif"/>
```

is therefore interchangeable with

```
<graphic source="file.gif"></graphic>
```

Attributes (Lines 7 and 22)

Element tags can include one or more optional or mandatory attributes that give further information about the elements they delimit.

Attributes can only be specified in the element start tag. The syntax for specifying an attribute is

```
<element.type.name attribute.name="attribute.value">
```

If elements were nouns, attributes would be adjectives. We could therefore say

```
<fruit taste=”sharp”>
```

or even

```
<problem size=”huge” cause=”unknown” solution=”run.away”>
```

An attribute can only be specified in an element start tag.

In direct contrast to SGML and HTML, in which multiple declarations are considered to be fatal errors, XML deals with multiple declarations of attributes in a unique manner. If an element appears once with one set of attributes and then appears again with a different set of attributes, the two sets of attributes are merged. The first declaration of an attribute for a particular element is the only one that counts, and any other declarations are ignored. The XML processor might warn you about the appearance of multiple declarations, but it is not required to do so and processing can continue as normal.

An XML processor is a software package, library, or module that is used to read XML documents. The XML processor makes it possible for an XML application, such as a formatting engine or a viewer, to access the structure and content of an XML document.

Logical Structure

In the earlier discussion of HTML markup, you learned about the conceptual differences in markup between XML (and SGML) and HTML. HTML uses its tags as if they were style switches. The start tag turns a feature on, such as underlining, and an end tag turns it off again. XML uses its start tags and end tags as containers. Together, the start tag, the content, and the end tag all form a single element. Elements are the building bricks out of which an XML document is assembled. Each XML document must have only one root element, and all the other elements must be *perfectly nested* inside that element. This means that if an element contains other elements, those elements must be completely enclosed within that element.

- Let’s look at what this means for the simple example in Listing 2.6. If you sketch out the structure of the elements in this XML document, you’ll obtain the kind of tree structure of elements shown in Figure 2.3.

As you can see from Figure 2.3, the document has a sort of tree-like structure, with the root element (<home.page>) at the top of the tree (or the base, depending on how you look at it). All the elements that are inside this element are neatly contained within each other. An XML document must contain one and only one root element, and there must not be any elements that are either partially or completely outside, before or after, that element.

To make it easier to refer to the relationships between elements and to elements with respect to other elements, you could say that an element is the *parent* of the elements that it contains. The elements that are inside an element are called its *children.* Elements that share the same parent element are called *siblings.*

In the simple example shown in Figure 2.3, <home.page> is the parent of all the other elements, <text> is the parent of <para>, <title> is a child of <head>, and <title> and <banner> are siblings. Going down the element tree, each child element must be fully contained within its parent element. Sibling elements may not overlap.

Figure 2.3 *The logical structure of elements.*

The arrangement of the elements in an XML document is called the *logical structure*. As you will see next, an XML document also has a physical structure. In order to be usable (technically, in order to be *well-formed*), the logical and physical structure of an XML document must be *synchronous*; they must be completely and properly nested inside each other.

Physical Structure

One of the key concepts in XML is that of the *entity*. To really understand XML, you need to understand what entities are. There are various types of entities, and the entities are far more important than the elements in determining how the XML processor deals with the XML code. You will learn about entities in some detail later, but for now it is enough to think of an entity as a physical storage unit. It's an object, although in fact most entities can usually be thought of as being separate computer files.

An *entity* is essentially a unit of information, but in the official terms of the XML language specification, an entity is a *storage object*. This object might be an element or an XML ENTITY object (normally an unparsed external file).

The main entity that you work with all the time, although you will hardly ever even notice that it's there, is the document entity. As you have seen, this document (or root) entity is logically divided into elements. (There are other logical components covered later, but for now it is enough to concentrate on the elements.)

Entities can reference other entities and can cause them to be included in the XML document. Surprisingly enough, you've already seen some entities. The entities listed in Table 2.1 that you used to escape markup characters in normal text are in fact internal entities, but more on those later. For now, let's examine the basic reference to a graphics file that is common in HTML Web pages, and the derivative example in Listing 2.6 (line 7):

```
<banner source="topbanner.gif"/>
```

The banner element's source attribute refers to an external entity (not contained in the current document), which is an external graphics file. If this were HTML code, the graphic would appear on your Web browser at this point in the document. In XML terms, this graphics file is called an *unparsed entity*; the XML processor ignores the content of the entity and passes it on to the application.

XML is a little stricter than HTML about the inclusion of external graphics files. As you will learn later, XML allows you to specify the *notation* or format that the graphic

is in, but it can also include more than just a simple graphic. This is where a lot of problems start!

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

XML can include entities that contain XML code, text, HTML code—almost anything. Depending on how the referenced entity is identified, if it too was a piece of XML code, it could be processed (parsed) by the XML processor as if that XML code had been in the original document (root entity) and not in an external file. To further complicate matters, that XML entity could then reference another entity, and so on to infinity. Apart from the practical problems that this might cause (just imagine trying to open a small document and getting several thousand linked pages!), this creates special problems when the included entities also contain markup.

Just as the arrangement of the elements gives a logical structure, so the arrangement of the entities gives a physical structure. Now, suppose that an included entity also contains elements. This doesn't seem like such a problem on the surface, but it certainly does become a problem if the included entity contains elements with the same element type names as elements in the root (or other) entities, and if it is important to the application to distinguish the differences. This problem is solved by a mechanism called *namespaces* that will be discussed later on Day 7, "Developing Advanced DTDs."

Besides element conflicts, suppose that you had opened an element in the root entity (your XML document) and referenced an external entity. Again, not a problem; it's a normal thing to do. But suppose that the external entity contained an end tag for the element you just opened. Suddenly your whole logical structure would be ruined.

To limit the occurrence of these problems, the logical and physical structures of XML

entities must be synchronous; logical entities cannot cross physical entity boundaries, and physical entities must be fully enclosed (nested) within logical entities. Sometimes it can be hard to work out whether this is the case, and when it *isn't* the case, it can cause a lot of problems. This requirement is unique to XML and for authors coming from either SGML or HTML. It is probably the most difficult point to grasp and the most common source of mistakes.

Summary

This chapter looked at an XML document as an object constructed of various components. You've learned about markup and the distinction that XML makes between markup and character data. You have also learned how XML documents have both a logical and a physical structure, and you've seen the connection between these and XML's elements and entities.

Q&A

- **Q Why is XML case sensitive, whereas SGML and HTML are not?**
  A XML is designed to work with applications that might not be case sensitive and in which the case folding (the conversion to just one case) cannot be predicted. Rather than make dangerous assumptions, XML takes the safest route and opts for case sensitivity.
  Q Why are Web browsers so loose in interpreting markup?
  A This was a conscious decision aimed at making the Web easy to use. Rather than enforce a lot of rules concerning HTML code, the decision was made to accept everything and ignore anything that can't be interpreted.
  Q Why doesn't XML allow tag minimization?
  A XML processors are meant to be simple to create (and consequently inexpensive and fast). This means that they must not be asked to perform complicated processing, such as storing their current context and then looking ahead to see where the current element ends. Therefore, tagging has to be 100 percent explicit, thereby avoiding the processing overhead created by tag minimization.
  Q What purpose do the predefined entities serve?
  A Officially, they give you an easy way to escape XML delimiters. Unofficially, they mean you have no excuse for trying to use the delimiter characters as normal characters. Generally, they make it simpler for an XML processor because the delimiters can always be treated as delimiters and never as normal characters (although there are a few exceptions).
  Q Why must elements be properly nested?
  A Again, this is a question of avoiding processing overhead. If the XML processor has to remember which elements have been opened but not yet closed, it will have to store this information somewhere. This means memory (or disk) storage has to be used, which means more processing, more complexity, and less speed. This is one of many examples in XML in which a bit of (enforced) coding discipline can save a lot of programming effort.

Exercises

1. Mark up the following email message to identify its information content:

```
From: Simon North <north@synopsys.com>
To: Nick <sintac@xs4all.nl>
Subject: Hi
Hi Nick, this is just a quick message
to say I got the material. Thanks.
```

2. Now mark up the same message to identify its appearance:

```
From: Simon North <north@synopsys.com>
To: Nick <sintac@xs4all.nl>
Subject: Hi
Hi Nick, this is just a quick message
to say I got the material. Thanks.
```

3. Compare your two marked-up messages. Give two reasons why one type of markup could be more useful than the other.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 3
Using XML Markup

Yesterday, in “Anatomy of an XML Document,” you learned about the main features of XML markup for elements and entities. Today’s chapter will expand on this, and you’ll also learn about the following:

- Attributes
- Entity references and how to use them as shortcuts
- How to include comments in your code
- What CDATA sections are and how they are used
- Processing instructions

Markup Delimiters

Yesterday you learned about XML’s markup characters in fairly general terms. Now it’s time to get a little more technical and examine the exact details of XML’s markup declarations.

Table 3.1 identifies the parts of XML’s element tags. When the details get a bit more technical, it will be helpful if you’re familiar with these parts. (Although you don’t need to commit them to memory!)

Table 3.1 The Parts of an Element Tag

Symbol	*Description*
<	Start tag open delimiter
</	End tag open delimiter
foo	Element name
>	Tag close delimiter
/>	Empty tag close delimiter

It is worth remembering that, whereas HTML simply relies on recognizing preprogrammed tags, XML is triggered by these specific parts of the element tags, and the XML processor's behavior and what it expects to see next are directly controlled by the named symbols.

Element Markup

XML is concerned with element markup. This might sound like an obvious point to make, but it is worth repeating because it indicates a deeply rooted conceptual difference between XML as a markup language and an arbitrary tag language. As you have already seen, HTML often tends toward being a tag language rather than a markup language. This is a direct consequence of Web browsers being so intentionally lenient in accepting bad markup.

Instead of XML's tags being markers that indicate where a style should change or a new line should begin, XML's element markup is composed of three parts: a start tag, the contents, and the end tag. This is shown in Table 3.2. The start tag and end tag should be treated like wrappers, and when you think of an element, you should have a mental picture of a piece of text with both tags in place.

Table 3.2 The Parts of an Element

Symbol	*Name*	*Description*
<foo>	Start tag	At the start of an element, the opening tag
text	Content	In the middle of an element, its content
</foo>	End tag	At the end of an element, the closing tag

Note that the element name that appears in the start tag must be exactly the same as the name that appears in the end tag. For example, the following would be wrong:

```
<simple.element>This element won't close!</simple.Element>
```

The first versions of XML, before it became a full-blown proposal, were not case sensitive. There are still some XML software packages in circulation that are not case sensitive and will not signal an error if you mix up cases. For conformity with XML requirements, you must be careful to keep your use of upper- and lowercase consistent.

Attribute Markup

As you learned yesterday, attributes are used to attach information to the information contained in an element. The general form for using an attribute is

```
<!element.name property="value">
```

or

```
<!element.name property='value'>
```

The technical description of the markup of this attribute specification is given in Table 3.3.

Table 3.3 Specifying an Attribute

Symbol	*Description*
<	Start tag open delimiter
element.name	Element name
property	Attribute name
=	Value indicator
"	Literal string delimiter
'	Alternative literal string delimiter
value	Value of the attribute
>	Start tag close delimiter

Note that an attribute value must be enclosed in quotation marks. You can use either single quotes (<lie size='big'>) or double quotes (<lie size="massive">), but you cannot mix the two in the same specification.

When you are working without a DTD (none of the XML code shown in today's chapter requires you to associate a DTD with the XML document), you can simply specify the attribute and its value when you use the element for the first time, as shown in Listing 3.1. When you specify attributes for the same element more than once (as in Lines 3 and 4 of Listing 3.1), the specifications are simply merged.

Listing 3.1 Specifying Attributes

```
1: <?xml version="1.0"?>
2: <home.page>
3:  <para number="first">This is the first paragraph.</para>
4:  <para number='second' color="red">This is
5:   the second paragraph.</para>
6:  </home.page>
```

When the XML processor encounters line 3, it will record the fact that a para element has a number attribute. (Remember that this is in the absence of a DTD, which would explicitly declare what attributes a para element has.) Then, when it encounters line 4, it will record the fact that a para element also has a color attribute.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

There is one attribute that is reserved for XML's own use—the xml:lang attribute. This attribute is reserved to identify the human language in which the element was written. The value of the attribute is one of the ISO 639 country codes; some of the most common language codes are shown in Table 3.4.

Table 3.4 Common ISO 639 Language Codes

Code	*Language*
ar	Arabic
ch	Chinese
de	German
en	English
es	Spanish
fr	French
gr	Greek
it	Italian
ja	Japanese
nl	Dutch
pt	Portugese

ru	Russian

When there are several versions of a language, such as British and American English, the language code can be followed by a hyphen (-) and one of the ISO 3166 country codes. Some of the most common country codes are shown in Table 3.5. If you have spent much time on the Internet, you may well recognize these as the same codes that are used in email addresses and URLs. An element written in American English could be identified like this (note the cases; the language code is in lowercase and the country code is in uppercase):

```
<para xml:lang="en-US">My country 'tis of thee.</para>
```

Table 3.5 Common ISO 3166 Country Codes

Code	*Country*
AT	Austria
BE	Belgium
CA	Canada
CN	China
DE	Germany
DK	Denmark
EN	England
ES	Spain
FR	France
GR	Greece
IT	Italy
JA	Japan
NL	The Netherlands
PT	Portugal
RU	Russia
US	United States

The codes given in Tables 3.4 and 3.5 are not complete or exhaustive. There is another coding scheme registered by the Internet Assigned Numbers Authority (IANA), which is defined in RFC 1766. And if you really need to, you can devise your own language code. User-defined codes must be prefixed with the string x-, in which case you could declare an element as being in "computer geek" language like this:

```
<para xml:lang="x-cg">Do you grok this code?</para>
```

Naming Rules

So far you've learned about the markup used for elements and attributes, and all the descriptions mention that these markup objects have names. XML has certain specific rules governing which names you can use for its markup objects.

XML's naming rules are as follows:

- A name consists of at least one letter: a to z, or A to Z.
- If the name consists of more than one character, it may start with an underscore (_) or a colon (:). (Technically, there wouldn't be anything stopping you having an element called <_>, but that would not be very helpful.)
- The initial letter (or underscore) can be followed by one or more letters, digits, hyphens, underscores, full stops, and combining characters, extender characters, and ignorable characters. (These last three classes of characters are taken from the Unicode character set and include some of the special Unicode character symbols and accents. For a full list, refer to the XML recommendation online at http://www.w3.org/XML/REC-xml.)

The World Wide Web Consortium (W3C) regularly reorganizes its Web site, and the URLs for recommendations, notes, and working drafts change quite often. When you visit their Web site, you will find a pointer to the URL Minder service. This free service is one of the many wonders of the Web. By registering a Web page—any Web page—you will automatically be sent an email message if anything on that page changes. This is an excellent way to keep track of any new developments.

Note that spaces and tabs are not allowed in element names (<one two> would be interpreted as two separate names), and the only punctuation signs allowed are the hyphen (-) and full stop (.).

If you spend any time writing code in any other language (even Java or JavaScript), it's easy to get into the habit of using an underscore character (_) to separate long names into sensible chunks, as in: This_is_a_Long_Name. This use of underscores is illegal in XML. You would have to rewrite this as This.is.a.Long.Name.

There is no rule that says your choice of a name needs to make sense. As long as you obey the naming rules, you can call XML objects whatever you like and the names can be as long and meaningless as you like. However, one of the major benefits of using XML in the first place is that it is self-describing. If you have elements such as <thingy>, <whatever>, and <huh>, you're defeating the whole issue. Try to choose names that are at least slightly suggestive of the nature or purpose of the object. Don't forget that one of the XML's aims is to be readable by users. Being readable is one thing, but it also helps if they also make sense.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Comments

No self-respecting language, whether it's a programming language or a markup language, could hold its head up without allowing comments to be added to the code. From a maintenance point of view, it's also pretty important to have a lasting record of why you did particular things. The best way to document your code is to include the explanation with the code by using comments.

In keeping with the design constraint of keeping XML simple, its comment facilities are also simple. Comments have the form

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

> The comment start tag (<!--) and end tag (-->) must be used exactly as they are shown here. Inserting spaces or any other characters into these strings can lead to the tags, or anything inside the comment, mistakenly being interpreted by the XML processor as markup.

Provided that you use the comment start tag and end tag correctly, everything in the comment text will be completely ignored by the XML processor. The following comment is therefore quite safe:

```
<!-- These are the declarations for the <title> and <body> -->
```

There is only one restriction on what you can place in your comment text: the string -- is not allowed. This keeps XML backward-compatible with SGML. (The string --> will obviously end the comment.)

Comments can be placed anywhere in an XML document *outside* other markup. The following is therefore allowed:

```
<para>This is simple <!-- So everyone tells me --> to do.</para>
```

while this is not allowed:

```
<para <!-- blatant lie --> >This is simple to do.</para>
```

Character References

Unlike SGML (and, as a result, unlike HTML), which is very much ASCII-based, XML was developed right from the start to support languages other than English. XML therefore has far better support for accented and foreign language characters than either SGML or HTML.

In HTML, you can always enter the code for the foreign language character you want (è would be è, í would be í, and û would be û). As you will see later in this chapter, these codes are in fact entity references. The abbreviations egrave, iacute, and ucirc are taken from the ISO 8859/1 character set (SGML's character set), which is derived from the ISO/IEC 646 version of the ASCII alphabet (the first 128 characters). ISO 8859/1 is also the basis for the Microsoft Windows fonts.

Although these character entity references will allow you to deal with most European and Scandinavian languages, things would come to a sudden stop if you tried to display or write in an Asian or Middle Eastern language such as Japanese, Hindi, or Arabic. However, XML is based on Unicode and the even more extensive ISO/IEC 10646 standards (which even allow the use of Chinese characters). You needn't concern yourself too much with these character sets now (or not at all if you are only interested in publishing Western languages on the Web), but we will return to this topic later on.

The most important thing you need to know about these exotic characters is that you can still enter them even if your keyboard doesn't support them. You do this by entering a character reference.

A character reference consists of the string &#, followed by the number of the character in the ISO/IEC 10646 alphabet and terminated by a semicolon (;). The character number may be either a decimal number, in which case you enter the number as-is, or in hexadecimal form, in which case you must precede the number with the letter x, such as x12ABC. For example, the character reference for the copyright symbol (©)—written in HTML as ©—is © (in decimal) or © (in hexadecimal).

Predefined Entities

Character references allow you to enter characters that you might not be able to enter from your keyboard. A variation on this theme is the set of *predefined entities.* These are characters that you can enter normally, but you shouldn't because they can easily be mistaken for markup characters. To refresh your memory, the set of predefined entities is shown in Table 3.6.

Table 3.6 The Predefined Entities

Character	***Replacement***

&	& or &
‘	' or '
>	> or >
<	< or <
“	" or "

You can enter a named entity to represent the character, such as ', or you can enter a character reference, such as '. The character references for the ampersand (&) and the less-than (<) character are special cases, however, so the character references are double-escaped. The reasons for this will be explained in the following section.

Entity References

As you remember from yesterday’s discussion of the anatomy of an XML document, entities are normally external objects such as graphics files that are meant to be included in the document. To reference these external entities, you must have a DTD for your XML document. You will learn about these entities when you learn about DTDs, but there is one other type of entity that you can use already, called an *internal entity*. It can save you a lot of unnecessary typing.

Internal entities look very much like character references, but with one vitally important difference—you must declare an internal entity before you can use it.

Entity Declarations

The declaration of an internal entity has this form:

```
<!ENTITY name “replacement text”>
```

Now, every time the string &name; appears in your XML code, the XML processor will automatically replace it with the replacement text (which can be just as long as you like). Judiciously used, entity references can save you a lot of typing.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

The Benefits of Entities

You can almost think of an entity reference as a sort of macro. But whatever you call it, it can be a real time-saver when there is a piece of text that you want to use several times, or even if you want to use some kind of template text.

Consider the example shown in Listing 3.2, in which a copyright statement is used as an entity reference.

Listing 3.2 Using an Internal Entity

```
1:  <?xml version="1.0"?>
2:  <home.page>
3:    <head><title>Title Page</title></head>
4:    <body> <h1>The Title Page</h1>
5:      <para>(c) 1998, &rights;</para>
6:     </body>
7:  </home.page>
```

Given the following declaration for the rights entity:

<!ENTITY rights "All rights reserved. No part of this book, including interior design, cover design, and icons, may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior permission of the publishers.">

This would result in the following substitution being made in line 5 of Listing 3.2:

<para>(c) 1998, All rights reserved. No part of this book, including interior design, cover design, and icons, may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior permission of the publishers.>

Using an entity reference in this way, you would only have to enter the text once, in the entity declaration, instead of having to search for and change every occurrence of the string in the text. Used in this way, entity references can simplify the task of creating and maintaining XML documents. On Day 8, "XML Objects: Exploiting Entities," you will learn how to expand this feature to use external entities as a sort of boilerplate text facility, enabling you to declare these text entities in a common document that can be accessed by any number of other documents.

Some of the Dangers of Using Entities

You've seen how handy internal entity references can be as a sort of shorthand for entering pieces of text, and as a means of dealing with variable content. Obviously, with a little thought and advance preparation, entity references can save you a lot of time and effort later on.

Naturally, a feature this handy raises a very simple question: "Could I use this to insert markup too?" It's an attractive idea and a natural thing to want to do. Can you put markup inside the replacement text? Well, yes you can... but it's subject to a few restrictions, and you need to think it out quite carefully beforehand to avoid some unpleasant surprises.

The first thing you must remember is that XML will process the contents of the entity replacement text when it expands the entity reference. This means that you must not just escape any markup characters in the replacement text; you must *double escape* the characters. Consider this simple example:

```
<!ENTITY dangerous "Black &#38; White">
```

When the XML processor sees the entity reference &dangerous; in the XML document, it will immediately expand (dereference) the predefined entity before it inserts the replacement text. This XML code seems harmless enough:

```
<text>This is not a &dangerous; choice.</text>
```

But let's look at what happens, step by step:

1. The XML processor sees the entity reference &dangerous; and looks for the replacement text.
2. Finding Black & White, the XML processor dereferences this to Black & White.
3. The XML processor inserts the replacement text, and the resulting XML code is

```
<text>This is not a Black & White choice.</text>
```

4. The XML processor then tries to parse the ampersand and reports an error because & has not been declared as an entity.

Avoiding the Pitfalls

You've seen some of the problems that entity references can create when their contents are dereferenced. At worst, they can make a complete mess of your XML code. Of course, there's a way to avoid these problems—double escape any markup contained in the replacement text, like this:

```
<!ENTITY safe "Harry &#38;#38; Fred &amp;amp; Joe">
```

When the XML processor sees the entity reference &safe; in this XML document:

```
<text>The job was left to &safe; to fix.</text>
```

The expansion will still leave you with valid code. Let's see what happens as the XML processor dereferences the entity reference:

1. The XML processor sees the entity reference &safe; and looks for the replacement text.
2. Finding "Harry & Fred & Joe">, the XML processor dereferences this to Harry & Fred & Joe.
3. The XML processor inserts the replacement text, and the resulting XML code is

```
<text>The job was left to Harry &#38; Fred &amp; Joe to
⇒finish.</text>
```

4. The XML processor then parses the resulting code, sees the entity reference &, and dereferences that to produce

```
<text>The job was left to Harry & Fred & Joe to ⇒finish.</text>
```

As you can see from these examples, you can escape the markup by using either the entity reference form (in the

example, &) or the character reference form (&) of the predefined entity.

Synchronous Structures

Other than these problems, there is one very important restriction on using markup in entities. On Day 2, “Anatomy of an XML Document,” you learned that the logical and physical structures in the XML document must be synchronous.

At the time, the restriction might not have made too much sense because it’s difficult to imagine the two structures *not* being synchronous. Well, here’s an example of the two structures becoming asynchronous: The logical structure is composed of the elements in the XML document and in the replacement text. The physical structure is composed of the document entity (the root entity of the XML document containing the entity reference) and the internal entity (the replacement text). The two objects are discrete physical entities as far as XML is concerned, even though in this case they are actually in the same file. For the two structures to be synchronous, any element that is inside the replacement text must start and finish inside the replacement text (in other words, inside the entity).

The following would be allowed:

```
<!ENTITY safe “&#38#60;emph&#62;Harry&#38#60;/emph&#62; and Joe”>
<text>The job was left to &safe; to finish.</text>
```

because the dereferenced entity reference would yield this:

```
<text>The job was left to <emph>Harry</emph> and Joe to finish.</text>
```

The following could create a lot of problems, however:

```
<!ENTITY unsafe “”&#38#60;emph&#62;Harry and Joe”>
<text>The job was left to &safe;</emph> to finish.</text>
```

even though, when the entity reference has been dereferenced, the resulting markup would actually be quite legal:

```
<text>The job was left to <emph>Harry and Joe</emph> to finish.</text>
```

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Previous | Table of Contents | Next

Although we are still talking about *internal* entities, which are completely within our control, the restriction is really pretty logical. The same dereferencing mechanism applies for external entities as well as internal entities and, bearing in mind that one of XML's design goals is to be used easily on the Web, we have absolutely no control over what is contained in external entities. XML's developers could have made a distinction between internal and external entities, but this would go against two more of XML's basic design goals—simplicity and clarity.

Where to Declare Entities

You have learned what an internal entity reference looks like, and you've seen some of the benefits and drawbacks of using entity references. Before we move on to something else, you still need to learn *where* to put the entity declarations.

Entity references are normally only allowed in the DTD that accompanies the XML document. The declarations of element structures and entities are in fact the only reason for having a DTD at all. You will learn all about DTDs in detail later on; for now, all you need to know is illustrated in Listing 3.3.

Listing 3.3 Declaring an Internal Entity

```
1:  <?xml version="1.0"?>
2:  <!DOCTYPE home.page [
3:    <!ENTITY shortcut "This is the replacement text.">
4:  ]>
5:  <home.page>
```

Line 1 of Listing 3.3 is the now-familiar XML declaration. Line 2 is a document type declaration. This is the line that will later be used to make the association between the XML document and the DTD that describes its structure.

The *document type declaration* is the XML statement that declares what type of XML document follows and identifies the document type definition (DTD), which contains the description of the allowed structure of this type of document. (It is quite easy to confuse these two terms.)

The document type declaration takes this form:

```
<!DOCTYPE name external.pointer [ internal.subset ]>
```

where external.pointer points to a separate file that contains the *external subset* of the DTD. Don't worry too much about this for now; the trick is that you can leave this out and concentrate on the *internal subset* of the DTD. The declaration you will need, then, looks like this:

```
<!DOCTYPE name [ internal.subset ]>
```

In this internal subset you can declare as many elements, attributes, and entities as you like without having an external DTD at all.

As you will discover later, there are all sorts of other tricks you can do with the internal DTD subset. Anything you put in the internal subset takes precedence over anything in an external subset. For example, you can declare a default set of global values for a whole suite of XML documents and then override the global values in an individual XML document when you want to, but that is another story.

Before we leave the subject of DTDs altogether, there is one last thing that you should get into the habit of doing, even if it doesn't make much sense at this point. Although you aren't using an external DTD yet, if and when you do, the name that you give to the document type must be the same as the name of the root element in the XML document. This is shown in Listing 3.3, where the document type name (home.page on line 2) is the same as the root (first) element name (line 5). This isn't a requirement when there isn't an external DTD, but it is still a good habit to get into.

CDATA Sections

You have learned how to escape markup characters by using the predefined entities and character references. Replacing every markup character in a piece of text could be a long and tedious process. Besides, there might be cases when you want to keep all those characters exactly as they are (like when you're sending the XML code on for further processing by a different application).

The way to do this is to use a CDATA (character data) section, like this:

```
<![CDATA[This is the text < 5 lines > that I want the &!%# XML processor
to leave alone!]]>
```

Nothing, absolutely nothing, that appears between the opening tag (<![CDATA[) and the closing tag (]]>) will be recognized as markup. You do not need to escape any markup characters in a CDATA section. (In fact, you can't anyway because the escape itself won't be recognized.) The only thing that will be recognized is the end-of-section tag (]]>), so obviously this string cannot be included in a CDATA section. As a logical consequence of this, you cannot put one CDATA section inside another.

Using markup characters in a CDATA section like this in an XML document, which is built around markup, rather goes against the grain. An XML processor is intended to prevent you from breaking this unwritten rule, and it's very unforgiving of any mistakes. The opening string and closing string of a CDATA section must be used exactly as it is shown here. The slightest deviation, a tab or a space character somewhere inside one of the strings, will be punished immediately. The content of the CDATA section will either be treated as markup, or the rest of your document (up to the next CDATA section that is closed properly) will be treated as part of the CDATA section and all the markup will be ignored. You have been warned!

CDATA sections are one of the recommended ways to embed application code (JavaScript, VBasic code, Perl code, and so on) in your XML code. You could place the embedded code in comments, as is often done in HTML documents, but the XML processor is not required to pass the comment text to an

application. Therefore, there's always the risk that the contents of comments will be stripped out before the application sees them.

Even though it is quite legal to declare your own type of element to contain the embedded code (like the <script> element in HTML 4), you'd be implicitly breaking the spirit of generic markup. Nor would it prove to be much help if your embedded code contained characters that could be interpreted as markup, because the contents of these elements would be parsed in the normal way by the XML processor.

The other way to embed code, and probably the best way, is by using processing instructions.

Processing Instructions

Probably without even noticing it, you have already seen processing instructions. The XML declaration at the start of every XML document (or at least it *should* be there) is a processing instruction:

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

XML markup is meant to be generic, and in a perfect world it would be. However, there will always be times when you need to enter instructions for specific applications. One of these applications could be a script interpreter, and so, like CDATA sections, processing instructions are good places to put embedded code. While CDATA sections are purely a way of avoiding characters being interpreted as markup, better still, processing instructions can be targeted to your application. For example, this would allow you to have two or more sets of embedded script code, intended for different processors or interpreters, and identify them separately, as shown in Listing 3.4.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Previous | Table of Contents | Next

Listing 3.4 Embedding Code in Processing Instructions

```
1:  <para>This is text containing two
2:  processing instructions,
3:    <?javascript I can put whatever I like here?>
4:    <?perl And I can put whatever I like here too?>
5:  one for each interpreter.</para>
```

There are no restrictions on the content of the processing instructions (the XML processor doesn't even consider the content to be part of the document's character data), but the name that you choose must comply with XML's naming rules.

Summary

In this chapter you learned the details of XML's markup language. You also learned how to declare and use internal entities, as well as some of the benefits and dangers of using them. You were introduced to character references for entering characters not available on your keyboard, and you saw how you can use the characters that are normally reserved for markup in your character data by using character references and the predefined entities.

To conclude, you learned how to use comments and CDATA sections to hide text that could be interpreted as markup by the XML processor, and how you can extend this by using processing instructions to pass code through for processing by other applications.

Q&A

Q Which of these element names is valid and which is not?

a) **<para 1>**
b) **<para,1>**
c) **<para.1>**
d) **<Pa3A1>**
e) **<para!>**

A Only c and d are legal; a contains a space, b contains a comma, and e contains an exclamation mark.

Q What is wrong with the following code fragment?

```
<para size="12pt">'twas brillig and
the slithey toves <!-- I've no idea
what these are --> did gyre and gymble
in the wabe.</para>
```

A Comments may not be placed inside elements. They must be outside other markup.

Q Where do you declare entities?

A You can declare entities inside either the internal subset or the external subset of the DTD. If you have an external DTD, you will have to create a complete DTD. If you only need the entities and nothing else, you can get away with an internal DTD subset. Entity references in XML documents that have external DTD subsets are only replaced when the document is validated.

Q Why do I need an XML declaration? It should be obvious that this is XML code.

A Strictly speaking, you do not need an XML declaration. XML has also been approved as a MIME type, which means that if you add the correct MIME header (xml/text or xml/application), a Web server can explicitly identify the data that follows as being an XML document, regardless of what the document itself says. (MIME, or Multipurpose Internet Mail Extensions, is an Internet standard for the transmission of data of any type via electronic mail. It defines the way messages are formatted and constructed, can indicate the type and nature of the contents of a message, and preserves international character set information. MIME types are used by Web servers to identify the data contained in a response to a retrieval request.)

The XML declaration is not compulsory for practical reasons; SGML and HTML code can often be converted easily into perfect XML code (if it isn't already). If the XML declaration was compulsory, this wouldn't be possible.

Q Can I use entities in attribute values as well as in content? This would allow me to parameterize elements.

A Yes and no. You can use entity references in attribute values, but an entity cannot be the attribute value. There are strict rules on where entities can be used and when they are recognized. Sometimes they are only recognized when the XML document is validated. For details, see the XML recommendation itself (http://www.w3.org/XML/REC-xml).

Q Can I put binary data in a CDATA section?

A Technically there's nothing stopping you, even though it's really a character data section. Because the XML processor doesn't consider the contents of a CDATA section to be part of the document's character data, it will never know or care what you put in there. However, you would have to live with the increase in file size and all the transportation problems that would imply. Ultimately, it would be a shame to jeopardize the portability of your XML documents when there is a far more suitable feature of XML you can use for this purpose. Entities, which you learn about on Day 8, allow you to declare a format and a helper application for processing a binary file (possibly displaying it) and associate it with an XML document by reference.

Exercises

1. There are two mistakes in the following fragment of code. What are they?

```
<![CDATA [This is the hidden &markup!] ]>
```

You can check your answers by running the code through one of the XML parsers, as explained on Day 5, "Checking Well-formedness."

2. Yesterday you marked up an email message. Using the appropriate entities, change the markup to turn the XML code into a boilerplate for email messages to anyone.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 4
Working with Elements and Attributes

Yesterday you learned the basics of XML specifications and how to use them to mark up elements and attributes in an XML document. Then, using the internal subset of the DTD without actually having to create an external DTD, you learned how to create internal entity declarations that you can then use in the XML document as a kind of macro.

Today you will extend this knowledge of markup by learning about the following:

- How to add element and attribute declarations to the internal DTD subset
- Element content models
- The basics of document modeling
- The principles of well-formedness

Armed with this knowledge, you'll take your first steps into the world of information modeling. There, using the element content models you declare in XML, you can control and check the contents of XML documents that conform to the DTDs that you will create.

By the end of today, you should be able to create quite complex XML documents and ensure that they are well-formed. You will then be ready to move on to Day 5, "Checking Well-formedness," where you will check whether you have put everything you've learned to good use and created well-formed XML documents.

Markup Declarations

Before we get into the details of actually declaring elements and attributes, let's quickly review where these declarations are made in the XML document. Listing 4.1 shows the basic skeleton for declaring an internal DTD subset in an XML document.

Listing 4.1 The Declaration of an Internal DTD Subset

```
1:  <?xml version="1.0"?>
2:  <!DOCTYPE page [
3:    <!-- this is where the internal DTD subset is located. -->
4:  ]>
5:  <page>
6:    <!-- this is the content of the (only) element -->
7:  </page>
```

As shown in Listing 4.1, the XML document begins with the XML declaration (line 1). At this stage you are still working without an external DTD, so the declaration as shown is sufficient.

Line 2 is the start of the DOCTYPE declaration, which ends on line 3. All the markup declarations for the XML document are therefore entered between the square brackets ([]).

Although the full syntax is somewhat more complex than described here (you will learn all about the full syntax when you need it), when it's used with an internal DTD subset only, the syntax takes this form:

```
<!DOCTYPE document.type.name [ internal.subset ]>
```

where the document type name is the same as the name of the XML document's root element (<page> in Listing 4.1).

> At this stage it's not necessary for the document type name to be the same as the root element name, but it will become a requirement later on when you validate your XML documents. It's a good idea to get into the habit of using this naming scheme all the time because it can save some unnecessary reworking.

Element Declarations

The first kind of declaration that you will use inside a DTD, whether it's an internal or external subset, is the element declaration. This takes the following form:

```
<!ELEMENT name content>
```

The name is a standard XML name, constructed in accordance with the naming rules you learned yesterday.

The content part of the element declaration either describes a specific content in the form of the keyword EMPTY or ANY, or it consists of a content model that describes the sequence and repetition of elements that are contained inside (are children of) this element.

Before we take a look at these special sorts of element declarations, let's look at the very simple example of an email message, as shown in Listing 4.2.

Listing 4.2 A Very Simple Email Message in XML

```
1:  <message>
2:    <header>
3:      <date>14 May 1998</date>
4:      <From>Me</From>
5:      <To>You</To>
6:      <Subject>Test Message</Subject>
7:     </header>
8:     <Body> ...
9:     </Body>
10:   <Sig>Some smart saying
11:   </Sig>
12: </message>
```

You haven't learned enough yet to declare the content of the elements right down to the level of the actual text. But given what you have learned, you should still be able to sketch out a skeleton set of content models, as shown in Listing 4.3.

Listing 4.3 A Partial DTD for the Email Message

```
1: <!ELEMENT message (Header, Body, Sig) >
2: <!ELEMENT Header (Date, From, To, Subject) >
```

Empty Elements

As you learned yesterday, empty elements have no content (they are forbidden to have any) and are marked up as either

```
<empty.element/>
```

or

```
<empty.element></empty.element>
```

An empty element is simply declared like this:

```
<!ELEMENT empty.element EMPTY>
```

You could use empty elements for things like external graphics (where, as you will learn later, you declare the name of the external file by using it as an attribute value).

Unrestricted Elements

The opposite of an empty element is an unrestricted element, which can contain any element that is declared elsewhere in the XML document's DTD (in either the internal or external DTD subset). You aren't using an external DTD subset at this point, so obviously there is no way the XML processor can know about any elements declared in one.

An unrestricted element's content is declared like this:

```
<!ELEMENT any.element ANY>
```

Obviously, you cannot declare that the contents should be in any order.

Element Content Models

Empty elements and unrestricted elements are special types of elements that are so straightforward that they more or less speak for themselves; they either allow everything or they allow nothing.

Of much more interest to you are elements that contain other elements. Using XML's element declarations, you can precisely specify which other elements are allowed inside an element, how often they may appear, and in what order. You do this by specifying an element content model.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

> As you will learn later, you don't just *want* to be specific—you *must* be specific. One of the things that an XML processor (or an SGML parser, for that matter) cannot deal with is ambiguity. Your content models must be capable of being interpreted in only one way. This takes a bit of practice and can cause you a lot of head-scratching, especially in the beginning. You will learn all about ambiguous content models in due course.

An element content model consists of a very simple but very specific description of the elements that may appear in the current element, the order in which they may or must appear, and how often they may or must appear. More than this, though, you can arrange elements into groups and specify special container elements. In fact, you can organize the elements into a whole class of XML documents (the document type) to reflect the significance of and relationships between chunks of information.

Element Sequences

The simplest form of element content model consists of a list of the possible elements, enclosed in parentheses and separated by commas:

```
<!ELEMENT counting (first, second, third, fourth)>
```

This example says that a counting element must consist of a first element, followed by a second element, followed by a third element, and ending with a fourth element.

In this example, all four elements must be present in a counting element, and each one may only be present once. (You can specify how often an element may appear by using an occurrence indicator, which you will learn about later in this chapter.)

Element Choices

A choice of elements in an element content model is indicated by a vertical line (|) between the alternatives:

```
<!ELEMENT choose (this.one | that.one)>
```

In this example, a choose element consists of either a this.one element or a that.one element. When use these

elements in an XML document, you can write this:

```
<choose><this.one>I chose this one</this.one></choose> and
then <choose><that.one>I chose that one</that.one></choose>
```

- Notice once again that without an occurrence indicator, the chosen element can appear only once.

Also note that only one element can be selected, no matter how long the list of alternatives is:

```
<!ELEMENT choose (this.one | that.one | the.other.one | another.one |
⇒no.that.one.silly)>
```

Combined Sequences and Choices

You can combine content sequence and choices by grouping the element content into model groups. For example:

```
<!ELEMENT lots.of.choice (maybe | could.be), (this.one, that.one)>
```

Here, a lots.of.choice element can consist of either a maybe element or a could.be element, followed by one this.one element and then one that.one element.

Ambiguous Content Models

You can combine sequences and choices in element content models, but be very careful. Although it is not an XML requirement (or at least not if you aren't going to validate your documents), you can create compatibility problems if your content models can be interpreted in more than one way.

- Consider this possibility:

```
<!ELEMENT confused ((this.one, that.one) | (this.one, the.other.one))>
```

- When the XML processor validates the XML document (checks its content to see if the elements are in an allowed order), it won't be able to decide what is allowed and what isn't. Having seen a this.one element, it's impossible for it to work out which element is supposed to come next.

Of course, the XML processor could read further and then check to see if what does occur is allowed, but XML processors are not meant to look ahead. They're meant to be simple and fast. For the processor to look ahead, it has to save what it has seen in memory, look ahead, read in the next part, save that in memory, compare the two memory contents, and then decide. All this takes extra processing time.

- With careful consideration, you can avoid ambiguous content models with a little rewriting:

```
<!ELEMENT unconfused ( this.one, ( that.one | the.other.one ) ) >
```

Generally, anytime you start combine these two operators, you should be on the lookout for ambiguities.

- Consider another example:

```
<!ELEMENT confused.again ( this.one, that.one , the.other.one |
⇒no.that.one ) >
```

- This could easily lead you (and the XML processor) to believe that the no.that.one element is an alternative to all of the other elements. Or is it just an alternative to the the.other.one element?

- Again, some rewriting can resolve the ambiguity and make your intention a bit clearer:

```
<!ELEMENT explained ( this.one, that.one , ( the.other.one | no.that.one )
⇒) >
```

You might have noticed that one of my ambiguous element content models wasn't really *completely* ambiguous; the XML processor would have been able to make sense of it. However, you may remember that one of XML's design goals is to be reasonably easy for human beings to understand without needing the assistance of software. It is an excellent idea to extend this same principle to XML DTDs as well. If you can, try to arrange your content models into groups so that it is easier to understand what they mean. After all, you have no idea how much time might pass before you will see the DTD again, but by which time you will probably have forgotten all of the wonderful reasoning that went into the design of your now-cryptic element content models!

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Element Occurrence Indicators

By using an element occurrence indicator, you can specify how often an element or group of elements may appear in an element. There are three occurrence indicators (without an occurrence indicator, the element or group of elements must appear just once):

- The ? character indicates that the element or group of elements may be omitted or may occur just once. Consider this content model:

```
<!ELEMENT testing (one, two?, three)>
```

This would allow you to have

```
<testing><one>tock</one><two>tock</two><three>tock</three></testing>
```

or

```
<testing><one>tock</one><three>tock</three></testing>
```

in your XML document.

- The * character indicates that an element or group of elements may be omitted or may appear zero or more times. Consider this content model:

```
<!ELEMENT nice (mmm, mmmm*)>
```

This would allow you to have

```
<nice><mmm>I can't complain.</mmm></nice>
```

or

```
<nice><mmm>I like this one.</mmm><mmmm>More, </mmmm><mmmm>more, </mmmm>

<mmmm>more, </mmmm><mmmm>more, <mmmm>more.</mmmm></testing>
```

in your XML document.

- The + character indicates that an element or group of elements must appear at least once and may appear one or more times. Consider this content model:

```
<!ELEMENT funny (ha, haha+)>
```

This would allow you to have

```
<funny><ha>Who?</ha><haha>is he?</haha></funny>
```

or

```
<funny><ha>I laughed </ha><haha>until </haha><haha>I </haha>

<haha>thought </haha><haha>I'd <haha>die!</haha></funny>
```

in your XML document.

As you can see, occurrence indicators give you a little control over how often an element or group of elements occur—not at all, once, or an unlimited number of times. This "all or nothing" approach is a little too loose for a lot of possible XML applications.

One of the promises that XML has made is that it might finally be possible to seriously use it to model databases. (This was always a practically unachievable dream in SGML.) However, the syntax of XML element content models is not rich and precise enough to do this kind of modeling. Databases, for example, might need to specify an exact number of occurrences, rather than just 0, 1, or infinite.

Another example that programmers would probably welcome is some kind of conditional content model along the lines of "if element A and then B, choose between C and D; otherwise choose between E and F."

Complex content models like examples simply cannot be expressed in the syntax used in XML DTDs. To address this problem and fulfill the needs of a much wider range of users, a search has begun for alternative ways of describing and declaring XML content models (DCD, XML-Data, and RDF). You will learn about a few of the most important of these schemas on Day 17, "Using Meta-Data to Describe XML Data."

Character Content

There is one more type of element content, and I have been intentionally saving it because it is a little bit special.

When text—and only text—is allowed inside an element, this is identified by the keyword PCDATA (parseable character data) in the content model. To prevent you from confusing this keyword with a normal element name (and to make it impossible for you to use it as a name), the keyword is prefixed by a hash character (#), which is called the reserved name character (RNI).

Consider these element declarations:

```
<!ELEMENT para (title, text)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT text (#PCDATA)>
```

These would allow you to write the following in your XML document:

```
<para><title>My Life</title><text>My life has been
very quiet of late.</text></para>
```

A parseable character data element cannot contain any further markup and the ends of the title and text element start tags is therefore where the markup stops and “normal” text takes over.

Don’t lose sight of the fact that XML’s content models are only concerned with the structure of an XML document and make no attempt to control its content. An element that is totally devoid of data content will still match a #PCDATA content model.

Mixed Content Elements

Elements that can contain text (parseable character data), elements, or both are a real problem sometimes. They are called mixed content models, and they require extra care.

The important point is that it is difficult for an XML processor to distinguish between unintentional PCDATA (spaces, tabs, line endings, and so on) and element content. An accidental space between an end tag and the next start tag could lead to chaos.

To declare mixed content, you use the content model grammar you have learned so far, but you must use it in a particular way. The content model has to take the form of a single set of alternatives, starting with #PCDATA and followed by the element types that can occur in the mixed content, each declared only once. Except when #PCDATA is the only option (as you saw earlier), the * qualifier must follow the closing parenthesis:

```
<!ELEMENT pick (#PCDATA | eeney | meeney | miney | mo)*>
```

Attribute Declarations

In XML, you can declare only one element at a time. (In SGML, and by implication HTML, you can put whole groups of elements in one declaration and give them all the same content model at once.) However, elements can have lots of attributes, which are all declared at once in an attribute declaration list.

An attribute declaration list has the following form:

```
<!ATTLIST element.name attribute.definitions>
```

It is normal practice to keep an element’s attribute declaration list close to the declaration of the element itself, but there is absolutely no requirement to do so. It just makes the DTD easier to understand and maintain.

An attribute declaration list consists of one or more attribute declarations. (For readability, they are often put on separate lines, but there is no requirement to do this.) It does the following for an element:

- It declares the names of allowed attributes.
- It states the type of each attribute.
- It may provide a default value for each attribute.

Each attribute declaration consists of a simple attribute name and an attribute type pair statement in the following form:

```
attribute.name attribute.type
```

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Attribute Types

There are three types of attributes:

- A string attribute, whose value consists of any amount of character data.
- A tokenized attribute, whose value consists of one or more *tokens* that are significant to XML.
- An enumerated attribute, whose value is taken from a list of declared possible values.

String Attribute Types

The values of string types are simple strings of characters. Any attribute that is used in an XML document that does not have a DTD (either an internal or external DTD subset) is automatically treated as a string type attribute.

Here's an example of a string type attribute declaration:

```
<!ATTLIST owner CDATA>
```

You would use it like this:

```
<book owner="Hammersmith Public Library">
```

You can also use an internal entity (in this case it's given the more generic name *general entity*) in the value of a string type attribute:

```
<book owner="&my.local; Public Library">
```

Tokenized Attribute Types

Tokenized attributes are classified according to what their value, or values, can be:

- ID—This attribute serves as an identifier for the element. No two elements can have the same ID

attribute value in the same document. An ID value must comply with the standard XML naming rules. An ID attribute type can be applied to any attribute, but it is standard practice to restrict its use to an attribute that is also called ID to make it easier to find.
Here's an example of an ID type declaration:

```
<!ATTLIST book
          id ID>
```

You would use it like this:

```
<book id=”A51”>
```

- IDREF—This attribute is a pointer to an ID (an ID reference). The value must match the value of an ID type attribute that is declared somewhere in the same document.
- IDREFS—The value of this attribute consists of one or more IDREF type values, separated by spaces.

Here's an example of an IDREFS type declaration:

```
<!ATTLIST book
           authors IDREFS>
```

You would use it like this:

```
<book authors=”A51 A62 B87”>
```

- ENTITY—This attribute is a pointer to an external entity that has been declared in the DTD, in either the external or internal DTD subset. The value of the attribute is the name of the entity, which must consist of name characters. The XML document can no longer be a standalone document if you use external entities. It is a little early to learn about external entities, but we will return to them on Day 8, “XML Objects: Exploiting Entities.”
- ENTITIES—The value of this attribute consists of one or more ENTITY type values, separated by spaces.

ENTITY and ENTITIES type attributes are normally used to refer to things like graphics files and other unparsed data:

```
<!ELEMENT graphic  EMPTY >

<!ATTLIST graphic   boardno   ENTITY >
```

- NMTOKEN—The value of this attribute is a name token string consisting of any mixture of name characters.
- NMTOKENS—The value of this attribute consists of one or more NMTOKEN type values, separated by spaces.

Enumerated Attribute Types

Enumerated attributes have values that are simply lists of possible values. Each value has to be a valid name token (NMTOKEN). Here is an example:

```
<!ATTLIST paint
          COLOR (RED | YELLOW | GREEN) “RED”>
```

When a list of possible values is prefixed by the keyword NOTATION (which you will learn about on Day 8), the notations listed as possible values must all have been declared already:

```
<!ATTLIST image
          type NOTATION (GIF | JPEG | PNG) “GIF”>
```

When matching an attribute value against the allowed values specified in the attribute definition, the XML processor carries out a non–case-sensitive match for all attributes, except those that are of the CDATA, IDREF, or IDREFS type.

Strictly speaking, there is no such thing as an optional attribute in XML. However, there is a little-known trick using enumerated types that allows you to mimic the behavior of an optional attribute. Consider this attribute declaration list:

<!ATTLIST clever.element

clever.att SORT (A | B | C) “” >

Now, if you don’t explicitly declare this attribute but simply leave it out, as shown here:

<clever.element>This element has no declared

attribute value.</clever.element>

the default value would be assigned to the element, which in this case is empty.

Attribute Default Values

You can add a keyword to the end of an attribute specification to specify the action the XML processor should take when you leave out (or forget) the attribute in a particular start tag.

There are three possible keywords:

- #REQUIRED means that the attribute is required and should have been there. If it’s missing, it makes the document invalid.

Here’s an example of a required declaration:

```
<!ATTLIST book
          author ID #REQUIRED>
```

Normally, ID type attribute values are specified as being required. You will learn later that they must be specified as required if the document is to be validated.

- #IMPLIED means that the XML processor must tell the application that no value was specified. It is then up to the application to decide what it is going to do.

Here’s an example of an implied declaration:

```
<!ATTLIST section
          number #IMPLIED>
```

Implied attribute values are often used for things like section and list item numbering, where the application can calculate the value itself simply by counting. You would also use this implied value for attribute values that you want an element to inherit from its parent.

- If the default value is preceded by the keyword #FIXED, any value that is specified must match the default value or the document will be invalid.

The following are examples of attribute declarations with default values:

```
<!ATTLIST termdef
          id ID #REQUIRED
          name CDATA #IMPLIED>
```

```
<!ATTLIST list
          type (roman | arabic | Roman | Arabic) “roman”>

<!ATTLIST form

          method CDATA #FIXED “POST”>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Well-Formed XML Documents

Elements, attributes, and entities are the three primary building blocks of XML documents. Just having elements is already enough for you to be able to create true XML documents. Using all three objects, you can create quite complex XML documents and fulfil the needs of 90% of the applications for which you would use XML.

For such XML documents to be properly usable—in other words, for an XML processor to process these documents successfully—they must be well-formed. According to the XML standard, a data object is not officially an XML document until it is well-formed. You have already encountered most of the rules that an XML document must obey in order to be well-formed, but let's review them. A document that you can create now using just elements, attributes, and entities is well-formed if:

- It contains one or more elements.
- It has just one element (the document, or root element) that contains all the other elements.
- Its elements (if it contains more than one element) are properly nested inside each other (no element starts in one element and ends in another).
- The names used in its element start tags and end tags match exactly.
- The names of attributes do not appear more than once in the same element start tag.
- The values of its attributes are enclosed in either single or double quotes.

- The values of its attributes do not reference external entities, either directly or indirectly.
- The replacement text for any entity referenced in an attribute value does not contain a < character (it can contain the string <).
- Its entities are declared before they are used.
- None of its entity references contain the name of an unparsed entity.
- Its logical and physical structures are properly nested.

There are many ways to check whether an XML document is well-formed, and many tools to help you do it. At the very simplest, there are some public-domain Perl scripts and some very useful tools (you will learn about some of these tomorrow). Or you could even try loading your XML code into Mozilla, the development version of Netscape 5 (which you will learn more about on Days 13, “Viewing XML in Other Browsers,” and 18, “Styling XML with CSS”).

Summary

Today you learned the nuts-and-bolts details of element declarations and element content models. You have also learned how to declare and use element attributes. Put this together with what you have already learned about processing instructions, the internal DTD subset, and internal entities, and you have enough to get started using XML in applications.

In fact, if you’re not at all interested in including external objects (graphics and referenced data) or validating your XML documents, you could probably skip the rest of this book! There’s a lot more to come, though. Tomorrow you will learn how to use an XML parser to check that what you have created is well-formed, and after that we can really start to get technical with the next step: valid XML documents.

Q&A

Q Why is element grouping a useful tool in declaring element content models?
A They can be an important aid in avoiding ambiguous content models, and, when used intelligently, they can help to make DTDs easier to understand.
Q Why do mixed content models require special attention and care?
A It is essential for the XML processor to be able to distinguish between intentional whitespace, which it will treat as character data, and accidental characters. The order in which the content model is declared and the use of occurrence indicators is therefore very specific and must be followed exactly.
Q Is there any limit to the length of an element or attribute name?
A No. As long as you obey the restrictions on which characters you are allowed, a name can be as long as you want it to be. (In SGML, you have to make special arrangements to allow longer names.)
Q What is so different about an ID attribute value?
A An ID attribute value must be unique in an XML document.
Q What is the difference between a name and a name token?
A Name tokens are a little less restricted than names. Names have to begin with a letter, an underscore, or a colon; name tokens can start with any valid name character.

Q How can you tell if an XML document is well-formed?
A As well as using an XML processor, you can use one of many free utilities written in a variety of languages, including Perl, JavaScript, Java, and C, that can parse an XML document and point out any well-formedness errors.

Exercises

1. A hot topic for discussion in SGML circles (and it will become one in XML circles) is whether to use an element to describe information or to put the information in an attribute instead. Write down two reasons for each side of the argument.
2. The following fragment of XML code is taken from a simple parts catalog. Add an internal DTD subset to it.

```
<?xml version="1.0"?>
<parts>
  <part>
    <name>Widget</name>
    <catalog.number>11037</catalog.number>
    <price.code>4</price.code>
    <quantity>d</quantity>
  </part>
  <part>
    <name>Bolt</name>
    <catalog.number>11497</catalog.number>
    <thread>w</thread>
    <price.code>1</price.code>
    <quantity>100</quantity>
  </part>
  <part>
    <name>Screw</name>
    <catalog.number>10020</catalog.number>
    <type>countersunk</type>
    <price.code>7</price.code>
    <quantity>c</quantity>
  </part>

</parts>
```

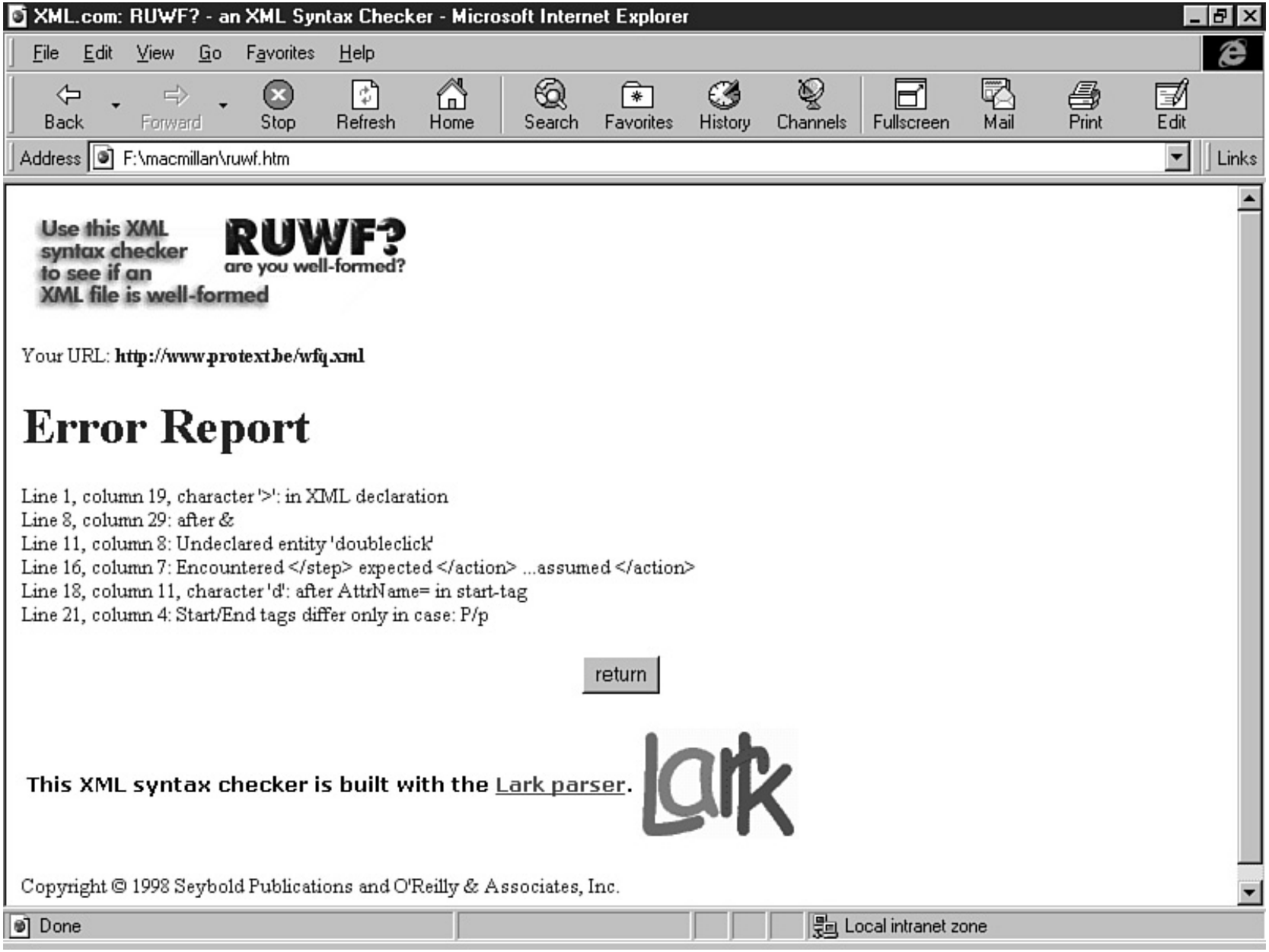
3. The following content models are ambiguous. Rewrite them so they aren't.
- **a.** (section?, section)
- **b.** ((one, two) | (one, three))

To access the contents, click the chapter and section titles.

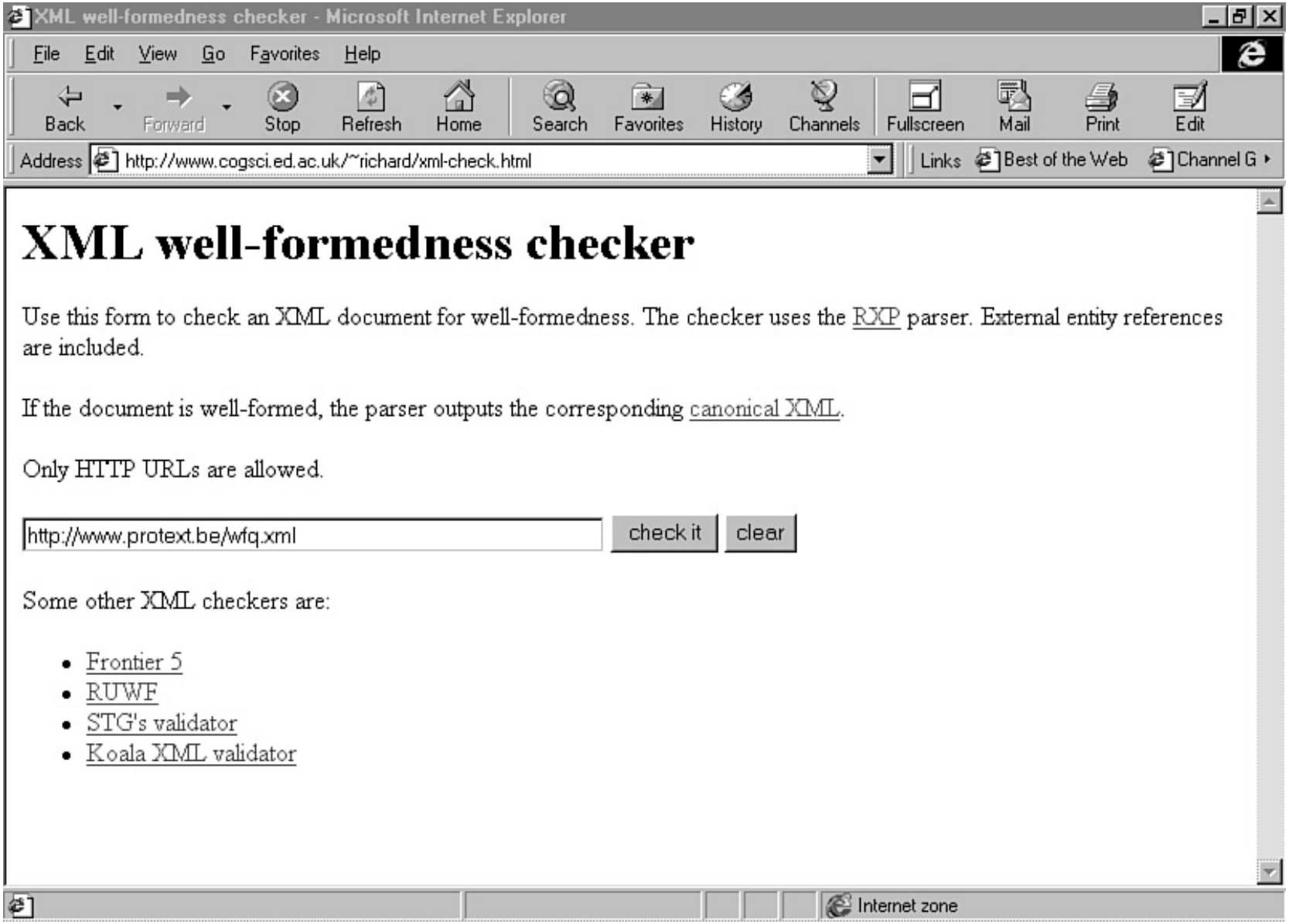
Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 5
Checking Well-formedness

During the previous days, you have learned how to write well-formed XML. But how can you be sure that your XML files are indeed well-formed? Well, some help is at hand. There’s now software whose main purpose is to check the syntax of your files according to the well-formedness rules specified in the XML recommendation. These programs are called non-validating parsers.



Validating parsers are discussed on Day 9, "Checking Validity," when you have acquired knowledge about document type definitions.



In addition to checking your XML files, XML processors give other applications access to the structure and content of your XML files.

In this chapter you will learn the following:

- Where you can find these XML parsers
- How to use the expat parser
- How to use the DXP parser from the DataChannel company
- How to check your XML files over the Web using RUWF
- How to check your XML files over the Web using other validation services

Where to Find Information on Available Parsers

Lists of available parsers can be found at the following URLs:

- Robin Cover manages the most complete reference site on SGML and XML. If you want to know anything about XML or SGML, start at http://www.oasis-open.org/cover/.
- Lars Marius Garshol maintains http://www.stud.ifi.uio.no/~larsga/linker/XMLtools.html#C_Parsers. Here you'll find a good overview of free XML tools, with lots of clarifying comments and all necessary pointers.
- Lisa Rein's http://www.finetuning.com/parse.html is another good resource page.
- James Tauber's http://www.xmlsoftware.com/parsers/ offers the clearest overview. He lists the parsers according to the programming language used, the interfaces they offer, if they are validating or not, and so on.

Checking Your XML Files with expat

In this section you'll learn how to use the expat XML parser program to check your XML files. The expat program was written by James Clark, the technical lead of the W3C XML Working Group (WG) and author of other acclaimed free software for the SGML/XML community. The code is platform-independent C.

Mozilla 5, the next release of Netscape's browser, will use the expat parser. For more info, see http:/www.mozilla.org/rdf/doc/xml.html.

For Perl, there's an extension module named XML::Parser that is a Perl interface to expat. More information can be found at http://www.netheaven.com/~coopercc/xmlparser/intro.html.

Installing expat

expat is available for download from ftp://ftp.jclark.com/pub/xml/expat.zip. This ZIP file is 138KB large (version 19981122). Just unzip this distribution to your disk and you're ready to go.

Using expat

What's of interest to us is the xmlwf application. The executable is located inside the bin subdirectory of the expat distribution. This application takes as arguments one or more files to be checked for well-formedness.

The typical usage is as follows:

```
xmlwf file
```

For example:

```
xmlwf c:\xmlfiles\parse1.xml
```

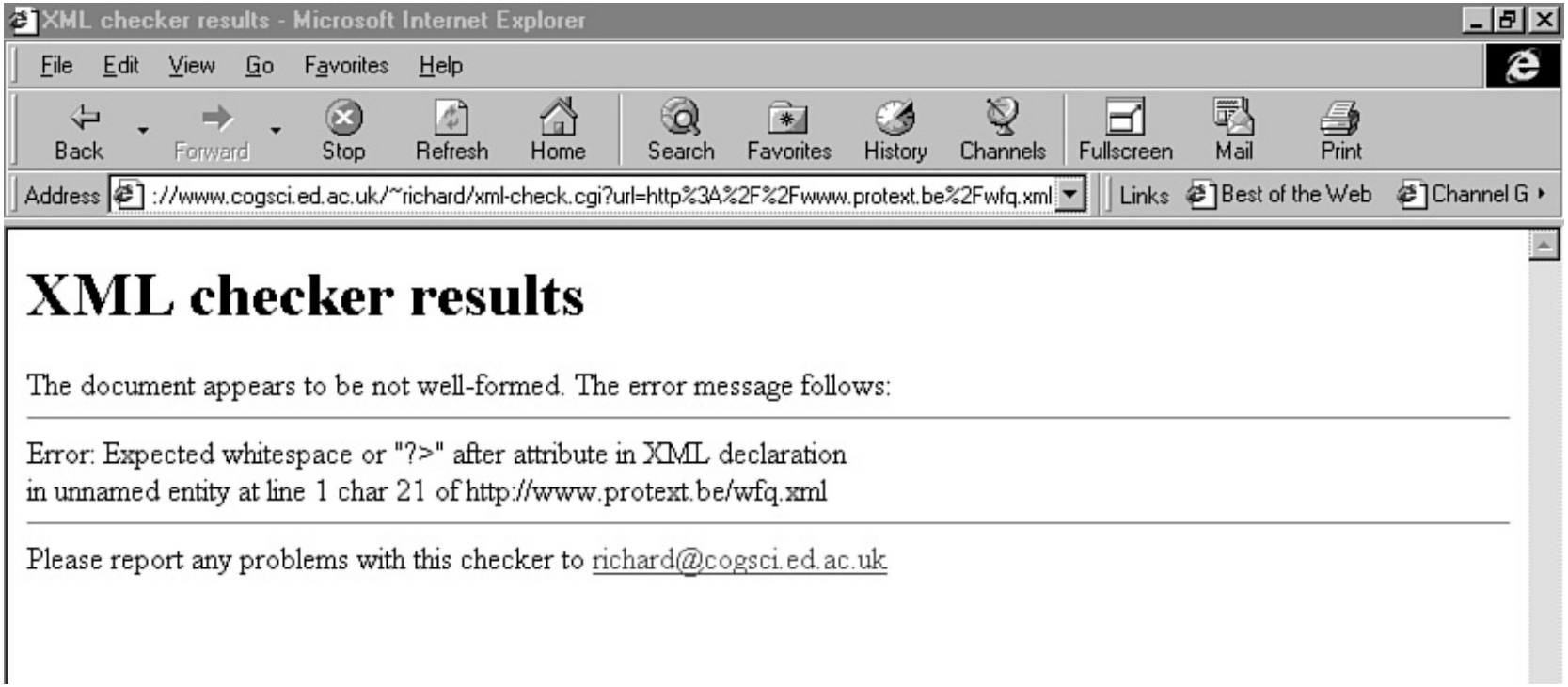
Checking a File Error by Error

Let’s go over a file step-by-step and see in detail which errors expat detects. We’ll use the XML file shown in Listing 5.1.

Listing 5.1 wfq.xml—Checking for Syntax Errors

```
1: <?xml version=“1.0”>
2: <?protext objid=“I5678” ?>
3: <helptopic>
4: <title keyword=“printing,network;printing,shared printer”>How
⇒to use a shared network printer?</title>
5: <procedure>
6: <step><action>In <icon>Network Neighborhood</icon>, locate and
⇒double-click the computer where the printer you want to use is
⇒located. </action>
7: <tip targetgroup=“beginners”>To see which computers have
⇒shared printers attached, click the <menu>View</menu> menu,
8: click <menu>Details</menu>, & look for printer names or
⇒descriptions in the Comment column of the Network
⇒Neighborhood window.</tip>
9: </step>
10: <step>
11: <action>&doubleclick; the printer icon in the window that
⇒appears.</action>
12: </step>
13: <step>
14: <action>
15: To set up the printer, <xref linkend=“id45”>follow the
⇒instructions</xref> on the screen.
16: </step>
17: </procedure>
18: <rule form=double>
19: <tip>
20: <P>After you have set up a network printer, you can use it as
⇒if it were attached to your computer. For related topics, look up
⇒&quot;printing&quot; in the Help Index.
21: </p>
22: </tip>
23: </helptopic>
```

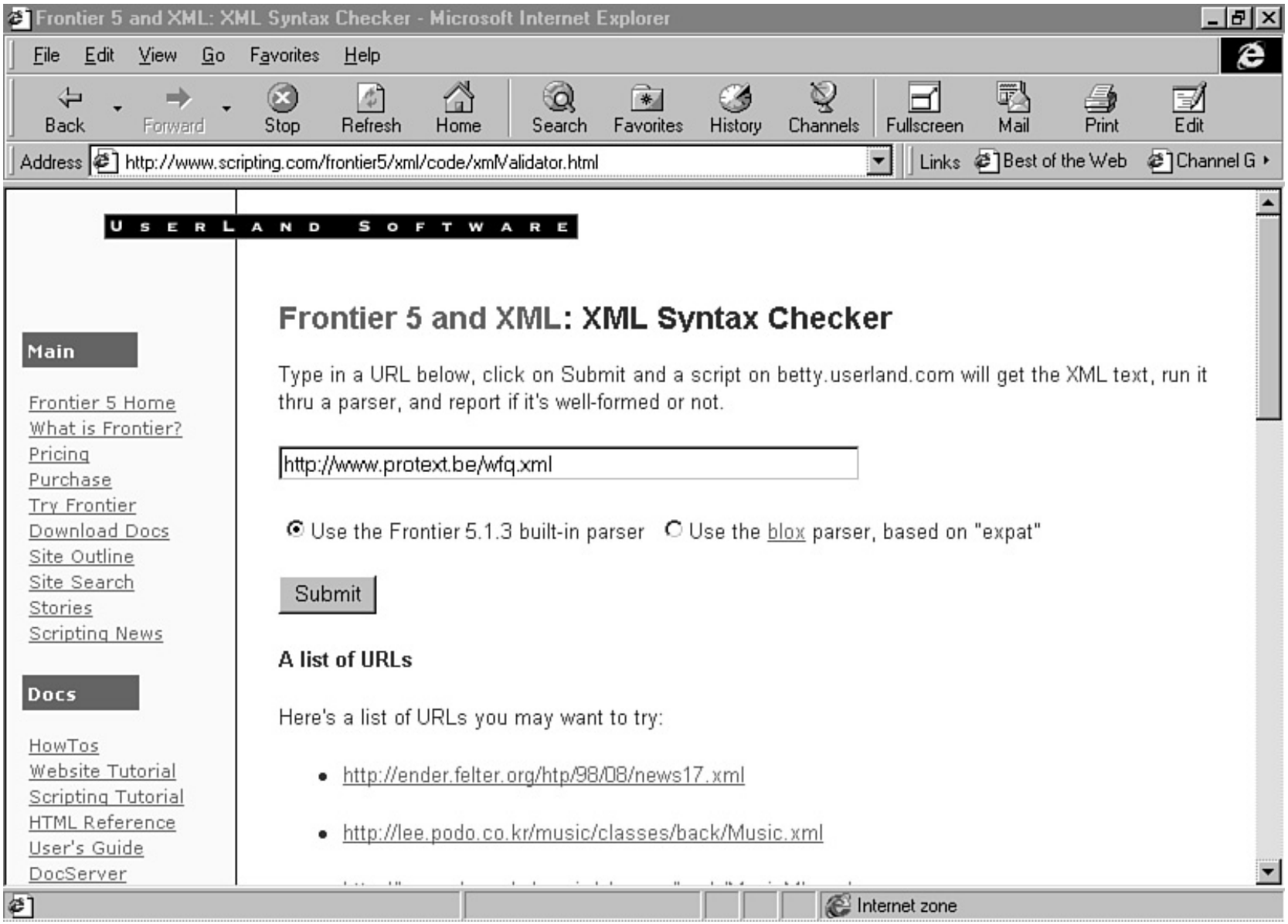
Let’s see what expat discovers.

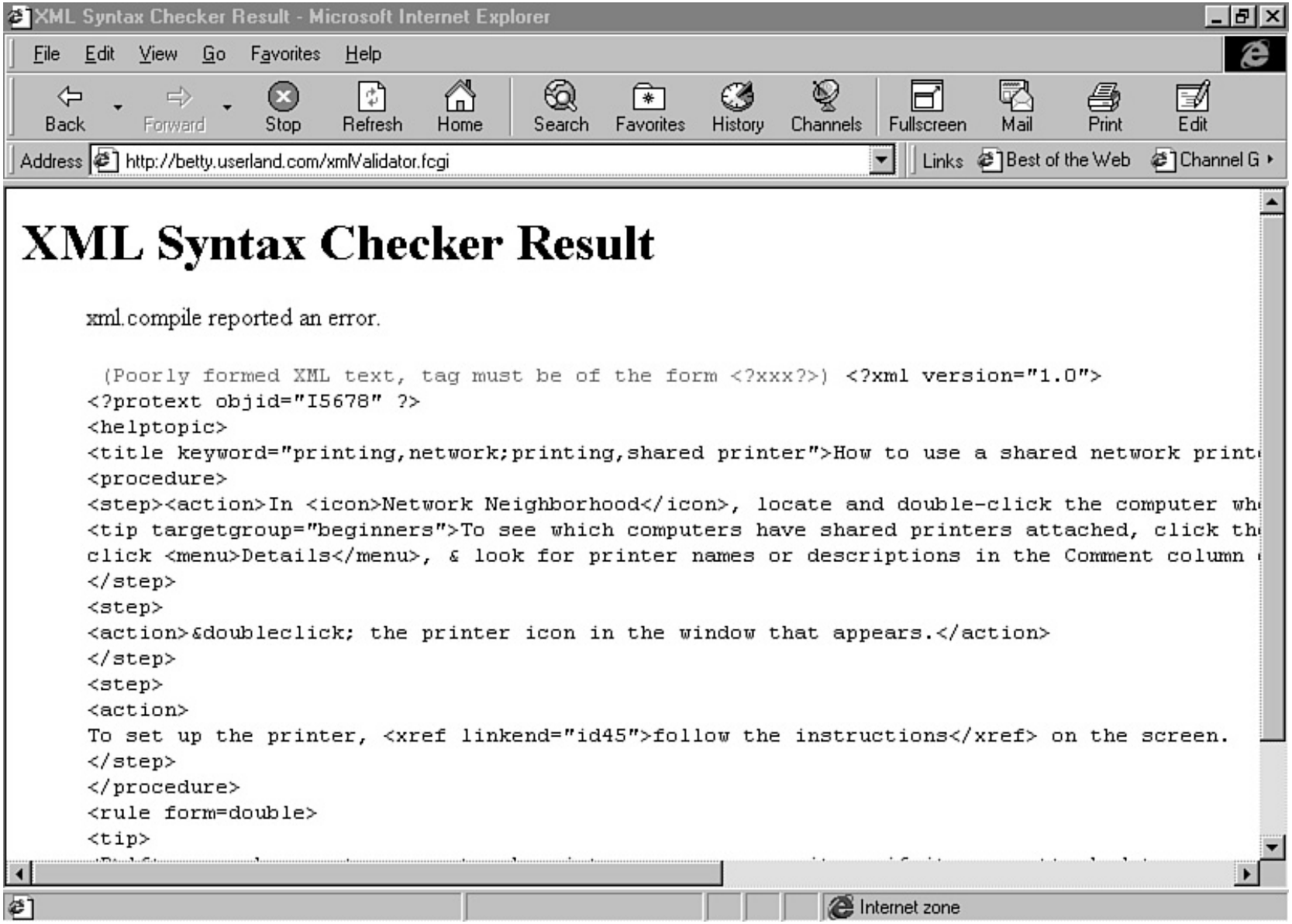


xmlwf
wfq.xml

The first error message is

wfq.xml:1:19:
syntax error


Frontier 5 and XML: XML Syntax Checker - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print Edit
Address http://www.scripting.com/frontier5/xml/code/xmlValidator.html
Links Best of the Web Channel G
USERLAND SOFTWARE
Main
Frontier 5 Home
What is Frontier?
Pricing
Purchase
Try Frontier
Download Docs
Site Outline
Site Search
Stories
Scripting News
Docs
HowTos
Website Tutorial
Scripting Tutorial
HTML Reference
User's Guide
DocServer
Frontier 5 and XML: XML Syntax Checker
Type in a URL below, click on Submit and a script on betty.userland.com will get the XML text, run it thru a parser, and report if it's well-formed or not.
http://www.protext.be/wfq.xml
Use the Frontier 5.1.3 built-in parser
Use the blox parser, based on "expat"
Submit
A list of URLs
Here's a list of URLs you may want to try:
http://ender.felter.org/http/98/08/news17.xml
http://lee.podo.co.kr/music/classes/back/Music.xml
Internet zone



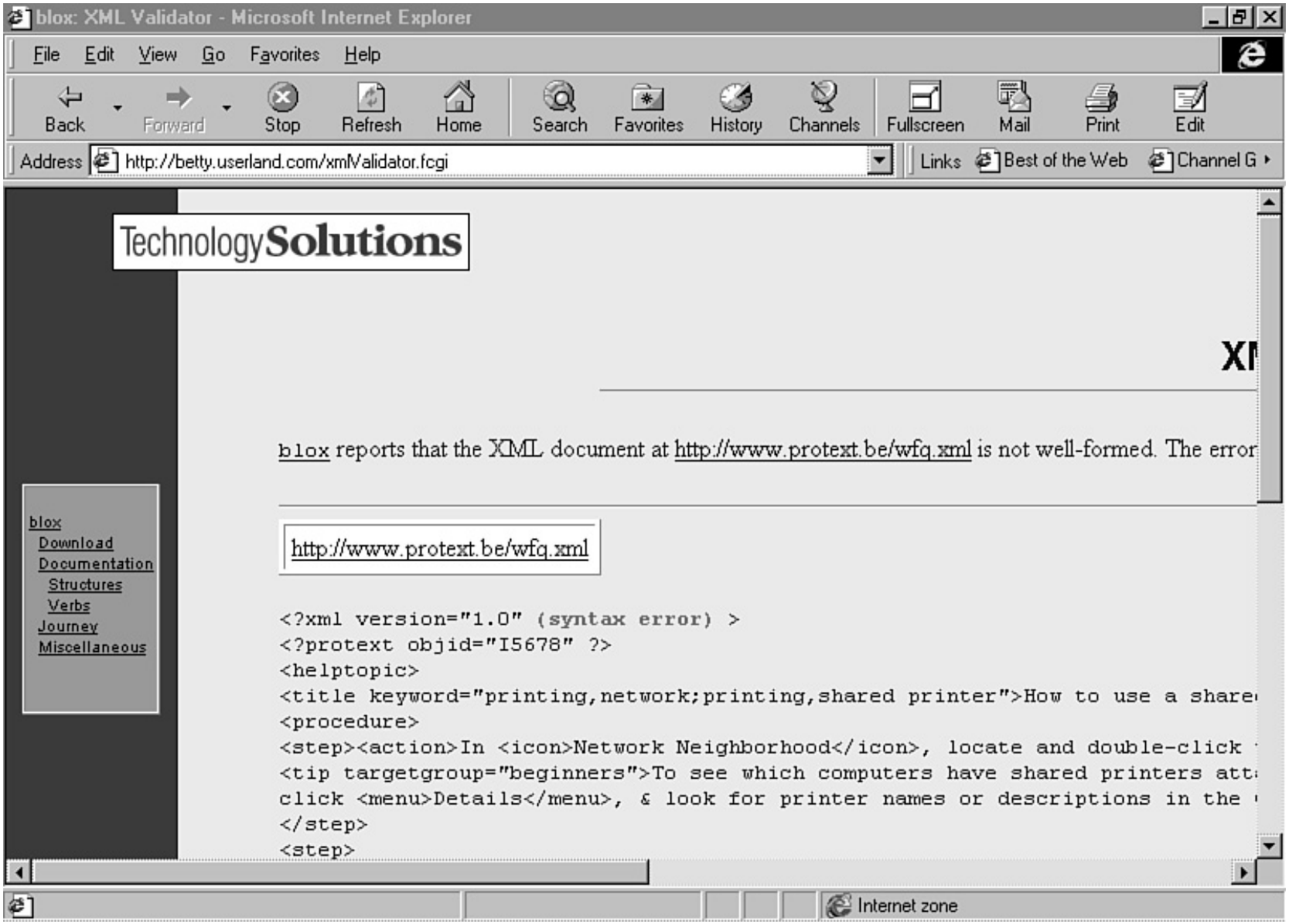
This message shows four pieces of information:

- The name of the file where the error was encountered
- The line number with the error
- The position on that line where the error was detected
- A description of the error

In this file, there was a syntax error detected on line 1, character 19, which is in front of the > character. Indeed, the XML declaration needs to end with ?>, which wasn't the case in this file. Let's correct that:

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

Run expat again and you'll receive a second error message:



wfq.xml:8:29: not well-formed

- [ ] The problem seems to be with the ampersand character used. The XML specification says that "the ampersand character (&) may appear in its literal form only when used as a markup delimiter... If it is needed elsewhere, it must be escaped."

This escaping can be done by using the predefined entity &:

```
8: click <menu>Details</menu>, &amp;amp; look for printer names
⇒or descriptions in the Comment column of the Network
⇒Neighborhood window.</tip>
```

The next error message is:

- wfq.xml:11:18: undefined entity
- You have used an entity reference to the entity doubleclick, but this entity hasn't been declared.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Let's examine what the specification says: "A textual object is said to be a well-formed XML document... if, for each entity reference which appears in the document, either the entity has been declared in the document type declaration or the entity name is one of [the following]: amp, lt, gt, apos, quot."

So you need to add an entity declaration to your document, more specific to the document type declaration, which you don't have at the moment.

Let's start by adding one:

```
3: <!DOCTYPE helptopic []>
```

Now add the declaration of the entity doubleclick:

```
3: <!DOCTYPE helptopic [
4: <!ENTITY doubleclick "Double-click">
5: ]>
```

The XML file looks like Listing 5.2.

Listing 5.2 wfq.xml—An Entity Declared in the Document Type Declaration Internal Subset

```
1: <?xml version="1.0" ?>
2: <?protext objid="I5678" ?>
3: <!DOCTYPE helptopic [
4: <!ENTITY doubleclick "Double-click">
```

```
5: ]>
6: <helptopic>
7: <title keyword=“printing,network;printing,shared printer”>How
⇒to use a shared network printer?</title>
8: <procedure>
9: <step><action>In <icon>Network Neighborhood</icon>, locate and
⇒double-click the computer where the printer you want to use is
⇒located. </action>
10: <tip targetgroup=“beginners”>To see which computers have
⇒shared printers attached, click the <menu>View</menu> menu,
11: click <menu>Details</menu>, &amp; look for printer names or
⇒descriptions in the Comment column of the Network
⇒Neighborhood window.</tip>
12: </step>
13: <step>
14: <action>&doubleclick; the printer icon in the window that
⇒appears.</action>
15: </step>
16: <step>
17: <action>
18: To set up the printer, <xref linkend=“id45”>follow the
⇒instructions</xref> on the screen.
19: </step>
20: </procedure>
21: <rule form=double>
22: <tip>
23: <P>After you have set up a network printer, you can use it as
⇒if it were attached to your computer. For related topics, look up
⇒&quot;printing&quot; in the Help Index.
24: </p>
25: </tip>
26: </helptopic>
```

Now, let’s parse again:

- wfq.xml:19:2: mismatched tag

- You encounter an end tag for the step element. Take a close look at what’s happening inside the step element: an action element was encountered, but it isn’t closed at the end of the step.

Let’s examine the XML specification once more: “For all other elements (other than the root), if the start tag is in the content of another element, the end tag is in the content of the same element. More simply stated, the elements, delimited by start and end tags, nest within each other.”

Therefore, line 19 needs to become

```
19: </action></step>
```

After this correction, the next parsing error you encounter is

- wfq.xml:21:11: not well-formed

The problem is in the line

```
21: <rule form=double>
```

Let's take a look at the attribute form. An attribute value needs to be inside quotes:

```
21: <rule form="double">
```

The next expat message is

wfq.xml:24:2: mismatched tag

On line 24 there's an end tag for the element p, but there is no element p open. P is open. Both the start and end tag needs to be in the same case. Change the start tag to p:

```
23: <p>After you have set up a network printer, you can use it as
⇒if it were attached to your computer. For related topics, look up
⇒&quot;printing&quot; in the Help Index.
24: </p>
```

The next parse result we receive is:

wfq.xml:26:2: mismatched tag

This is a tricky one. Do you see an error on line 26? Not immediately. The root element is being closed here. Remember that all other elements have to be nested inside this root element. Is this the case? Has every open element been closed?

No, the element rule is still open. But let's examine this element a little bit more closely. Does this element rule contain any content? No, it seems to be an empty element. And according to the specification, empty elements must take one of the following forms:

- <tag></tag>
- <tag/>

The following is what's needed:

```
21: <rule form="double"/>
```

After all these corrections, the file should look like Listing 5.3.

Listing 5.3 wf.xml—The Corrected XML File

```
1: <?xml version="1.0" ?>
2: <?protext objid="I5678" ?>
3: <!DOCTYPE helptopic [
4: <!ENTITY doubleclick "Double-click">
5: ]>
6: <helptopic>
7: <title keyword="printing,network;printing,shared printer">How
⇒to use a shared network printer?</title>
```

```
8: <procedure>
9: <step><action>In <icon>Network Neighborhood</icon>, locate and
⇒double-click the computer where the printer you want to use is
⇒located. </action>
10: <tip targetgroup="beginners">To see which computers have
⇒shared printers attached, click the <menu>View</menu> menu,
11: click <menu>Details</menu>, &amp;amp; look for printer names or
⇒descriptions in the Comment column of the Network
⇒Neighborhood window.</tip>
12: </step>
13: <step>
14: <action>&doubleclick; the printer icon in the window that
⇒appears.</action>
15: </step>
16: <step>
17: <action>
18: To set up the printer, <xref linkend="id45">follow the
⇒instructions</xref> on the screen.
19: </action></step>
20: </procedure>
21: <rule form="double"/>
22: <tip>
23: <p>After you have set up a network printer, you can use it as
⇒if it were attached to your computer. For related topics, look up
⇒&quot;printing&quot; in the Help Index.
24: </p>
25: </tip>
26: </helptopic>
```

Let's run another parse. No messages appear: The file seems to be error-free.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Checking Your XML Files with DXP

DXP stands for DataChannel XML Parser and is based on NXP (Norbert Mikula's XML Parser), one of the first XML parsers. It is written in Java.

> Java is the most popular language for XML development. That's easy to understand. XML is SGML made easy and optimized for use on the Web. One of the design principles of SGML was to be independent of hardware, operating system, software, and so on. That's also exactly what Java does: it allows you to run your programs on different machines, on different operating systems, even on Web browsers. They make a perfect couple.
> So if you want to become involved in serious XML development, you need to learn Java.

You can use this parser not only to check the well-formedness of your files, but also to validate them, a subject treated during Day 9.

Installing DXP

You can download DXP from http://www.DataChannel.com/xml.resources/ dxp.shtml#download. The actual file to download, DXP.zip, is 547KB. Next to the needed Java classes, you'll find a fair amount of documentation and examples.

How to install DXP:

1. Unzip the DXP.zip file.
2. Make sure you have Java Virtual Machine version 1.x running. You can use SUN's Java Development Kit or JDK (http://java.sun.com/products/jdk/1.1/), Sun's Java Runtime Environment or JRE (http://java.sun.com/products/jdk/ 1.1/jre/index.html), or Microsoft's SDK for Java (http://www.microsoft.com/ java/).
   To install a Java Virtual Machine, see the book *Sams Teach Yourself Java 1.2 in 21 Days* or

the documentation included with the preceding tools.

3. Make sure that your Java Virtual Machine can find the DXP classes by doing one of the following:

Adding to your classpath environment variable the packages directory of the DXP parser, c:\DataChannel\dxp\classes, or

In the case of JRE, setting the –cp parameter to the path on the command line: jre –cp c:\DataChannel\dxp\classes.

Using DXP

At the command line (DOS prompt), type

```
jre -cp .;c:\DataChannel\dxp\classes dxpcl -s c:\xmlex\wfq.xml
```

where

- -cp sets the classpath (where to find the classes used). In this case two paths are specified: the first (.), referring to the current working directory, and the second (c:\DataChannel\dxp\classes), both separated by;.

The path separator on UNIX machines is a :.

- dxpcl is the name of the Java program (class).
- -s stands for silent mode.

For the moment, we are only interested in the errors in this file. By asking for silent mode, the only output generated are the error messages.

- c:\xmlex\wfq.xml is the file to be checked.

Checking a File Error by Error

Let’s go over this same file step-by-step and see in detail which errors DXP detects.

Checking the file wfq.xml results in the following error:

```
FATAL ERROR: encountered “>”. Was expecting one of: <S> , “?”
Location: file:/c:/xmlex/wfq.xml:1:20

Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)
```

For every error message, you receive

- The kind of error with an explanation, and
- On a second line, the location of the error (line 1, character 20 in file wfq.xml)

The XML specification makes a distinction between errors and fatal errors. Violations of well-formedness constraints are fatal errors. A XML processor must detect such errors and must not continue normal processing.

At the end you’ll find a summary with the numbers of errors, fatal errors, and warnings.

You know already what the problems are with this XML file. The first error was the missing ? at the end of the XML declaration. We'll change that. Let's run the parser again:

```
FATAL ERROR: after the "&" there is no "entity-name" and ";"
⇒following in the entity reference
Location: file:/c:/xmlex/wfq.xml:7:146

FATAL ERROR: reference to undefined entity "doubleclick"
Location: file:/c:/xmlex/wfq.xml:10:21

FATAL ERROR: name of the start-tag "action" doesn't match the
⇒name of the end-tag "step"

Location: file:/c:/xmlex/wfq.xml:15:7

Found errors/warnings: 3 fatal error(s), 0 error(s) and 0 warning(s)
```

Now we're making some progress. There are three more errors, all familiar:

- The not-escaped &
- The undefined entity doubleclick
- The missing end tag of the element action

Correct them and run the parser again:

```
FATAL ERROR: encountered "double". Was expecting one of: "\"" , "\'"
Location: file:/c:/xmlex/wfq.xml:20:12

Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)
```

Attribute values need to be quoted. After this correction, you receive the following:

```
FATAL ERROR: name of the start-tag "P" doesn't match the name of
⇒the end-tag "p"
Location: file:/c:/xmlex/wfq.xml:23:4
Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)
```

Get rid of this mismatch and parse once more:

```
FATAL ERROR: name of the start-tag "rule" doesn't match the name
⇒of the end-tag "helptopic"
Location: file:/c:/xmlex/wfq.xml:25:12

Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)
```

Make the rule element empty. After this correction, you have a correct file. The parser doesn't generate any more messages.

Checking Your Files Over the Web Using RUWF

RUWF, the XML syntax checker, is a service provided by the Web site of XML.COM, a partnership between Seybold Publications and O'Reilly with the technical aid of Tim Bray, featuring the king of XML gossip, Xavier McLipps. You can reach this service at http://www.xml.com/xml/pub/tools/ruwf/check.html.

This service was built using Lark, the non-validating parser written by Tim Bray.

Using RUWF

1. Place your XML file on a Web server so it can be referenced by an HTTP address (such as http://www.protext.be/wfq.xml).
2. Go to http://www.xml.com/xml/pub/tools/ruwf/check.html.
3. Put your URL in the input field of the RUWF page and submit the page.

If you send wfq.xml to the URL, the HTML page shown in Figure 5.1 is returned.

Figure 5.1 *The result of the syntax checking by RUWF.*

Previous | Table of Contents | Next

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Checking Your Files Over the Web Using Other Online Validation Services

Here are some other online validation services:

- XML well-formedness checker from Richard Tobin at http://www.cogsci.ed.ac.uk/~richard/xml-check.html.
- XML Syntax Checker from Frontier at http://www.scripting.com/ frontier5/xml/code/syntaxChecker.html. You can use the Frontier 5.1.3 built-in parser or the blox one, which is based on expat.
- The Koala XML Validation Service at http://koala.inria.fr:8080/XML/.
- The Techno 2000 Project XML Validation Service at http:// xml.t2000.co.kr/xmlval/.
- WebTech's Validation Service at http://valsvc.webtechs.com/.

Using XML Well-formedness Checker

At http://www.cogsci.ed.ac.uk/~richard/xml-check.html, you'll find the page shown in Figure 5.2.

Figure 5.2 *The submit page of the XML well-formedness checker.*

After supplying the URL of the file to be checked, you receive the results page shown in Figure 5.3.

Figure 5.3 *The results page of the XML well-formedness checker.*

Using XML Syntax Checker from Frontier

Figure 5.4 shows you what the submit page looks like at http://www.scripting.com/frontier5/xml/code/syntaxChecker.html.

Figure 5.4 *The submit page of the XML Syntax Checker of Frontier.*

Depending on the parser chosen, you get what's shown in Figure 5.5...

Figure 5.5 *The results generated by the built-in parser of Frontier.*

...or what's shown in Figure 5.6.

Figure 5.6 *The results generated by Blox, using expat.*

Summary

In an XML environment, you need to be sure that your documents are well-formed according to the XML recommendation. Non-validating parsers can help you meet this requirement.

A lot of these tools are available for free on the Web. We did our quality checks with the following:

- expat, a parser written by James Clark
- DXP, a parser written by DataChannel
- RUWF using Lark, written by Tim Bray
- Other online validation services

Do
Do check your XML files for well-formedness.
This is very important because when an XML file violates the specified well-formedness constraints, it contains fatal errors and the normal processing must be stopped.

Q&A

Q Why should I bother to have well-formed XML files?
A XML processors act as helpers for other applications, giving them access to the content and structure of the XML files. If an XML file contains violations against the specified well-formedness constraints, it has fatal errors. And in the case of fatal errors, the normal processing must be stopped.

This is indeed different from HTML, in which browsers try to make sense out of the rubbish they sometimes receive. Consequently, 75 percent of the code in Internet Explorer and Netscape Navigator is just for handling bad HTML.

Q I cannot make sense out of the error message I received. What can I do?

A If you have trouble understanding the message, use the annotated XML specification at http://www.xml.com/axml/testaxml.htm. This specification is heavily indexed and can help you find the relevant section.

Or submit the file to a different parser. Parsers vary substantially in the extent and clarity of their error messages.

Q What other well-formedness parsers are available?

A Other examples of non-validating parsers are ®lfred, developed by David Megginson (http://www.microstar.com/XML/index.html), and Microsoft XML Parser in C++, which is included Internet Explorer 4.0.

But for a complete overview, see the links at the beginning of the chapter.

Q Are HTML files well-formed?

A Not normally, but they can be made well-formed.

The most obvious violations against well-formedness in HTML files are

- Not all start and end tags are explicitly included
- The syntax of empty elements

Remember that in XML, empty elements need to encoded in one of two possible ways:

<name></name>

<name/>

Exercises

1. See the following XML file:

Listing 5.4 function.xml—XML File to Check for Well-formedness

```
1: <?xml version=“1.0”?>
2: <!DOCTYPE functiondescription [
3: <!NOTATION GIF SYSTEM “” >
4: <!ENTITY idhelp782 SYSTEM “idhelp782.txt”>
5: <!ENTITY idhelp785 SYSTEM “idhelp785.txt”>
6: <!ENTITY idhelp645 SYSTEM “idhelp645.txt”>
7: <!ENTITY buttonleft SYSTEM “http://www.protext.be/button.gif”
⇒NDATA GIF>
8: ]>
9: <function>
10: <title>ctime</title>
11: <funcsynopsis>
12: <funcdef>
13: <function>ctime</function>
14: </funcdef>
15: <paramdef>
16: <parameter>time</parameter>
17: <parameter role=“opt”>gmt</parameter>
18: </paramdef>
19: </funcsynopsis>
20: <para>This function converts the value
21: <parameter>time</parameter>, as returned by <function>time()
⇒</function>
```

```
22: or <function>file_mtime()</function>, into a string of the
⇒form produced by <function>
23: time_date()</function>. If the optional argument <parameter>
⇒gmt</parameter>
24: is specified and non-zero, the time is returned in <parameter>
⇒gmt</parameter>.
25: Otherwise, the time is given in the local time zone.</para>
26: <para><emphasis role="strong" targetgroup="beginners role=
⇒"strong">Related topics</emphasis></para>
27: <para>&doubleclick; the <mousebutton>
28: LEFT &buttonleft;</mousebutton> mouse button on a topic:
⇒</para>
29: <itemizedlist>
30: <listitem><para><ulink url="&idhelp782;"><function>
⇒file_mtime()</function>
31: built-in function</ulink></para>
32: </listitem>
33: <listitem><para><ulink url="&idhelp785;"><function>time()
⇒</function> built-in
34: function</ulink></para>
35: </listitem>
36: </itemizedlist>
37:</function>
```

Using this XML file:

- Detect the errors and predict the error messages that the parser of your choice will generate
- Check the file with that parser
- Correct all problems discovered

2. Using an XML file you've made during the previous days:

- Introduce errors

- Parse the file with different parsers
- Study the differences between the results of different parsers

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 6
Creating Valid Documents

So far you have learned about XML tags, elements, and attributes, and you have seen how you can use simple tools to test whether an XML document is well formed. This chapter looks a little further and introduces the most obvious feature of XML that identifies it as being firmly on the side of SGML rather than next to HTML as just another set of tags. Today you will learn

- What a Document Type Definition (DTD) is
- The contents and purpose of the internal subset
- The contents and purpose of the external subset
- The basics of developing a DTD

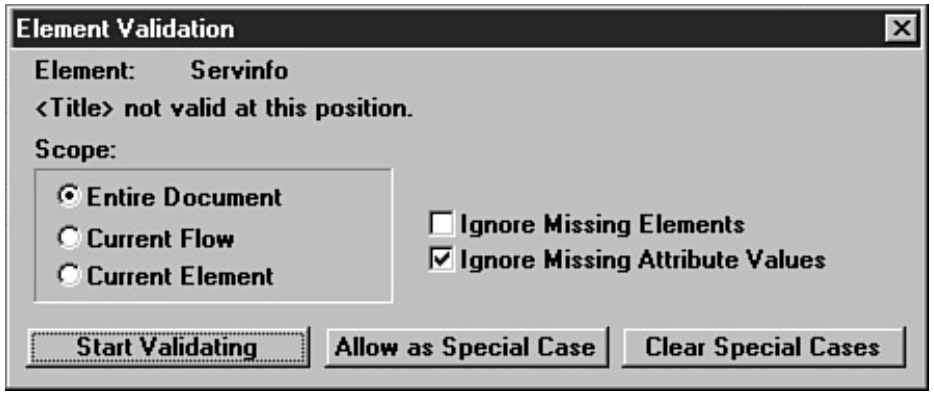
This chapter is not intended to be a complete treatment of DTDs, but it should be enough to get you started. On Day 7, “Developing Advanced DTDs,” we will look at the more advanced aspects of DTD development and cover some of the approaches to information modeling as it can be applied to DTD development.

XML and Structured Information

If you only wanted “proper tagging”—with all the text neatly wrapped inside elements—creating XML documents is quite possibly unnecessary. You could probably achieve just as much by using the standard HTML elements and inventing classes to do the work for you. For example, the HTML code shown in Listing 6.1 could do almost as much as the equivalent XML code.

Listing 6.1 Using HTML to Mimic XML

```
1: <HTML>
2:   <HEAD>
3:     <TITLE>HTML Mimicking XML</TITLE>
4:   </HEAD>
5:   <BODY>
6:     <DIV ID=“1” CLASS=“CHAPTER”>
7:      <H1 CLASS=“HEAD1”>Chapter 1</H1>
8:          <P CLASS=“PARA”>This actually comes quite close
9:             to being acceptable as pseudo-XML.</P>
10:         <DIV ID=“1.1” CLASS=“SECTION”>
11:           <H2 CLASS=“HEAD2”>Section 1.1</H2>
12:              <P CLASS=“PARA”>We can even bring in some
13:                 kind of pseudo-structure by using
14:                 attributes. </P>
15:           </DIV>
16:    </BODY>
17: </HTML>
```

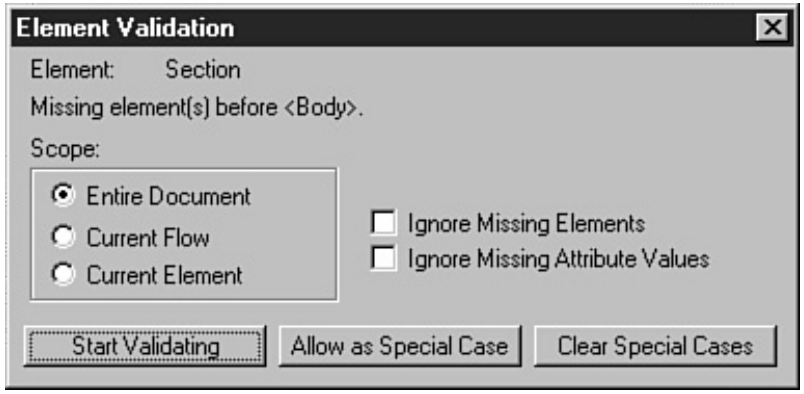


Listing 6.1 shows an example of how HTML code can come quite close to being as rich as XML markup. By using HTML DIV elements (lines 6 and 10) and ID and CLASS attributes on as many elements as necessary, it is possible to build almost as much information into the markup as might be achieved with custom XML elements.

Now try comparing the HTML code shown in Listing 6.1 with the almost equivalent XML code shown in Listing 6.2.

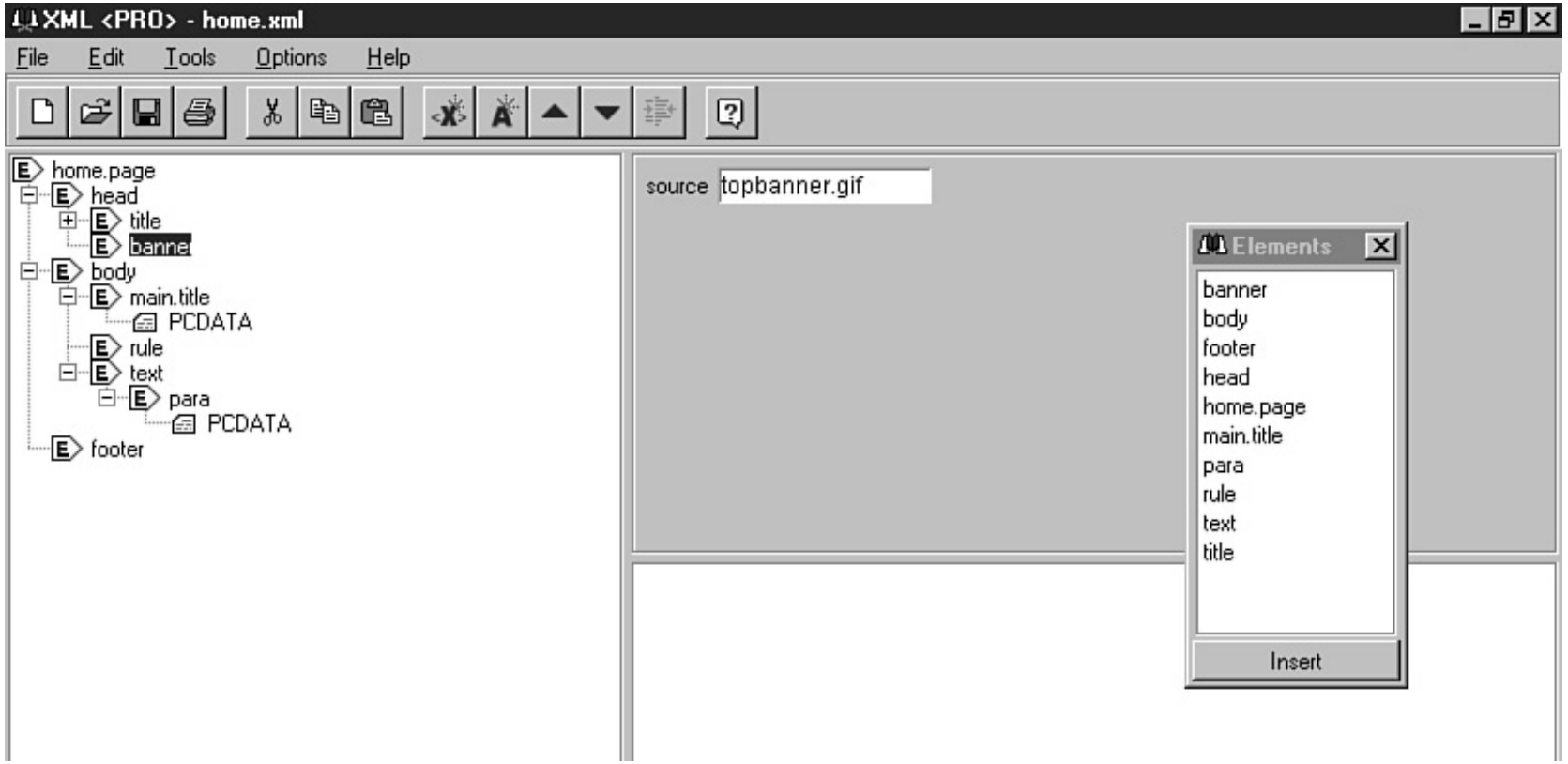
Listing 6.2 XML Using Elements to Match HTML Attributes

```
1: <?xml version="1.0"?>
2:   <DOCUMENT>
3:     <TITLE>XML Being What it is</TITLE>
4:   <CHAPTER>
5:      <HEAD>Chapter 1</HEAD>
6:          <PARA>This is quite
7:             acceptable XML.</PARA>
8:         <SECTION>
9:           <HEAD>Section 1.1</HEAD>
10:              <PARA>The structure lies in the
11:                 elements themselves. </P>
12:           </SECTION>
13:    </CHAPTER>
14: </DOCUMENT>
```



In Listing 6.2, the XML markup uses elements to achieve what HTML, which doesn't have the freedom to invent its own elements (or at least, not officially), is forced to use attributes for. On this small scale (or superficial level of complexity) you could use HTML just as readily as XML.

So what do you gain from using XML instead of HTML? The most obvious difference is that in HTML you are very limited in what attributes you can give an element (in most cases not much more than an ID and a CLASS), and not all elements can have these attributes. XML, in contrast, enables you to not only have as many attributes as you like, but to call them whatever you like (subject to the normal naming rules, of course). These attributes allow you to add a wealth of information to describe an element, as in the declaration of a graphic element shown in Listing 6.3.

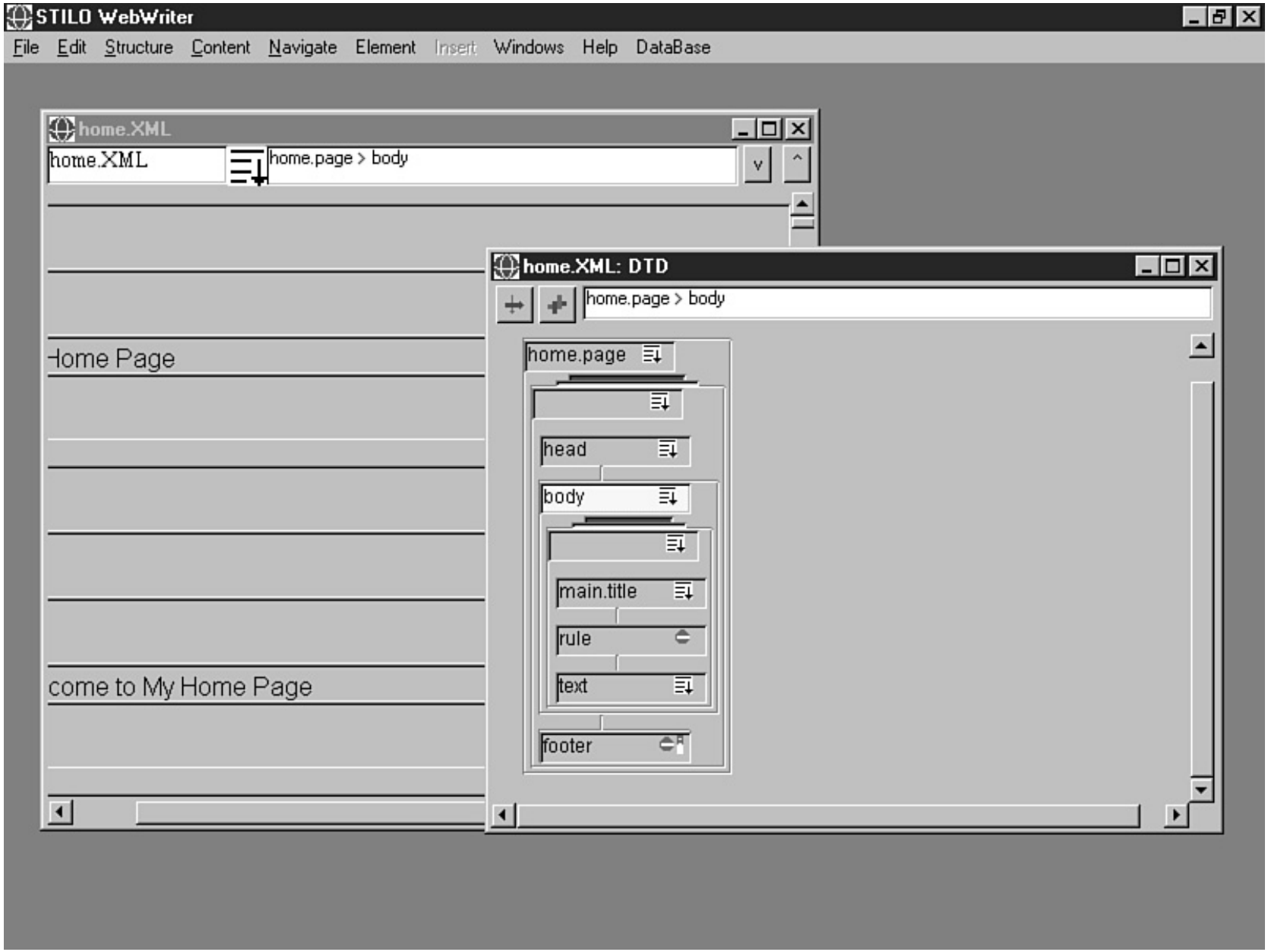


The set of graphics attributes shown in Listing 6.3 is derived from a Department of Defense SGML DTD and is far more complex than anything you are ever likely to encounter. It is certainly far more complex than anything you will ever need to create yourself.

Listing 6.3 A Typical Graphic Element Declaration

```
1:<!ELEMENT graphic  EMPTY >
2:<!ATTLIST graphic boardno        ENTITY         #REQUIRED
3:                      graphsty   NMTOKEN        #IMPLIED
4:                      llcordra   NMTOKEN        #IMPLIED
5:                      rucordra   NMTOKEN        #IMPLIED
6:                      reprowid   NMTOKEN        #IMPLIED
7:                      reprodep   NMTOKEN        #IMPLIED
8:                      hscale     NMTOKEN        #IMPLIED
9:                      vscale     NMTOKEN        #IMPLIED
10:                     scalefit   %yesorno;      #IMPLIED
11:                     hplace     (left | right |
12:                                center | none) #IMPLIED
13:                     vplace     (top | middle |
```

```
14:                          bottom | none)  #IMPLIED
15:                 coordst  NMTOKEN         #IMPLIED
16:                 coordend NMTOKEN         #IMPLIED
17:                 rotation NMTOKEN         #IMPLIED>
```



The attributes in Listing 6.3 describe all the physical and placement properties for the image. This is information relevant to the processing of the element, but not information in the same way that the text in the XML document is. This meta information is rightly kept as attributes instead of being part of the normal element content of the document.

As you saw earlier, however, although HTML doesn't have the freedom to define new elements, it is perfectly legal to add as many attributes as you like.

So what about structure? XML is structured by its very nature; HTML has little or no structure. HTML, like any other SGML application, does have a DTD... it's just that unless you get involved with the technicalities of HTML you will probably hardly ever notice it and, depending on what software packages you use to create HTML code, you might never need to know that it's there. The HTML DTD is a fairly loose DTD, requiring very little and giving lots of room to choose the order of elements that suits you best. This is, however, purely a matter of usage. If you wanted to, you could apply a lot of self-discipline and turn out extremely well-structured results, as shown in Listing 6.4.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Listing 6.4 Structured HTML

```
1:  <HTML>
2:   <HEAD>
3:      <TITLE>A Tale of Two Computers</TITLE>
4:   </HEAD>
5:    <BODY>
6:      <DIV CLASS="CHAPTER">
7:        <H1>CHAPTER ONE</H1>
8:        <P>It was a dark and stormy night.</P>
9:             <DIV CLASS="SECTION">
10:              <H2>At Home</H2>
11:              <P>The author huddled over his manuscript.</P>
12:            </DIV>
13:            <DIV CLASS="SECTION">
14:              <H2>At Work</H2>
15:              <P>The writer huddled over his draft.</P>
16:            </DIV>
17:     </DIV>
18:   </BODY>
19:  </HTML>
```

If you really want to do so, you can use HTML in a structured way. In contrast, it is quite

possible to use just P tags and FONT attributes to make an HTML look like a very highly structured document when it's displayed in a Web browser. If you were to take a look at the raw HTML code, though, it would be almost impossible to find the actual text amongst the code, much less make sense of it.

Done carefully, you can encode almost as much information in HTML code as you can in XML. The problem is that you cannot count on any help from your tools. You certainly won't get any help from a Web browser; they have intentionally been built to accept any and all tags. If the Web browsers can't make sense of the tags, they will simply ignore them. The early Web browser developers were too worried about scaring people away from using HTML by enforcing rigid rules.

Yes, there are some HTML code verification packages (they even call themselves *validators*, but this is stretching the interpretation of the term just a little too far) and they are generally very good. The verifiers check that you have used the HTML tags properly and that your attributes are properly used, but they cannot help you by checking the structure of your document. If you use an H4 tag before an H2 tag, that's your problem. If you want to use <P><FONT SIZE="7"> instead of <H1>, it's all allowed by "the" HTML DTD (there are at least 19 different officially recognized HTML DTDs in circulation).

Why Have a DTD at All?

We've talked about DTDs and about how HTML has one too. Later in today's chapter you will discover that XML allows you to have many, but this begs the question of whether you need a DTD at all. Hasn't XML been touted as DTD-less SGML? True, as you have already learned, as long as it is well formed, there is no need to have a DTD. You will even learn later today that it is possible to derive a DTD just by looking at the XML document. There are, however, some important restrictions on an XML document that does not have a DTD.

If you want to be able to validate an XML document without a DTD:

- All the attribute values in the XML document must be specified; you cannot have default values for them.
- There can be no references to entities in the XML document (except of course amp, lt, gt, apos, and quot).
- There can be no attributes with values that are subject to normalization. (For example, when they contain entity references. We will return to entities on Day 9, "Checking Validity.")
- In elements with content consisting of only elements, there can be no whitespace (space, tab or other whitespace characters) between the starting tag of the container element and the starting tag of the first element contained in it. For example, this would be illegal:

```
<CHAPTER>  <SECTION>... </SECTION></CHAPTER>
```

This is a complicated point, but without the help of a DTD to tell the XML processor whether this whitespace is to be treated as meaningful (as PCDATA or as preserved whitespace), the XML processor has no way of knowing whether to delete it or not.

DTDs and Validation

The DTD describes a model of the structure of the content of an XML document. This model

says what elements must be present, which ones are optional, what their attributes are, and how they can be structured with relation to each other. While HTML has only one DTD, XML allows you to create your own DTDs for your applications. This gives you complete control over the process of checking the content and structure of the XML documents created for that application. This checking process is called *validation*.

Of all the markup languages, SGML and XML are almost unique in that their documents can be truly validated. Yes, there are software packages that can "validate" HTML code, but this is validation of a completely different nature. HTML code validation is basically little more than tag syntax checking—looking for spelling mistakes and omissions. True validation in SGML and XML terms goes much farther than this. A valid XML document is one that isn't just syntactically correct, it's one with internal structure that complies with the model structure you declare in the DTD.

Depending on what you, as the DTD developer, want to achieve, you can exercise almost complete control over the structure and create a strict DTD. When you validate XML documents created using this strict DTD, you can insist that certain elements are present and you can enforce the set order you require. You can check that certain attribute values have been set and, to a limited degree, even check that these attribute values are the right general type.

On the other hand, you can also make almost everything optional and create a loose DTD. You could even have parallel versions of the same DTD, one that allows you to create draft versions of the XML that aren't complete and another that rigidly checks that everything is present. Going even further, and you will learn how to do this on Day 9, it is even possible to insert switches into a DTD that can be used to turn the strictness on and off again.

Based on what you have declared in the DTD, when the completed XML document is validated, what is allowed and what is not will be completely determined by the choices you made in designing the DTD. The author of the document can then be warned if required elements are missing, as shown in Figure 6.1, and warned when elements are not in the right place, as shown in Figure 6.2.

Figure 6.1 *Missing element warning.*

Figure 6.2 *Faulty structure warning.*

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Document Type Declarations

So you've decided to use a DTD. How do you associate a DTD with an XML document? You use a document type declaration. Confusing? Even within established SGML circles there is a lot of confusion about the very similar terms *document type declaration* and *document type definition*. These are two completely different things, so it is good to know which is which. The document type definition (DTD) is an XML (and SGML) description of the content model of a type (or class) of documents. The document type declaration is a statement in an XML (or SGML) file that identifies the DTD that belongs to the document. If an external DTD file is used, the document type declaration identifies where the DTD entity (the file) can be found.

At its very simplest, the syntax of a document type declaration looks like this:

```
<!DOCTYPE DTD.name  [ internal.subset  ]>
```

DTD.name is the name of the DTD. When you read about validity later in this chapter you will discover that the DTD name should be the same as the root element of the document. So, a DTD designed for a document would be called book or something similar and the root element in the document would also be book. Don't forget that XML is case sensitive—if you call the DTD BooK then you should have a root element BooK.

internal.subset is the contents of the internal DTD subset, the part of the DTD that stays in the XML document itself. We will investigate the internal DTD subset shortly; it contains local element, attribute, and entity declarations. Without the internal DTD subset there wouldn't really be much point in including a document type declaration at all.

Internal DTD Subset

For an XML document to be well formed, all the external entities must be declared in the DTD. If you design your application carefully, it may be possible for you to put all the declarations in the internal DTD subset. With all the declarations in the internal DTD subset, the XML processor would not need to read and process external documents.

The existence of an internal DTD subset does not affect the XML document's status as a standalone document. This can be a little confusing at first, since you might assume that being standalone means the document doesn't need to reference a DTD at all. As long as the declarations in the internal DTD subset follow a few basic rules, the XML document qualifies as a standalone document.

When you start off the XML document, the first line is the XML declaration, which can include a standalone document declaration:

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

The statement standalone="yes" means that there are no markup declarations external to the document entity (recalling our discussion of physical and logical structures on Day 2, "Anatomy of an XML Document"). In the XML document it is still perfectly acceptable to reference external entities (graphics files, included text, and so on) provided that the declarations of the external entities are contained inside the document entity (in other words, inside the internal DTD subset).

Standalone XML Documents

A document type declaration and the contents of an internal DTD subset are all you need to define the structure of an XML document. Without any external support and without referring to any other files, an XML document containing an internal DTD subset contains enough information to be used for quite complex applications.

Given what you have already learned about declaring elements and attributes, you should already be able to produce something like the basic catalog shown in Listing 6.5.

Listing 6.5 A Standalone XML Document with Internal DTD Subset

---

```
1:  <?XML version="1.0" standalone="yes"?>
2:  <!DOCTYPE CATALOG [
3:
4:  <!ELEMENT CATALOG (PRODUCT+)>
5:
6:  <!ELEMENT PRODUCT (SPECIFICATIONS+, PRICE+, NOTES?)>
7:  <!ATTLIST PRODUCT NAME CDATA #REQUIRED>
8:
9:  <!ELEMENT SPECIFICATIONS (#PCDATA)>
10: <!ATTLIST SPECIFICATIONS SIZE CDATA #REQUIRED
11:                          COLOR CDATA #REQUIRED>
12:
13: <!ELEMENT PRICE (#PCDATA)>
14: <!ATTLIST PRICE WHOLESALE NMTOKEN #REQUIRED
```

```
15:           RETAIL NMTOKEN #REQUIRED
16:           SALES.TAX NMTOKEN #IMPLIED>
17:
18: <!ELEMENT NOTES (#PCDATA)>
19: ]>
20:
21: <CATALOG>
22:   <PRODUCT NAME="T-shirt">
23:     <SPECIFICATION SIZE="XL" COLOR="WHITE"/>
24:     <PRICE WHOLESALE="9.95" RETAIL="19.95" SALES.TAX="2.56"
⇒ SHIPPING="5.00"/>
25:     <NOTES>Dilbert</NOTES>
26:   </PRODUCT>
27:   <PRODUCT NAME="Shirt">
28:     <SPECIFICATION SIZE="38" COLOR="BLACK"/>
29:     <PRICE WHOLESALE="69.95" RETAIL="79.95" SALES.TAX="4.54"
⇒ SHIPPING="10.00">
30:     Euro</PRICE>
31:   </PRODUCT>
32: </CATALOG>
```

The XML markup shown in Listing 6.5 starts with the now familiar XML prolog that identifies what follows as being an XML version 1.0 document. I have added the STANDALONE statement to make it explicitly clear to the XML processor that it doesn't need to look for an external DTD.

Line 2 declares the document type to be a CATALOG and then we open the internal DTD subset.

The meat of the DTD is lines 4 to 18. We have a CATALOG element that contains one or more PRODUCT elements. A PRODUCT element (line 6) contains one or more SPECIFICATION elements, followed by one or more PRICE elements and then optional NOTES elements.

The SPECIFICATION and PRICE elements, as you can see in the markup (lines 21 to 31) are actually empty and their information is included in the form of attributes (lines 23, 24, 28, and 29).

I could have declared these elements as empty, but in this case I left them open. Just because an element isn't declared as empty, it doesn't stop someone leaving the element empty.

Validation can check the markup, but it actually does little or nothing to check what's between the markup (the content), other than looking for more markup. You can have perfectly structured garbage if you want, or even perfectly structured space and tab characters.

In this case I didn't really want to specify a currency attribute, so all the prices (the PRICE attribute values) are assumed to be in the local currency. This might not always be so, and I still have the option of adding this information as text inside the PRICE element (lines 29 and 30) without having to go back later and change everything.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Note that I have used elements to identify the main objects and properties of those objects in the DTD; a PRODUCT has SPECIFICATION elements inside it. Most of the real property data, however, is declared as the attribute values of the SIZE, COLOR, and PRICE elements.

It is difficult to decide when to use elements and when to use attributes. We will explore this problem a little further tomorrow. I have declared most of the attribute values as being #REQUIRED, which means that they must be given. There is one exception though; the SALES.TAX attribute, which is #IMPLIED. This means that if the value isn't specified to the computer system (actually the XML application running on the computer system), the system will still be able to calculate the value for me.

The XML document I have described can be validated. It contains enough of a DTD, the internal DTD subset, to allow the content and structure of the XML document to be checked. In this form the XML document is pretty portable and there are good arguments for leaving it at that. By adding some 14 lines to the XML markup (far less if you take out the line breaks and spaces added to make it easier to read) we've succeeded in making the document reasonably self-describing. The recipient can even perform a rough check that parts of it are complete. The fact that a complete product is missing could only be detected by the lack of a closing CATALOG tag (assuming the file was clipped in transit). You wouldn't be able to tell through validation how many products were missing (although you could easily modify the DTD—even on-the-fly—to allow this to be checked).

Getting Sophisticated, External DTDs

You have already learned that you can achieve quite a lot with a standalone XML document. While you have all the benefits of portability by keeping the DTD inside the XML document itself, you are only just touching the surface of what can be achieved when you take the next logical step and use an external DTD subset.

The fact that it's called a document *type* definition already gives us a clue that a DTD is intended for use with more than one XML document. Indeed, by not using an external DTD subset you miss out on many XML features as well as the ability to use the DTD as a sort of template for a limitless set of XML documents.

In XML contexts the external DTD subset is often called an *external* DTD, but in SGML contexts it's more

often called *the* DTD. XML has one DTD, but it's a composite of both the internal DTD subset and the external DTD subset.

In XML, the internal DTD subset is read before the external DTD subset and so takes precedence. This allows you to use an external DTD subset to make global declarations and then override them on an individual basis by putting different declarations in the internal DTD subset.

You've already learned how to associate an internal DTD subset with an XML document. The association of an external DTD subset with an XML document is more complicated and uses either a system identifier (SYSTEM keyword) or a public identifier (PUBLIC keyword) and a system identifier:

```
<!DOCTYPE name public.identifier system.identifier [ internal.subset ]>
```

Both the system identifier and the public identifier are entity identifiers. We will return to these identifiers on Day 8, "XML Objects: Exploiting Entities." For now we will examine them in just enough detail for them to make sense when you use them to reference external DTDs.

System Identifier

A system identifier is a *URI* (Universal Resource Identifier), that can be used to retrieve the DTD. If you've ever opened a Web page in a browser or downloaded a file from an FTP site, you have already seen one form of a URI called a URL (Uniform Resource Locator). URLs are special forms of URIs intended for network (Internet) use. You will see the more technically precise name of URI used more often than URL, but for the majority of uses the two terms are virtually interchangeable.

A system identifier can reference an absolute location, as in:

```
<!DOCTYPE book SYSTEM "/mount/usr/home/dtds/book.dtd">
<!DOCTYPE book SYSTEM "<http://wwwin.synopsys.com/~north/dtds/book.dtd>">
```

or it can be a reference to a relative location:

```
<!DOCTYPE book SYSTEM "dtds/book.dtd">
<!DOCTYPE book SYSTEM "../../dtds/book.dtd">
```

Public Identifier

A public identifier is the officially recorded identifier for a DTD. Obviously it would be impossible to register every DTD (it was already impossible with SGML). Instead, the person or company who creates the DTD registers their own. The International Standards Organization (ISO) is responsible under the provisions of ISO 9070 for the registrations, but authority to issue identifiers and the associated record keeping is delegated to the American Graphic Communication Association (GCA).

A public identifier has the following form:

```
reg.type  // owner  //DTD description  // language
```

reg.type is a plus (+) if the owner is registered according to the ISO 9070 standard. Normally, this will not be the case and so reg.type is simply a minus (-) sign.

- owner is the name of the owner; the name of you or your company.
- description is a simple text description. You can make this description as long as you like, but it's a good idea to keep it as short and informative as possible. Spaces are allowed in the description, so you could make it something really meaningful like "Simple Email Message."
- language is the two-character language code taken from the ISO 639 standard.

An example of the public identifier for a DTD developed by me could then be:

```
-//Simon North//DTD Simple Web Page//EN
```

An XML processor attempting to retrieve the DTD's content may use the public identifier to try to generate a location. For reasons that will be explained on Day 9, this is not always possible and so the public identifier must be followed by a so-called *system literal*, which is simply a URI. Using the public identifier and the system literal, the documentation type declaration would then look like this:

```
    <!DOCTYPE home.page PUBLIC "-//Simon North//DTD Simple Web Page//EN"
⇒ "home.dtd">
```

Note that before a match is attempted for a public identifier, all the strings of whitespace in it are normalized to single space characters, and any leading and trailing whitespace is removed.

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Developing the DTD

There are many ways to describe information models, technically known as *schemas*. Indeed, there are several XML development activities devoted to defining schemas for describing XML data. One such schema is the XML DTD, which is inherited straight from SGML.

The task of developing a DTD can be as simple or as difficult as you make it. It all depends on what you want to do with the information you intend to model with the DTD, and what you intend to do with the information once it has been marked up. I cannot even hope to cover the subject of DTD development thoroughly here; to do so would require a book much thicker than the one before you now. In tomorrow's chapter I will try to address some of the major steps that you should go through while developing your DTD. For now, let's just look at some of the quick and easy methods.

There are two methods that may immediately spring to mind as possible shortcuts to creating a DTD:

1. Modify an existing SGML DTD
2. Create the DTD from the XML document, either automatically or manually

Modifying an SGML DTD

SGML and XML have DTDs. XML is a subset of SGML, so it would seem a logical step to simply take an SGML DTD (there are hundreds in circulation on the Internet) and modify it for use with XML document. Unfortunately, life is never that simple.

There are many differences between an SGML DTD and an XML DTD and, unlike SGML and XML documents, the two are a long way from being compatible with each other. The conversion of an XML DTD into an SGML DTD could be a relatively simple task (and will ultimately be a trivial task once the SGML has been amended to make XML SGML-compliant). On the other hand, it can be an extremely difficult task to convert an SGML DTD into an XML DTD.

The fact that there are so many SGML DTDs publicly available could tempt you to consider converting an SGML DTD into XML. I would advise you not to this, or at least not until you have learned enough about XML DTDs to appreciate and understand the differences. There are many features that SGML allows in a DTD that XML does not (and some of these features are so basic that they are used almost routinely) that you may at best be forced to remodel the structure in an XML approximation. At worst, you could even find it necessary to take a step backward (if you are lucky enough to have this option) and change the SGML model to make conversion possible. We will look at the problems of converting SGML DTDs into XML DTDs tomorrow.

Developing a DTD from XML Code

The easiest, quickest, and simplest method of creating an XML DTD is to create an XML document and work backward. Listing 6.6 shows a simple Web page with basic markup.

Listing 6.6 A Basic Web Page

```
1:   <?xml version="1.0"?>
2:   <page>
3:     <head>
4:       <title>My Home Page</title>
5:     </head>
6:     <body>
7:       <title>Welcome to My Home Page</title>
8:       <para>
9:         Sorry, this home page is still
10:        under construction. Please come
11:        back soon!
12:      </para>
13:    </body>
14:  </page>
```

Given the XML document shown in Listing 6.6, we can already sketch out the rough hierarchy of the elements, where I have used spaces to show the hierarchy:

```
<page>
        <head>
                <title>
        <body>
```

```
<title>
<para>
```

If we now transpose this into DTD syntax, we arrive at an approximation like the following:

```
<!DOCTYPE page [
     <!ELEMENT page (head, body)>
     <!ELEMENT head (title)>
     <!ELEMENT body (title, para)>
]
```

> Note that I have drafted this DTD as it would appear as an internal DTD subset at the start of the XML document. Keeping the DTD inside a document during its development can save you a lot of file swapping until you are sure that the DTD works. You can move the DTD to an external file once it has been finalized.

All I have said is this:

- The page is the root element.
- The page element consists of a head followed by a body.
- A head element contains a title element.
- A body element contains a title element followed by a para element.

This is the hierarchy of the elements. I now need to add occurrence indicators to specify any restrictions to the order of appearance of the elements and how many times they may appear (if at all). (Don’t worry about the occurrence indicators—you will learn all about them tomorrow.)

Without leaping too far ahead into the details of DTD development, there are a few things we can already say about element modeling. In the beginning you should try to leave things as open as you can. Placing unnecessary restrictions will make the DTD harder to use and may force you to change it later on. It is better to go from an unrestricted, open model to a tight, closed one than the other way around. The following might be what you’d want to aim at, for it ensures that there is a title inside a head, and at least one para inside the body:

```
<!ELEMENT page (head, body)>
<!ELEMENT head (title)>
<!ELEMENT body (title?, para)+>
```

You should keep things simple at first, and you might do well to leave the DTD like this:

```
<!ELEMENT page (head, body)>
<!ELEMENT head (title)>
<!ELEMENT body (title, para)*>
```

All I have done is add the '*' indicator to my content model for the body element. This is the *optional repeatable* indicator, which means that a body can be empty or it can contain an endless string of title and para elements in any order.

To complete the DTD, all we have to do is fill in the terminal (leaf) elements, those that contain the actual text or at least don't contain any more elements. The best thing to do is declare these as being text:

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT para  (#PCDATA)>
```

Don't worry too much about the details of this right now—you'll see this in some detail tomorrow. All I'm trying to do now is give you a little feel for the major steps of DTD development.

Creating a DTD from an *instance* (a single XML document) like this goes completely against the grain for most SGML diehards, but SGML was primarily designed for longevity. The whole idea was that you could mark up your data in SGML and preserve it for a very long time, long beyond the lifetime of any one software package or any one computer system. XML is the other side of the coin—it's about information delivery right now, with a maximum lifetime that may be measured more in months than in years.

Granted, if you're intending to use XML as the back-end of a database, you have to think in terms of longer periods than a few months. In many ways, though, the database itself could probably give you most of the DTD design for free without you having to do much extra work (you'll learn about database modeling in DTDs tomorrow).

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Creating the DTD by Hand

Although tools that will make the task of creating a DTD much simpler are starting to appear, there's little substitute for doing it with a pencil and a piece of paper.

My personal preference is to use a technique loosely based on the storyboarding techniques used in the film industry for planning a shooting sequence. The important thing is that you work visually, identifying the elements, arranging them into a structure, and then adding all the additional details such as the attributes. Once everything is in place you can simplify the structure, collecting common attributes and sub-element structures into parameter entities.

> The temptation to make a DTD as sophisticated as possible can be very strong. Bear in mind that a DTD isn't meant to be a showplace for your skill as a DTD developer. Think of the poor souls who have to try to understand what you created and why (this could also be you several years later). You should therefore resist the temptation to use advanced features such as parameter entities or, if you can justify their use (for example, on the grounds of ease of modification), you should document them clearly.

Whatever you decide to use to help you develop your DTDs (there are many different modeling and notation schemes in circulation), there's a good chance that you will eventually develop your own preferred method. The following sections describe the major steps and particular aspects of a DTD that I find help me.

Identifying Elements

For relatively simple Web pages, and some of the simple applications that we have discussed so far, simply typing out the intended XML document and then marking it up will most likely give you a flying start on developing your DTD. Before you do, though, make sure you have a clear idea of what you want to achieve with the DTD. You will learn about this in more detail tomorrow, but you

need to be aware of what kind of markup you want to support. It can be

- **Content**—Here you are trying to describe what the information represents. You would then be looking for abstractions that represent real-world objects, such as part numbers and house addresses.
- **Structure**—Here you are more concerned with grouping elements such as lists, paragraphs, and tables. These are elements that break the information into units but do not really add anything informational.
- **Presentation**—Here you are concerned only with the way things look. You should think of line breaks, horizontal lines, and character attributes such as blinking and underlining. Avoid them.

Avoiding Presentation Markup

As far as generic markup is concerned (and the portability of your XML documents) presentation elements are the worst kind of element and should be avoided as much as possible.

Where you feel you need some kind of typographic embellishment in an XML document, like boldface, try to relate it to a function by asking yourself why you want it bold in the first place. Is it a keyword? Well, then, call it a KEYWORD element.

Compare this fragment that uses strictly presentational markup:

```
<para><bold>This is a <em>very</em> important aspect.</bold></para>
```

with this fragment that uses a more structural, semantic approach:

```
<note><para>This is a <emphasis>very</emphasis> important
aspect.</para></note>
```

Note that by creating a separate note element I can actually increase the flexibility of the code. If I decide that the text isn't that important after all, I can simply "unwrap" the para element. I can let the note elements determine the appearance of the text, and identifying them as information units makes them much easier to find and deal with than seemingly arbitrary format instructions.

While you should be able to get rid of all the purely presentation elements, there will inevitably be some left. A line break element might be useful (although this should really be more properly solved by inserting a processing instruction addressed to the formatting or page layout application), but some of the more familiar candidates that you might think you'd need aren't worth it.

An example of this kind of presentation element is the horizontal line element, HR, in HTML. Yes, you can do some neat things with horizontal lines, but you should be asking yourself where you actually use them. If you can couple a presentation feature to an element, like putting a horizontal line before the start of a section, or you can link it to a context, like indenting paragraphs inside lists, you should think carefully about whether you could achieve the required effect by using a style sheet of some kind. For example, this XML code fragment

```
<section>
          <title>Section 4</title>
          <rule/>
</section>
```

could be much better written like this:

```
<section>
```

```
            <title type="rule under">Section 4</title>
    </section>
```

but you could probably simply write this and let the style sheet handle all of the presentation-oriented aspects:

```
    <section>
            <title>Section 4</title>
    </section>
```

Structure the Elements

Having identified and named the elements in your XML document, the next step is to arrange the elements into some kind of hierarchical (tree) arrangement. The complexity of the tree that you make will largely depend on your application. If you're modeling a database, then you might want to keep the hierarchy fairly flat. At the very least, though, you must obey the rules for setting up well-formed elements and have just one root element that contains all the others. You will probably find that a pencil and lots of paper for sketching the structure will be a great help (you will learn some of the modeling techniques tomorrow). I prefer to scribble the name of each element on a yellow Post-It note and use a large surface like a wall to position them hierarchically. I find this allows me to shuffle elements around much more easily and if I do need to get a global picture I can literally take a few steps back and view the structure from an objective distance.

While structuring, look for group and container elements. Group elements are things like lists, definition lists, and glossaries that arrange sets of elements as units. Sometimes you will need to consciously create extra elements to collect elements so that it's easier for you to process them. For instance, a container list for a set of numbered paragraphs makes it easier to number the paragraphs and reset the number each time you start a new set of numbered paragraphs.

Previous | Table of Contents | Next

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

A useful clue that you need a container element is you find yourself thinking of a set, a list, or a group of elements. If that's the case you should automatically start thinking container.

Examples of container elements are the HEAD and BODY elements in HTML, or perhaps separate MESSAGE.HEADER and MESSAGE.TEXT elements for an email message DTD.

A rough general rule for containers is to look at the content model for the element. If you can't easily understand what it means (try reading it out loud) without having to stop and think, it's too complex and you should consider breaking it up with a container element. You will also find that these containers also give someone who's creating an XML using a DTD that implements containers a lot more flexibility in moving elements around. Consider Listing 6.7, which shows a fragment of an XML document that uses several containers to create a list.

Listing 6.7 A Typical XML Fragment Using Container Elements

```
1:   <para>There are several techniques:</para>
2:   <list>
3:      <item>
4:           <body>
5:                <para>Blame someone else</para>
```

```
6:           </body>
7:      </item>
8:      <item>
9:           <body>
10:               <para>Pretend it wasn't you</para>
11:          </body>
12:     </item>
13:     <item>
14:          <body>
15:               <para>Be honest</para>
16:               <para>Not advisable!</para>
17:          </body>
18:     </item>
19:  </list>
```

Note how breaking an item into a body that contains para elements allows the author to move para elements between items without having to worry about constantly wrapping and unwrapping elements. This structure would also make it much easier to nest lists within lists and so on.

Enforce the Rules

So far, you should have been concentrating on simply getting everything identified and organized. Only when these stages have been completed should you start to think about enforcing your model, making elements optional and required. As I suggested earlier, think carefully about how strict you want to be and how much validation you need. If your XML code is to be computer-generated, perhaps from a database, there probably isn't much point in wasting time and energy in tightening up the DTD. After all, you then have complete control over the generation of the XML markup. On the other hand, if the XML markup is to be created by humans, you will probably want to make sure that certain elements aren't forgotten.

Be careful when you make elements optional that you don't make the content model ambiguous.

> Remember that XML processors aren't very complex and are unable to look ahead at what comes next in the XML document. At each point in the content model, it must be absolutely clear to the processor what element is allowed next based on what is visible at that point.

Sometimes, as you will learn tomorrow, you will have to use some pretty clever tricks to get round these ambiguities.

Assigning Attributes

Once you have arranged your elements into a hierarchical structure and grouped them as necessary, you can assign attributes to them (size, color, ID, and so on). You may find that you want to move some of the information into attributes, which is why it helps to keep the two tasks separate.

There are no real rules about when to use attributes and when to use elements, although there is a lot of discussion about it. You will learn more about it tomorrow, but for now you should simply go by feel and common sense. I, for example, have a head and a height. It's a pretty easy choice to say that my head should be an element, a physical part, and my height, which is not something I can hold in my hands, should be an attribute. You could therefore model this in the XML DTD something like the following:

```
<!ELEMENT me (head)>
<!ATTLIST me height CDATA " ">
```

The matching XML document code would then look something like the following:

```
<me height="6 feet">
     <head/>
</me>
```

Tool Assistance

There are already a number of very good XML editors available. Although some of them are still in the early stages of development and it's still unclear what features are actually required by users, a trend is apparent. In less than a year, along a path that very roughly parallels that followed by HTML tools, we have moved from markup-aware text editors to a sort of word processor package attuned to editing XML code.

The XML editors that are now appearing on the market are beginning to add in dedicated XML capabilities such as validation.

XML Pro (http://www.vervet.com) and Stilo WebWriter (http://www.stilo.com) are both excellent. XML Pro provides a superb and cheap editing environment that allows you to edit and validate XML documents. WebWriter goes one step further and allows you to actually generate a raw DTD straight from the XML document. I predict that more and more XML tools will incorporate this facility, as it's reasonably easy to extract at least the core of a DTD from the markup in an XML document.

A Home Page DTD

You have now learned the basics of XML DTDs and DTD modeling. It's time to put this knowledge to use and create a DTD for a very simple type of document, a beginner's Web home page. We'll reuse the example we used in Day 2, shown again in Listing 6.8.

Listing 6.8 A Typical XML Instance

```
1:  <?xml version="1.0"?>
2:   <home.page>
3:     <head>
```

```
4:       <title>
5:         My Home Page
6:       </title>
7:       <banner source="topbanner.gif"/>
8:     </head>
9:     <body>
10:      <main.title>
11:        Welcome to My Home Page
12:      </main.title>
13:      <rule/>
14:      <text>
15:        <para>
16:          Sorry, this home page is still
17:          under construction. Please come
18:          back soon!
19:        </para>
20:      </text>
21:    </body>
22:    <footer source="foot.gif"/>
23:  </home.page>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

If we pull this markup into the XML Pro editor, without a DTD (see Figure 6.3) we can then add elements, delete them, and move them around as much as we want. Once you've finished playing and are satisfied with the results, you can use the tree structure shown in the left window to see the way you have structured things. You can easily explore your tree by expanding and collapsing branches. The tree structure gives you a good head start on describing the structure you want in your DTD.

Figure 6.3 *Exploring the structure in XML Pro editor.*

If you pull the same XML markup into Stilo WebWriter, it isn't quite as easy to navigate the structure as in XML Pro, but WebWriter compensates by allowing you to actually generate the DTD. See Figure 6.4.

Figure 6.4 *Creating the DTD in Stilo WebWriter.*

Let's see what WebWriter makes of our simple home page. Listing 6.9 shows the extracted DTD (I have added a few blank lines and spaces to make it a little more readable).

Listing 6.9 The Extracted DTD

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

```
2: <!ELEMENT home.page  (#PCDATA | head | body | footer)* >
3:
4:  <!ELEMENT head    (#PCDATA | title | banner)* >
5:
6:  <!ELEMENT title    (#PCDATA) >
7:
8:  <!ELEMENT banner     EMPTY >
9:   <!ATTLIST banner     source CDATA “” >
10:
11:  <!ELEMENT body    (#PCDATA | main.title | rule | text)* >
12:
13:  <!ELEMENT main.title (#PCDATA) >
14:
15:  <!ELEMENT rule     EMPTY >
16:
17:  <!ELEMENT text    (#PCDATA | para)* >
18:
19:  <!ELEMENT para    (#PCDATA) >
20:
21:  <!ELEMENT footer     EMPTY >
22:  <!ATTLIST footer     source CDATA “” >
23:
24:  <!ENTITY lt “<”>
25:  <!ENTITY gt “>”>
26:  <!ENTITY apos “‘”>
27:  <!ENTITY quot “””>
28:  <!ENTITY amp “&”>
```

Obviously, WebWriter has sensibly played it a little safe in allowing every element to have mixed content (#PCDATA is allowed in all the content models), and it was wise not to guess about the nature of the attributes, but what you have is a good framework. All you need to do is tighten up the content models and fill in the attribute declarations. One possible end result is shown in Listing 6.10.

Listing 6.10 The Finalized DTD

```
1: <!ELEMENT home.page  (head | body | footer) >
2:
3:  <!ELEMENT head    (title | banner?) >
4:
5:  <!ELEMENT title    (#PCDATA) >
6:
7:  <!ELEMENT banner     EMPTY >
8:  <!ATTLIST banner
9:            src CDATA  #REQUIRED
10:           alt CDATA #IMPLIED
11:           align (top | middle | bottom) #IMPLIED >
12:
13:  <!ELEMENT body    (main.title |rule | text) >
14:
15:  <!ELEMENT main.title (#PCDATA) >
16:
```

```
17:  <!ELEMENT rule     EMPTY >
18:  <!ATTLIST rule
19:            align (left|right|center) #IMPLIED
20:            size  NMTOKEN #IMPLIED
21:            width CDATA #IMPLIED >
22:
23:  <!ELEMENT text    (#PCDATA | para)* >
24:
25:  <!ELEMENT para    (#PCDATA) >
26:
27:  <!ELEMENT footer     EMPTY >
28:  <!ATTLIST footer
29:            src CDATA  #REQUIRED
30:            alt CDATA #IMPLIED
31:            align (top|middle|bottom) #IMPLIED >
32:
33:  <!ENTITY lt “<”>
34:  <!ENTITY gt “>”>
35:  <!ENTITY apos “‘”>
36:  <!ENTITY quot “””>
37:  <!ENTITY amp “&”>
```

Note that, in accordance with the XML specification, now that we are using a DTD, we have included the declarations of the default character entities (lt, gt, apos, quot, and amp).

Summary

You learned about the XML DTD and how it’s divided into two subsets, an internal subset that is included in the XML document and an external subset that is contained in an external file. You have also learned how to reference the external DTD subset using a public identifier and a system identifier.

You have been introduced to some of the factors governing whether or not to use a DTD, and whether to include it externally or internally, and you are should be fairly familiar with the major content of a DTD.

You should now have an idea of the basics of DTD modeling and a few of the problems that you may encounter. Tomorrow we will pick up the DTD thread again and examine modeling and its problems in some detail. Don’t let DTDs scare you off—they are complex, but they can also be very, very simple. If all else fails, this is still a perfectly acceptable, valid XML document with an internal DTD subset and it might be all you need:

```
<?xml version=“1.0” standalone=“yes”?>
<!DOCTYPE DOC [
<!ELEMENT DOC (#PCDATA)>
]>
<DOC>I can now put all this text in
my document and forget about any other
markup. </DOC>
```

Q&A

Q Is there any limit to the length of an element or attribute name?
A No, not in XML. As long as you obey the restrictions on what characters you are allowed, a name can be as long as you want it to be (in SGML you have to make special arrangements—you have to modify the SGML declaration—to allow longer names).
Q How many elements should you have in a DTD?
A It all depends on your application. HTML has about 77 elements, and some of the industrial DTDs have several hundred. It isn't always the number of elements that determines the complexity of the DTD. By using container elements in the DTD (which add to the element count), authoring software is able to limit the possible choices and automatically enter required elements. Working with one of the major industrial DTDs can actually be far easier than creating HTML! Because HTML offers you more free choice, you have to have a much better idea of the purpose of all the elements rather than just the ones you need.
Q Do you have to validate your XML documents?
A No, but unless you are certain that your documents are valid you cannot predict what will happen at the receiving end. Although the XML specification lays down rules of conduct for XML processors and specifies what they must do when certain invalid content is parsed, the requirements are often quite loose. Validation certainly won't do any harm—though it might create some extra work for you.
Q I can validate my XML documents, but how do I check that my DTD is correct?
A Simply validate an XML document that uses the DTD. There aren't as many tools specifically intended for checking DTDs as there are for validating documents. However, when an XML document is validated against the DTD, the DTD is checked and errors in the DTD are reported.

Exercises

1. Create a content-based XML DTD for a simple email message.
2. Create (part of a) DTD to deal with lists. Try creating two versions of the element structures, one using an attribute to describe the type of numbering (for example, bulleted or numbered), and one that uses separate item elements (such as numberlist and bulletlist). Now compare the two and consider the two versions and their advantages and disadvantages relating to ease of authoring and ease of formatting via a style sheet.

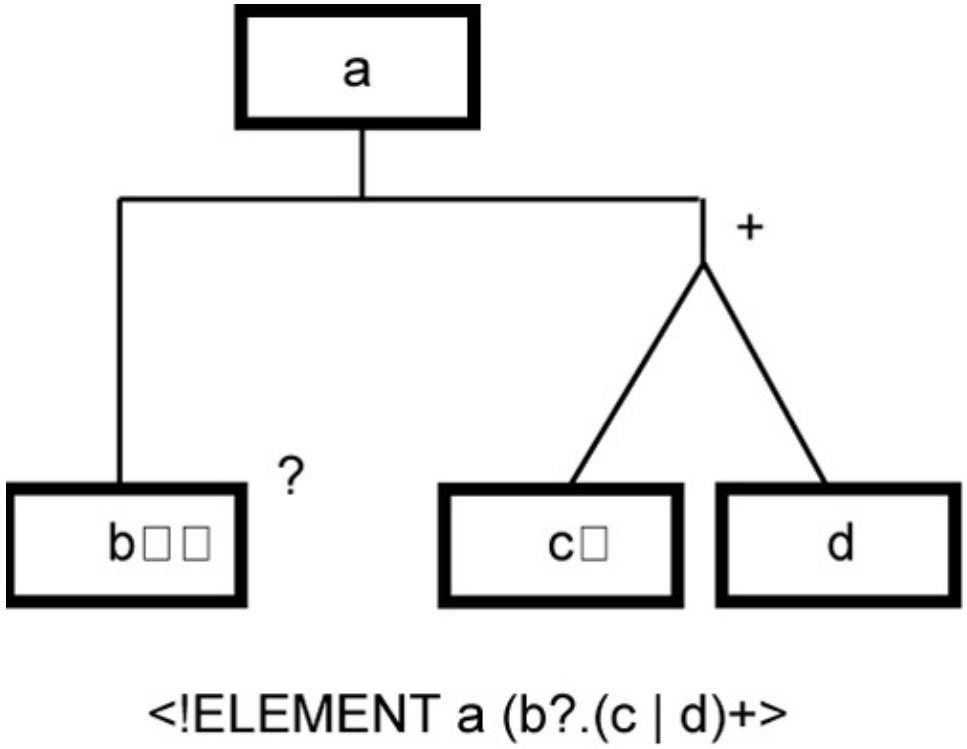
To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 7
Developing Advanced DTDs

Yesterday you learned about XML DTDs and mastered the basics of DTD development. Today's chapter will expand and deepen your knowledge of DTDs. You will do the following:

- Explore some of the issues concerning DTD development and information modeling raised yesterday.
- Look at some of the more technical aspects of DTDs.
- Learn some new features and tricks that will make creating a DTD much easier.



A word of warning: I added the word *advanced* to the title of this chapter with the express intention of scaring you off. So far in this book I have assumed you have far more knowledge of HTML than I would have liked. I've also made a lot more reference to SGML than I should have. But when it comes to DTDs, XML comes very close to SGML and it's hard to ignore it completely.

Although this chapter doesn't require you to know anything about SGML, it might make things clearer if you did.

Information Richness

Yesterday you learned that the content and complexity of a DTD is pretty much determined by the application. Essentially, the XML document and its markup follow what I call the Law of NINO: *Nothing In, Nothing Out*. This means that if you want to do something with a piece of information in an XML document, you must identify it. It doesn't matter whether you mark it up using elements or attributes. However, if you haven't marked it up, you either won't be able to find it all or, if you *do* find it, you won't be able to do anything with it.

Here's another acronym I invented to help bring the point home: YCEWIT, or You Can't Extract What Isn't Tagged. It's pronounced "you sow it," harkening back to the Biblical admonition that you shall reap what you sow.

Humor aside, a DTD that contains a lot of (informative) markup is called a *rich DTD*. As the DTD designer, you will always face a difficult compromise between the complexity of the DTD and the information-richness of XML documents that conform to that DTD. When humans are the authors of those documents, the complexity of the DTD can become a major obstacle. Always choose the richest information model you can:

- It is far easier to throw away information than it is to add it afterwards.

Consider what happens when you go from XML, or even HTML, to plain text with no markup at all. It's a very simple conversion to make because you're moving from a higher level of information richness to a lower one. However, going the other way is not easy at all. The same applies for moving from SGML to XML, and from XML to HTML. At each conversion you throw information away, moving from an information-rich level (SGML) to a pretty information-poor level (HTML).

- You cannot always account for all media and all intended uses. No matter how carefully you research your application in advance, there will always be something you haven't thought of.

With SGML, often it's only after you've got all your data marked up that you really start to appreciate what you can do with it. If you limit yourself to the minimum

necessary for the current application, you may design yourself into a corner and have no room left for expansion.

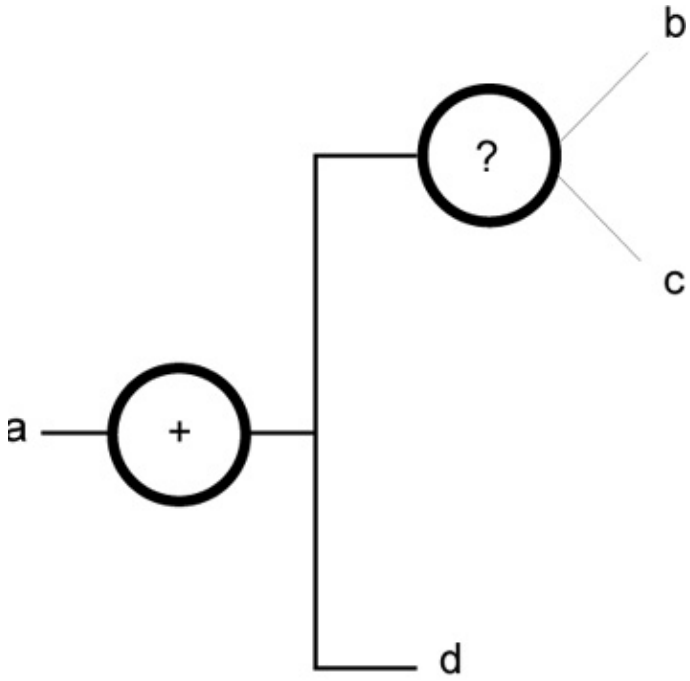
As in mechanical systems, information objects are subject to a sort of entropy law. In mechanical systems, the law of entropy says that a system will always tend to become more disorganized. Release some gas into a room, and after a while its molecules will be pretty evenly scattered. The chances of all the molecules magically finding their way back into the original container are very small (although it's actually possible at the quantum level). In the same way, the information-richness of an XML document will always have a tendency to slide down the slippery slope into garbage (high information entropy). The higher up the slope you start (the richer your start point), the longer you can fight off the decline.

- Remember the curse of legacy data.

In SGML circles, data that either hasn't been marked up or has been marked up to an earlier version of a DTD is called *legacy data*. It's the stuff that's dumped on you, like the Biblical sins of the fathers being visited on the sons. It's also often a far greater problem than any of the new material you produce.

Although you can achieve a lot by remapping attributes (as you will learn on Day 11, "Using XML's Advanced Addressing"), you are more or less stuck with the elements you started with.

There is a proposal to implement a feature taken from HyTime called *architectural forms* for XML, but it is unlikely that the proposal will be adopted. Unfortunately, whereas SGML is now understood by quite a large number of people, even the eyes of SGML gurus have been known to glaze over at the very mention of HyTime. Architectural forms are not the easiest things to understand, and there is still very little software that can support them.




```
<!ELEMENT a ((b | c)? d )+>
```

Don't underestimate the speed with which you will amass legacy data, and don't underestimate the longevity of your data. It's all too easy to start out with a DTD that works, thinking that it's only small-scale and will be superceded in a few months. If you do this, you're opening the door to a lifelong career as an editor.

You'll have to go back to modify every XML document you have created so far, and just when you think you've finished, something will change and you'll have to repeat the exercise all over again. This is the curse of legacy data.

Visual Modeling

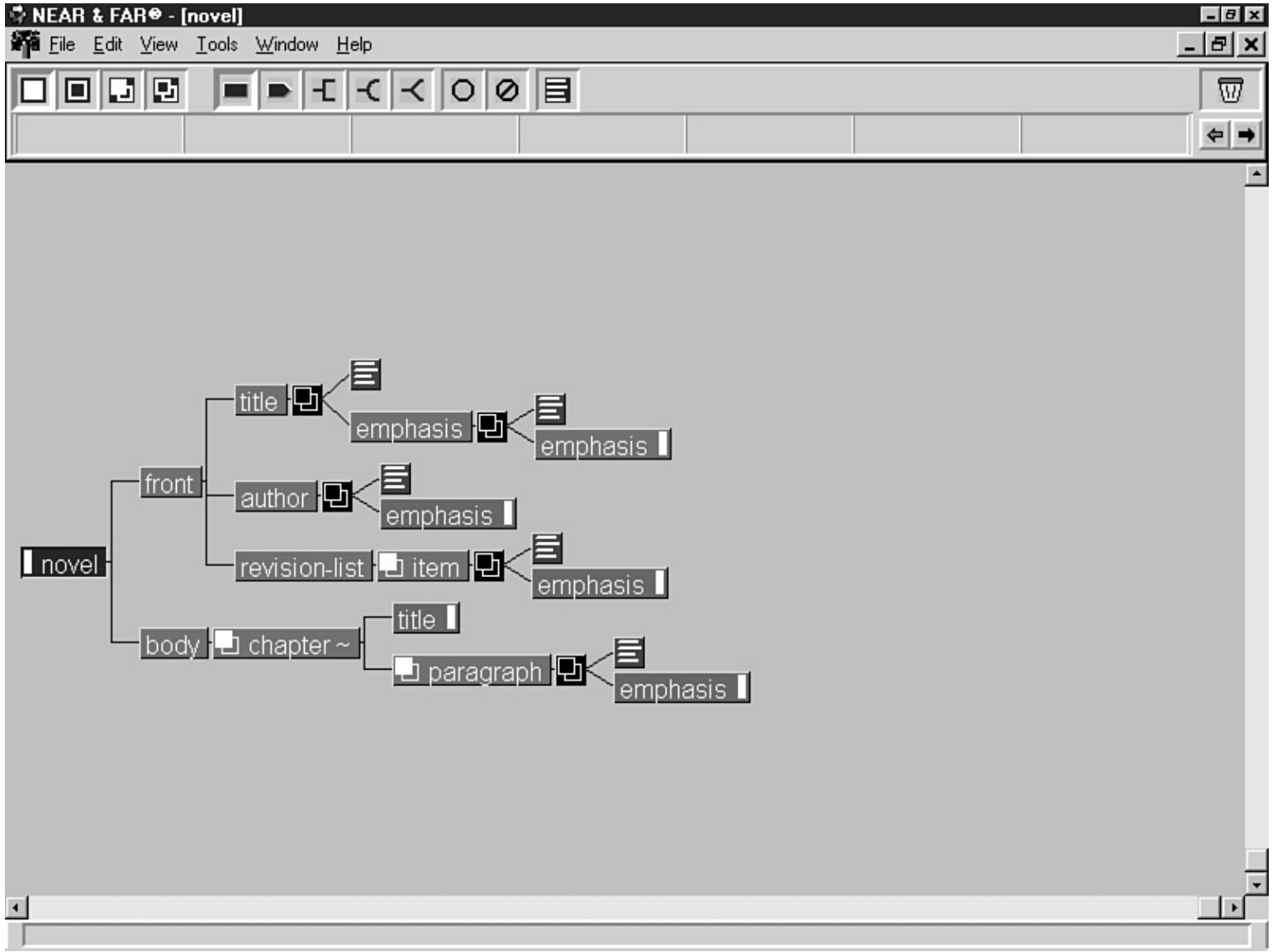
Yesterday you learned that one of the basic DTD design process steps is to construct the hierarchy—use the content models in the DTD to arrange the elements in an XML

document that uses the DTD into a hierarchical tree-like structure.

There are many information-modeling techniques, but detailed discussion of them falls outside the scope of this book. The important thing is that all of them recommend some kind of visual aid to help you. The same applies for DTD development, but each expert has his or her own set of symbols. Some prefer to model information using the blocks so familiar from flow diagrams, as shown in Figure 7.1.

Figure 7.1 *Block diagram DTD modeling.*

An alternative is the somewhat sparser representation form shown in Figure 7.2.



Note that both types of modeling graphics try to represent the sequence and OR models (a model in which there are two or more alternatives and one must be chosen) by using different kinds of connecting lines between the objects (diagonal line for an OR choice, straight line for a simple relationship, and rectangular connections for sequences). This type of graphics modeling is also incorporated into the top-of-the-line SGML DTD modeling package, Microstar's Near & Far (http://www.microstar.com), an example of which is shown in Figure 7.3.

Figure 7.2 *Simplified diagram DTD modeling.*

Figure 7.3 *Visual modeling in Near & Far.*

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

It is difficult to think of a better way of modeling DTDs visually than these kinds of models, especially in a software package where you can expand and collapse branches of the DTD tree as you work on them. However, as the DTD gets larger and larger, it becomes harder and harder to maintain a sensible overview. You shouldn't forget that these are primarily SGML tools, and people using them are mostly working on a completely different scale than is probably useful in XML. Whereas an XML application might need a few dozen elements and a couple of hundred lines in a DTD, SGML applications routinely use several hundred elements and several thousand lines of declarations in a DTD.

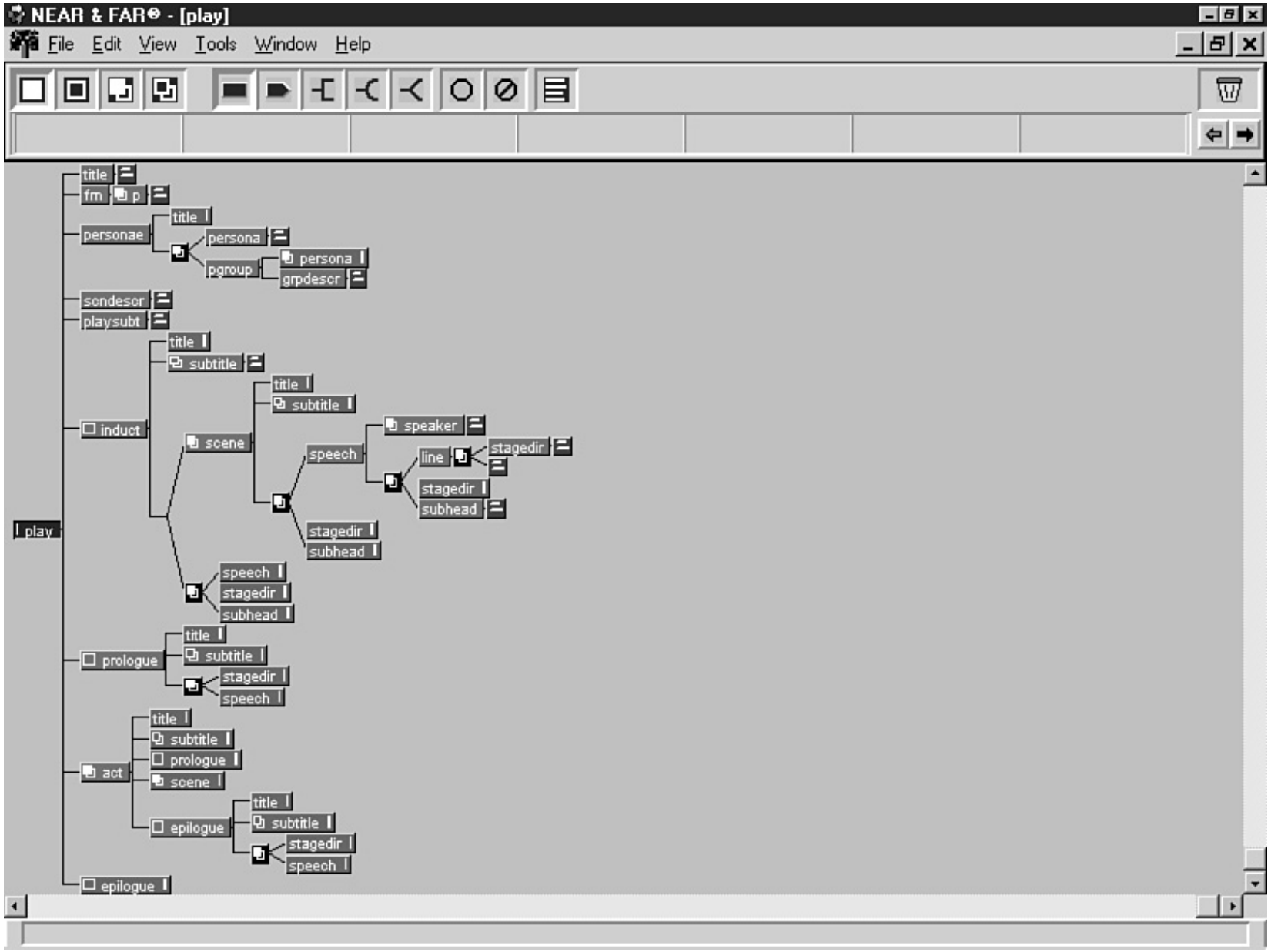
It isn't just the number of elements that makes the DTD bulky, though. Often, elements that have a presentational function (like emphasized text) will be reused in almost every element. When you expand these models, even a well-designed visual presentation can simply become too cluttered to be much use unless you start finding other ways to manage DTDs of this scale or complexity (see Figure 7.4).

Figure 7.4 *Reaching the limits of useful visual modeling.*

There are techniques for mastering DTDs of this scale, of course, such as breaking the DTD into modules of a more manageable size (as you will learn later in today's chapter). But there also comes a point where you must ask yourself whether it is really worth continuing in this way. Figure 7.5 shows a public-domain

software package called EZDTD that gives a sort of tabular display of the elements in the DTD (go to http://www.download.com and search for “EZDTD”).

Figure 7.5 *Tabular modeling in EZDTD.*



All you have here is a list of elements, and you can display the content model for one model at a time. This kind of tool might be all you will ever need for simple XML applications and smaller DTDs, especially as you become more proficient in creating DTDs. Unfortunately, there is no way of displaying any kind of tree to show the relationships between elements. Again, once the scale of the DTD increases beyond a certain point, the usefulness

starts to tail off (see Figure 7.6).

Figure 7.6 *Reaching the limits of usefulness of a simplified form of visual modeling.*

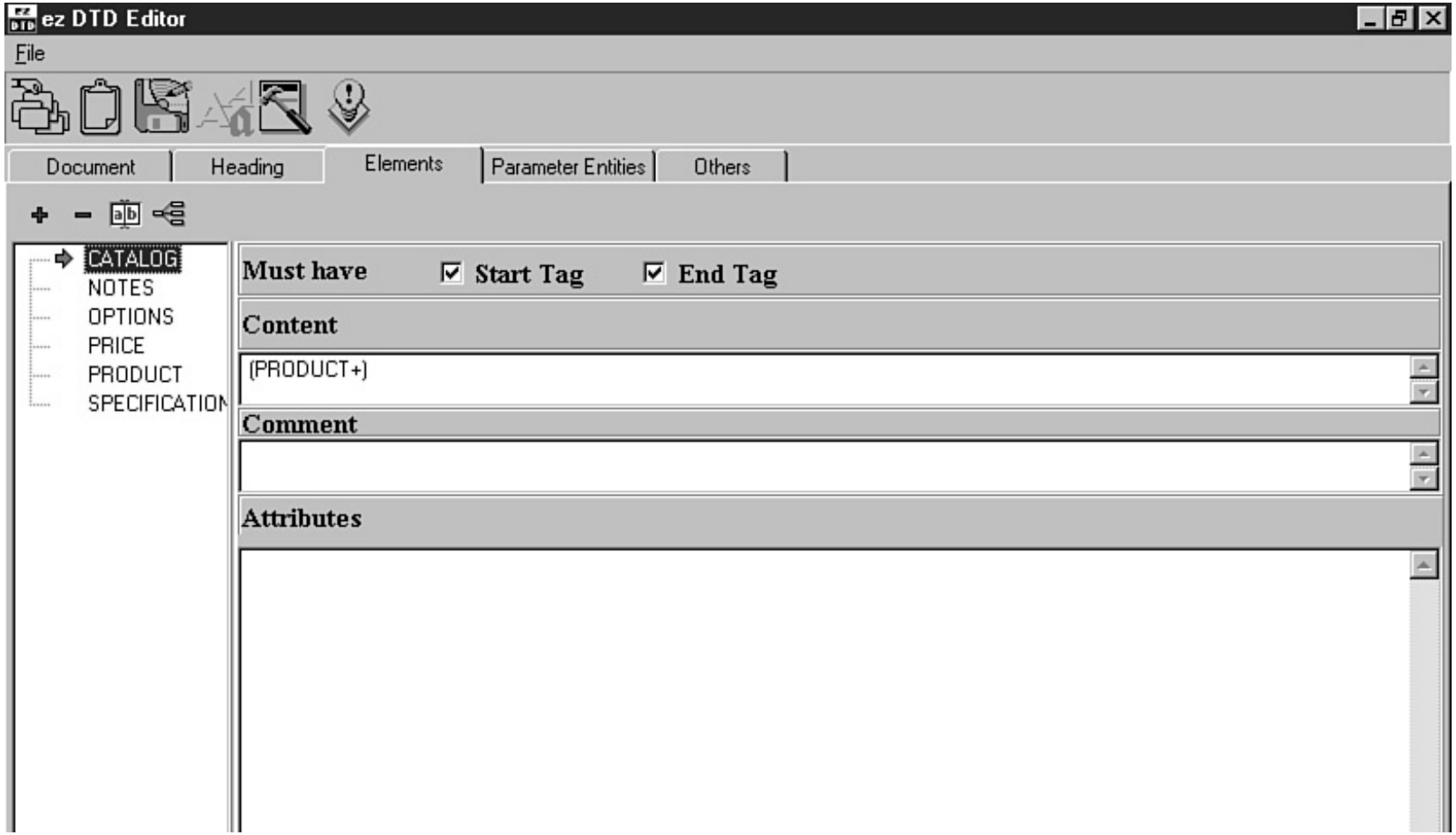
It may make more sense to give up on trying to visualize the DTD altogether, although I can highly recommend Earl Hood's public-domain DTD2HTML package for Perl (http://www.oac.uci.edu/indiv/ehood/perlSGML.html). This package converts a DTD into a set of interlinked HTML pages. The output isn't a great deal of help while you're actually developing the DTD, but it's extremely useful for checking the results of your work and is an excellent way to document the completed DTD.

Instead of visually modeling the DTD, you may ultimately find it easier to use a more conservative package, such as Innovation Partner's DTD Generator package (
http://www.mondial-telecom.com/oddyssee), shown in Figure 7.7.

Figure 7.7 *Creating a DTD with the XML DTD Generator package from Innovation Partners.*

This no-frills package is a sort of halfway point between the two camps. You have all the ease of dedicated buttons that allow you to add the parts of content models quickly, but there is no visual display. Instead, you can very quickly flip between the ease-of-use interface and

the raw DTD code.



XML DTDs from Other Sources

As suggested earlier, one alternative to getting involved in the complexities of DTD design and development is to leave the DTD design to one side and simply work on a representative XML document. You must consider all the likely variations of the document, of course, and you will probably need several documents as your XML application becomes larger. However, once you have explored all the possibilities of the XML document, you can then use tools to create at least the core of the DTD. While this would be unthinkable, even heretical, to attempt with an SGML application, this approach is often a realistic choice for XML applications. As you can see from the real-life document loaded in XML Pro (http://www.vervet.com) in Figure 7.8, a good XML editor will allow you to browse through the structure of the document, adding elements and attributes wherever you need them. Although you may find yourself repeating some steps unnecessarily, it is easy enough to consolidate the multiple declarations afterwards.

Figure 7.8 *Modeling the document before creating a DTD.*

There is also one very significant benefit to this approach that no specialized DTD tool can give you: By exposing you to what amounts to a completed XML document using your (proposed) DTD, it gives you firsthand taste of what it would be like for a human author to work with your DTD.

Yet another way to leapfrog yourself into the middle of the DTD development process is to take a sort of “back door”. Many of the mainstream software packages, such as Adobe’s FrameMaker and even Microsoft’s Office suite, will be supporting XML as an output format in the near future. This opens up all sorts of possibilities for perfecting the structure of an intended XML in one of those software packages and then creating the DTD from that document, or even skipping the DTD altogether.

An interesting development in this direction is the Majix software package from Tetrasix (http://www.tetrasys.fr/ProduitsFrame.htm). This software package (seen in Figure 7.9) allows you to create an XML document from a Microsoft RTF document. For example, if you use Microsoft Word and you can discipline yourself to use styles properly (remember, a DTD is just one way to describe the structure of an XML document), you can use these styles to drive the conversion into XML elements.

Figure 7.9 *Converting RTF into XML with Majix from Tetrasix.*

I’m not much of a fortune teller, but it’s a safe bet that there will be a lot more of these conversion tools in the future, and they will make everyone’s lives a lot easier.

Modeling Relational Databases

In all this discussion of developing DTDs, I have consistently talked of structuring the elements hierarchically. However, sometimes you may not want, or even need, to have too much of a hierarchy and want to keep the model “flat.” For example, you might use XML to model a relational database, an application that XML is ideally suited for (subject to the planned XML data-typing initiatives bearing fruit).

Consider the very simple relational database description shown in Listing 7.1. Although the topic of database modeling in XML deserves far more extensive treatment than I can give it here (there are some excellent books devoted to the topic), even a quick appraisal shows that you don’t really need that much depth at all. In fact, the simple database shown here needs only four levels:

- The database (the root element, music)
- The database tables (the artists and disks elements)
- The database records (the artist and disk elements)
- The database fields (the name, label, title, date, and number elements)

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Listing 7.1 A Simple Database in XML

```
1: <!DOCTYPE music SYSTEM “mymusic.dtd”>
2: <music>
3:   <artists>
4:     <artist>
5:       <name>Cyndi Lauper</name>
6:       <label>Sony</label>
7:       <title>Twelve Deadly Guns</title>
8:     </artist>
9:     <artist>
10:      <name>Kate Bush</name>
11:      <label>EMI</label>
12:      <title>Hounds of Love</title>
```

```
13:    </artist>
14:   </artists>
15:   <disks>
16:     <disk>
17:       <title>Twelve Deadly Guns</title>
18:       <date>1994</date>
19:       <number>EPC 477363 2</number>
20:     </disk>
21:     <disk>
22:       <title>Hounds Of Love</title>
23:       <date>1985</date>
24:       <number>CDP 7 46164 2</number>
25:     </disk>
26:   </disks>
27: </music>
```

Once you have this structure, it's a simple matter to add table and record keys by assigning attributes. In this way you could create a forced division, using the elements to describe the content and structure of the database itself and reserving the attributes for the (internal) data that describes the relationships between the data.

Elements or Attributes?

Separating elements and attributes in an XML database description is just one of the many ways to make a distinction between elements and attributes. This question surfaces regularly in SGML circles and has never really been answered with any degree of conviction. The database example shows one possible answer, but it is just one of many. I have no quick answer either, but I can offer the following thoughts to help you decide which to choose:

- Elements can be remapped using architectural forms, and attributes can be remapped using XLink. From this perspective there is not a decisive reason to select either the element or the attribute (because both can be remapped).
- Historically, attributes have been reserved for intangible, abstract properties such as color and size, while elements are reserved for physical components. To deviate from this usage may confuse people, especially in an SGML context.
- It may be easier to edit elements than attributes with XML editing tools, and attribute values may not be as readily displayed as element content. This would favor elements when dealing with human authors.
- An XML processor can check the content of an attribute value more easily than it can check the content of an element. It can't check an attribute value for much, but it can barely check the content of an element at all.
- The attributes of elements from separate XML documents can be merged quite easily. It can be extremely hard to merge elements, so attributes may have the edge over elements in a distributed environment.
- In collaborative, multiple-author environments, it is reasonably easy to split XML documents into fragments based on the element structure rather than attribute values.
- Sensibly named elements can make authoring easier and less confusing. For example, when someone selects an element to contain a list, it's pretty obvious that she should choose a list element. However, the selection of the correct attribute can be less than obvious and not very visible (you will often see such cryptic attribute values as ordered and unordered, which come from HTML). Choosing container elements instead, like list.numbered

and list.bulleted, may make the author's life much simpler.

You may notice a general trend here. Generally speaking, when human beings are expected to create or work with XML documents, it's better to use elements than attributes.

Saving Yourself Typing with Parameter Entities

You will learn a lot more about entities on Day 9, "Checking Validity." For the moment, we're concerned with one particular type of entity called the parameter entity, which can be extremely useful in XML DTDs.

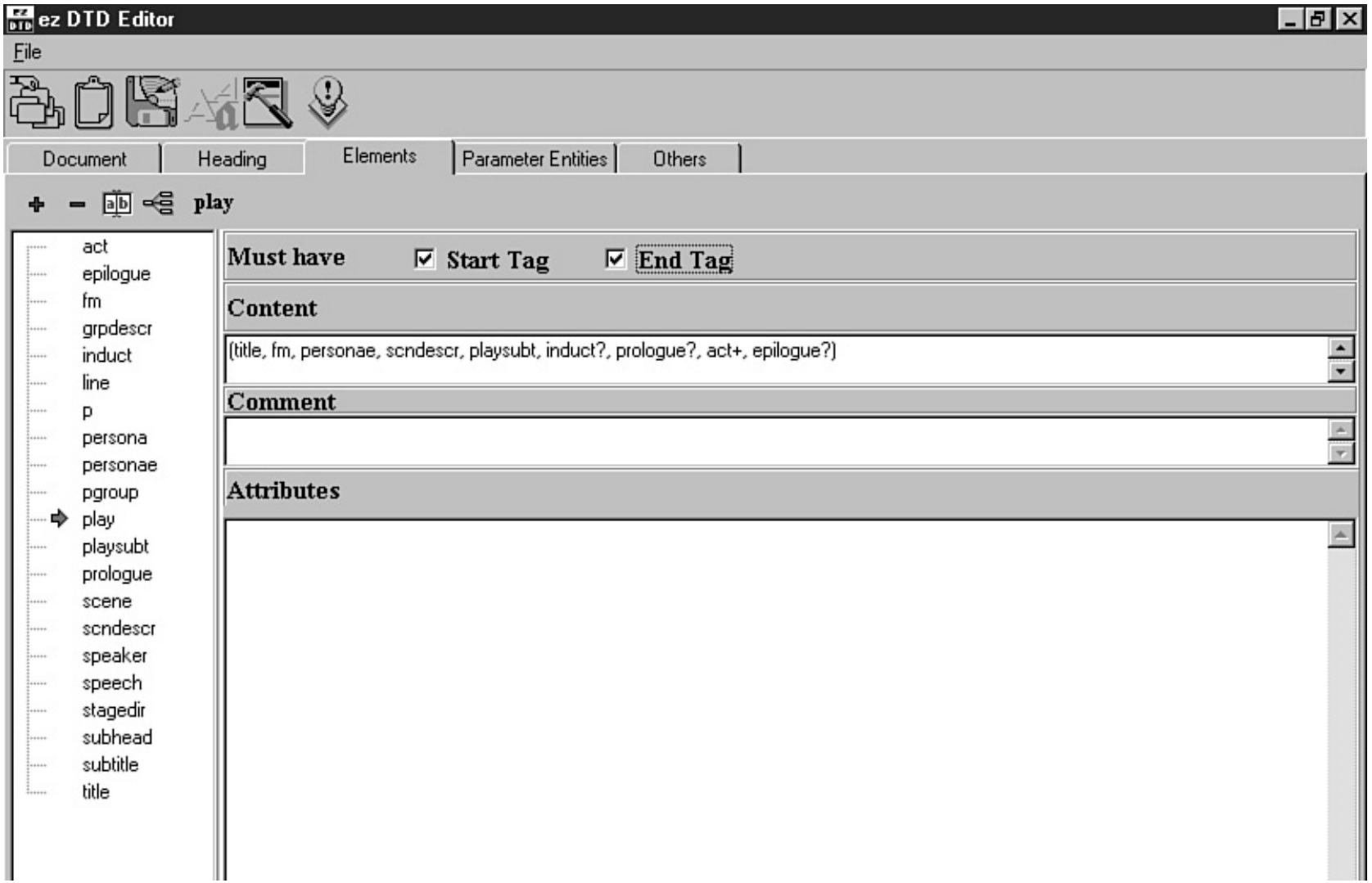
Parameter entities are basically just like the character entities you learned about earlier. They too behave like macros and can be used as abbreviations for strings. However, whereas character entities serve as abbreviations for character strings, parameter entities serve as shortcuts for markup declarations and parts of declarations. Obviously, because they are concerned with markup declarations, they can be used only in DTDs. In fact, their use is even a little more restricted than this:

- In the internal DTD subset, parameter entities can be used only between other markup declarations.
- In the external DTD subset, parameter entities can be used both between and inside other markup declarations.

To make sure there's no confusion between parameter entities and character entities, the syntax for declaring and referring to parameter entities is quite different than the syntax for character entities. A parameter entity uses a percent sign (%), as in this example:

```
<!ENTITY % heading "H1 | H2 | H3 | H4 | H5 | H6" >
<!ENTITY % body.content "(%heading | %text | %block | ADDRESS)*">
<!ELEMENT BODY %body.content>
```

The heading parameter entity saves a lot of typing (note that you can only use this particular trick in the external DTD subset).

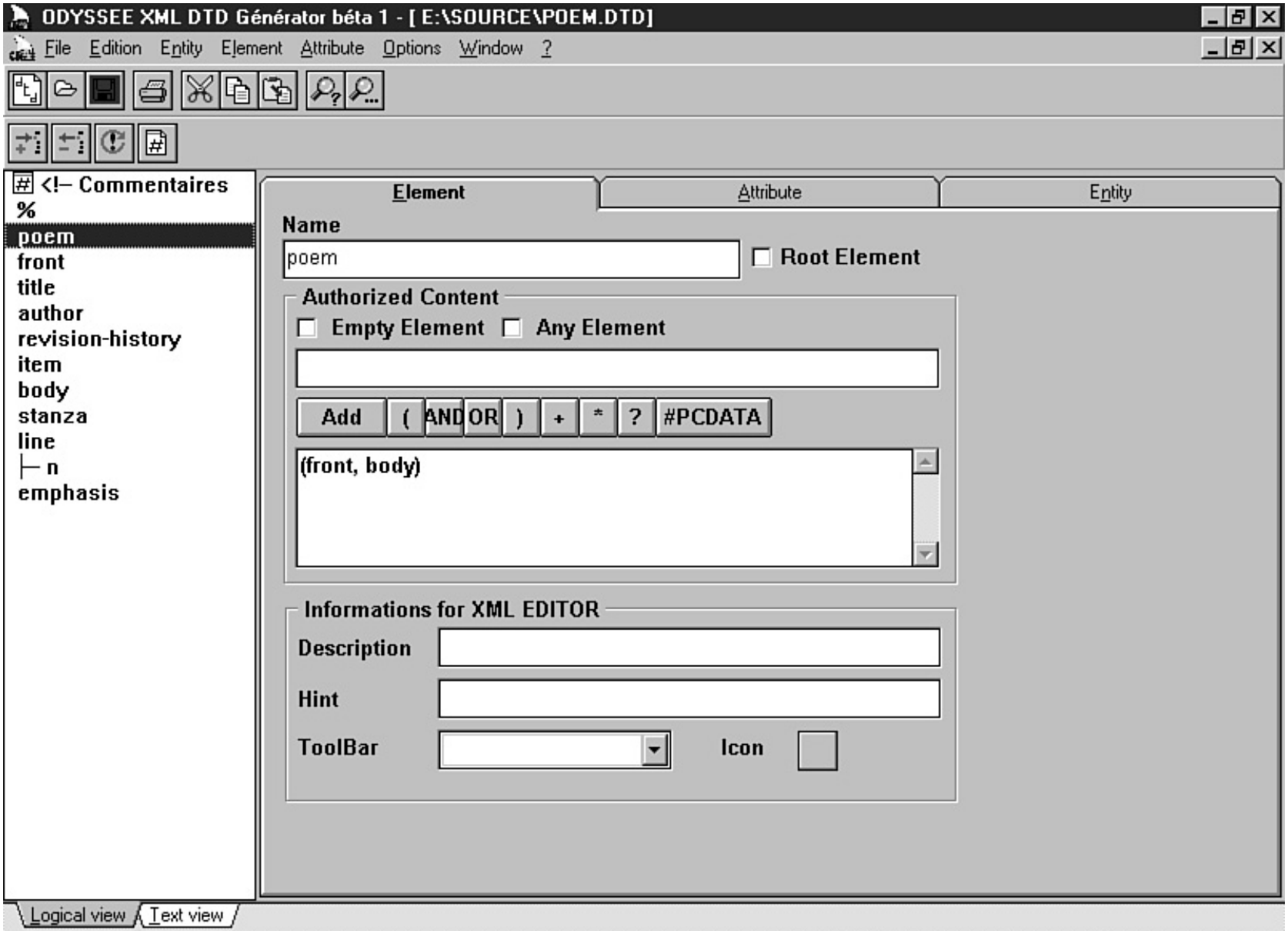


You can also use a parameter entity to make your DTD declarations a little more comprehensible, as in Listing 7.2.

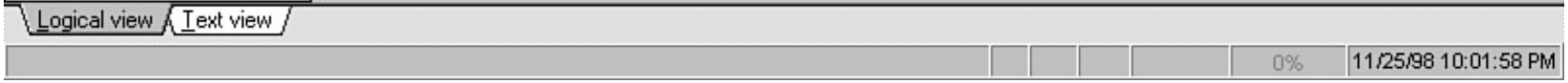
Listing 7.2 Simplifying a DTD with a Parameter Entity

```
1:<!ENTITY % color “CDATA” >
2:
3:<!ENTITY % body-color-attrs “
4:        bgcolor %color #IMPLIED
5:        text    %color #IMPLIED
6:        link    %color #IMPLIED
```

```
7:        vlink   %color #IMPLIED
8:        alink   %color #IMPLIED
9:        ">
10:
11:<!ELEMENT BODY  %body.content>
12:<!ATTLIST BODY
13:          background %URL #IMPLIED
14:          %body-color-attrs; >
```



Both examples shown here use internal parameter entities. No external file is required for the XML processor to determine the contents of the entities. It is also possible to have external parameters, as you will learn next, and these are used for quite different purposes.



Modular DTDs

When I talked about the size and complexity of DTDs earlier in this chapter, I mentioned the possibility of splitting up the DTD into modules. This is another use for external parameter entities. For example, the following DTD fragment declares and then immediately references a set of character entity declarations contained in an external file (this is such a common use for parameter entities that there is an extensive set of public identifiers that covers most of the less commonly used characters):

```
<!ENTITY % ISOnum PUBLIC “ISO 8879:1986//ENTITIES Numeric and Special
Graphic//EN”>
%ISOnum;
```

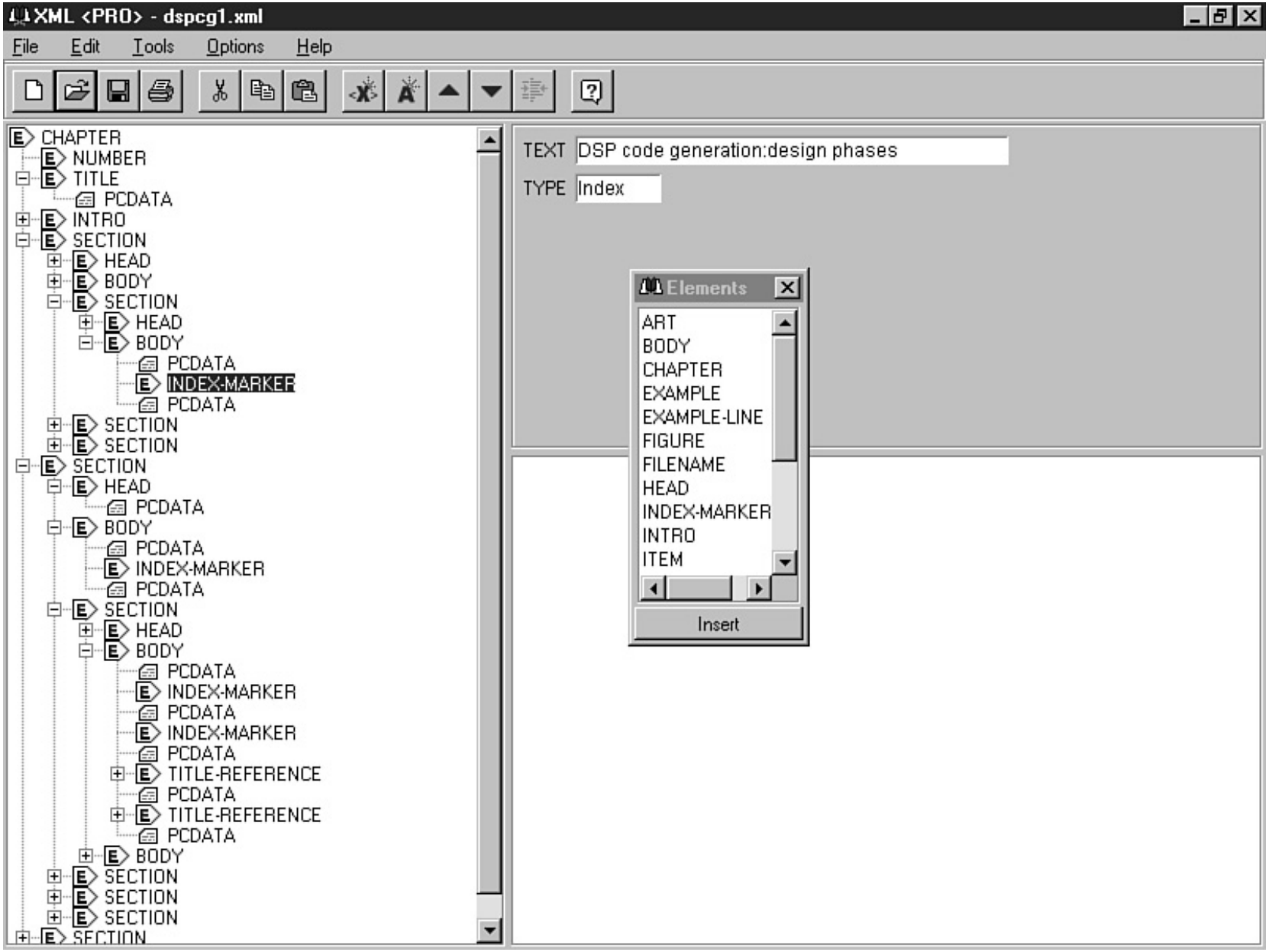
Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

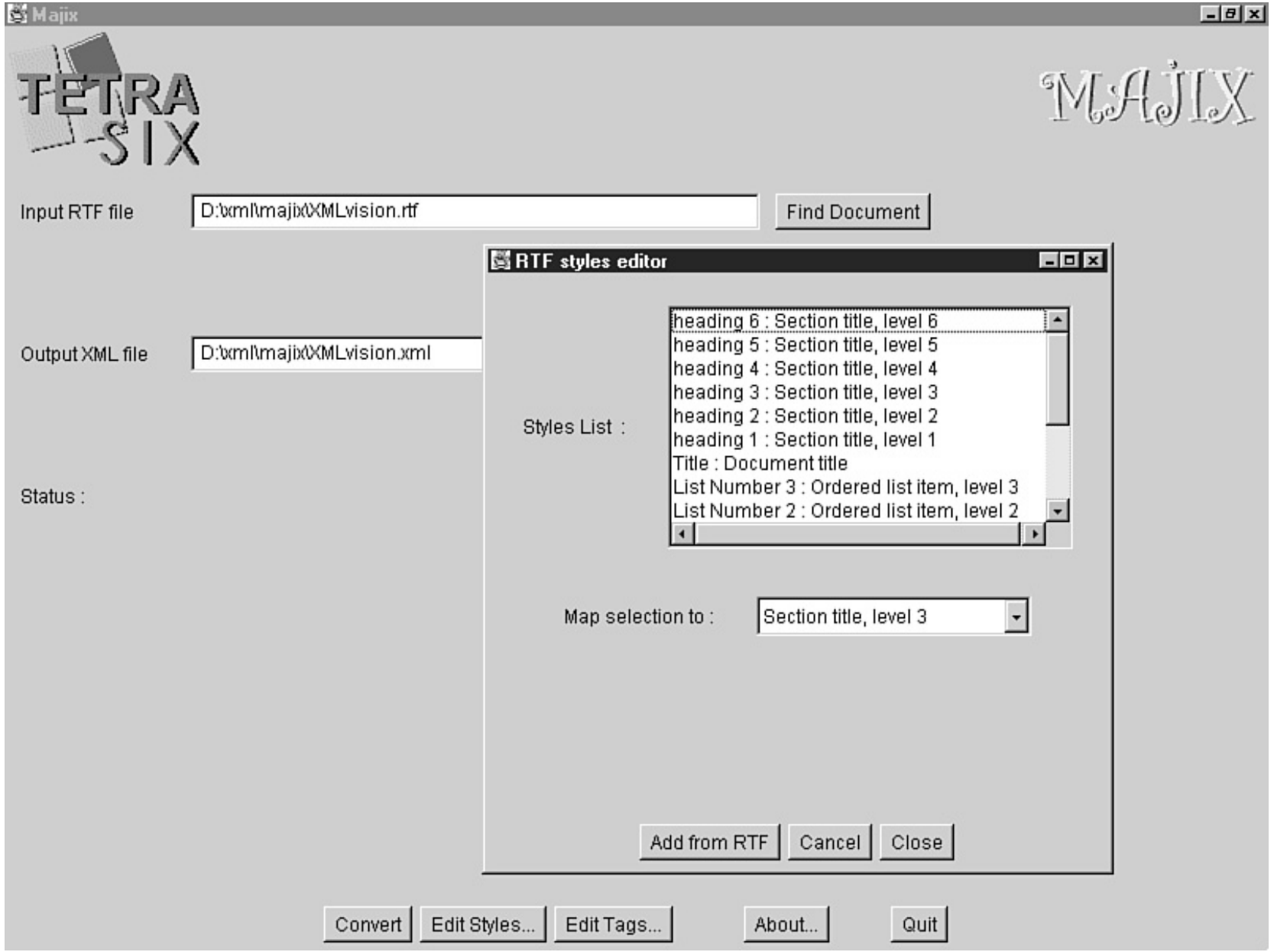
Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

External parameter entities really come in handy when you decide to reuse the results of work that you have invested in the development of other DTDs. For example, over the years the declarations of the elements in a table have become standardized around a model called the *CALS table* model (named after the American Department of Defense Computer Assisted Logistic Support initiative, under which a lot of early SGML developments were funded). It is hard to improve on this model (tables are pretty standard), so it is often provided as a common DTD fragment that all other DTDs can reference when they need it (it's also part of the official HTML DTDs):

```
<?xml version="1.0" standalone="no"?>
  <!DOCTYPE book SYSTEM "book.dtd" [
    <!ENTITY % calstable SYSTEM "cals.dtd">
    %calstable;
  ]>
```



Note that this external parameter entity declaration, and the reference to it, are located in the internal DTD subset. Note, too, that I have explicitly stated that the XML document is *not* a standalone document.



If you're not going to validate an XML document, you must always be wary of what can go wrong with external parameter entities that contain markup declarations.

Listing 7.3 shows an external parameter entity being used in an internal DTD subset to include some declarations from an external file.

Listing 7.3 A Dangerous External Parameter Entity

```
1: <?xml version="1.0" standalone="yes"?>
```

```
2: <!DOCTYPE menu [
3:   <!ELEMENT menu (#PCDATA, front, meals+, back)* >
4:   <!ATTLIST menu title CDATA “Carte Blanche”>
5:
6:   <!ENTITY % entrees SYSTEM “entrees.xml”>
7:   %entrees;
8:   <!ATTLIST menu desserts CDATA “Sweet Temptations”>
9:  ]>
10: <menu><front>Sunday, July 11, 1998</front> ... </menu>
```

When this XML document is parsed, but not validated, the value of the menu element’s title attribute will be Carte Blanche. The XML processor sees the declaration of the title attribute (line 4) even though it is inside the internal DTD subset (lines 2 to 9). Because it isn’t validating (the document claims that it is a standalone document in line 1), the XML processor is not allowed to process any external entity references. The value of the desserts attribute has to remain unknown.

The default value (Sweet Temptations) has been declared, but the XML processor isn’t allowed to use it. Maybe it seems odd, but what is contained in the entrees.xml file that is referenced by the entrees external parameter entity? I don’t know, but it doesn’t really matter because the XML processor is not allowed to know either. Suppose, though, that the entrees.xml file contained element declarations; the entity reference is inside the internal DTD subset, so it would be quite legal to do so. Suppose too that the file contained this attribute declaration:

```
<!ATTLIST menu desserts CDATA “Bitter Experiences”>.
```

This would mean that the desserts attribute could not have the value Sweet Temptations.

So be careful with external parameter entities in the internal DTD subset, and consider the possible confusion they could cause when the recipient is not validating (something you have little or no control over). The only way you can protect yourself from this is to always use a standalone document declaration.

The purpose of this declaration is really to tell the recipient that the DTD could change the document, and that the DTD should be retrieved if the recipient wants to be certain that it’s seeing the same thing as the application that created the document. Parsing the DTD would mean that external entity references would be dereferenced, so the external declaration would then be found. Although the declaration has no effect on the XML processor, changing the value of the standalone document declaration to “no” will inform the recipient that validation is needed if the document is to be seen as intended.

Conditional Markup

Yesterday, when you learned about the DTD design step of constraining the model, I said that you could always consider making two versions of the same DTD. You could use a loose DTD during authoring when, for example, not all the content has been filled in but you still want to validate the document, and another, tighter DTD for the finished XML document. There is a way to do this using *conditional sections* of the DTD and parameter entities. (You will learn more about conditional sections of a DTD on Day 9 when you learn about entity declarations.)

This trick involves wrapping the two variant declarations of the DTD inside blocks and starting each block with a reference to a parameter entity, as shown

in this fragment:

```
<![%AuthoringSwitch;[
    <!ENTITY % body “chapter, intro?, section*”>
]]>
<![%FinalSwitch;[
    <!ENTITY % body “chapter, intro, section+”>
]]>
```

The parameter entities AuthoringSwitch and FinalSwitch can then be declared as the keyword INCLUDE or IGNORE, according to which version of this entity you want to use. For the first, loose version, you would insert the declarations like this, before the conditional sections:

```
<!ENTITY % AuthoringSwitch “INCLUDE”>
<!ENTITY % FinalSwitch     “IGNORE”>
<![%AuthoringSwitch; [
    <!ENTITY % body “chapter, intro?, section*”>
]]>
<![%FinalSwitch; [
    <!ENTITY % body “chapter, intro, section+”>
]]>
```

And for the second, tighter version, you would insert the declarations like this, before the conditional sections:

```
<!ENTITY % AuthoringSwitch “IGNORE”>
<!ENTITY % FinalSwitch     “INCLUDE”>
<![%AuthoringSwitch; [
    <!ENTITY % body “chapter, intro?, section*”>
]]>
<![%FinalSwitch; [
    <!ENTITY % body “chapter, intro, section+”>
]]>
```

Conditional sections in DTDs might seem like a trivial feature with only a small section affected (as in the examples), but you can use pairs of parameter entities like this to control many separate sections of a DTD. By changing the values of just these two parameter entities, you could radically change the whole DTD.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Optional Content Models and Ambiguities

When you review a DTD you are working on, take a close look at the occurrence indicators. If there are a lot of optional elements in a content model, there may be something wrong with the model.

An example of this comes from the SGML world, where paper books are common items for markup. Consider the back part of a book. I am creating an XML DTD for a book, and I've created a container element BACK to make it easier to process the parts together. My BACK element consists, as books often do, of any number of appendices (including zero), followed by an optional glossary and then an optional index. When you can describe a content model in plain English like this, it is quite easy to model:

```
<!ELEMENT BACK (appendix*, glossary?, index?)
```

The content model might look good, but I now have a problem. Everything is now optional, making it possible for me to have an empty BACK element, which certainly wasn't the idea.

One way to get around this is to break up the content model into inner groups, where each group explicitly describes one of the possibilities:

```
<!ELEMENT BACK ( (appendix+, glossary?, index?) |
                 (appendix*, glossary, index?) |
                 (appendix*, glossary?, index) )>
```

That seems to cover it. Let's just check:

- Group 1: I can have one or more appendices, followed by an optional glossary and then an optional index.
- Group 2: I can have zero or more appendices, which must be followed by a glossary and then an optional index.
- Group 3: I can have zero or more appendices, followed by an optional glossary and then there must be an index.

Seems pretty complete—and no empty BACK elements allowed—except that I now have an even bigger problem. This markup won't work!

> XML processors aren't very intelligent. They are not able to remember what they've seen, look ahead at what comes next, and then backtrack to confirm that your markup agrees with content model after all. All processors can do is read a piece of markup, check if it's valid, read the next piece, check that, and so on. Because of the limitations of processors, element content models have to be absolutely clear and capable of being interpreted in only one way.

When the XML processor sees an appendix element, it really cannot tell what's supposed to happen next—should there be a glossary element next or not? This content model is "ambiguous."

One way to get around this problem is to adopt what is called the waterfall approach, where you take each element in turn and work out only the new cases you need to cover. For the first element you'd have to consider a lot of possibilities, fewer for the second, and even fewer for the third until with the last element there are hardly any possibilities left. Let's adopt this approach here and follow the process step by step, re-examining the three groups I defined earlier as we go:

- Group 1: I can have one or more appendices, followed by an optional glossary and then an optional index. The first group seems OK.
- Group 2: I can have zero appendices (I've already covered the one or more possibility) which must be followed by a glossary and then an optional index.
- Group 3: I can have zero appendices and no glossary, but then there must be an index.

This is much better and once we write it out in the DTD it turns out to be the answer:

```
<!ELEMENT BACK ( (appendix*, glossary?, index?) |
                 (glossary, index?) |
                 (index) )>
```

Avoiding Conflicts with Namespaces

XML was designed with the Internet in mind, this is a given. XML also (as you will learn later) has extremely powerful mechanisms for linking documents that go much further than either HTML or even SGML. The linking features in XML actually allow you to go a step further than just "pointing" to another document: they allow you to pull the linked document (the link target) into the current document and include it as if it had been a physical part of the current document. This kind of linking is called *transclusion*.

Transclusion is a composite word, formed from *transversal* and *inclusion*. It means the activity of following a hypertext link from its source to its target and then copying the target to the point of reference as if it had been physically included at that point (much like the way graphics are included in HTML files by reference).

Now, suppose the link target is also an XML document. In itself, this needn't be a problem because XML has set rules for dealing with these conflicts:

- If the documents were created using the same DTD, with the same elements and the same element content models, this is probably not a problem at all.
- If they have the same elements but the elements have different attributes, the attributes are simply merged and the composite document elements have all the attributes.
- If they have different element declarations or the same attributes with different values, things start to get complicated. If you are validating an XML document, an element can only be declared once.

To get around this problem, under a proposal of the same name, each schema (this also applies for schemas other than DTDs) is considered to have a private "namespace" in which all the declarations are unique and have their own meanings. In between the XML declaration and the document type declaration, you then insert a declaration (an XML processing instruction) like the one shown in this fragment to identify the namespaces belonging to the schemas:

```
<?xml version="1.0" standalone="no">
<?xml:namespace name="<http://www.synopsys.com/>"

                        href="<http://www.synopsys.com/~north/xml/version1.dtd>"
                        as="v1"?>
```

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

The name attribute value locates the namespace owner, the href attribute value locates the schema itself, and the as attribute value declares a prefix that is used in front of every element and attribute that originates in this namespace. Now, when I have two attributes that mean different things in the different namespaces I don't have to discard one of them, and now I'm not going to get fatal validation errors because elements are declared twice. Instead I can keep them distinct by identifying them as coming from different namespaces and preserve the information they represent, as shown in this fragment:

```
<object v1.type:v2.sort="shirt">
<v1:size>10</size>
<v2:size>XXL</size>
</object>
```

Note that the prefix used to identify the namespace occurs locally. This not only means that you can call it whatever you like, it also means there are none of the inevitable problems associated with trying to centralize the names used for the infinite number of namespaces.

The namespaces proposal is only a proposal and is still subject to change. Although it is already being adopted in various XML applications (including XSL), some of the implementations are non-standard and there may be some heated discussions before a suitable compromise is found.

A Test Case

So far, much of this chapter has been a bit theoretical. To complete what you have learned about DTD development so far, it might be useful to follow through with a practical demonstration.

We can achieve only so much in such a short space and limited time (it isn't unusual for SGML DTDs to take several years and the involvement of hundreds of people to complete). But in a way, this is as it should be. After all, XML is much more suited to fast, lightweight applications.

Let's take something simple, like a set of address records. (I'm almost tempted to call it an "address database," which it isn't, but this kind of database-like information is perfect for XML-based applications.) A small fragment of the raw data is shown in Listing 7.4.

Listing 7.4 The Raw Data for an XML Application

```
1:  Fred Bloggs MISTC
2:  22 Chancery Lane
3:  London SW17 4QP
4:  England
5:  44 1 800 3451
6:  44 12 446 3382
7:  fbloggs@hk.co.uk
8:
9:  Dr. Jon F. Spencer
10: El Camino Real 44621
11: Sunnyvale
12: 95144 California
13: USA
14: 1 650 445 1273
15: 1 405 227 1877
16: jdspencr@hiflier.com
```

The raw data shown in Listing 7.4 consists of two different types of address details. This highlights some of the problems that you might encounter. For example, in the UK, designatory letters come after the name, although they can also come before, as shown in the US example. In a lot of European countries, they all appear before the name (like the Dutch example of "Prof. Mr. Dr. Ing. D.H.J. Heyden-Loods"). Also note the difference in the ZIP code formatting and position (no two countries have the same scheme) and in the telephone and fax numbers (again, no two countries have the same scheme).

Fortunately, the order in which these items appear in any rendered output is something that can be left to a style sheet. Our main concern is simply to identify information, not to specify how it will be rendered. However, this aspect cannot be ignored completely because inevitably there will be information that you want to identify in elements simply to apply some special rendering to it.

The first step is to abstract the information content out and identify it as elements. While you're doing this, you must look ahead and consider possibilities that might not actually occur in the test data, (such as addresses that consist of more lines than the number shown in Listing 7.4). Listing 7.5 shows my attempt to formalize the data as a hierarchy of elements.

Listing 7.5 The Formal Element Structure

```
1: People
2:    Person
3:        Name
4:            FirstName
5:            MiddleName
6:            FamilyName
7:            Title
8:        Address
9:           Street1
10:          Street2
11:          City
12:          State
13:          Country
14:          ZipCode
15:       PhoneNumber
16:       FaxNumber
17:       Email
18:       Notes
```

Notice how I've added an element for a middle name and one for a street address that can consist of two lines, even though neither of my two examples uses either. Also note that I haven't broken the telephone and fax numbers into separate numbers for the country code, area code, and so on. You might want to do this for your own records, especially if you want to select only people with a certain area code. For this application, we're only interested in the complete numbers, so there's nothing to be gained by adding extra elements.

Now that you have the basic element tree, you can start transcribing this into DTD syntax. Start with the top-level elements first:

```
1: <!DOCTYPE People [
2:
3:   <!ELEMENT People  (Person)>
4:
5:   <!ELEMENT Person  (Name, Address, PhoneNumber,
6:                      FaxNumber, Email, Notes)>
7:
8:   <!ELEMENT Name    (FirstName, MiddleName, FamilyName,
```

```
9:                          Title)>
10:
11:  <!ELEMENT Address (Street1, Street2, City, State,
12:                          Country, ZipCode)>
```

Once you've got the top-level elements sorted out, you can continue with the second level, and then the lower (leaf) elements:

```
13: <!ELEMENT FirstName   (#PCDATA)>
14: <!ELEMENT MiddleName  (#PCDATA)>
15: <!ELEMENT FamilyName  (#PCDATA)>
16: <!ELEMENT Title       (#PCDATA)>
17: <!ELEMENT Street1     (#PCDATA)>
18: <!ELEMENT Street2     (#PCDATA)>
19: <!ELEMENT City        (#PCDATA)>
20: <!ELEMENT State       (#PCDATA)>
21: <!ELEMENT Country     (#PCDATA)>
22: <!ELEMENT ZipCode     (#PCDATA)>
23: <!ELEMENT PhoneNumber (#PCDATA)>
24: <!ELEMENT FaxNumber   (#PCDATA)>
25: <!ELEMENT Email       (#PCDATA)>
26: <!ELEMENT Notes       (#PCDATA)>
27: ]>
```

Resist the temptation to specify the nature of element content unless it's really necessary, and even then, only do it when you're sure the data will always be of that type. Specifying a telephone number as being numerical data (NMTOKENS) can only create problems for you if you encounter numbers that include non-numeric data (such as dashes).

I have elected to play it safe in this DTD. All the leaf elements are simply PCDATA type (text) because either I cannot be sure what the contents of the elements will be, or I won't gain anything by adding a restriction on the data content.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Since this is such a simple DTD, I could safely leave it at this. However, there is one small optimization that almost demands implementation, and that's the long list of identical leaf element declarations. This is a perfect example of parameter entities saving you a lot of effort. The whole list can be collapsed like this:

```
1:  <!ENTITY % Data "(FirstName, FirstName, MiddleName,
2:                    FamilyName, Title, Street1, Street2,
3:                    City, State, Country, ZipCode,
4:                    PhoneNumber, FaxNumber, Email, Notes)">
5:
6:  <!ELEMENT %Data; (#PCDATA)>
```

Note that there is a space on either side of the % symbol in the declaration of the Data parameter entity, but there is no space when the entity is actually used.

We aren't quite finished yet. You now need to go back and add the occurrence indicators, as shown in Listing 7.6.

Listing 7.6 The Completed DTD

```
1: <!DOCTYPE People [
2:
```

```
3:   <!ELEMENT People  (Person)+>
4:
5:   <!ELEMENT Person  (Name, Address?, PhoneNumber?,
6:                      FaxNumber?, Email?, Notes?)>
7:
8:   <!ELEMENT Name    (FirstName?, MiddleName?, FamilyName,
9:                      Title?)>
10:
11:  <!ELEMENT Address (Street1?, Street2?, City?, State?,
12:                     Country?, ZipCode?)>
13:
14:  <!ENTITY % Data “(FirstName, FirstName, MiddleName,
15:                   FamilyName, Title, Street1, Street2,
16:                   City, State, Country, ZipCode,
17:                   PhoneNumber, FaxNumber, Email, Notes)”>
18:
19:  <!ELEMENT %Data; (#PCDATA)>
```

Almost every element in the complete DTD shown in Listing 7.6 can occur just once or can be omitted altogether. These elements are indicated with a ?. There are a few exceptions. For example, there must be at least one Person, and there can be as many as you like; the Person element is therefore marked with a +. A Person must have some kind of name or the record would be pretty pointless (although all the other data is optional). I therefore made the Name element a compulsory part of the Person element, and the FamilyName element must appear just once within the Name element (Name and FamilyName therefore don’t have an occurrence indicator).

We’re almost there. The very last step is to review the model and decide on any attributes. In this example, I’ve left a choice open. I added a Notes element that allows me to add descriptive information, but I won’t be able to do much with anything that’s inside the element because as far as the XML application is concerned, it simply contains text. You could consider adding an attribute to describe what kind of person this is. (This is an example where your choices can only be driven by what you want to do with the data.) For example, you could declare this:

```
1:   <!ELEMENT Person  (Name, Address?, PhoneNumber?,
2:                      FaxNumber?, Email?, Notes?)>
3:
4:   <!ATTLIST Person Type (Business | Personal) “Business”>
5:
6:   <!ATTLIST Title Position (Before | After) “Before”>
```

Here I’ve declared a Type attribute for a Person. Since the content of the attribute is a string, it’s CDATA by default and I don’t need to declare the attribute type. What I can do, though, is declare Business as the default value, which means I only need to specify a value in the XML data for all the people who are personal type entries.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Where the attributes come into their own is in solving presentation and rendering problems, like the title position problem that I mentioned earlier. Here I have specified a Position attribute for the title, with two options, Before and After. The Before value is the default, so I only need to explicitly specify the attribute when the title needs to appear after the name. The application and the style sheet control the rendered position.

So far I've been writing the DTD so that it can be used inside a test document as an internal DTD subset. All I have to do now is trim off the first line (<!DOCTYPE People [) and the last line (]>) and save the DTD in a separate file.

If you're going to create more than just a few DTDs, you should be careful to choose filenames that describe the content. Generally, giving the file the same name as the root element will be enough, but you might want to be more specific (such as DOC_SHORT and DOC_FULL). Get this right first; going back to change all the references to a DTD in a lot of documents can be a real pain when all you want to do is change a filename.

You're now ready to mark up the XML and validate it against the DTD. Listing 7.7 shows my marked-up data.

Listing 7.7 The Marked-Up Data for an XML Application

```
1:  <?xml version=“1.0” ?>
2:  <!DOCTYPE People SYSTEM “People.DTD”>
3:
4:  <People>
5:
6:    <Person Type=“Personal”>
7:      <Name>
8:        <FirstName>Fred</FirstName>
9:        <FamilyName>Bloggs</FamilyName>
10:       <Title Position=“After”>MISTC</Title>
11:     </Name>
12:     <Address>
13:       <Street1>22 Chancery Lane</Street1>
14:       <City>London</City>
15:       <Country>England</Country>
16:       <ZipCode>SW17 4QP</ZipCode>
17:     </Address>
18:     <PhoneNumber>44 1 800 3451</PhoneNumber>
19:     <FaxNumber>44 12 446 3382</FaxNumber>
20:     <Email>fbloggs@hk.co.uk<Email>
21:   </Person>
22:
23:   <Person>
24:     <Name>
25:       <FirstName>Jon</FirstName>
26:       <MiddleName>Jefferson</MiddleName>
27:       <FamilyName>Spencer</FamilyName>
28:       <Title>Dr.</Title>
29:     </Name>
30:     <Address>
31:       <Street1>El Camino Real 44621</Street1>
32:       <City>Sunnyvale</City>
33:       <State>California</State>
34:       <Country>USA</Country>
35:       <ZipCode>915144</ZipCode>
36:     </Address>
37:     <PhoneNumber>1 650 445 1273</PhoneNumber>
38:     <FaxNumber>1 405 227 1877</FaxNumber>
39:     <Email>jdspencr@hiflier.com</Email>
40:   </Person>
41: </People>
```

There you have it—an XML instance, complete with its own DTD.

Summary

Today you have learned a lot more about DTDs and XML content modeling. You have also learned quite a few of the advanced tricks that can make things a lot simpler. You’ve seen for yourself some of the problems that can occur, and some of the ways to

get yourself out of trouble. You should now be able to create complex XML DTDs, and you should have a pretty good idea of the best tools to use. On Days 8 and 9, we'll flesh out some of the more complex topics you learned yesterday and today.

Q&A

Q Why are ambiguities so important?
A Ambiguous content models are among the most common sources of problems for DTD developers, especially at the beginning. They're also some of the hardest problems to locate and fix. Anything you learn about them now can save you a lot of head-scratching later on.
Q At what point should I split up a DTD into pieces?
A There's no absolute value; it's simply a matter of how happy you are dealing with one unit. Size and complexity aren't the only criteria, though. You might want to split up a DTD earlier than strictly necessary in order to introduce some modu-larity.
Q I can have modular DTDs, so can I also have modular documents?
A Of course. To modularize a XML document, simply break it into text entities and combine them by using character entities in the main document. Don't forget the rules about parallel logical and physical structures. You can also use transclusions (inclusion by reference), which you'll learn about on Day 10, "Creating XML Links."

Exercises

1. Design a DTD for a basic Web home page (you can cheat and look at earlier examples).
2. Using your home page DTD as the basis, expand the DTD so it can be used to cover a complete Web site. Think in terms of internal DTD subsets and external parameter references.
3. Take any DTD you like and sketch it as a tree diagram (you can use whichever symbols you like). Now set the DTD elements in a table in which the first column contains the root element, the second column contains its children, and so on. Compare the two representations. This should convince you of the importance of modeling a DTD using a clear visual representation.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Part II

Chapter 8
XML Objects: Exploiting Entities

In yesterday’s chapter, you learned much of what you need to know about markup. Now it’s time to retrace some of these steps and go into greater technical detail about the real nuts and bolts of XML documents—the data you put into them. In this chapter, you will learn about

- XML entities
- Notations

- XML character sets
- Using (multiple) character encoding

Entities

Without getting too involved in the terminology (especially since this chapter takes a few liberties with some of the terms), XML brings markup a step closer to the world of object orientation, as in object-oriented programming. The basic object in XML is the entity, whether it's the XML document entity itself, the elements it contains, or the internal and external entities it references.

Of course, the DTD itself is an external entity that the document references (a special type of parameter entity, in fact), but the relationship is a little more complex than this. The DTD describes a class of XML document entities, of which the actual XML document is an instantiation.

The entities are divided into three types (excluding the XML document entity itself, which really isn't of any further interest in this context): character entities, general entities, and parameter entities. General entities can be further subdivided into internal entities and external entities. To confuse things a little more, external entities are subdivided into parsed entities (containing character data) and unparsed entities (usually containing binary data). Finally, common usage is that parsed general entities are usually referred to as internal and external text entities, and unparsed external general entities are often just called binary entities.

Personally, I find it easier to think in terms of text entities, which can be either internal or external, and binary entities, which have to be external. This minimal classification is actually helped by the way you declare the entities. An internal text entity declaration looks like this:

```
<!ENTITY name "replacement text">
```

An external text entity declaration looks like this (you will learn about the syntax of these declarations shortly):

```
<!ENTITY name SYSTEM "system.identifier">
<!ENTITY name PUBLIC public.identifier "system.identifier">
```

A character entity declaration (which is just a special case of an internal text entity) looks like this:

```
<!ENTITY name "&#code;" >
```

When you refer to these entities, the text and character entity references look like this:

```
&name;
```

Now that you've been introduced to the basic types of entities, let's look at them in more detail.

Internal Entities

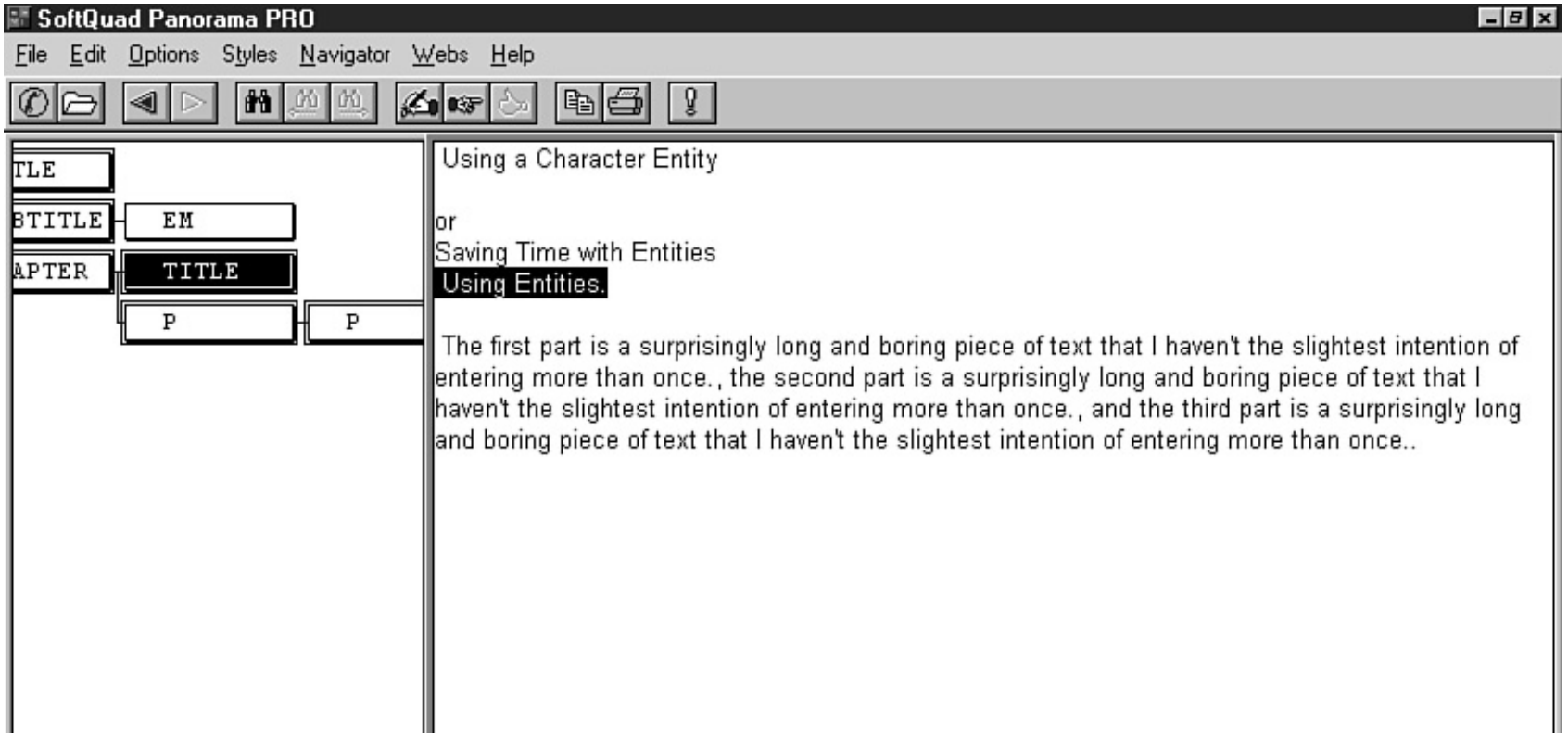
Internal entities are those whose definitions contain their values:

```
<!ENTITY intent “I am an internal entity, my declaration is self-contained”>
```

There is no separate physical storage object (file), and the content of the entity is given in the declaration. (It may be necessary for the XML processor to resolve any entity and character references in the entity value to produce the correct replacement text.)

Internal entities are parsed and must not contain references to themselves, either directly or indirectly.

In Chapter 4, you learned how to declare entities in a DTD so you can insert multiple occurrences of the text in a file without having to retype it every time. I won’t recap the mechanism in any detail here, but let’s look at a practical example.



I’m going to add the following code to my DTD:

```
<!ENTITY quick “a surprisingly long and boring piece
               of text that I haven’t the slightest
               intention of entering more than once.”>
```

Now I can reuse this text over and over again in my XML document by referencing it wherever I want it:

```
<p>The first part is &quick;,
the second part is &quick;, and
```

```
the third part is &quick;.</p>
```

When these references are displayed in an XML browser, they will be resolved and replaced with the entity text, as shown in Figure 8.1.

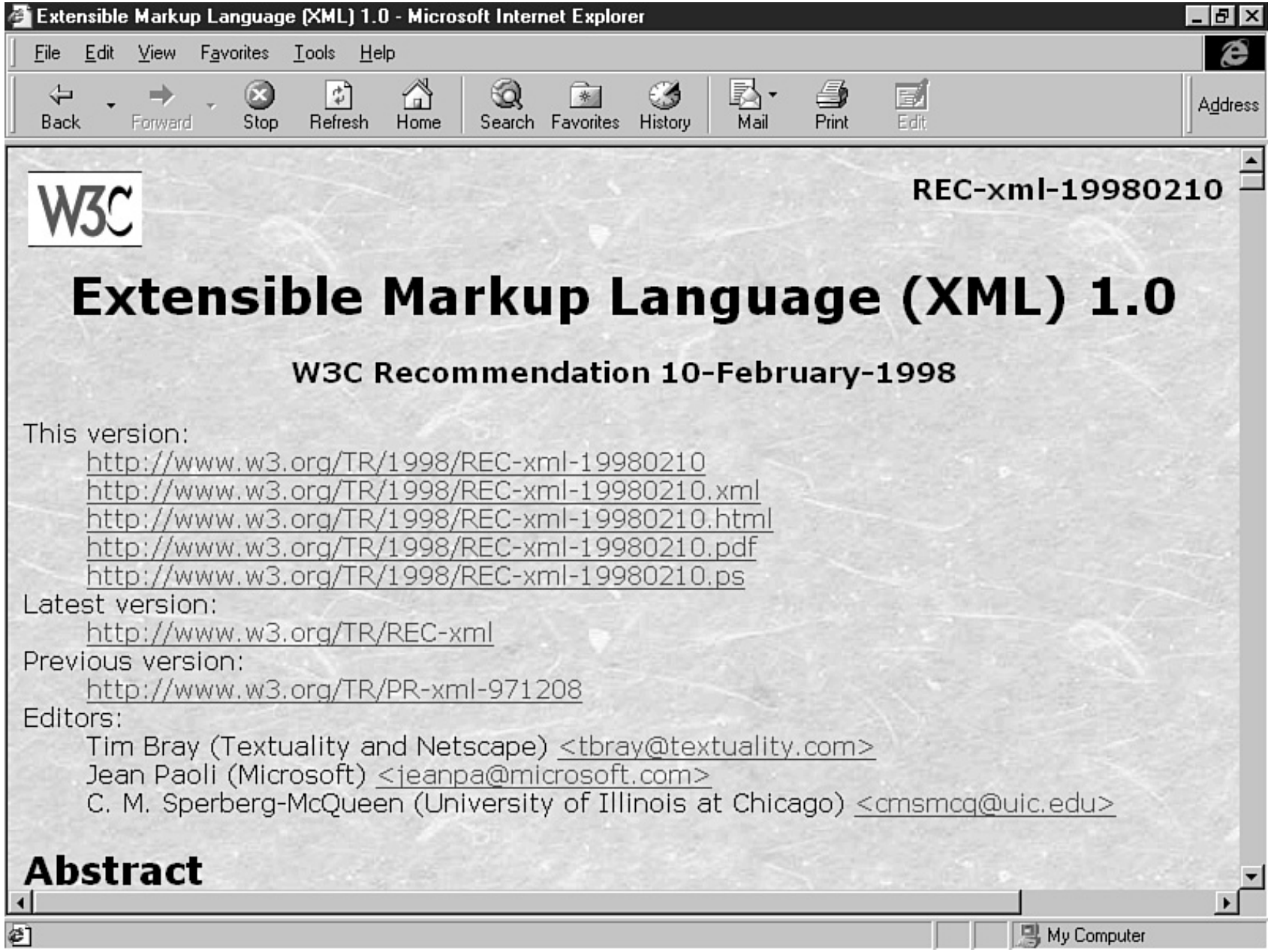
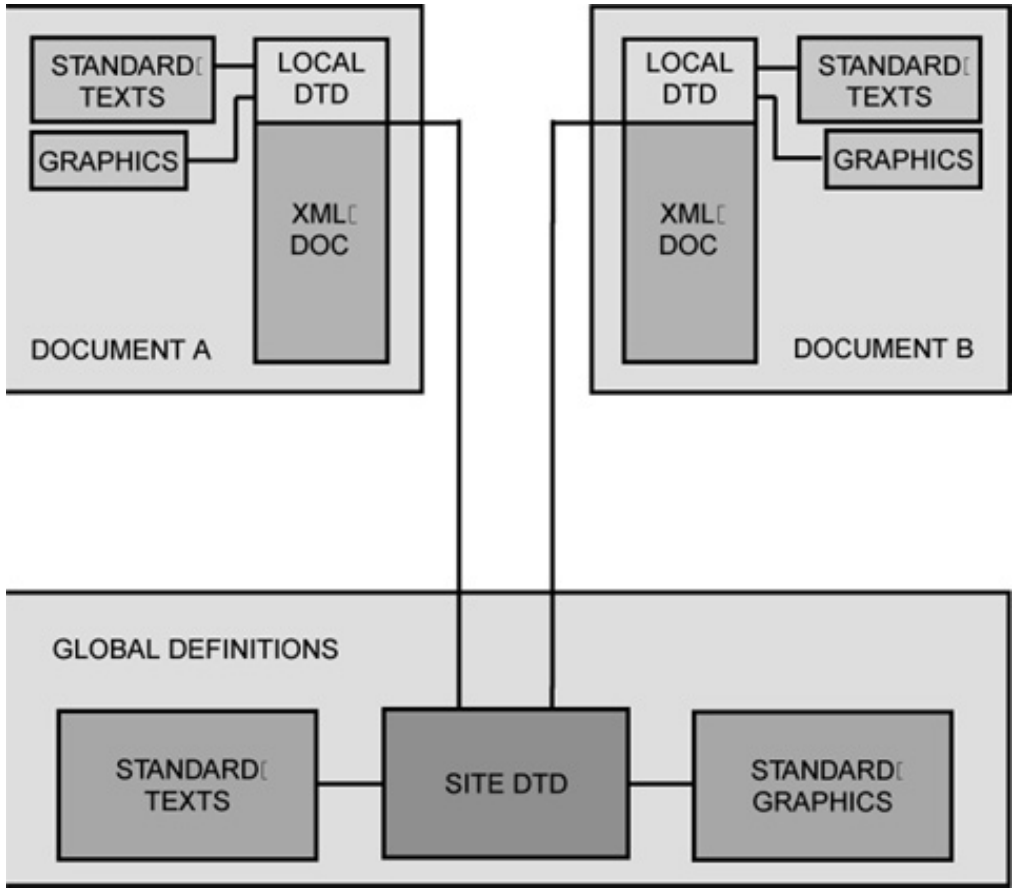


Figure 8.1 *How an entity is resolved in an XML browser.*



It would have been far more satisfying to have a true XML browser in Figure 8.1, but none of the current Web browsers can resolve entity references. Instead, the display shows SoftQuad's Panorama SGML browser. Fortunately, you can easily display XML code that doesn't differ too much from normal SGML code (and that has a DTD). An evaluation copy of Panorama can be downloaded at http://www.sq.com.

Binary Entities

Binary entities contain unparsed data (graphics, sound, and so on). When they are declared, they must be identified with a notation. The notation must also have been declared in the DTD:

```
<!NOTATION notation.name "public.identifier" "helper.application">
<!ENTITY entity.name NDATA notation.name>
```

Binary entities can only be referenced in the value of an attribute that has been declared to be of type ENTITY or ENTITIES in the DTD:

```
<!ELEMENT element.name EMPTY>
<!ATTLIST element.name
          attribute.name NDATA notation.name>
```

And this binary entity would then be referenced in the XML document, like this:

```
<element.name attribute.name="entity.name"/>
```

Listing 8.1 shows a typical example of a binary entity reference in an HTML file. An identical reference could also be used in an XML file.

Listing 8.1 A Typical Binary Entity Reference

```
1:  <H4>
2:    <A HREF="http://www.w3.org/">
3:      <IMG alt="W3C" SRC="http://www.w3.org/
4:           pub/WWW/Icons/WWW/w3c_home.gif"
5:        WIDTH="72" HEIGHT=48>
6:    </A>Parsing Entities
7:  </H4>
```

When this code is displayed in a Web browser, you will see the graphic displayed at the place where the reference was made, as shown in Figure 8.2.

Figure 8.2 *How a binary entity (a graphics file) is resolved in a Web browser.*

Note that the reference to a binary entity appears as the value of an element attribute. Text entity references can appear in element content, but binary (unparsed) entity references can only appear inside attributes.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Notations

Notations identify by name the format of unparsed entities (binary files such as external graphics files), the format of elements that bear a notation attribute, or the helper application (capable of processing the data) to which a processing instruction is addressed.

A notation must be declared in the DTD before it is used. The absolute minimum acceptable form of a notation declaration is

```
<!NOTATION  Name SYSTEM “”>
```

where Name is something meaningful to you or the registered public identifier for a particular format.

If your system supports it (as Microsoft Windows will, provided that the extension is associated with an application), you may be able to get away with using a notation declaration like this:

```
<!NOTATION GIF SYSTEM “GIF”>
```

This declaration takes advantage of the fact that as far as the XML processor is concerned, there doesn’t have to be anything on the system that can interpret data in this notation. Interpreting the data, or handling the error when it can’t handle the data, is entirely the application’s problem.

If you know there’s an application on the system that can handle data in a certain notation, you can help the application out by also pointing to the application:

```
<!NOTATION TIFF SYSTEM ‘C:\PROGRAM FILES\PaintShop Pro 5\psp.exe”>
```

Obviously, this will only work if you know the name and location of the application that can handle a particular notation, and if no one moves it. This isn't a lot of use on the Internet.

It is also possible to use an existing SGML facility and a public identifier. There are a lot of registered public identifiers for notations, covering everything from the C programming language (ISO/IEC 9899:1990//NOTATION (Programming languages—C)) to time itself (ISO 8601:1988//NOTATION (Representation of dates and times)). Some of the best-known notations, and their SGML public identifiers as they would be used in an SGML notation declaration, are shown in Listing 8.2.

Listing 8.2 Some Standard SGML Notation Declarations

```
<!NOTATION JPEG PUBLIC "ISO/IEC 10918:1993//NOTATION
  Digital Compression and Coding of
  Continuous-tone Still Images (JPEG)//EN">

<!NOTATION BMP PUBLIC
   "+//ISBN 0-7923-9432-1::Graphic Notation//NOTATION
  Microsoft Windows bitmap//EN">

<!NOTATION CGM-CHAR PUBLIC
 "ISO 8632/2//NOTATION Character encoding//EN">

 <!NOTATION CGM-BINARY PUBLIC
 "ISO 8632/3//NOTATION Binary encoding//EN">

<!NOTATION CGM-CLEAR PUBLIC
 "ISO 8632/4//NOTATION Clear text encoding//EN">

<!NOTATION FAX PUBLIC "-//USA-DOD//NOTATION
  CCITT Group 4 Facsimile Type 1 Untiled Raster//EN">

<!NOTATION GIF87a PUBLIC "-//CompuServe//NOTATION
  Graphics Interchange Format 87a//EN">

<!NOTATION GIF89a PUBLIC "-//CompuServe//NOTATION
  Graphics Interchange Format 89a//EN">

<!NOTATION PCX PUBLIC
"+//ISBN 0-7923-9432-1::Graphic Notation//NOTATION
  ZSoft PCX bitmap//EN">

<!NOTATION WMF PUBLIC
"+//ISBN 0-7923-9432-1::Graphic Notation//NOTATION
  Microsoft Windows Metafile//EN">
```

There's no reason why you can't use these SGML declarations in XML documents, and there are some very good reasons why you should. But the SGML public identifiers have to be combined with system identifiers to allow you to do so, like this:

```
<!NOTATION GIF89a PUBLIC "-//CompuServe//NOTATION
```

```
   Graphics Interchange Format 89a//EN"
    'C:\Program Files\lviewpro.exe'>
```

Note that you use a system identifier, but you don't need the SYSTEM keyword.

After (and *only* after) you have declared the notation, you can use that notation by name in an entity declaration with the NDATA (notation data) keyword:

```
<!ENTITY figure1 SYSTEM 'figure1.gif' NDATA BMP>
```

You can also use the notation in one of the attribute declarations for an element by using the NOTATION keyword:

```
<!ELEMENT IMG  EMPTY >
<!ATTLIST IMG
        src     %URL     #REQUIRED
        alt     CDATA    #IMPLIED
        type    NOTATION (GIF | JPEG | BMP) "GIF" >
```

Note that when you use enumerated notations in an attribute declaration, every notation you name must be declared in the DTD before the XML processor reaches that part of the DTD. For example, if you use notations in the internal DTD subset, you must declare the notation in the internal DTD subset too, and not in the external DTD subset. (As you may remember, the internal DTD subset is read before the external DTD subset.)

Typing in the public identifiers for notations can very quickly become a tedious and error-prone business. There's also a good chance that you will use the same notations again and again. Therefore, it's a good idea to collect all your notation declarations into one file. Give the file an obvious name like graphics.ent so that when you reference it in an XML document, it will always be easy to trace back what is going on:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE chapter SYSTEM "chapter.dtd" [
<!ENTITY % myentities SYSTEM "mysymbols.ent">
%myentities;
]>
<chapter><number/> ... </chapter>
```

By using an obvious filename and then referencing that file in all the DTDs you create by using an external entity declaration that points to it, you can save yourself a lot of unnecessary work. Why an obvious filename? Simple. You may return to a DTD after weeks, even months, and any extra clues about its content are bound to be welcome.

If for no other reason, keeping all your notation declarations in a separate file will mean that you'll only have to edit the file once if something changes, instead of having to edit every separate DTD. Also, there is less chance of you forgetting a declaration you need.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Identifying External Entities

Internal entities are self-contained, meaning that their definition is contained in their declaration. External entities are contained elsewhere, as their name suggests. There are two ways to identify the location of an external entity's content: through a system identifier or through a public identifier.

System Identifiers

A system identifier can be either a relative path to a filename (for example, ..\..\graphics\home.gif) or an absolute path to a filename (for example, c:\Program Files\LView\lview.exe):

```
<!ENTITY my.file SYSTEM "c:\inetroot\filez\file1.xml">
```

A system identifier can also be a Universal Resource Identifier (URI). The URI is an enhancement of the Universal Resource Locator (URL) system used for World Wide Web addresses: http://www.xs4all.nl/~sintac/books.htm

In this case, www.xs4all.nl is my service provider's Web server and ~sintac is a pointer that the UNIX system translates into my login (home) directory. The Web server then directs the Web browser to the designated Web page directory coupled to my login name, and books.htm is obviously the name of the file. A URI is a type of Universal Resource Name (URN), which is a sort of superset. The two are more or less synonymous, so I will simply use "URL" and be done with it. (Old habits do die hard.) The syntax for a full URL looks like this:

```
scheme://login-name:password@ host:port//path
```

In this URL, scheme is a protocol and most of the host information (login-name, password, and port) is only entered when it is really needed. The scheme could be http (Hypertext Transfer Protocol), ftp (File Transfer Protocol), gopher (the Gopher protocol), news (Usenet news—this one breaks the rule because the protocol is actually nntp, for Net News Transfer Protocol), wais (for Wide Area Information Servers), or file (for local file access). There are several more schemes, but many of them, like mailto, wouldn't make much sense for retrieving information.

Public Identifiers

A system identifier is reasonably straightforward; it simply points to a file. The public identifier is an inheritance from SGML. As far as entities are concerned, public identifiers result in the same things as system identifiers: somewhere, they are resolved into filenames. For notations, however, they represent a more formal method of identification. Using public identifiers allows the inclusion of such extra details as the language, the owner (or copyright holder), and the author.

There is still no *official* method for resolving public identifiers in XML. However, there is an SGML method that has been used for years. Most of the major XML tools come from SGML developers, so the same method has quietly been implemented in XML tools without any real discussion about whether it was needed or not.

SGML uses a public identifier resolution mechanism based on an industry agreement published as Technical Resolution 9401 in 1997 by the SGML Open Consortium (which recently changed its name to OASIS—the Organization for the Advancement of Structured Information Standards). More commonly known as the SGML Open Catalog (SOC), this mechanism uses a *catalog file* that is located in the same directory as the document (the application is free to change this location, of course). This file is usually called catalog.soc or, more frequently, just catalog. (A particular application will normally allow you to specify which file should be used as the catalog file.)

There is little point in going into all the technical details because the catalog file is really an SGML facility that in turn draws on some of the features of HyTime, and SGML and HyTime are well outside the scope of this book. As far as we are concerned, the catalog file is an ASCII file consisting of lines that couple a public identifier (officially a Formal System Identifier (FSI)) with a system object identifier. A *system object identifier* is basically a file, but it could also be some other kind of identifier that the system can convert into something meaningful. An example of a typical catalog file is shown in Listing 8.3.

Listing 8.3 A Typical Catalog File

```
1:  -- catalog: SGML Open style entity catalog for HTML --
2:  -- $Id: catalog,v 1.3 1995/09/21 23:30:23 connolly Exp $ --
3:  -- Hacked by jjc --
4:  -- Ways to refer to Level 2: most general to most specific --
5:  PUBLIC     “-//IETF//DTD HTML//EN”                    “html.dtd”
6:  PUBLIC     “-//IETF//DTD HTML 2.0//EN”                “html.dtd”
7:  PUBLIC     “-//IETF//DTD HTML Level 2//EN”            “html.dtd”
8:  PUBLIC     “-//IETF//DTD HTML 2.0 Level 2//EN”        “html.dtd”
9:
10: -- Ways to refer to Level 1: most general to most specific --
11: PUBLIC     “-//IETF//DTD HTML Level 1//EN”            “html-1.dtd”
12: PUBLIC     “-//IETF//DTD HTML 2.0 Level 1//EN”        “html-1.dtd”
13:
14: -- Ways to refer to Strict Level 2: most general to most specific --
15: PUBLIC     “-//IETF//DTD HTML Strict//EN”             “html-s.dtd”
16: PUBLIC     “-//IETF//DTD HTML 2.0 Strict//EN”         “html-s.dtd”
17: PUBLIC     “-//IETF//DTD HTML Strict Level 2//EN”     ‘html-s.dtd”
18: PUBLIC     “-//IETF//DTD HTML 2.0 Strict Level 2//EN” “html-s.dtd”
19:
20: -- Ways to refer to Strict Level 1: most general to most specific --
21: PUBLIC     “-//IETF//DTD HTML Strict Level 1//EN”     “html-1s.dtd”
22: PUBLIC     “-//IETF//DTD HTML 2.0 Strict Level 1//EN” “html-1s.dtd”
23:
24: -- ISO latin 1 entity set for HTML --
25: PUBLIC     “ISO 8879-1986//ENTITIES Added Latin 1//EN//HTML”
⇒ISOlat1.sgm
```

Note that the example shown in Listing 8.3 is a modified XML version of an SGML catalog file. In XML the filename has to be enclosed in quotes, while in SGML it doesn't.

There is nothing to prevent you from creating a catalog file with a text editor, but there are a few free catalog management packages (also called *entity management packages*) available on the Internet. Some software packages have their own built-in facility, often called an *entity manager*, for resolving entities. The only thing you need to know to make the mechanism work is that the left-hand part of a line (the FSI) in a catalog file must exactly match the declaration in the XML document.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Parameter Entities

A parameter entity is a completely separate sort of entity because of one major restriction: parameter entity references may only appear in a DTD.

To keep parameter entities distinct from general entities (and to prevent them from being used in a document), they're declared and referenced with a percent sign (%):

```
<!ENTITY % "front | body | back" >
```

Parameter entities are extremely useful as shortcuts for parts of declarations that occur often in a DTD. However, they can't contain markup (complete declarations). They can only contain parts of declarations:

```
<!ENTITY common "(para | body | text)">
<!ELEMENT chapter ((%common;)*, section+)>
<!ELEMENT section (%common;)>
```

When a parameter entity reference is resolved, one leading space and one trailing space is added to the replacement text to make sure that it contains an integral number of grammatical tokens.

Entity Resolution

The rules governing entity resolution—when entities are interpreted and when they are ignored—can be quite complicated. Table 8.1 shows what happens to entity references and character references. The leftmost column describes where the entity reference appears:

- **Inside an element**—The reference appears anywhere after the start tag and before the end tag of an element.
- **In an attribute value**—The entity reference occurs within the value of an attribute in a start tag, or in a default value in an attribute declaration.
- **As a name in attribute value**—The entity reference appears as a name, not as an entity reference, as the value of an attribute that has been declared as being of type ENTITY or ENTITIES:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE graphic SYSTEM "graphic.dtd" [
  <!ELEMENT graphic (icon)+>
  <!ELEMENT icon EMPTY>
  <!ATTLIST icon
            height   NMTOKEN #IMPLIED
            nsoffset NMTOKEN #REQUIRED
            width    NMTOKEN #IMPLIED >
  <!ENTITY icon8 SYSTEM "icon813.gif" NDATA gif>
]>
<graphic>
  <icon source="icon8"
        height="0.391in" nsoffset="0.000in"
        width="0.429in"/>
</graphic>
```

- **In an entity value**—The reference appears in a parameter or entity's value in the entity's declaration.
- **In the DTD**—The reference appears within either the internal or external subset of the DTD, but outside of an entity or attribute value.

Table 8.1 Entity Resolution

Where Referenced	***Entity Type***			***Character Reference***
	Parameter	***Internal***	***External***	
Inside an element	Ignored	Replaced	Replaced if validating	Replaced
In an attribute value	Ignored	Replaced	Not allowed	Replaced
Name in Attribute value	Ignored	Not allowed	Passed to application	Ignored

In an entity value	Replaced*	Ignored	Not allowed	Replaced
In the DTD	Replaced if validating	Not allowed	Not allowed	Not allowed

When the entity reference appears in an attribute value or a parameter reference appears in an entity value, single and double quotes are ignored so that the value isn't prematurely terminated.

An entity's replacement text may contain character references, parameter entity references, and general entity references. Character references and parameter entity references in the value of an entity are resolved when the entity is resolved. General entity references are ignored.

When an entity reference is replaced, the entity's replacement text is retrieved and processed, in place of the reference itself, as though it were part of the document at the location where the reference was recognized. The replacement text may contain both character data and markup (except for parameter entities). These are recognized in the usual way, except that the replacement text of the entities amp, lt, gt, apos, and quot is always treated as data.

Before a document can be validated, the replacement text for parsed entities also has to be parsed. If the entity is external and the document is not being validated, the replacement text doesn't have to be parsed. If the replacement text isn't parsed, it is up to the application to handle it.

When the name of an unparsed entity appears in the value of an attribute whose declared type is ENTITY or ENTITIES, the system and public (if any) identifiers for both the entity and its associated notation are simply passed to the application. The application is then responsible for any further processing (such as displaying the graphic in a window).

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Getting the Most Out of Entities

As you may remember, there is a fixed order in which the external and internal DTD subsets are read and interpreted. First the internal DTD subset is read, and then the external DTD subset is read. If something is declared in the internal DTD subset, its declaration cannot be changed in the external DTD subset. However, some things (such as additional attributes) can be added to declarations.

With a little careful planning, you can use this hierarchical arrangement to construct your information collection, such as a Web site, to make the maximum use of shared data (see Figure 8.3):

- Global declarations can be placed in a central DTD that governs all the documents. As far as each document is concerned, this central DTD is the (common) external DTD subset.
  The central DTD references entities that contain common graphics (company logo graphics and letterhead material, special characters, and so on).
- Local declarations can be placed in the internal DTD subset of each document. This allows each document to override the global declarations and tailor them for its own purposes.
  The internal DTD subset in each document references any graphics and text that the particular document needs.

Figure 8.3 *Using local and global entities.*

Breaking the documents into modules and exploiting the order in which the internal and external DTD subsets are parsed allows you to modularize a lot of documents, but you can go even further than this when you need to. You can even modularize the DTD itself and collect the parts that you need for a particular document type (although in this context you could almost call it a subtype).

There are many large industrial implementations of SGML that use this scheme. Some people call this the *pizza model*, where you build your DTD out of a core of essential elements and then add the toppings that you need for each particular document. Examples of major DTDs that use this model are the Text Encoding Initiative DTD and the DTDs that the American Department of Defense uses for Interactive Electronic Technical Manuals (IETM).

This same approach could be used just as easily in XML. In fact, it could probably be used a lot *more* easily because XML can include Internet URLs as the locators for fragments (which is being retrofitted to SGML). This makes it far easier to physically distribute data while still bringing it all together as a coherent set of information.

Character Data and Character Sets

As you have learned, XML entities contain data that can be either parsed (processed by the XML processor) or unparsed (this is normally non-XML data). Parsed data is made up of characters, some of which form character data and some of which form markup.

Taken at its most basic, an XML entity simply consists of a sequence of *characters*, each of which is an atomic unit of text—a unit of text that cannot be divided into anything smaller.

Internally, computers generally have used seven bits to store letters and characters in digital form. This representation was standardized as ISO/IEC 646, the now-familiar ASCII scheme in which each letter, number, and punctuation symbol is given a different seven-bit code. For example:

- The letter *x* is stored as 1111000.
- The letter *m* is stored as 1101101.
- The letter *l* is stored as 1101100.

Rather than write out the binary representation in full, these bit patterns are commonly represented by a *hexadecimal* number, such as *6B* standing for the letter *k*.

The range of *legal characters*, the characters that can appear in an XML document, are those with the following hexadecimal values:

- 09 (the tab character)
- 0D (the carriage return character)
- 0A (the line feed character)

- 20 to D7FF, E000 to FFFD, and 10000 to 10FFFF (which are the legal graphics characters of Unicode and ISO 10646)

Unicode and ISO 10646 are standardized *character sets*, which you will learn about next.

Character Sets

There are only 128 different seven-bit patterns possible in the basic ASCII alphabet, so seven-bit ASCII can represent only 128 different characters. The eighth bit (there being eight bits in a byte, the basic unit of computer storage) is used as a *check bit* to make sure that the byte is stored or transmitted correctly. These 128 characters are known as the standard ASCII character set and have been the basis of computing for many years.

As computers have become more advanced and more of an international phenomenon, extra characters have become needed to cover things like the accented characters used in European languages. The eighth bit has been repurposed to give eight-bit character sets, thereby doubling the number of possible characters to 256. This is standardized as the ISO 8859 character set. In fact, there are many ISO 8859 variants, each tailored for a specific language. The version you'll probably see most often is 8859/1, which is the character set used for HTML and understood by Web browsers. This character set includes accented characters, drawing shapes, a selection of the most common Greek letters used in science and technology, and various other symbols. The first 128 characters of ISO 8859/1 are exactly the same as ISO 646, so it's backward compatible.

Eight bits are fine for most Western languages, but next to useless for languages such as Arabic, Chinese, Urdu, and so on. To cater to these languages, Unicode (with 16-bit encoding) and ISO 10646 took the next logical steps to support up to 32-bit patterns to represent characters. This allows more than two billion characters to be represented. ISO 10646 provides a standard definition for all the characters found in many European and Asian languages. Unicode is used in Microsoft Windows NT.

Unicode actually includes a number of different encoding schemes, which are named according to the number of bits they need. UCS-2 uses 16 bits (two bytes), which is identical to Unicode, and UCS-4 uses a full 32 bits (four bytes).

ISO 10646 is even more sophisticated. Using mapping schemes called UCS Transformation Formats (UTF), ISO 10646 allows a variable number of bits to be used. There's little point in using so many bits if you're only sending the basic 128 ASCII characters, so ISO 10646 allows you to claim extra bits as you need them.

XML supports two UTF formats: UTF-8 (eight-bit to 48-bit encoding), and UTF-16 (up to 32-bit encoding, but using a mapping that gives more than a million characters).

Entity Encoding

Every text entity in XML can use a different encoding for its characters. For example, you can declare separate text entities or elements to hold sections of an XML document that contain Chinese or Arabic characters, and assign the 16-bit UCS-2 encoding to these sections. The rest of the document can then use the more efficient eight-bit

encoding.

By default, the ISO 10646 UTF-8 encoding is assumed. If the text entity uses some other encoding, you must declare what that encoding is at the beginning of the entity:

```
<?xml encoding="Encoding.Name" ?>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Where Encoding.Name is a character set name consisting of only the Roman alphabetic characters (A to Z and a to z), Arabic numerals, full stops, hyphens, and underscores. The XML processor has to recognize the following character set names:

- UTF-8
- UTF-16
- ISO-10646-UCS-2
- ISO-10646-UCS-4
- ISO-8859-1 to -9
- ISO-2022-JP
- Shift-JIS
- EUC-JP

Examples of encoding declarations are

```
<?xml version="1.0" encoding='UTF-16'?>
<?xml version="1.0" standalone="yes" encoding="EUC-JP"?>
```

The default (UTF-8) encoding is detected by the first four bytes of an XML text entity having the hexadecimal values 3C, 3F, 58, and 4D, which are the first characters of the encoding declaration. If there is no declaration, or if none of the other encoding schemes can be made to fit, the entity is simply assumed to be in UTF-8.

All XML processors can read entities in either UTF-8 or UTF-16. The standard ASCII alphabet is contained in the first part of the UTF-8 character set, so you do not need to declare the entity's character set if it is encoded in plain ASCII.

If an XML text entity is encoded in UCS-2, it must start with an appropriate encoding signature called the Byte Order Mark. This is FF FF for the standard byte order and FE FF for the byte-reversed order. These characters are not considered to be part of the markup or character data of the XML document.

Other than using the encoding declaration, the XML processor is allowed to assume that a certain encoding has been used based on the first few bytes it sees. In addition, the text/xml MIME type identification, which could be added by a Web server, can also be used to identify the character encoding used.

Entities and Entity Sets

Switching to a different encoding isn't the only way to represent characters that are not included in the UTF-8 character set. Don't forget that you can always reference any character by quoting its ISO 10646 character number in a character reference (such as &).

You can also declare an entity that represents the character you need, like this declaration of the degree sign (°) taken from the ISO 8859-1 character set:

```
<!ENTITY deg    "&#176;" >
```

You can then reference this entity in an XML document wherever you need it:

```
<para>The temperature today in the south will be 82 &deg;C.</para>
```

Not all computer systems and transfer media can handle the advanced character sets that you have learned about. The seven-bit ASCII character set is still the common denominator. These kinds of character entity declarations have been around since the early days of SGML and have been collected into *entity sets*.

These entity sets are included as part of the SGML standard (ISO 8879) and go under the somewhat cryptic names of ISOlat1 (Roman alphabet, accented characters), ISOnum (numeric and special characters), ISOcyr1 (Cyrillic characters used in Russian), and so on. They are really SGML facilities and cannot be used as they are in XML. (XML does not allow the use of an SDATA (system data) notation.) However, XML versions of the most important of these entity sets are being made publicly available, as you can see from the XML version of the ISOdia (diacritical marks) entity set shown in Listing 8.4.

Listing 8.4 The Modified ISOdia Entity Set for XML

```
1:  <!-- (C) International Organization for Standardization 1986
2:       Permission to copy in any form is granted for use with
3:       conforming SGML systems and applications as defined in
4:       ISO 8879, provided this notice is included in all copies.
5:  -->
6:  <!-- Character entity set. Typical invocation:
7:       <!ENTITY % ISOdia PUBLIC
8:              "ISO 8879:1986//ENTITIES Diacritical Marks//EN//XML">
9:       %ISOdia;
10:
11: -->
12: <!-- This version of the entity set can
13:      be used with any SGML document
14:      which uses ISO 10646 as its document character set.
15:      This includes XML documents and ISO HTML documents.
16:      This entity set uses hexadecimal numeric character references.
17:
18:      Creator: Rick Jelliffe, Allette Systems
19:
20:      Version: 1997-07-07          -->
21:
22: <!ENTITY acute  "&#180;" ><!--=acute accent-->
23: <!ENTITY breve  "&#x2D8;" ><!--=breve-->
24: <!ENTITY caron  "&#x2C7;" ><!--=caron-->
25: <!ENTITY cedil  "&#184;" ><!--=cedilla-->
26: <!ENTITY circ   "^" ><!--=circumflex accent-->
27: <!ENTITY dblac  "&#x2DD;" ><!--=double acute accent-->
28: <!ENTITY die    "&#168;" ><!--=dieresis-->
```

```
29: <!ENTITY dot    “&#x2D9;” ><!--=dot above-->
30: <!ENTITY grave  “`” ><!--=grave accent-->
31: <!ENTITY macr   “&#175;” ><!--=macron-->
32: <!ENTITY ogon   “&#x2DB;” ><!--=ogonek-->
33: <!ENTITY ring   “&#x2DA;” ><!--=ring-->
34: <!ENTITY tilde  “&#x2DC;” ><!--=tilde-->
35: <!ENTITY uml    “&#168;” ><!--=umlaut mark-->
```

The entity sets are provided as separate files, one for each set. If you know that you’re going to need a particular set of characters for an XML document, you can just include the necessary declaration in your XML DTD, as shown in Listing 8.5.

Listing 8.5 Typical Notation Declarations in a DTD

```
1:  <?xml version=”1.0” standalone=”no”?>
2:  <!DOCTYPE chapter SYSTEM “chapter.dtd” [
3:
4:  <!ENTITY % ISOlat1 PUBLIC “ISO 8879-1986//ENTITIES
5:                     Added Latin 1//EN//XML” “isolat1.xml”>
6:  %ISOlat1;
7:
8:  <!ENTITY % ISOnum PUBLIC “ISO 8879:1986//ENTITIES
9:             Numeric and Special Graphic//EN//XML” “isonum.xml”>
10: %ISOnum;
11:
12: <!ENTITY % ISOpub PUBLIC “ISO 8879:1986//ENTITIES
13:             Publishing//EN//XML” “isopub.xml”>
14: %ISOpub;
15: ]>
16: <chapter><number/> ... </chapter>
```

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Note that the form of the entity declaration in the DTD differs slightly from that used in the entity file itself (Listing 8.4). A system identifier has to be used as well as the public identifier, but there is no SYSTEM keyword.

Summary

You've now learned several methods for getting characters into an XML text entity that aren't normally available from your keyboard. Not only have you learned about character encoding schemes, you've learned how to incorporate whole sets of special characters through entity sets.

You have also learned how to include external graphics files by declaring them with specific notations and then referencing them in the XML document, as well as how to declare a helper application that can be used to process the data.

Apart from the fact that you have to use external entities to include binaries such as graphics files in your formatted XML output (in the raw XML code, you just include entity references), entities can be a wonderful tool for breaking up a complex collection of many documents into manageable chunks. By using internal and external text entities, combined with the hierarchical ordering applied to interpreting the two DTD subsets, you can implement a staged hierarchy of template content. At the global level, you can safely set up default settings and then let individual XML documents implement local overriding changes when you need to.

As you will learn on Day 10, “Creating XML Links,” XML’s extremely powerful linking facilities make this task even easier. Tomorrow, you will put everything you’ve learned to the test by actually validating XML documents—the make-or-break evaluation of whether your XML code is going to work or not.

Q&A

Q How can you put unparsed (non-XML) text in an external entity?
A There are several ways. You could declare a TEXT notation, but this would not allow you to physically include the text in the XML document. (It would go the helper application you designated in the notation declaration.) The best way would probably be to declare the file containing the text as an external text entity and put the text in that file in a CDATA section.

Q You cannot use parameter entities in an XML document, but can you use general entities in a DTD?
A No. Parameter entities are exactly the same as general entities. They contain markup text (the only text you can have in a DTD, other than comments), so they perform exactly the same function.

Q Are there any limits on the size of entities?
A None that XML imposes. SGML does set some limitations on the size of certain objects (such as the lengths of names), but these have all been removed in XML. The only limits would be those imposed by your XML application or by the computer on which you’re working.

Q Can you use an ANSI code for a character?
A No. Entering the ANSI code for a character was a feature available in some Microsoft Windows packages, and it has nothing to do with XML. There are fewer and fewer packages that support this shortcut, and a shortcut was all it was. All it did was give you another method of entering a character, in the current encoding scheme, that isn’t available on the keyboard. If your editing package supports this and you want to use it, do. Don’t confuse this with character encoding, however.

Q What happens if the software declared for a notation isn’t there or can’t be executed for some reason?
A As far as XML’s concerned, nothing. All the XML processor will do is check that the declaration is correct. It’s up to the XML application to process the contents of an entity that uses the notation. The application is also responsible for figuring out what to do if the helper application isn’t there (it might have its own internal viewer, after all).

Exercises

1. Develop a simple DTD for a basic business letter. It needn’t be complex—just enough for a letterhead, address blocks, and basic text.
2. Create a simple letter in XML using your letter DTD. Extend the XML code in the document and the DTD to include a standard letterhead that contains a GIF-format company logo.
3. Now set up your XML application so that you can quickly and easily put your name in the signature block.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 9
Checking validity

During the previous days you have learned how to write document type definitions (DTDs), the formal specifications of the structure of your XML files.

Now you may have two questions:

- How can I check that my DTD is correct?
- How can I check that my documents comply with the rules defined in the DTD?

Validating parsers can do these jobs for you. In this chapter you will learn to:

- Check your DTD using DXP
- Check your DTD using XML for Java
- Check if your XML files comply with the rules of your DTD using the DXP parser
- Check if your XML files comply with the rules of your DTD using the XML for Java parser from IBM

> For overviews and lists of available parsers, refer to Day 5, “Checking Well-Formedness.”

Checking Your DTD with DXP

Finding and installing the DXP parser was discussed on Day 5. Now let’s see how it can be used to check your DTDs.

At the DOS prompt, type

```
jre -cp .;c:\datachannel\dxp\classes dxpcl -s -v c:\xmlex\file.xml
```

where

- jre invokes the Java Runtime Engine.
- -cp sets the classpath (where to find the classes used). In this case two paths are specified: the first (.) refers to the current working directory, and the second is c:\datachannel\dxp\classes; they're separated by ;.
- dxpcl is the name of the Java program (class).
- -s stands for silent mode.

For the moment, we are only interested in the errors in your file. When silent mode is specified, the only output generated is the error messages.

- -v stands for validation on.
- c:\xmlex\file.xml is the file to be checked.

Walkthrough of a DTD Check with DXP

Now we're going to check the markup declarations shown in Listing 9.1.

Listing 9.1 dtdv.dtd—Declarations to Be Checked Using DXP

```
1: <!ENTITY % admonitions "(tip | warning | note)" >
2: <!ENTITY % paracontent "(#PCDATA | icon | menu | xref |
⇒iconbmp)*" >
3:
4: <!ELEMENT helptopic (title, rule, procedure, rule?,
⇒%admonitions;) >
5: <!ATTLIST helptopic id ID #IMPLIED>
6:
7: <!ELEMENT title (#PCDATA) >
8: <!ATTLIST title keyword CDATA>
9:
10: <!ELEMENT procedure (step+)>
12: <!ELEMENT step (action, (%admonitions;)*) >
13:
14: <!ELEMENT action %paracontent; >
15: <!ELEMENT tip %paracontent; >
16: <!ATTLIST tip targetgroup (beginners | specialists)
⇒"beginners" >
17:
18: <!ELEMENT warning %paracontent; >
19: <!ELEMENT note %paracontent; >
20: <!ELEMENT icon (#PCDATA) >
21: <!ELEMENT menu (#PCDATA|shortcut)+>
22:
23: <!ELEMENT xref (#PCDATA) >
24: <!ATTLIST xref linkend idref #REQUIRED>
25:
```

```
26: <!ELEMENT shortcut (#PCDATA)>
27: <!ELEMENT tip (#PCDATA) >
28:
29: <!ELEMENT iconbmp EMPTY>
30: <!ATTLIST iconbmp src ENTITY #REQUIRED
31:                type NOTATION (bmp | gif) “gif”>
```

This is a perfect opportunity to test your knowledge. Before letting the parser point out the errors to you, try to discover them by yourself: find how many there are, describe them, and figure out how you would solve them.

Markup declarations can appear in two places:

- In the external or
- In the internal subset of the document type declaration

When external, the markup declarations exist in an external file (a special kind of external entity), and the document type declaration in the XML file needs to point to this external file:

```
<!DOCTYPE helptopic SYSTEM “http://www.protext.be/help.dtd” []>
```

In our case, the XML file should be as shown in Listing 9.2.

Listing 9.2 dtdv.xml—Simple XML File to Parse Using DXP

```
1:  <?xml version=“1.0” ?>
2:  <!DOCTYPE helptopic SYSTEM “dtdv.dtd” [

3:  ]>
```

You started your XML file with a document type declaration that refers to the external subset in file “dtdv.dtd”.

Bear in mind that on the file system, your XML file (dtdv.xml) needs to be in the same location (subdirectory) as “dtdv.dtd”.

Now let’s run DXP to expose any errors:

```
jre -cp .;c:\datachannel\dxp\classes dxpcl -s -v c:\xmlex\dtdv.xml
```

The first error message appears:

FATAL ERROR: encountered “>”. Was expecting one of: <EOF> , <S>
Location: file:///c:/xmlex/dtdv.dtd:8:30

Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)

For the structure of the error messages generated, see Day 5.

The problem has to do with this line:

```
“<!ATTLIST title keyword CDATA>”
```

Here you declare the attributes for the element title; in this case there’s only one (keyword).

For each attribute, you need to have a definition with the following parts:

- The name of the attribute
- The type of the attribute
- A default

These are all separated by a space.

In this case, you have defined only the following:

- The name, keyword
- The type, CDATA or stringtype, which may take any literal string as value

You’ve forgotten to include the default. The missing default needs to be one of the following:

- #REQUIRED
- #IMPLIED
- An attribute value optionally preceded by #FIXED

And it needs to be preceded by a space.

You opt for #IMPLIED, so the line becomes

```
<!ATTLIST title keyword CDATA #IMPLIED>
```

Parse again and receive

- FATAL ERROR: encountered “+”. Was expecting: “*”
  Location: file:///c:/xmlex/dtdv.dtd:21:34

  Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)
- It concerns <!ELEMENT menu (#PCDATA|shortcut)+>.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Search this book:

Go!

Previous | Table of Contents | Next

When you have mixed content—character data interspersed with child elements—the content model needs to have the * occurrence indicator. The line needs to become

```
<!ELEMENT menu (#PCDATA|shortcut)*>
```

The next error we receive is:

FATAL ERROR: encountered “idref”. Was expecting one of:
⇒”ID” , “IDREF” , “IDREFS” , “ENTITY” , “ENTITIES” , “NMTOKEN” ,
⇒“NMTOKENS” , “NOTATION” , “CDATA” , “%”, “(“

Location: file:///c:/xmlex/dtdv.dtd:24:24

Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)

All keywords (including “idref”) need to be in uppercase in XML. After this correction we receive:

ERROR: element declared twice “tip”
Location: file:///c:/xmlex/dtdv.dtd:27:11

ERROR: notation not declared “bmp”
Location: file:///c:/xmlex/dtdv.dtd:31:41

ERROR: notation not declared “gif”
Location: file:///c:/xmlex/dtdv.dtd:31:44

FATAL ERROR: encountered end of file
Location: :3:4

Found errors/warnings: 1 fatal error(s), 3 error(s) and 0 warning(s)

Now this is interesting. You'll have three errors and one fatal error over here.

The element tip is declared twice. The specification clearly states that no element type may be declared more than once. This is a validity constraint. Violations against validity constraints are considered errors. Correct this by deleting the second element declaration of tip.

You receive errors on the notations, which is understandable because according to the specification, all notation names in the declaration must have been declared.

Add notation declaration for BMPs and GIFs at the start of your file:

```
1: <!NOTATION bmp SYSTEM "paint.exe">
2: <!NOTATION gif SYSTEM "">
```

If you'd added those two notation declarations at the end of the file, the problems still wouldn't be resolved because notations (and entities) need to be declared before they are referenced.

After this, you are stuck with one more error:

FATAL ERROR: encountered end of file
Location: :3:4

Found errors/warnings: 1 fatal error(s), 0 error(s) and 0 warning(s)

Obviously, encountered end of file is the error message, because our file contains only a prolog and no element.

Because our interest is in simply checking the DTD, the missing element is not a concern for the moment.

You end up with no errors in your DTD, as shown in Listing 9.3.

Listing 9.3 dtdv.dtd—The DTD with No Errors

```
1: <!NOTATION bmp SYSTEM "paint.exe">
2: <!NOTATION gif SYSTEM "">
3:
4: <!ENTITY % admonitions "(tip | warning | note)" >
5: <!ENTITY % paracontent "(#PCDATA | icon | menu | xref |
⇒iconbmp)*" >
6:
7: <!ELEMENT helptopic (title, rule, procedure, rule?,
⇒%admonitions;) >
8: <!ATTLIST helptopic id ID #IMPLIED>
```

```
9:
10: <!ELEMENT title (#PCDATA) >
11: <!ATTLIST title keyword CDATA #IMPLIED>
12:
13:<!ELEMENT procedure (step+)>
14:<!ELEMENT step (action, (%admonitions;)*) >
15:
16:<!ELEMENT action %paracontent; >
17:<!ELEMENT tip %paracontent; >
18:<!ATTLIST tip targetgroup (beginners | specialists)
⇒“beginners” >
19:
20:<!ELEMENT warning %paracontent; >
21:<!ELEMENT note %paracontent; >
22:
23:<!ELEMENT icon (#PCDATA) >
24:<!ELEMENT menu (#PCDATA|shortcut)*>
25:
26:<!ELEMENT xref (#PCDATA) >
27:<!ATTLIST xref linkend IDREF #REQUIRED>
28:
29: <!ELEMENT shortcut (#PCDATA)>
30:
31: <!ELEMENT iconbmp EMPTY>
32: <!ATTLIST iconbmp src ENTITY #REQUIRED
33:               type NOTATION (bmp | gif) “gif”>
```

When internal, the declarations appear between the brackets ([and]) of the document type declaration:

```
<!DOCTYPE helptopic [
<!ELEMENT helptopic (title, procedure)>
<!ATTLIST helptopic id ID #REQUIRED>
...
]
<helptopic>....
```

Remember that the DTD of a document consists of both subsets, external and internal, taken together.

For the moment, all your declarations are in the external subset of your document type declaration.

What will happen if you copy the content of your file inside your internal subset and don’t refer anymore to the external file with the declarations, as shown in Listing 9.4?

Listing 9.4 dtdv.xml—XML File with Declarations in the Internal Subset

```
1: <?xml version=“1.0” ?>
2: <!DOCTYPE helptopic [
3: <!NOTATION bmp SYSTEM “paint.exe”>
4: <!NOTATION gif SYSTEM “”>
5:
6: <!ENTITY % admonitions “(tip | warning | note)” >
7: <!ENTITY % paracontent “(#PCDATA | icon | menu | xref |
⇒iconbmp)*” >
8:
9: <!ELEMENT helptopic (title, rule, procedure, rule?,
⇒%admonitions;) >
10: <!ATTLIST helptopic id ID #IMPLIED>
11:
12: <!ELEMENT title (#PCDATA) >
13: <!ATTLIST title keyword CDATA #IMPLIED>
14:
15: <!ELEMENT procedure (step+)>
16: <!ELEMENT step (action, (%admonitions;)*) >
17:
18: <!ELEMENT action %paracontent; >
19: <!ELEMENT tip %paracontent; >
20: <!ATTLIST tip targetgroup (beginners | specialists)
⇒“beginners” >
21:
22: <!ELEMENT warning %paracontent; >
23: <!ELEMENT note %paracontent; >
24:
25: <!ELEMENT icon (#PCDATA) >
26: <!ELEMENT menu (#PCDATA|shortcut)*>
27:
28: <!ELEMENT xref (#PCDATA) >
29: <!ATTLIST xref linkend IDREF #REQUIRED>
30:
31: <!ELEMENT shortcut (#PCDATA)>
32:
33: <!ELEMENT iconbmp EMPTY>
34: <!ATTLIST iconbmp src ENTITY #REQUIRED
35:                 type NOTATION (bmp | gif) “gif”>
36:
37: ]>
```

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Parsing this file gives you the following:

FATAL ERROR: parameter entity reference in entity value in
⇒internal subset for “admonitions”

Location: file:/c:/xmlex/dtdv.xml:9:65

In the internal subset, parameter-entity references (%admonitions; and %paracontent;) can occur only where markup declarations can occur, not within markup declarations such as the declaration of the element tip. This means that the following is allowed:

```
<!DOCTYPE helptopic [
<!ATTLIST helptopic a CDATA “A11”>
<!ENTITY % buttons SYSTEM “button.ent”>
%buttons;
]>
```

You can include the parameter entity reference %buttons; at that place because markup declarations could be entered also.

The following is *not* allowed:

```
<!DOCTYPE helptopic [
<!ENTITY % paracontent “(#PCDATA | emphasis)*”>
<!ELEMENT tip %paracontent; >
]>
```

This is because the parameter entity reference appears in a markup declaration. You can solve this by using either of these methods:

- Bring everything back to the external subset.
- Replace the parameter references with their declared content, as shown in Listing 9.5.

Listing 9.5 dtdv.dtd—Corrected File with Parameter Entity References Replaced

```
1: <?xml version=“1.0” ?>
2: <!DOCTYPE helptopic [
3: <!NOTATION bmp SYSTEM “paint.exe”>
4: <!NOTATION gif SYSTEM “”>
5: <!ELEMENT helptopic (title, rule, procedure, rule?,
⇒(tip | warning | note)) >
6: <!ATTLIST helptopic id ID #IMPLIED>
7:
8: <!ELEMENT title (#PCDATA) >
9: <!ATTLIST title keyword CDATA #IMPLIED>
10:
11: <!ELEMENT procedure (step+)>
12: <!ELEMENT step (action, ((tip | warning | note))*) >
13:
14: <!ELEMENT action (#PCDATA | icon | menu | xref |
⇒iconbmp)* >
15: <!ELEMENT tip (#PCDATA | icon | menu | xref | iconbmp)* >
16: <!ATTLIST tip targetgroup (beginners | specialists)
⇒“beginners” >
17:
18: <!ELEMENT warning (#PCDATA | icon | menu | xref |
⇒iconbmp)* >
19: <!ELEMENT note (#PCDATA | icon | menu | xref | iconbmp)* >
20:
21: <!ELEMENT icon (#PCDATA) >
22: <!ELEMENT menu (#PCDATA|shortcut)*>
23:
24: <!ELEMENT xref (#PCDATA) >
25: <!ATTLIST xref linkend IDREF #REQUIRED>
26:
27: <!ELEMENT shortcut (#PCDATA)>
28:
29: <!ELEMENT iconbmp EMPTY>
30: <!ATTLIST iconbmp src ENTITY #REQUIRED
31:                type NOTATION (bmp | gif) “gif”>
32:
33: ]>
```

Checking Your DTD with XML for Java

XML for Java is a validating XML parser written in 100% pure Java. The package (com.ibm.xml.parser) contains classes and methods for parsing, generating, manipulating, and validating XML documents. XML for Java is a robust XML processor and is very complete.

Installing XML for Java

- Download XML for Java from http://alphaworks.ibm.com/formula/xml. The file xml4j_1_1_9.zip is 1.288KB.
- Unzip xml4j_1_1_9.zip into a new directory, such as c:\xml4j.

Because this is another Java implementation, make sure you have Java Virtual Machine version 1.x running. Refer to Day 5 for details.

Using XML for Java

At the DOS prompt, type

```
jre -cp c:\xml4j\xml4j.jar trlx c:\xmlex\dtdv.xml
```

where

- jre is used to invoke the Java Runtime Engine.
- -cp is used to set the classpath. In this case you refer to the jar (Java archive file) with the name xml4j.jar.
- trlx is the Java class that does the parsing.
- c:\xmlex\dtdv.xml is the file to be checked.

Walkthrough of a DTD Check with XML for Java

You'll be using the same file as in Listing 9.1. Here are the error messages received:

```
dtdv.dtd: 8, 30: Spaces are expected.
dtdv.dtd: 8, 30: '#REQUIRED' or '#IMPLIED' or '#FIXED' or
⇒attribute value is expected.
dtdv.dtd: 21, 35: This content model is not matched with
⇒the mixed model '(#PCDATA|FOO|BAR|...|BAZ)*': '(#PCDATA|shortcut)+'
dtdv.dtd: 24, 29: 'CDATA' or 'ID' or 'IDREF' or 'IDREFS'or
⇒'ENTITY' or 'ENTITIES' or 'NMTOKEN' or 'NMTOKENS' or 'NOTATION'
⇒or '(' is expected.
dtdv.dtd: 27, 24: Element 'tip' is already declared.
dtdv.dtd: 31, 27: NOTATION 'bmp' is not declared.
dtdv.dtd: 31, 33: NOTATION 'gif' is not declared.
c:\xmlex\dtdv.xml: 3, 3: The document has no element.
```

Every error message consists of the following:

- The file in which the error appears
- The line number
- The character position where the problem was detected
- A description

With XML for Java, you receive all errors in one run with very clear error messages.

The last error message has to do with not having a complete document, but only a pointer to the DTD.

With XML for Java, it is possible to check the DTD directly. You have to add the parameter –dtd to the command line and read the file with the external subset directly:

```
jre -cp c:\xml4j\xml4j.jar trlx -dtd c:\xmlex\dtdv.dtd
```

This is a much cleaner way of doing it. Of course, you will receive the same results except for the last error, The document has no element.

Checking Your XML Files with DXP

At the DOS prompt, type the following:

```
jre -cp .;c:\datachannel\dxp\classes dxpcl -s -v c:\xmlex\wfq.xml
```

where

- jre invokes the Java Runtime Engine.
- -cp sets the classpath (where to find the classes used). In this case you've specified two paths: the first (.) refers to the current working directory, and the second is c:\datachannel\dxp\classes; they're separated by ;.
- dxpcl is the name of the Java program (class).
- -s stands for silent mode.
- -v stands for validation on.
- c:\xmlex\wfq.xml is the file to be checked.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Walkthrough of an XML File Check with DXP

On Day 5, you made the file wfq.xml well-formed with the help of parsers. This file (shown in Listing 9.6) contains a description of a help topic.

Listing 9.6 wfq.xml—The Well-Formed Help File

```
1: <?xml version=“1.0” ?>
2: <?protext objid=±I5678” ?>
3: <!DOCTYPE helptopic [
4: <!ENTITY doubleclick “Double-click”>
5: ]>
6: <helptopic>
7: <title keyword=“printing,network;printing,shared printer”>How to
⇒use a shared network printer?</title>
8: <procedure>
    <step><action>In <icon>Network Neighborhood</icon>, locate
⇒and double-click the computer where the printer you want to
⇒use is located. </action>
9: <tip targetgroup=“beginners”>To see which computers have shared
⇒printers attached, click the <menu>View</menu> menu,
10: click <menu>Details</menu>, &amp;amp; look for printer names or
⇒descriptions in the Comment column of the Network
⇒Neighborhood window.</tip>
11: </step>
12: <step>
13: <action>&doubleclick; the printer icon in the window
⇒that appears.</action>
14: </step>
15: <step>
16: <action>
17: To set up the printer, <xref linkend=“id45”>follow the instructions
⇒</xref> on the screen.
18: </action></step>
```

```
19: </procedure>
20: <rule form=“double”/>
21: <tip>
22: <p>After you have set up a network printer, you can use it as
⇒if it were attached to your computer.
⇒For related topics, look up &quot;printing&quot; in the Help Index.
23: </p>
24: </tip>
25: </helptopic>
```

Earlier in this chapter you corrected a DTD describing the structure of a help topic. (Refer to Listing 9.3.) Now you want to relate both. Do this by referring to the file dtdv.dtd in the DOCTYPE declaration of wfq.xml:

```
<!DOCTYPE helptopic SYSTEM “dtdv.dtd” []>
```

The change is shown in Listing 9.7.

Listing 9.7 wfq.xml—The DTD Is Now Referred to in the DOCTYPE Declaration

---

```
1: <?xml version=“1.0” ?>
2: <?protext objid=“I5678” ?>
3: <!DOCTYPE helptopic SYSTEM “dtdv.dtd” [
4: <!ENTITY doubleclick “Double-click”>
5: ]>
6: <helptopic>
7: <title keyword=“printing,network;printing,shared printer”>How
⇒to use a shared network printer?</title>
8: <procedure>
9: <step><action>In <icon>Network Neighborhood</icon>, locate and
⇒double-click the computer where the printer you want to use is
⇒located. </action>
10: <tip targetgroup=“beginners”>To see which computers have shared
⇒printers attached, click the <menu>View</menu> menu,
11: click <menu>Details</menu>, &amp; look for printer names or
⇒descriptions in the Comment column of the Network Neighborhood
⇒window.</tip>
12: </step>
13: <step>
14: <action>&doubleclick; the printer icon in the window that
⇒appears.</action>
15: </step>
16: <step>
17: <action>
18: To set up the printer, <xref linkend=“id45”>follow the
⇒instructions</xref>   on the screen.
19: </action></step>
20: </procedure>
21: <rule form=“double”/>
22: <tip>
23: <p>After you have set up a network printer, you can use it as
⇒if it were attached to your computer. For related topics,
⇒look up &quot;printing&quot; in the Help Index.
24: </p>
25: </tip>
```

```
26: </helptopic>
```

Let’s start checking:

```
jre -cp .;c:\datachannel\dxp\classes dxpcl -s -v c:\xmlex\wfq.xml
```

The result is

ERROR: Invalid content : procedure
Possible: rule
Location: file:/c:/xmlex/wfq.xml:8:2

ERROR: element not declared in DTD “rule”
Location: file:/c:/xmlex/wfq.xml:21:2

ERROR: attribute hasn’t been declared in the DTD “form”
Location: file:/c:/xmlex/wfq.xml:21:7

FATAL ERROR: java.lang.NullPointerException:
Location: :0:0

What’s the defined content of your help topic? Your help topic needs to start with a title, followed by a rule, and after that the procedure. This isn’t the case in your document.

Let’s start to add the rule to your document:

```
7: <title keyword=”printing,network;printing,shared printer”>How to
⇒use a shared network printer?</title><rule/>
```

rule is an empty element. Remember this special syntax:
<rule></rule>
Or
<rule/>

Let’s run your parser again:

ERROR: element not declared in DTD “rule”
Location: file:/c:/xmlex/wfq.xml:8:2

ERROR: element not declared in DTD “rule”
Location: file:/c:/xmlex/wfq.xml:21:2

ERROR: attribute hasn’t been declared in the DTD “form”
Location: file:/c:/xmlex/wfq.xml:21:7

FATAL ERROR: java.lang.NullPointerException:
Location: :0:0

The specification defines the following validity constraints:

- An element is valid if there is a declaration.
- An attribute must have been declared.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

If this isn't the case, you have errors. So add declarations for the missing elements and attributes:

```
<!ELEMENT rule EMPTY>
<!ATTLIST rule form (single | double | dotted) “single”>
```

When you parse again, you receive the following:

ERROR: Invalid content : p
Possible: iconbmp, icon, menu, #PCDATA, , xref
Location: file:/c:/xmlex/wfq.xml:23:2

ERROR: element not declared in DTD “p”
Location: file:/c:/xmlex/wfq.xml:23:2

ERROR: unknown ID referred “id45”
Location: file:/c:/xmlex/wfq.xml:26:14

Found errors/warnings: 0 fatal error(s), 3 error(s) and 0 warning(s)

The tip after the second rule contains a p, which isn't allowed by the DTD. Let's remove the start and end tag of p within element tip from wfq.xml.

ERROR: unknown ID referred “id45”
Location: file:/c:/xmlex/wfq.xml:25:14

Found errors/warnings: 0 fatal error(s), 1 error(s) and 0 warning(s)

- IDREF values must match the value of some ID attribute in the XML document, which isn't the case here. There's no element in your document with an attribute of type ID with the value "id45".

Remove the xref element, which leads to a valid document.

Checking Your XML Files with XML for Java

At the DOS prompt, type the following:

```
jre -cp c:\xml4j\xml4j.jar trlx c:\xmlex\wfq.xml
```

where

- jre is used to invoke the Java Runtime Engine.
- -cp is used to set the classpath. In this case you refer to the jar (Java archive file) with the name xml4j.jar.
- trlx is the Java class that does the parsing.
- c:\xmlex\wfq.xml is the file to be checked.

Walkthrough of an XML File Check with XML for Java

- Use the file in Listing 9.5. With XML for Java, you'll generate the following error messages:

- c:\xmlex\wfq.xml: 21, 22: Attribute 'form' of element 'rule'
⇒is not declared.
c:\xmlex\wfq.xml: 21, 22: Can't find content model of '<rule>'.
c:\xmlex\wfq.xml: 24, 5: Can't find content model of '<p>'.
c:\xmlex\wfq.xml: 25, 7: Content mismatch in '<tip>'. Content model is ''(#PCDATA|icon|menu|xref|iconbmp)*'.
c:\xmlex\wfq.xml: 26, 13: Content mismatch in '<helptopic>'.
⇒Content model is '(title,rule,procedure,rule?,(tip|warning|note))'.
c:\xmlex\wfq.xml: 18, 45: ID 'id45' is not defined in the document.

The problems encountered are

- No element and attribute list declarations for the element rule.
- Using a p inside the element tip, which isn't allowed according to the DTD.
- The title needs to be followed by a rule element, according to the DTD.
- The xref element, through its attribute linkend of type IDREF, refers to a unique identifier ID, which doesn't exist in the XML document.

Summary

In this chapter you used two parsers written in Java, DXP from Datachannel and XML for Java from IBM, to do the following:

- Check the declarations in the DTD.

Remember that the treatment is different for declarations in the external subset compared to the internal subset.

- To check if XML documents comply with the rules defined in the Document Type Definition (DTD).

Q&A

Q Why should I bother to have valid XML files?
A Well, it is much easier to use valid XML files. If you know that the element tip may appear only in this or that context, you only need to worry about what to do with the element tip in your program or your style sheets for the contexts known. Whereas if a tip can appear everywhere, your program or style sheet will become much more complicated and elaborate.
In addition, it is much easier to exchange files when you know that they use the same grammar.
Q Does a browser care?
A No, a browser can work with a well-formed XML file. A well-formed file carries enough information for the browser to create a tree structure and render the file.

The quality of rendering depends on the styles co-delivered, of course.

Q Is there other software that cares?
A Authoring tools will care about validity. If you want to start editing a new XML file, the software will ask you which DTD to use. After that, the software will guide you through the editing process by only making available those tags that are allowed by the content model. In this way, you're sure to have well-formed and valid XML files.
Q Are there other validating parsers available?
A Yes. There's MSXML from Microsoft and Larval from Tim Bray at Textuality, both written in Java.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Exercises

1. This exercise involves the following XML file, valq.xml:

```
1: <!DOCTYPE references SYSTEM “docb.dtd” [
2: ]>
3: <refentry id=“refentry.xref”>
4:     <refmeta>
5:         <refentrytitle>delete_mark</refentrytitle>
6:     </refmeta>
7:     <refnamediv>
8:         <refname>delete_mark</refname>
9:         <refpurpose>Deletes the currently
⇒selected region and puts it in the specified paste buffer
⇒</refpurpose>
10:     </refnamediv>
11:     <refsynopsisdiv>
12:         <title>Synopsis</title>
13:         <cmdsynopsis>
14:             <command>delete_mark</command>
15:             <arg><option>-append</option></arg>
16:             <arg><replaceable>buffername</replaceable></arg>
17:         </cmdsynopsis>
18:         <cmdsynopsis>
19:             <command>delete_mark</command>
20:             <arg>null</arg>
21:         </cmdsynopsis>
22:     </refsynopsisdiv>
23:     <refsect>
24:         <title>Description</title>
25:             <para>This command deletes the currently selected
```

```
⇒region and puts it in the paste buffer specified.
⇒If no buffer name is supplied, the current paste buffer is
⇒used (see <citerefentry><refentrytitle>set paste=buffername
⇒</refentrytitle></citerefentry>).
26: If the <literal><replaceable>buffername</replaceable>
⇒</literal>is <literal>null</literal>, the selection is
⇒deleted without copying it to a paste buffer.
27: This command corresponds to the <interface>Delete</interface>
⇒menu item on the default <interface>Edit</interface> menu.
28: If the <option>-append</option> option is selected, text
⇒is inserted at the end of the buffer rather than
⇒completely replacing its contents.</para>29 <command>dm
⇒</command> is a synonym for <command>delete_mark</command>.
30: </para>
31: </refsect>
32: <refsect>
33:    <title>Examples</title>
34:    <screen>Command: <userinput>dm</userinput></screen>
35:    <screen>Command: <userinput>dm bufA</userinput></screen>
36:    <screen>Command: <userinput>dm -append buf2</userinput>
⇒</screen>
37:</refsect>
38:</refentry>
```

And the following DTD file, docb.dtd, which is referred to in valq.xml:

```
1: <!ENTITY % in-line “option | replaceable | citerefentry
⇒| refentrytitle | literal | userinput | interface |
⇒command | arg” >
2: <!ENTITY % paracontent “(#PCDATA | %in-line;)*”>
3: <!ELEMENT refentry (refmeta?, refnamediv, refsynopsisdiv,
⇒refsect+)>
4: <!ATTLIST  refentry id ID #REQUIRED>
5: <!ELEMENT refmeta (refentrytitle, refmiscinfo*) >
6:    <!ELEMENT refentrytitle (#PCDATA)>
7:    <!ELEMENT refmiscinfo (#PCDATA)>
8: <!ELEMENT refnamediv (refname, refpurpose)>
9:    <!ELEMENT refname (#PCDATA)>
10:    <!ELEMENT refpurpose %paracontent;>
11: <!ELEMENT refsynopsisdiv (title, cmdsynopsis+)>
12:     <!ELEMENT cmdsynopsis (command, arg+)>
13: <!ELEMENT refsect  (title, (para | screen)+ )>
14:    <!ELEMENT title (#PCDATA)>
15:    <!ELEMENT para %paracontent; >
16:    <!ELEMENT screen %paracontent; >
17: <!ELEMENT option (#PCDATA)>
18: <!ELEMENT replaceable (#PCDATA)>
19: <!ELEMENT citerefentry (#PCDATA | refentrytitle)*>
20: <!ELEMENT literal (#PCDATA)>
21: <!ELEMENT userinput (#PCDATA)>
22: <!ELEMENT interface (#PCDATA)>
23: <!ELEMENT command (#PCDATA)>
24: <!ELEMENT arg (#PCDATA | option | replaceable)*>
```

- Detect the errors and the error messages that the parser will generate.
- Check the file with the parser of your choice.
- Correct all problems discovered.

2. Using the XML file you made during the previous days.
    - Introduce errors.

Use the W3C recommendation as your reference.

- Parse the file with different parsers.
- Study the differences between the results of different parsers.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

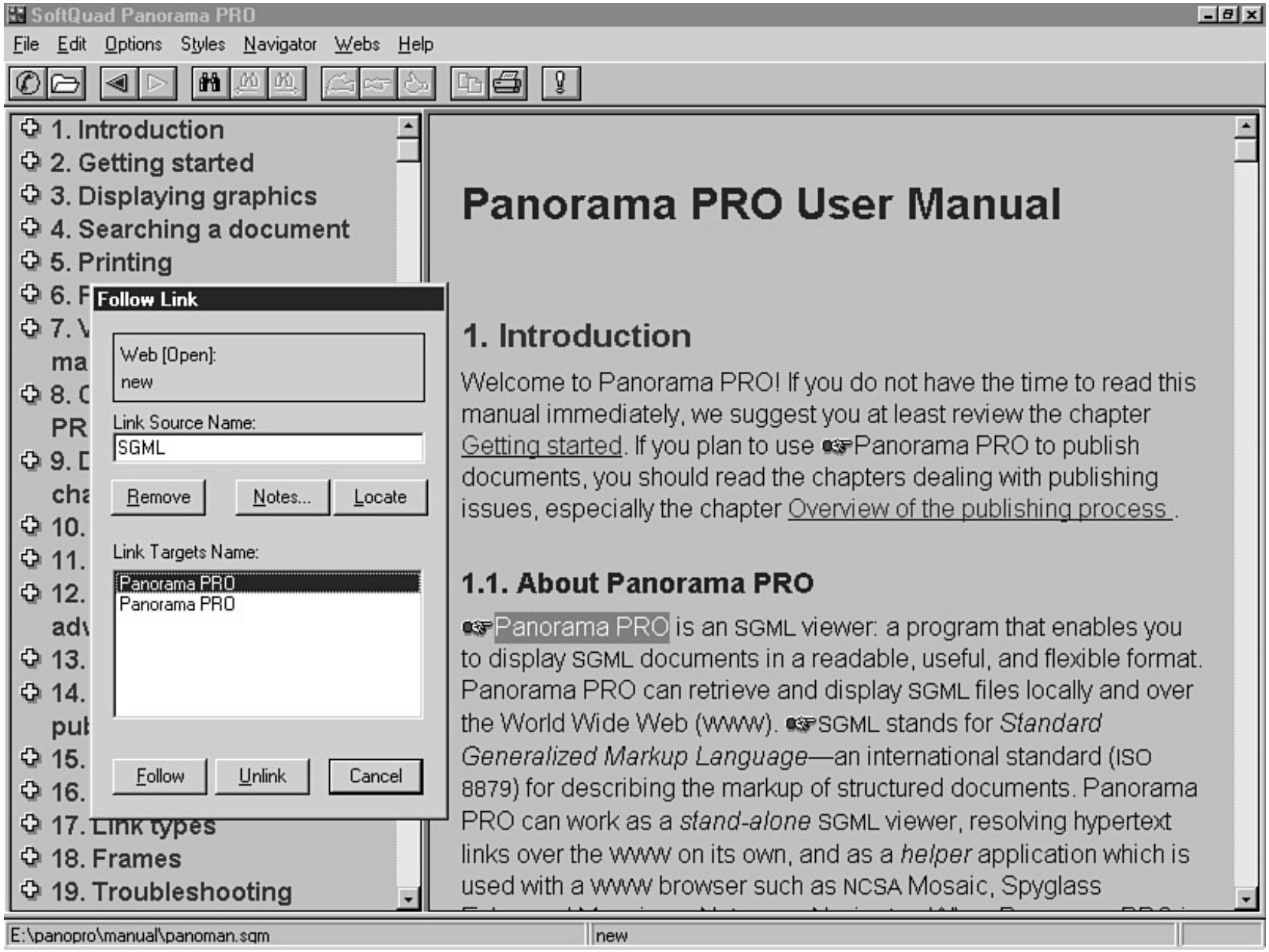
Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 10
Creating XML Links

Amidst all the hype and speculation surrounding XML as a language, little attention has been paid to a pair of related proposals, XLink and XPointer, that are potentially far more exciting than the XML language itself. These two proposals, under the generic name of XML linking, represent a revolution in the way documents can be linked. (In the first proposal, they were combined as a common XML Linking Language (XLL) document.)

Today you will learn the following:

- The differences between HTML and XML hyperlinks
- The various types of hyperlinks in XML and how they can be used to link multiple points, link read-only documents, and describe links
- How to use the various attributes of link elements to control how and when links are activated
- How to remap the attributes of elements in existing XML documents to add hyperlink behavior that was not originally present



Although XLink and HTML are constantly compared throughout this chapter, it is not at all necessary for you to understand any of the details of HTML hyperlinking. However, if you do understand at least a little of HTML, you will appreciate even more just how exciting the possibilities are that XLink opens up.

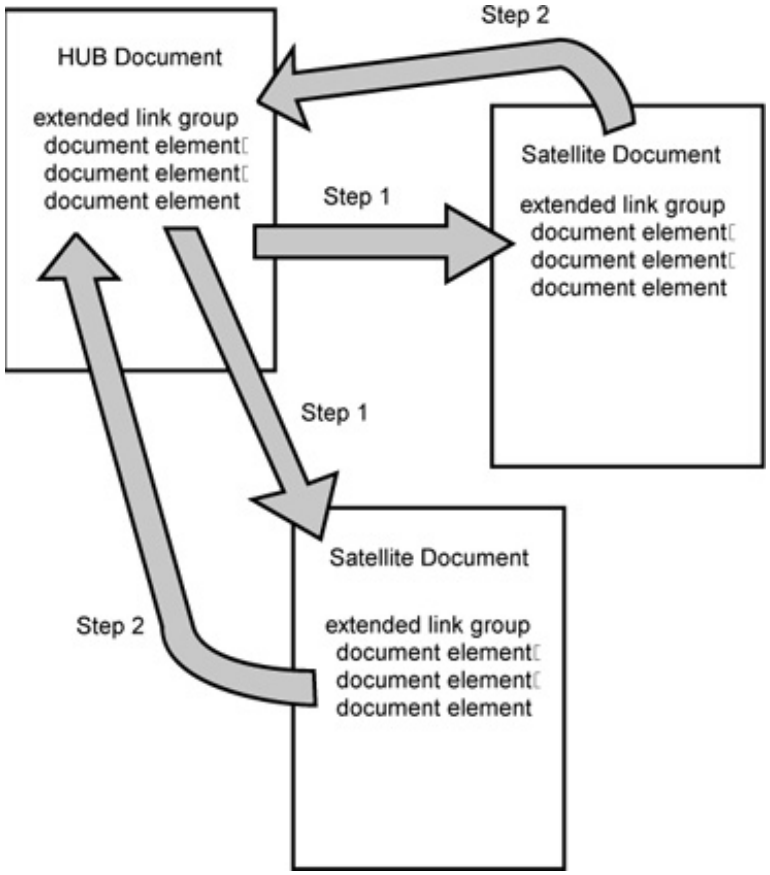
Hyperlinks

Anyone who has ever opened a Web page knows immediately what hypertext looks like. What distinguishes hypertext from normal text are hyperlinks, the characteristic blue underlined text that identifies hot spots that will move you to other pages when they're clicked with the mouse.

Arguably, it was the capability to link documents together that made the World Wide Web such a great success. It was certainly one of the design goals behind the original HTML. (At CERN—the European Laboratory for Particle Research—in Switzerland, the wide distribution of projects and their related

documents prompted Tim Berners-Lee to look for some way to associate documents with each other regardless of their physical location on the computer network.)

A hyperlink (or link for short) is an association between two pieces of text, between a piece of text and an object (such as the graphics that are included on Web pages by placing a link to the image file), or even between an object and a piece of text (such as the image map mechanism in HTML that allows the link to point to a different document according to where you click on an image).



In the official terminology of the XLink proposal, a link is “an explicit relationship between two or more data objects or portions of data objects.” For the sake of simplicity, this chapter will just talk about links between XML elements, regardless of what they contain.

The standards also talk about link resources, objects that participate in links. I prefer to use the rather less accurate but more manageable concept of a target, even though actuating a link in XML can result in far more than HTML’s simple link transversal.

In the simple terms of HTML linking, a link is the association between a source and a target. The target may be a complete HTML page, in which case the description of the target (the locator) is a Universal Resource Identifier (URI), or it may be a named element within an HTML page, which is identified using the # symbol, called a fragment identifier, followed by the value of the NAME attribute of the target element. Tomorrow’s discussion of the XPointer language gives more details about using fragment identifiers with XML documents.

The HTML code for the source of the link would look something like this:

```
<H3>Simple Linking</H3>
<P>This <A HREF=“<http://writer.xs4all.nl/simplelink.html#S15>”>link</A>
points to a specific section in the document. </P>
```

And the HTML code for the section in the target document would look something like this:

```
<H4><A NAME="S15">Target Section</A></H4>
<P>This section is the one I want to link to. </P>
```

HTML can safely talk about the sources and targets of links because its linking mechanisms are so simple. XML's linking language, XLink, is so much more powerful than HTML's that it is no longer possible to refer to sources and targets. XLink talks about *linking elements* instead of sources—links can be bidirectional, so the end of a link could be both a source and a target. And it talks about *resources* instead of targets—a resource could be a piece of data, the result of a database query, or an external link that acts as an intermediary en route to the final destination.

When you click on a link in an HTML page, a new page is invariably opened (although it might be in a new window or frame). In this simple scenario, it makes sense to talk about *following* a link because you do appear to travel from one document to another. As you will learn later, XLink's syntax allows you to specify multiple or grouped link locators, and the way in which they are processed can be more of an implementation issue (browser software) than a linguistic matter. For this reason, XLink talks about *traversing* a link.

Now that we've got some of the terminology sorted out, let's look at what it all actually means in practice.

Locators

XML links work with link elements. In turn, the link elements contain locators, in the form of attributes or other elements, that point to specific locations within a resource.

Generally, a locator is a URI, a fragment identifier, or a URI combined with a fragment identifier. Locators for XML documents are extended pointers (XPointers), which you will learn about tomorrow.

The syntax of locators allows you to use the following variations:

- URI#fragment—This fetches the whole of the resource identified by the URI and then extracts the part identified by the fragment identifier.
- URI|fragment—The application can decide how it will process the URI in order to extract the resource. For example, this could be used to only retrieve a specific part of a document instead of the complete document.

If the fragment identifier is a character string that complies with XML's rules for a name, the string is treated as the value of the id attribute of an XML element. The locator afile.html#ht7 would therefore point to the element in the file afile.html, whose id attribute value is ht7 (for example, <para id="ht7">). This built-in interpretation is meant to encourage you to use ID addressing, which in itself is a good practice because it is completely unambiguous. This form of address does carry with it the risk of confusion, however, if your software does not ensure that all element ID values are unique within a document.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

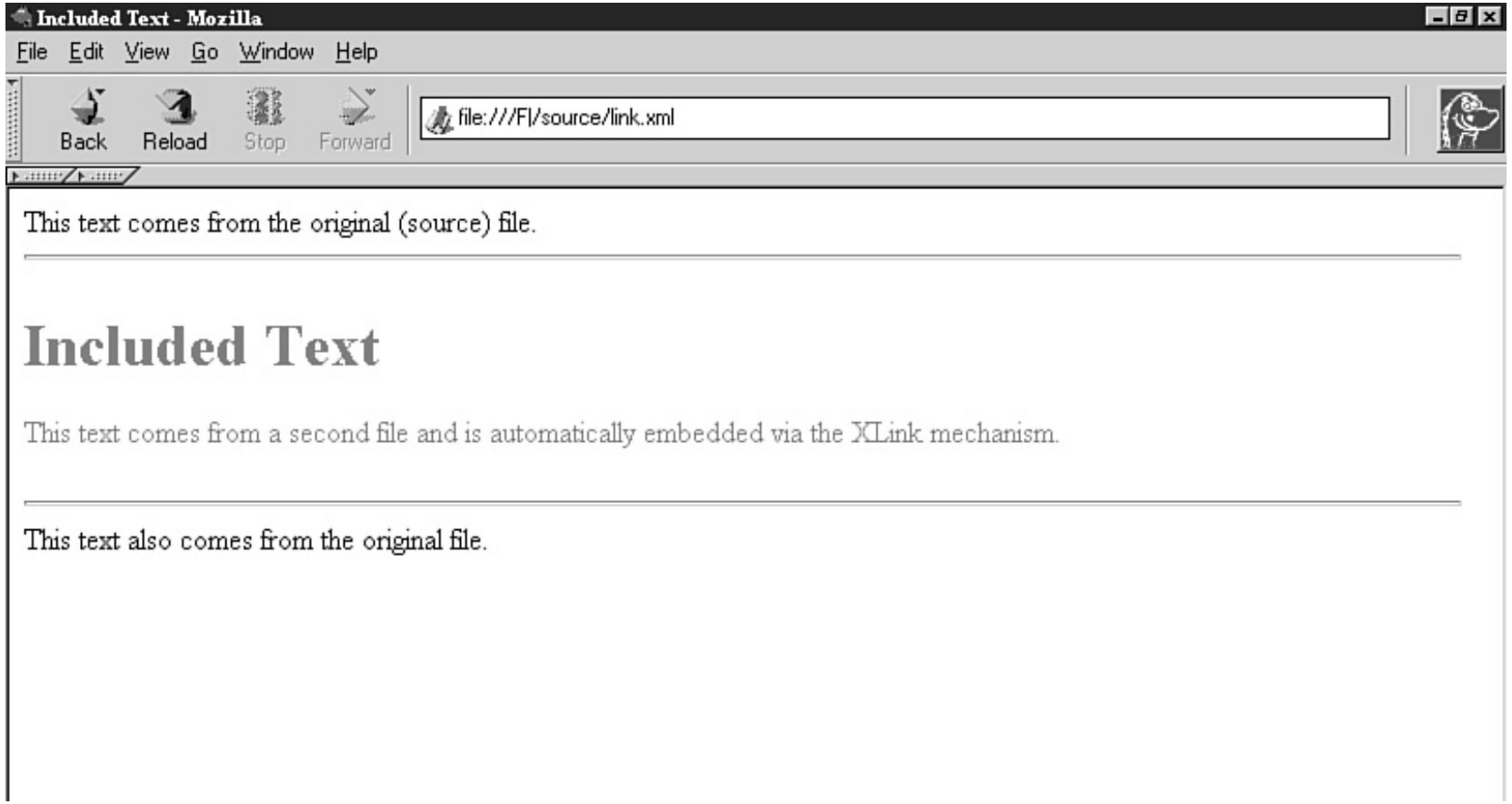
Link Elements

In HTML, there are only two elements that can act as links: A (anchor) and IMG (image). In XML, a link exists simply because the link element says it does. The existence of the link is identified by the element's attributes. This means that absolutely any XML element can act as a linking element if you ensure that it has the right attributes. Later today you will learn that you can actually redefine (remap) attributes as well. For now, we'll concentrate on the idea that a specific set of XLink attributes identify and describe a link.

The primary attribute that identifies an element as a link element is the xlink:form attribute, whose declaration in an XML DTD would look something like this (not forgetting that an element that is declared as a link element must also conform to the structure that you want to specify in the DTD):

```
<!ELEMENT CORRELATION ANY>
<!ATTLIST CORRELATION
     xlink:form CDATA #FIXED value >
```

Here, *value* is either simple or extended. (The value could also be locator, group, or document, but these are not linking elements in themselves but related elements.) The meanings of these values are explained in the following sections.



The current XLink working draft (March 3, 1998) describes the use of the xml:link attribute. Since publication of this draft, the XML Namespaces draft has made use of the xml: prefix inadvisable. In an email message on May 5, 1998, Eve Mailer, one of the editors of both the XLink and XPointer proposals, stated that this attribute was to be renamed xlink:form in the next version of the draft to avoid namespace problems. The publication of the next version has not been fixed, but I have already implemented the changed

syntax in this description.

Simple Links

Simple links closely approximate the simple HTML linking mechanism described earlier today. Simple links have only one locator, and only work in one direction:

```
<simple.link xlink:form="simple"
href="<http://me.com/title.xml>">see also</simple.link>
```

This simple link element contains a piece of text that acts as a resource, one end of the link. This kind of link is called an inline link. You'll learn about inline and out-of-line linking later today.

Don't confuse the term *inline link* with links that are contained inside the current document—internal links—and those that point to other documents—external links. An inline link is simply a link with a piece of text that acts as one end of the link (like the A elements in HTML).

Extended Links

In simple links, XLink doesn't really offer that much more than the basic linking that you're accustomed to seeing in HTML documents (although even here there are a few substantial improvements). It is in extended links that XLink, and thus XML, really comes into its own. If nothing else can be considered special about extended links, it's exciting enough that they do not need to be physically contained in one of the XML files that the links go either to or from. It goes further than this, though. Extended links allow XML documents to do the following:

- Link together any number of resources, resulting in multiple targets instead of a simple one-to-one relationship as in HTML.
- Link to and from resources that cannot contain the links themselves. This includes such things as graphics files, sound files, read-only documents, and so on, whose coding doesn't allow you to modify them to embed links.
- Enable the (dynamic) filtering, addition, and modification of links. Imagine being able to modify the links at a certain point so experienced readers of a technical manual can take a different path than novice readers.
- Enable application software to process the links in many other ways according to its own needs.

Remember that the XLink and XPointer developments are still working drafts. Unfortunately, this means that the proposal has not been implemented in the major Web browsers (other than very basic support of XLink in Netscape's Mozilla source code), and it is highly unlikely that it will be implemented until the proposal has become more stable.

Support is beginning to appear in other packages, however. The HyBrick software package (http://collie.fujitsu.com/hybrickSWA) supports it, there's an experimental implementation of parts of XPointer in the public domain Jade DSSSL processor (http://www.jclark.comSWA), and parts of it are being implemented in the IndelvIT viewer/editor package (http://www.indelv.comSWA).

An extended link doesn't actually point to anything or link anything together. An extended link element identifies itself through its xlink:form attribute value

and contains a set of locator elements that together form the extended link:

```
<comment xlink:form="extended">
     <opinion xlink:form="locator" href="note4"/>
     <reference href="#section2.1"/>
     <reference href="<http://here.com/appB.html>"/>
     <reference href="references.htm"/>
</comment>
```

As with the other link elements, the nature of the extended link element is specified by the xlink:form attribute. Here, a comment element declares itself to be an extended link and an opinion element declares itself to be a locator element. As you can see, this can be done in the element start tag without having to use a DTD. The declarations for these elements in a DTD could look something like this:

```
<!ELEMENT comment ANY>
<!ATTLIST comment
     xlink:form     CDATA     #FIXED "extended">

<!ELEMENT opinion ANY>
<!ATTLIST opinion
     xlink:form     CDATA #FIXED "locator">

<!ELEMENT reference ANY>
<!ATTLIST reference
     xlink:form     CDATA #FIXED "locator">
```

Note that I have used ANY as the content model for these elements, allowing any element to be contained inside one of these elements.

Don't forget that if you are going to validate the XML document, the elements used for linking must still comply with the structural rules declared in the DTD.

Where extended link elements contain locator elements that in turn contain elements themselves, only the locator elements themselves, and not the elements they contain, are considered to be link resources.

So how would this work in practice? Well, let's look at two paragraphs in an XML document that you want to cross-reference with each other. The easiest way to do this is to put a simple link in each paragraph that points to the other paragraph, as shown in Listing 10.1.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Previous | Table of Contents | Next

Listing 10.1 Cross-Referencing with Simple Links

```
1: <para id="idea1"><xref href="idea2">A first
2:   impression of the subject matter
3:   would inevitably lead to the
4:   conclusion that ... </xref></para>
5: <para id="idea2"><xref href="idea1">Contrary
6: to our earlier conclusion, it
7: would appear that the evidence points
8: to a different ... </xref></para>
```

Listing 10.1 shows one direct solution to cross-referencing the two paragraphs, but what happens if you then expand the thesis by including a third point that you want to bring into the discussion? At best, you would have to do some annoyingly repetitious link editing; at worst, you might have to completely rethink all the affected markup code.

Extended links, and in particular a variety of extended link called an out-of-line link (which is covered later in this chapter) allow associative links like these to be created and maintained much more easily. Putting the links somewhere else in the document, perhaps at the start to make them easier to find, or even in a separate document, can make it much easier to make the links, find them, and keep them up to date, as shown in Listing 10.2.

In an earlier chapter you learned how to place the DTD in an external file. Later on you learned how you can easily break up an XML document into parts that you can pull together using entities. Later in this chapter you will learn how to physically include pieces of text in the current document. Here, you are learning how the links themselves can be contained in an external document. The suggestion of modular document composition already should be dawning on you. These XML features open up possibilities for ease of document management and construction that go far beyond anything even remotely possible in HTML.

Listing 10.2 Cross-Referencing with Extended Links

```
1: <argument xlink:form="extended">
2:      <theme xlink:form="locator" href="idea1">
3:      <theme xlink:form="locator" href="idea2">
4: </argument>
.
.
5: <para id="idea1">A first impression of the subject matter
6: would inevitably lead to the conclusion that ... </para>
.
.
.
7: <para id="idea2">Contrary to our earlier conclusion, it
8: would appear that the evidence points to a different ... </para>
```

How a viewer behaves when you click on one of the extended link resources shown in Listing 10.2, or even how it identifies the existence of an extended link, is an application issue and is not addressed by the XLink specification. However, a special icon might be displayed to show that there is associated material, or a menu could pop up from which you could choose one of the link ends. (There are other descriptive attributes for linking elements that would make this selection more purposeful; these are described later on.)

Figure 10.1 shows the menu list that is displayed in SoftQuad's Panorama SGML browser.

Figure 10.1 *Picking a destination for a multiple-ended link.*

In Figure 10.1, the two link ends next to the phrase "Panorama Pro" (indicated by the pointing hand icon) both reference the link end next to the word "SGML." Clicking on either of the Panorama Pro link ends will simply bring you to the SGML link end. When you click on the SGML link end (in Panorama all links are two-way), there are two possibilities, so Panorama displays a list of possible link ends so you can choose the one you want.

It would have been more exciting to show you multiple links in a true XML package, but as yet there is no mainstream support for XML links (apart from the rudimentary support offered in Netscape's Mozilla).

SoftQuad's Panorama is an SGML browser with an embedded HyTime engine that supports a lot of the advanced HyTime linking facilities—some of the same facilities that were inherited from HyTime by XML. Despite the fact that it is meant to be used with SGML, Panorama will still do a reasonable job of rendering XML code, provided that the XML code you load into Panorama is valid and you do not drift too far from pure SGML code. (An evaluation version can be downloaded from http://www.sq.comSWA.)

Interestingly, it would be a simple task to add links between elements long after the document has

been completed without actually having to change any of the content (especially when the links aren't contained in the participating document). This will make a lot more sense tomorrow when you learn how to point to whole ranges of elements using extended pointers.

The fact that you can add links to a document after it has been completed, without changing any of the content of the document, means that you can also link to and from parts of read-only documents, such as documents contained on CD-ROM.

Extended Link Groups

You've already learned that links can be located in external documents; this is achieved through the use of extended link groups. Like an extended link, an extended link group element doesn't point to anything or link to anything. Instead, it contains a set of document elements in which each document (identified by a URI) contains the link resources:

```
<xternal.refs>
     <ref.doc href="<http://here.com/biblio.html>"/>
     <ref.doc href="lists.htm"/>
</xternal.refs>
```

These element and attribute declarations could look something like this in the XML DTD:

```
<!ELEMENT xternal.refs (ref.doc*)>
<!ATTLIST xternal.refs
     xlink:form     CDATA     #FIXED "group"
     steps          CDATA     #IMPLIED>

<!ELEMENT ref.doc EMPTY>
<!ATTLIST ref.doc
     xlink:form     CDATA     #FIXED "document">
```

As with so many of the other XLink elements, no rules are given as to how a browser or viewer should behave when it encounters groups of links like this. However, with the additional descriptive attributes described later on, a viewer could display a pop-up list of all the associations of a particular type and allow you to pick one or more.

When using extended link groups, you are going to run the risk of one of the link document elements also containing an extended link group. What happens if that extended link group points back to the original document, or to a document containing yet another extended link group? It would take many linked extended link groups to land you in an impossible situation of infinite links and link loops. To prevent this from happening, you can declare a value for the steps attribute of the group element. This is a numerical value specifying how many layers of nesting you, as the author, will permit. (The application is not required by XLink to obey this attribute, but one can hope that the more user-friendly software packages will.)

If you are building a web around a central hub document, it would make a lot of sense to specify a value of 2 for the steps attribute of the base extended link group element (see Figure 10.2). This would prevent any extended link group elements in satellite documents from bouncing back through the original extended link group and getting viewers completely tied in knots.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Inline and Out-of-Line Links

In XLink, a link can be inline or out-of-line. By default, all link elements are inline (the value of the inline attribute is true), so you only need to declare the attribute if the link element is out-of-line (inline is false).

There is a lot of confusion about the meanings of these terms, especially when it is complicated by the idea of links being inside or outside a document. Whether a link is inline or out-of-line has absolutely nothing to do with whether it's located inside the document or not.

Simple links are essentially inline all the time. An out-of-line simple link is possible, but it is rather hard to imagine the actual use of such a thing. It would be a single-ended link, really, a pointer to a location without any other association... not much use to anyone.

Figure 10.2 *Using steps to limit linking with a hub document.*

Where the distinction between inline and out-of-line links is relevant is in extended links. Listing 10.3 shows an example of an inline extended link in an XML document.

Listing 10.3 An Example of an Inline Extended Link

```
1: <para id=“para4”>This is a
2:   <xref.list>psychosomatic
3:      <see.also href=“med1.xml”/>
4:      <see.also href=“<http://www.medical.com/defs.htm>”/>
5:      <see.also href=“para6”/>
6:   </xref.list> condition, brought on by ...
7: </para>
```

Here, the xref.list element contains character data that serves as part of the associative trail formed by the links. Therefore, the link element framed in the official terms of the XLink specification contains a local resource of the link.

If you look again at the same set of extended links, but rewritten as an out-of-line link, you should be able to see the difference straight away, as shown in Listing 10.4.

Listing 10.4 Using an Out-of-Line Extended Link

```
1: <xref.list>
2:    <see.also href=“para4”/>
3:    <see.also href=“med1.xml”/>
4:    <see.also href=“<http://www.medical.com/defs.htm>”/>
5:    <see.also href=“para6”/>
6: </xref.list>
.
.
.
7: <para id=“para4”>This is a psychosomatic condition,
8:  brought on by ... </para>
```

When the extended link is rewritten as an out-of-line link, it is no longer located in the same place as the link resources and contains a *pointer* to the local link resource rather than the local link resource itself.

Link Behavior

Links in HTML documents are either passive, like the A (anchor) element links that do nothing until you actually click on them, or active, like the IMG (image) element that automatically inserts the graphics file that it references (you can normally suppress this behavior by changing the Web browser options so that images are not loaded automatically). The link behavior is tied to that particular element, and there is little you can do to affect it.

In XLink, not only can you control when a link is activated, or traversed, you can control what happens when a link is traversed. This link behavior is controlled by the show and actuate attributes.

Link Effects

In HTML, an image is always embedded at the point at which its anchor element (IMG) occurs. Other links are simple jumps to new documents, either in the same window, a new window, or a frame in either the current window or a new one.

In XLink, the show attribute allows you to formalize this behavior for all links—rather than relying on a link with a particular name—and allows you to do a little bit more:

- show="embed"—When the link is traversed, the designated resource is embedded at the point where the traversal started. This allows you to create virtual documents composed of fragments of other documents, for example. Or, more practically, you can create such things as dynamic tables of contents and indexes by picking up parts of other elements and (seemingly) physically copying them to the link location.
- show="new"—When the link is traversed, the designated resource is displayed (or processed) in a new context. This normally means that the resource is displayed in a new window. But the important point is that whatever is done to the target resource doesn't affect the display of the starting resource. This could be used to include a piece of text from an external document without its style declarations affecting the current document.
- show="replace"—When the link is traversed, the designated resource replaces the resource at which the traversal started. This is the equivalent of an HTML link in which the current HTML page is replaced with the new one.

When using extended links, it is possible for you to declare a default show attribute value for the extended link element that will then override any default values declared for the locator elements in the DTD. For example, in this fragment, the first location element has the default show attribute value of replace, but the second location element has a show attribute value of new that overrides the default definition:

```
<extended show="replace">
     <location href="mypage.html"/>
     <location href="hispage.htm" show="new"/>
</extended>
```

Link Timing

In HTML, an image is always automatically embedded at the point at which its anchor element (IMG) occurs. You don't have to do anything unless you have disabled automatic image loading in the Web browser itself. Other links are passive, meaning that absolutely nothing happens until you actually click on one.

In XLink, the actuate attribute allows you to formalize this behavior for all links rather than relying on a link with a particular name, and it allows you to do specify what happens when a link element is encountered:

- actuate="auto"—When any of the other resources of the same link are retrieved, this resource is to be retrieved automatically. Used together with the show="replace" attribute declaration, this link behavior can be used to automatically redirect a viewer to a new page, for example.
- actuate="user"—The resource is not to be retrieved until there is an explicit request to do so. This is the equivalent of an HTML link in which nothing happens until you click on the source of a link.

When using extended links, it is possible for you to declare a default actuate attribute value for

the extended link element that will then override any default values declared for the locator elements in the DTD. For example, in the following fragment, the first location element has the default actuate attribute value of user, and the second location element has an actuate attribute value of auto that overrides the default value:

```
<extended actuate="user">
     <location href="mypage.html" show="embed"/>
     <location href="hispage.htm" actuate="auto" show="embed"/>
</extended>
```

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Search this book:
Go!
Previous | Table of Contents | Next

With this code, you might not get the results you expected. The first resource (mypage.html) will be retrieved, as expected. However, while doing this, the second resource (hispage.html, which has actuate=auto) will be retrieved as well! Because they are both embedded, you will see them both, but with other values of the show attribute, you could end up with something completely different than what you thought would happen.

When there is more than one embedded link and the actuation is automatic, you might expect one resource to be actuated. It will be, but the actuation could be masked by a second actuation. The result will not be disastrous, but you should be alert to the problems that declaring this link behavior might cause.

The behavior Attribute

An application can do many things with links. It can display special icons, play sounds, change display characteristics... the possibilities are limitless.

In the latest beta release of Internet Explorer 5 (November 1998), Microsoft has implemented an extension to the CSS style sheet syntax that comes very close to the behavior attribute. This extension, called binary behaviors, allows a piece of executable code (such as a program) to be attached to an HTML or XML element.

XLink includes a behavior attribute that allows you to declare any other behavior associated with a link. The content and significance of any value that you give this attribute, and how the application reacts to that attribute value, is left for the application developer to decide.

Link Descriptions

One of the things that's sadly lacking in linking HTML documents is that it's an all-or-nothing situation. Either there's a link or there isn't, and all links are exactly the same. There is absolutely no

way of describing the nature of the relationship between two link resources. You might follow an HTML link only to find out that the link between the source and the destination were rather more real in their creator's head than they are in the Web browser.

This lack of a way to provide a description of a link, the semantics, becomes even more critical when there are multiple resources participating in the link, as in extended links. Here, it becomes meaningless to offer multiple locators if there is no sensible way of either describing the relationship or, at the very least, distinguishing between two parallel resources.

XLink provides several attributes that allow you to describe links and link resources:

- A link can have a role attribute that identifies the meaning of the link to the application. For example:

```
<extended role="bibliographies">
     <location href="myfile.htm"/>
     <location href="hisfile.html"/>
</extended>
```

- When a link is inline, a local resource can have a content-role attribute and a content-title attribute. The content-role attribute identifies the part that the resource plays in the link (in addition to the link's own role attribute). The content-title attribute acts as a caption to explain to users the part that the resource plays in the link. For example:

```
<mylink content-role="see also" content-title="Reference">
as mentioned in my other book.</mylink>
```

- A remote resource can have a role attribute and a title attribute. The role attribute identifies the part that the resource plays in the link (in addition to the link's own role attribute). The title attribute acts as a caption to explain to users the part that the resource plays in the link. For example:

```
<extended role="bibliographies">
 <location href="myfile.htm"
   role="reference" title="This Chapter"/>
 <location href="hisfile.html"
   role="reference" title="Whole Book"/>
</extended>
```

Having declared roles and titles for links can help make sense of the seemingly illogical linking applied so often to HTML documents. Consider studying a piece of literature and choosing to display only the links to documents by same author, or that were written between certain dates!

Mozilla and the role Attribute

An example of the use of the role attribute can be found in the current (August 1998) release of Netscape's Mozilla code (which will most likely become Netscape Communicator 5 eventually). Mozilla uses role="HTML" to embed linked HTML documents into the current HTML document, as shown in Listing 10.5 (the base document) and Listing 10.6 (the "included" document). In technical terms, importing by linking like this is called transclusion.

Listing 10.5 Mozilla Code to Embed in an HTML Document

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

```
2:   <page>
3:     <section>
4:       This text comes from the original (source) file.
5:     </section>
6:     <Foot XML-Link="LINK" Role="HTML"
7:        Show="EMBED" href="include.htm" />
8:     <section>
9:       This text also comes from the original file.
10:    </section>
11:  </page>
```

Listing 10.6 HTML Code in the Document to be Embedded

```
1:  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2:  <HTML>
3:    <HEAD>
4:      <META HTTP-EQUIV="Content-Type"
5:         CONTENT="text/html; charset=iso-8859-1">
6:      <META NAME="GENERATOR" CONTENT="Mozilla/5.0 [en]
7:        (WinNT; N ;Nav) [Mozilla]">
8:     <TITLE>Included Text</TITLE>
9:    </HEAD>
10:   <BODY>
11:     <HR>
12:     <H1>
13:     <FONT COLOR="#FF6666">Included Text</FONT></H1>
14:     <DIV color="red">
15:       <FONT COLOR="#FF6666">
16:         This text comes from a second file
17:         and is automatically embedded
18:         via the XML-Link mechanism.
19:       </FONT>
20:     </DIV>
21:     <HR>
22:   </BODY>
23: </HTML>
```

The resulting display shows the two documents merged seamlessly (see Figure 10.3).

Figure 10.3 *Text inclusion by reference in Mozilla.*

Note that the current version of Mozilla supports an older version of the XLink proposal, when it was still known as XML-Link or XLL (XML Link Language). The combination of uppercase and lowercase characters (for example, "Role" and "Show"), also dating back to an earlier version of the proposal, has to be exactly as shown in Listing 10.5 or it won't work properly.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Attribute Remapping

Earlier today I suggested that it would be fairly easy to add links into an XML document after you had finished it.

As you will remember, any element can be a link. It is declared as being a link of a particular type through the xlink:form attribute. Although the names of the link elements can be freely chosen, the names of their attributes are fixed—actuate, behavior, content-role, content-title, href, inline, role, show, steps, and title.

Now, suppose that you want to use existing elements in an XML document as links. This very easy to do by using extended links located in an external document. The problems start when the elements in that document already have attributes with the same names as the XLink attributes. You can overcome this by using the xml:attributes attribute to remap existing attribute names onto new ones.

Consider this element declaration:

```
<!ELEMENT book (front, body, back) >
<!ATTLIST book
     title     CDATA                #REQUIRED
     role     (single|volume)     #IMPLIED>
```

If you wanted to use this element as a link element, the title and role attributes would have to be remapped. You can do this by putting a declaration in an internal DTD subset. (Remember that a DTD can be split into an internal and an external DTD subset and that the internal DTD subset takes precedence.) The declaration you put in the internal DTD subset could look something like this:

```
<!ATTLIST book
   xlink:form     CDATA  #FIXED “simple”
   xml:attributes CDATA  #FIXED “title mytitle role myrole”>
```

This XML code in the document would be correctly recognized as a simple link:

```
<book title=“Presenting XML” role=“main” myrole=“Reference”
```

```
mytitle="Other Books" href="<http://www.amazon.com/search?Light+North>"/>
```

Summary

This chapter looked at XLink, the linking part of XML's linking facilities. You've learned what the various types of links are, how they can be used, and their many attributes. Tomorrow you will learn about extended pointers (XPointers) and how they can be used to locate blocks of text, elements, groups of elements, and many other parts of an XML document.

Q&A

Q How would you define a link in XML to mimic HTML's behavior?
A A link element with show="replace" and actuate="user" would mimic HTML's behavior for a normal link to another HTML page.
Q What's the difference between an inline link and an out-of-line link?
A An inline link element contains one of its link resources. If it's a simple link, you could more or less say that the link element contains one end of the link.
Q Can a simple link be an out-of-line link?
A In theory, yes. In practice, it probably wouldn't be much use because such a one-ended link could do little more than associate certain semantics with a location.

Exercises

1. You have an XML document containing a chapter element divided into section elements. You knew you would want to make a table of contents, so each section element has been given an id attribute whose value is the number of the section. Draft the XML code (using an out-of-line extended link) to demonstrate how you would create an embedded table of contents.
2. You have a collection of XML documents, each representing the chapter of a book. You now want to create a brief table of contents that contains each chapter's title element from each of these documents. Draft the XML code (using an out-of-line extended link group) to accomplish this.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 11
Using XML's Advanced Addressing

In yesterday's lesson you learned about linking in XML, but that's only half of the story. Today we'll learn about *locators*—the means of tying links to actual parts of XML files.

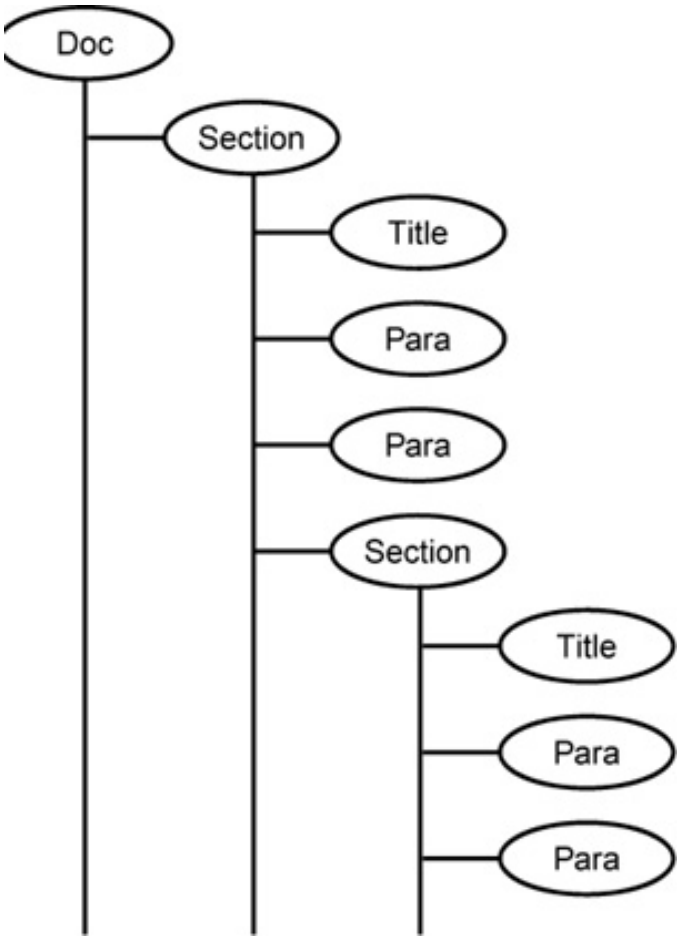
You have already learned one method of locating, using universal resource identifiers (URIs). In XML URIs can be used in much the same way that they are used in HTML—for navigating the World Wide Web. In this chapter I'm going to focus on a different kind of locator, the extended pointer (XPointer). In today's lesson you will:

- Learn a little about groves and XML's view of element trees
- Learn how to locate resources using absolute references (root(), origin(), id, html)
- Learn how to locate resources using relative terms (child, descendant, ancestor, preceding, psibling, following, fsibling)
- Learn how to locate link resources consisting of multiple elements and ranges of elements (spans)
- Learn how to select link resources by instance, element, attribute, and element content

Extended Pointers

As you may remember from yesterday's lesson, linking in XML is concerned with associating link resources with each other. (A link resource is really a pointer to one or more locations, or a means of resolving a pointer into a location.) You can use almost anything to resolve a location, such as a Web server CGI script or a database query. XML's extended pointers (XPointers) enable you to point into XML resources in a wide variety of ways. You can, as you will see, make full use of the structure of the document that is identified by means of the XML elements and attributes.

Extended pointers are described in a separate proposal called the *XML Pointer Language* (XPointer), which used to be part of the common XLL (XML Link Language) proposal.



You can use an extended pointer with a URI to locate a link resource more precisely. If you point into an XML document, you must use an extended pointer. If you are pointing into something other than an XML document, the XPointer recommendation does not specify or restrict the syntax of the locator in any way, but you can still use some extended pointers.

Documents as Trees

Before you can understand how extended pointers work and how to use them properly, it's important for you to understand how an XML link processor (whatever software package is resolving a link to find the actual ends) "sees" the structure of an XML document.

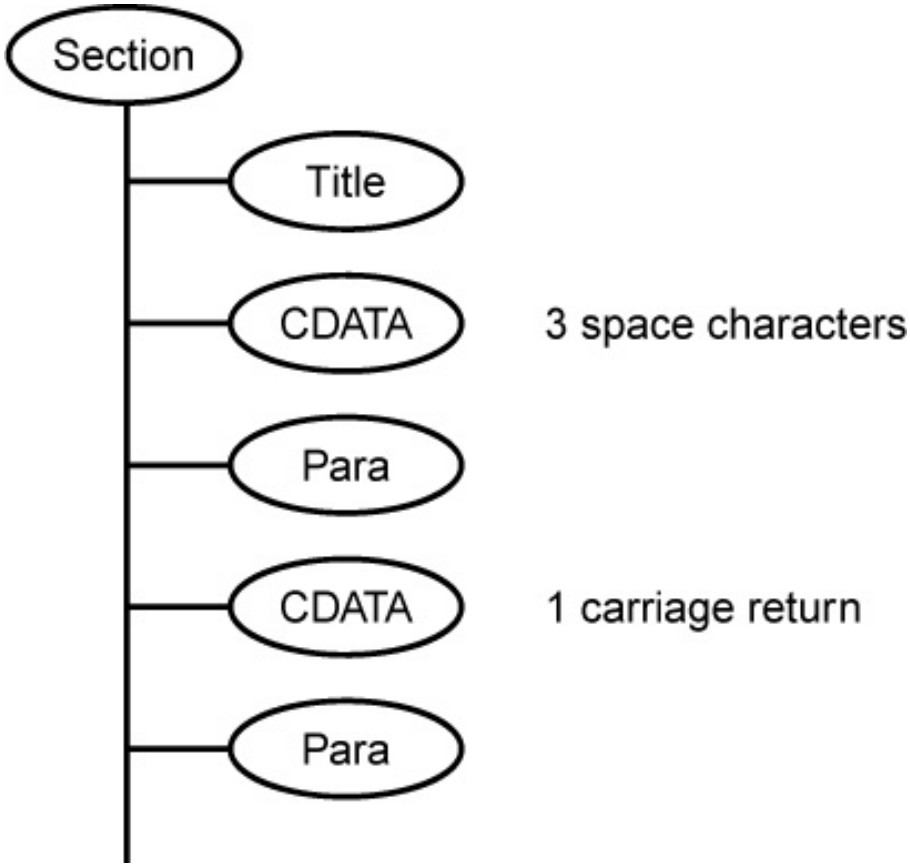
As you have worked your way through this book, you have no doubt become used to the idea of thinking of an XML document as being composed of trees of elements; the "root" element is at the top or base (depending on how you want to visualize it), and all the other elements branch off from it.

Figure 11.1 *A typical XML element tree.*

An ordinary tree structure is, however, too simple for discussing extended pointers. If I combine an absolute location with a relative link that begins somewhere deep inside the document, the resolution mechanism has to combine the tree that starts with the root element with the tree that starts with the element that acts as the start point for the relative location. From the viewpoint of an extended pointer, an XML document isn't just one element tree, it is hundreds.

A tree can be constructed for every possible entry point or reference point, so instead of talking about a tree we use the term *grove*. A grove is essentially a collection of trees; it is too small to be a forest but is more than just a super tree.

Each point in the tree is called a *node*. We've been talking about trees of elements, and so you'd suppose that nodes are just elements. Not so. A node can be an element, but it can just as easily be a piece of text (character data) within an element.



When you learn about DSSSL you'll see that a grove can be constructed using elements and attributes. This is not important for linking because you cannot use attributes as link resources.

Consider an element structure declared as follows:

```
<!ELEMENT section (#PCDATA | title | para )>
```

This is a typical mixed content element. It can contain both raw (untagged) character data and elements (title and para). Look at the following code fragment:

```
<section><title>Bert Goes to Bed</title> <para>Hello, Bert.</para>
<para>Today is going to be a very special day.</para></section>
```

Let's now try and plot this fragment as a tree. It looks simple enough; we have a section element that includes one title element and two para elements. Or does it? Look at the fragment again, then look at what I make of it in Figure 11.2.

Figure 11.2 *An element tree with hidden nodes.*

In Figure 11.2 I identified one section element containing a title element, a node of character data (3 whitespaces), a para element, an intervening character data node (a carriage return character or even, depending on the underlying operating system, a carriage return and a line feed character), and a last para element.

While this is a carefully chosen example of what can go wrong, these types of hidden nodes (not so much hidden as forgotten) can lead to some very unexpected results when you try to use extended pointers that employ node counting to find the link resource.

Location Terms

There are two types of location terms: absolute terms and relative terms.

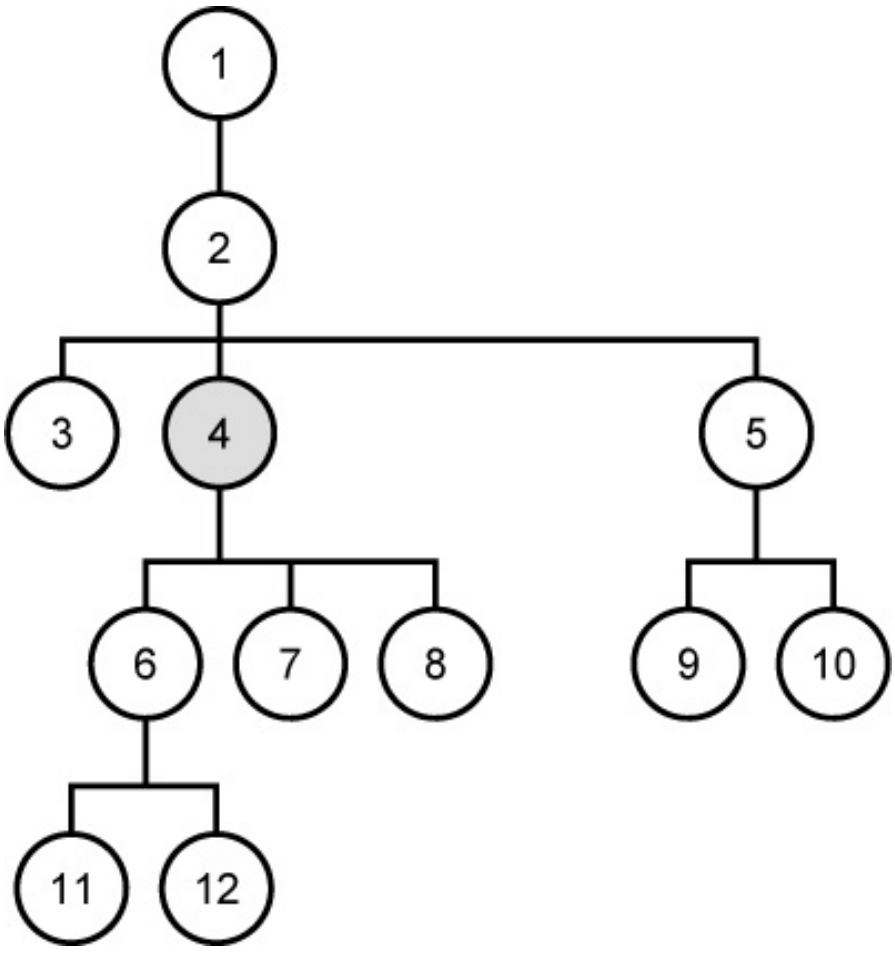
Absolute terms are much like the location pointers that you may be more used to seeing in HTML. These terms more or less ignore any structural relationships and point directly to specific elements.

Relative terms are easier to appreciate, and much more powerful than absolute terms because they mimic the kind of route description you might give to a friend to find your office.

Absolute Terms

An extended pointer can use one of the following absolute location terms:

- root()—States that the location source for the location term is the root element of the current document. Note that the keyword root is followed by a pair of brackets (technically called an *empty argument*) to prevent it from being confused with IDs.
- origin()—States that the location source for the location term is the link resource at which traversal started, rather than the current root element. Note that the keyword origin is followed by a pair of brackets (technically called an empty argument) to prevent it from being confused with IDs.
- id(name)—States that the location source for the location term is the element (it has to be a unique element) that has an attribute declared as being an ID class. The value of that attribute is name. The actual name of the attribute could be absolutely anything, hence the importance of explicitly declaring attributes that you intend to use as IDs.
- html(name-value—States that the location source for the first location term is the first element of type A that has a name attribute with a value of name-value (this is exactly the same as selecting a target in HTML using a # symbol).



If you do not specify the location source of the first location term in a traversal, the *root element* of the XML document that is the containing resource is used by default. This means that, officially at least, you shouldn't ever need to specify root(). However, including root() can't do any harm and it can make the intention of your extended pointer more clear to a human reader.

Relative Terms

In order to use a relative location term, you must already have a starting point (a location source). You can use relative location terms to move around the target XML document by describing a location in relation to where you are at that moment.

The ultimate goal of an extended pointer is to direct you to the correct location within an XML document. However, doing this is a bit like directing someone to a particular place in a city; it is difficult to give perfect directions on the first attempt. You might start with a known landmark (an absolute location), give a direction, count blocks or streets, and then describe buildings:

```
go to Central Station
go north on El Camino
take the third road on the left
go into the second building on the left
ask for Brian
```

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Each step of these instructions depends on the previous step being followed correctly. After one mistake the rest of the instructions will only get you lost.

Extended pointers work in exactly the same way. They are a sequence of *location terms*, which are either absolute locations or a series of locations where each is relative to the previous one. Extended pointers describe elements and nodes within an XML document in terms of various properties, such as their type or attribute values, or simply by counting them.

Each relative location term consists of a *keyword*, followed by one or more *steps*. Each term is a hop along a path to the final location of the link resource. In the same way as you'd tell someone to: "take the third right, go three blocks, turn left and it's the second on the right," so with extended pointers you could say, for example, "take the third child of this element, find its oldest brother, and take the second child element of that one." This route description could then be rewritten using extended pointers to look something like this:

```
child(3,DIV).psibling(-2).descendant(4.para).string(5,"bottle",1,1)
```

The following are the relative term keywords that you can use:

- child—Selects the child elements of the location source.
- descendant—Selects the elements that appear within the content of the location source.
- ancestor—Selects the elements in whose content the location source is found.
- preceding—Selects elements that appear before the location source.
- following—Selects elements that appear after the location source.
- psibling—Selects the preceding sibling elements of the location source.
- fsibling—Selects the following sibling elements of the location source.

Figure 11.3 shows a sample grove. Let me now try to explain these keywords with reference to a point in this grove, element number 4.

Figure 11.3 *Relative locations in an element tree.*

Using element number 4 as the location source, let's see what each of the absolute term keywords would select.

- child—Selects elements 6, 7, and 8 (the immediate children but not 11 and 12).
- descendant—Selects elements 6, 7, 8, 11, and 12 (the immediate children and their children).
- ancestor—Selects elements 1 and 2 (the elements that contain element 4).
- preceding—Selects elements 1, 2, and 3 (all the elements that appear before the location source).
- following—Selects elements 6, 7, 8, 11, 12, 5, 9, and 10 (all the elements that appear after the location source).
- psibling—Selects element 3 (the elements at the same level of hierarchy or generation that come before the location source).
- fsibling—Selects element 5 (the elements at the same level of hierarchy or generation that come after the location source).

Selection

So far you have learned to specify a type of location by using an absolute or relative term. These terms tell the processor what kind of element you're locating, but you still need to be able to specify which of those elements (or perhaps all of them) are of interest. Extended pointers offer several ways to select elements, as you will see.

Selecting by Instance Number

By specifying a number, you can pick out a particular element instance or range of element instances, for example:

```
child (2)
```

selects the second child element.

Specifying a negative number counts from the last instance backwards. For example:

```
(-1,PARA)
```

selects the last <PARA> element.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Selecting by Node Type

In addition to selecting a resource by number, you can specify a particular node type. To select a particular node type, the instance is followed by a comma and a node type, which can be one of the following values:

- A name value selects the elements that have a particular name.
For example, child(2,para) selects the second para element that is a child of the current element.
- The keyword #element selects XML elements. This is also the default type if you do not specify a type, so you will probably never need to use this keyword.
- The #pi keyword selects XML processing instructions. Obviously, you can't go on to locate any attributes of a processing instruction, or any elements inside one (both are impossible), but you can locate text inside one (with string).
- The #comment keyword selects XML comments. Obviously, you cannot go on to locate any attributes of a comment, or any elements inside one (both are impossible), but you can locate text inside one (with string).
- The #text keyword selects text regions directly inside an element's character data (CDATA) sections. You cannot go on to specify any attributes, but you can locate specific parts of text inside a text region (with string).
- The #cdata keyword selects text regions inside character data (CDATA) sections. You cannot go on to specify any attributes, but you can locate specific parts of text inside a CDATA section (with string).
- The #all keyword selects all node types.

Selection by Attribute

The element type, if specified, can be qualified by an attribute name by using the attr keyword. You will get the value of the attribute you name.

Selecting Text

All of the absolute and relative location terms described so far select complete elements. The string location term actually allows you to look inside an element and select text located within it. String selectors are extremely useful when linking into non-XML data (such as text files) or into XML data that contains large blocks of text.

The string location term takes the following four arguments:

- A number that indicates which occurrence of the specified string is required. To select all the occurrences you can use the keyword all.
- The character string, enclosed in single or double quotes (for example, “this was not”), to be matched. Instead of a string, you can specify a number to identify a particular character.
- A position number that indicates how many characters to count forward from the start of the matched string to locate the actual string required. You may not specify 0 as a value. If you don’t specify a value, 1 is assumed. A positive number (for example, +2) counts from the left end of the string to the right, a negative value (-4) counts backwards (to the left) from the right end of the string.
- A number that indicates the length of the string (the number of characters in the string) to be selected. If you specify 0, or don’t specify a value, this is assumed to be the position immediately before the character indicated by the position number. You can specify an empty string (a pair of quotes: “”) to specify the point just after the character string you’re matching.

The following are examples of the use of string:

- string(3,“the”,1,2) selects the characters “he” from the third occurrence of the word “the”.
- Assuming that you’ve already selected the word, string(2,“”) indicates the point between the character “t” and “h” in word “the”.
- string(all,“(”,-2,1) selects the last character of every word that appears before an opening bracket.
- string(1,“amazing”,-6,3) selects the characters “maz” from the first occurrence of the word “amazing”.

Selecting Groups and Ranges (spans)

A resource location can contain a single extended pointer or two extended pointers separated by a span keyword. If it has two extended pointers, the location is assumed to be everything from the start of the first extended pointer’s target to the end of the second one.

Spans are a major exception to the rest of the extended pointer language in that span locations are *not* trees. You cannot assume that a span is a meaningful chunk of an XML document. It is very unlikely to be a single element, or even a whole number of elements. This limits what an XML application can do with spans, but it is up to the XML software to find a sensible way of dealing with this.

Summary

This chapter looked at an XML document as an object constructed of various components. You've learned about markup and the distinction that XML makes between markup and character data. You have also seen how XML documents have both a logical and a physical structure and looked at the connection between these and XML's elements and entities.

Q&A

Q What is a grove?
A An XML document has a hierarchical tree-like structure. The view of an XML document that is used for linking is a collection of trees, called a grove. A leaf is called node and the branches are called children.
Q Why do you have to be careful with string **and** spans?
A Unlike the other extended pointers, which select whole elements or groups of elements, string and spans can select parts of elements. This could result in an XML document no longer being valid, or even no longer well-formed.
Q Does the root() **extended pointer serve any real function?**
A Not really, but it can make your linking more readable for a human (remember that one of the design goals of both XML and XLL is that it should be reasonably readable for human beings).
Q Why are root() **and** origin() **always written with brackets after them?**
A The brackets are so-called empty arguments that prevent them from being confused with IDs.

Exercises

1. The following data is taken from a simple XML book catalog:

```
<catalog>
<title>Simon's XML Book Catalog</title>
<book>
<title>Presenting XML</title>
<author>Richard Light</author>
<author>Simon North</author>
<publisher>Sams.Net</publisher>
<date>1997</date>
<ISBN></ISBN>
<cd>no</cd>
</book>
<book>
<title>XML Complete</title>
<author>Steven Holzner</author>
```

```
<publisher>McGraw-Hill</publisher>
<date>1998</date>
<ISBN>0-07-913702-4</ISBN>
<cd>yes</cd>
</book>
<book>
<title>Designing XML Internet Applications</title>
<author>Michael Leventhal</author>
<author>David Lewis</author>
<author>Mathew Fuchs</author>
<publisher>Prentice Hall PTR</publisher>
<date>1998</date>
<ISBN>0-13-616882-1</ISBN>
<cd>yes</cd>
</book>
<book>
<title>Structuring XML Documents</title>
<author>David Megginson</author>
<publisher>Prentice Hall PTR</publisher>
<date>1998</date>
<ISBN>0-13-642299-3</ISBN>
<cd>yes</cd>
</book>
<book>
<title>The XML Companion</title>
<author>Neil Bradley</author>
<publisher>Addison-Wesley</publisher>
<date>1998</date>
<ISBN>0-201-34285-5</ISBN>
<cd>no</cd>
</book>
<book>
<title>XML: Extensible Markup Language</title>
<author>Elliotte Rusty Harold</author>
<publisher>IDG Books</publisher>
<date>1998</date>
<ISBN>0-7645-3199-9</ISBN>
<cd>yes</cd>
</book>
</catalog>
```

What extended pointer could you use to select the title of each book?

2. What extended pointer could you use to select the title of just the third book?

3. What extended pointer could you use to select the complete entry for the second and third books?

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

CHAPTER 12
Viewing XML in Internet Explorer

Today we will discuss:

- Microsoft’s vision for XML
- How to view XML data in Microsoft Internet Explorer 4
- How to view XML data in Microsoft Internet Explorer 5

Microsoft’s Vision for XML

Let’s start with a framework. XML can be used for marking up three different kinds of information:

- Data—Structured sets of information normally handled by relational databases
- Documents—Less structured sets of information with a specific characteristic— that the sequence of the information is important
- Meta-data—Data about other information, for instance who the author is, keywords, the target group, and so on

Previously, Microsoft's idea was that if you wanted to display documents to people, HTML was the markup language to use. They saw XML as the preferred vehicle for exchanging structured data between applications.

Here's an example of what they envisioned: A relational database is queried. The result of this query is that XML-tagged structured data is sent over the network to the client where the data can be queried, altered, displayed, computed, and so on.

Internet Explorer 4 already has those facilities for reading and displaying XML-tagged data.

Microsoft also uses XML for meta-data. Their Channel Definition Format is an XML application describing sites, which is meta-information (information about sites).

But the release of Internet Explorer 5 beta 2 brought us in addition direct XML document support.

Viewing XML in Internet Explorer 4

One can use XML with IE4 in different ways. We'll explore these ways more profoundly.

Overview of XML Support in Internet Explorer 4

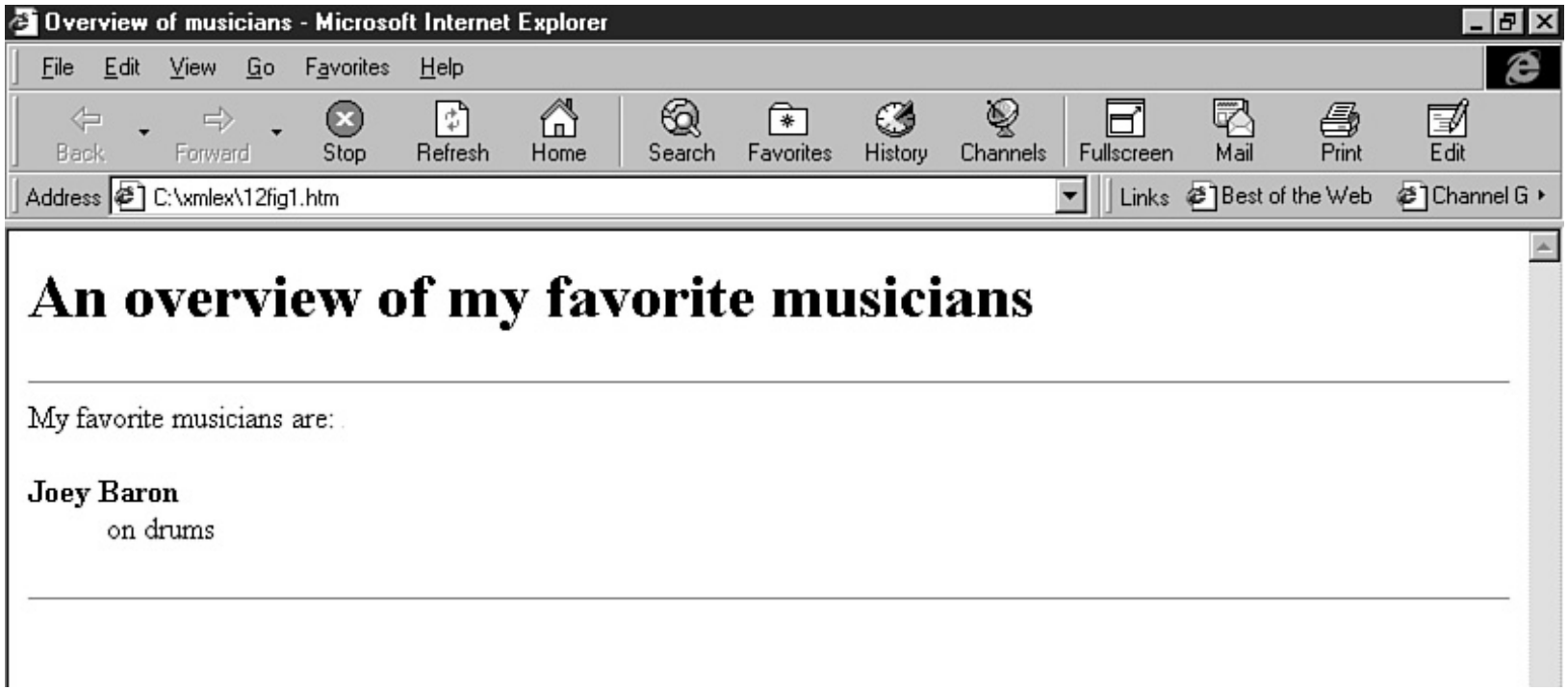
Internet Explorer 4

- Performs CDF processing

CDF stands for Channel Definition Format, which is an XML application

- Has two XML parsers: a non-validating parser in C and a validating parser written in Java

Those parsers weren't covered in Days 5 and 9 because at the time of this writing, they were not fully conformant to the W3C recommendation.

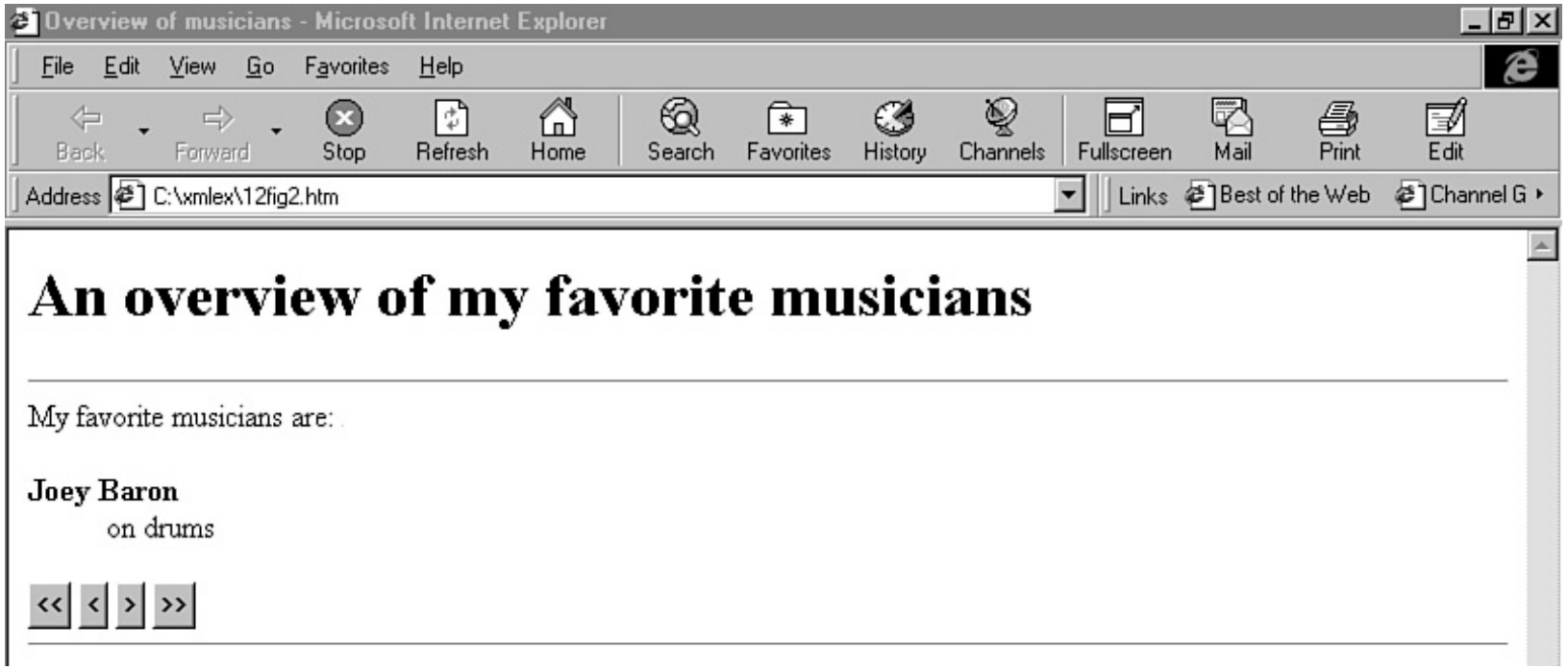


- Offers an XML Data Source object
- Offers an XML Object Model, and an interface (API) to let developers interact with XML data in the browser

It must be stressed that Internet Explorer 4 doesn't have facilities for natively displaying XML.

Viewing XML Using the XML Data Source Object

Data Source objects are used for what Microsoft calls *data binding*. Data binding is Microsoft's way of bringing data manipulation to the browser (client) and away from the server. Normally, if you want a new view on the data, you resubmit a query to the server. The server performs the necessary calculations and sends a new HTML page to the client. This doesn't happen with data binding. The server sends an HTML page together with the data to the client. They are stored locally and can be manipulated locally without reconnecting to the server.



Transferring processing from the server to the client is not a bad idea in a world where there is always a shortage of bandwidth.To implement this data binding you need first to include a data source object in your page. The XML data source object is just one of the data source objects shipped with Internet Explorer 4.0.

- Some of the other data source objects are as follows:

  - The Tabular Data Control (TDC)
  - The Remote Data Service (using OLE-DB or ODBC)
  - The JDBC Applet

More information about data source objects can be found at the following Web site: http://www.microsoft.com/workshop/c-frame.htm#/workshop/author/default.asp.

After the insertion of the data source object, you need to define HTML elements that are able to read data from the data source: they are called “data consumers.”

Let’s make this concrete by using an example. First we need an XML file with structured data. See Listing 12.1.

Listing 12.1 musicians.xml—An XML File with Structured Data

```
1:   <?xml version=”1.0” ?>
2:   <musicians>
3:   <musician>
4:   <name>Joey Baron
5:   </name>
```

```
6:   <instrument>drums
7:   </instrument>
8:   <NrOfRecordings>1
9:   </NrOfRecordings>
10:  </musician>
11:  <musician>
12:  <name>Bill Frisell
13:  </name>
14:  <instrument>guitar
15:  </instrument>
16:  <NrOfRecordings>3
17:  </NrOfRecordings>
18:  </musician>
19:  <musician>
20:  <name>Don Byron
21:  </name>
22:  <instrument>clarinet
23:  </instrument>
24:  <NrOfRecordings>2
25:  </NrOfRecordings>
26:  </musician>
27:  <musician>
28:  <name>Dave Douglas
29:  </name>
30:  <instrument>trumpet
31:  </instrument>
32:  <NrOfRecordings>1
33:  </NrOfRecordings>
34:  </musician>
35:  </musicians>
```

An HTML file is needed as well. See Listing 12.2.

Listing 12.2 HTML File to Add an XML Data Source Object

```
1:   <HTML>
2:   <HEAD>
3:   <TITLE>Overview of musicians</TITLE>
4:   </HEAD>
5:   <BODY>
6:   <H1>An overview of my favorite musicians</H1>
```

```
7:   <HR>
8:   <P>My favorite musicians are:</P>
9:   <!-- Things to come -- >
10:  <HR>
11:  <!-- some other stuff -- >
12:  </BODY>
13:  </HTML>
```

After that the XML Data Source object needs to be added. See Listing 12.3.

Listing 12.3 An XML Data Source Object

```
1:  <APPLET
2:      code=”com.ms.xml.dso.XMLDSO.class”
3:      id=”xmldso”
4:      width=”0”
5:      height=”0”
6:      mayscript=”true”>
7:  <PARAM NAME=”URL” VALUE=”...”>
8:  </APPLET>
```

... stands for the URL that points to an XML file.

Normally, the XML data come from the same place as the HTML file containing the APPLET tag.

Listing 12.4 places the applet in our HTML file.

Listing 12.4 HTML File with an XML Data Source Object Added

```
1:   <HTML>
2:   <HEAD>
3:   <TITLE>Overview of musicians</TITLE>
4:   </HEAD>
5:   <BODY>
6:   <H1>An overview of my favorite musicians</H1>
7:   <HR>
8:   <P> My favorite musicians are: </P>
9:   <APPLET
10:      code="com.ms.xml.dso.XMLDSO.class"
11:      id="xmldso"
```

```
12:      width="0"
13:      height="0"
14:      mayscript="true">
15:  <PARAM NAME="URL" VALUE="musicians.xml">
16:  </APPLET>
17:  <!-- More things to come -- >
18:  <HR>
19:  <!-- some other stuff -- >
20:  </BODY>
21:  </HTML>
```

The next step in our procedure is to include some HTML elements that are capable of rendering the data supplied by our XML data source object.

Not all HTML elements support data binding. The following lists the most important elements that do bind data:

- DIV
- SPAN
- TABLE

To bind information in the external data source to these elements, Microsoft added some attributes to these HTML elements. The most important of these are shown in Table 12.1.

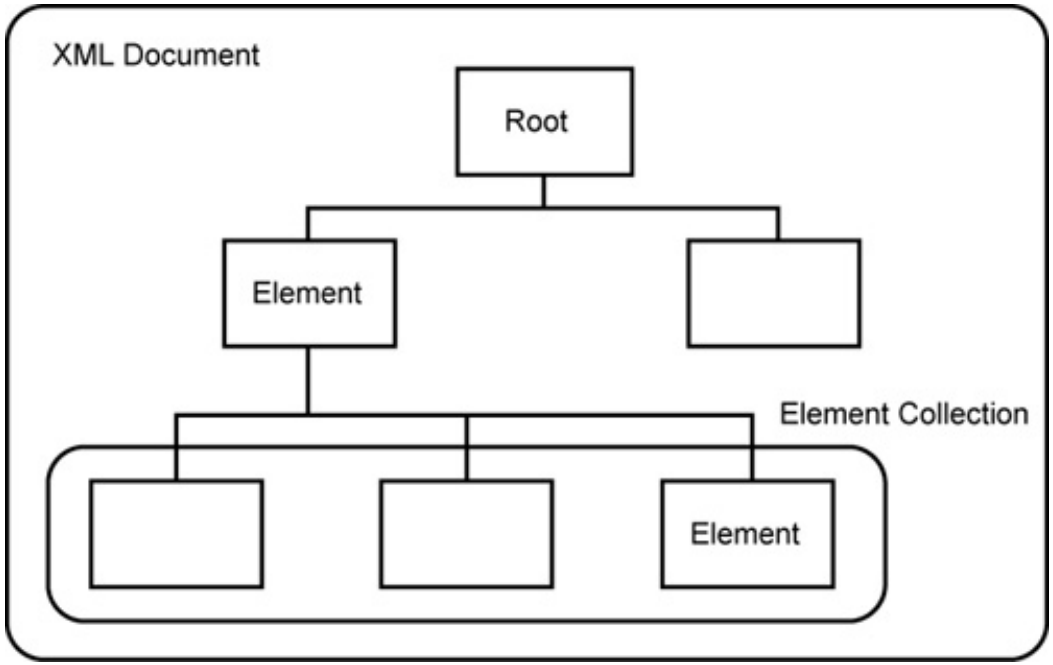
Table 12.1 Attributes for Connecting a Data Source Object to a Data Consumer

Attribute	*Function*
DATASRC	Refers to the name of the data source object
DATAFLD	Specifies the data to which the element is bound

Let’s make our example more complete by adding

```
<P><SPAN DATASRC=”#xmldso” DATAFLD=”name”></SPAN></P>
```

where the value of attribute DATASRC is referring to the ID of our applet, which is xmldso.

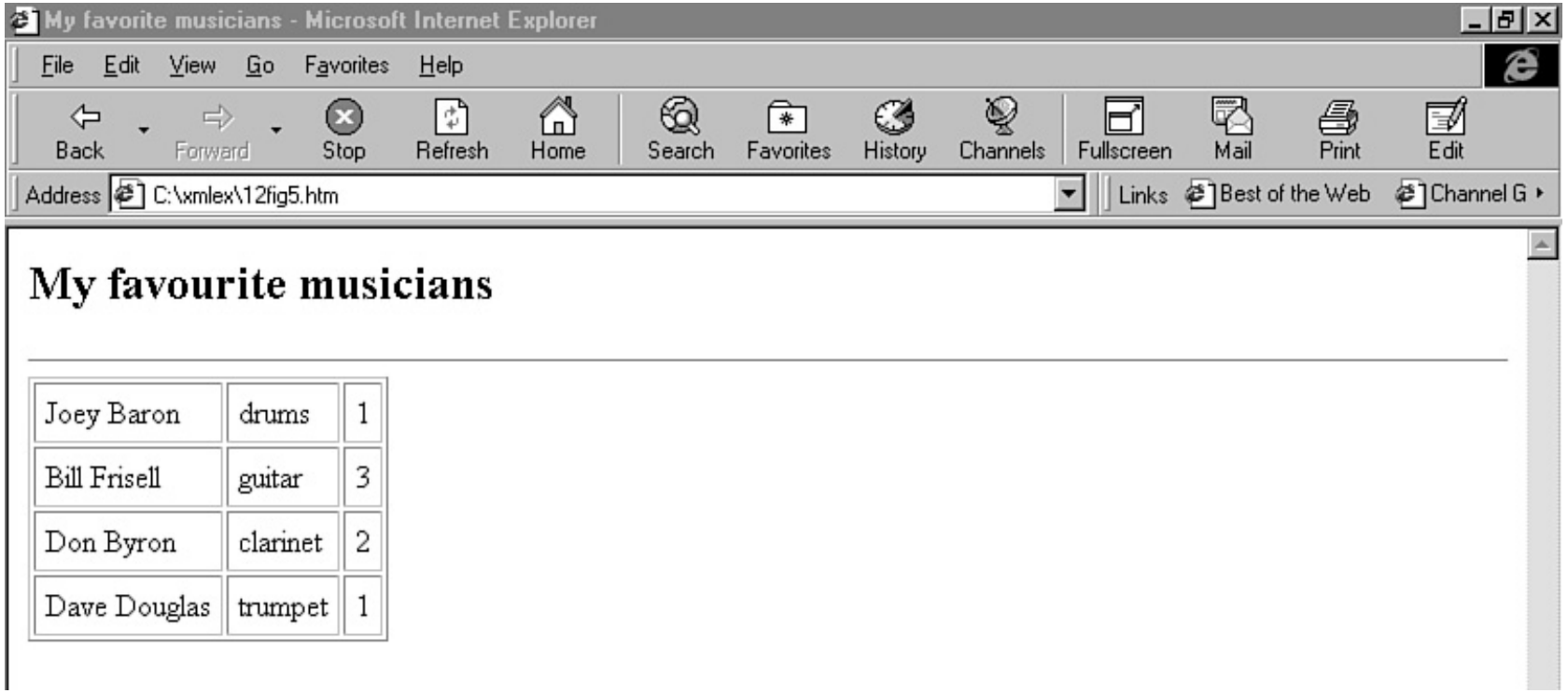


The # is used because it is a reference to a name inside the same document.

The value of the DATAFLD attribute refers to the name of an element used in our XML file.

Listing 12.5 shows how these data consumers are connected to the data source object.

Listing 12.5 Data Source Object Connected to Data Consumers




```
1:   <HTML>
2:   <HEAD>
3:   <TITLE>Overview of musicians</TITLE>
4:   </HEAD>
5:   <BODY>
6:   <H1>An overview of my favorite musicians</H1>
7:   <HR>
8:   <P>My favorite musicians are:</P>
9:   <APPLET
10:      code=”com.ms.xml.dso.XMLDSO.class”
11:      id=”xmldso”
12:      width=”0”
13:      height=”0”
14:      mayscript=”true”>
15:  <PARAM NAME=”URL” VALUE=”musicians.xml”>
16:  </APPLET>
17:  <DL>
18:  <DT><B><SPAN DATASRC=”#xmldso” DATAFLD=”name”></SPAN></B></DT>
19:  <DD>on <SPAN DATASRC=”#xmldso” DATAFLD=”instrument”></SPAN></DD>
20:  </DL>
```

```
21:  <HR>
22:  <!-- some other stuff -- >
23:  </BODY>
24:  </HTML>
```

Figure 12.1 shows this HTML file loaded into Internet Explorer 4.0.

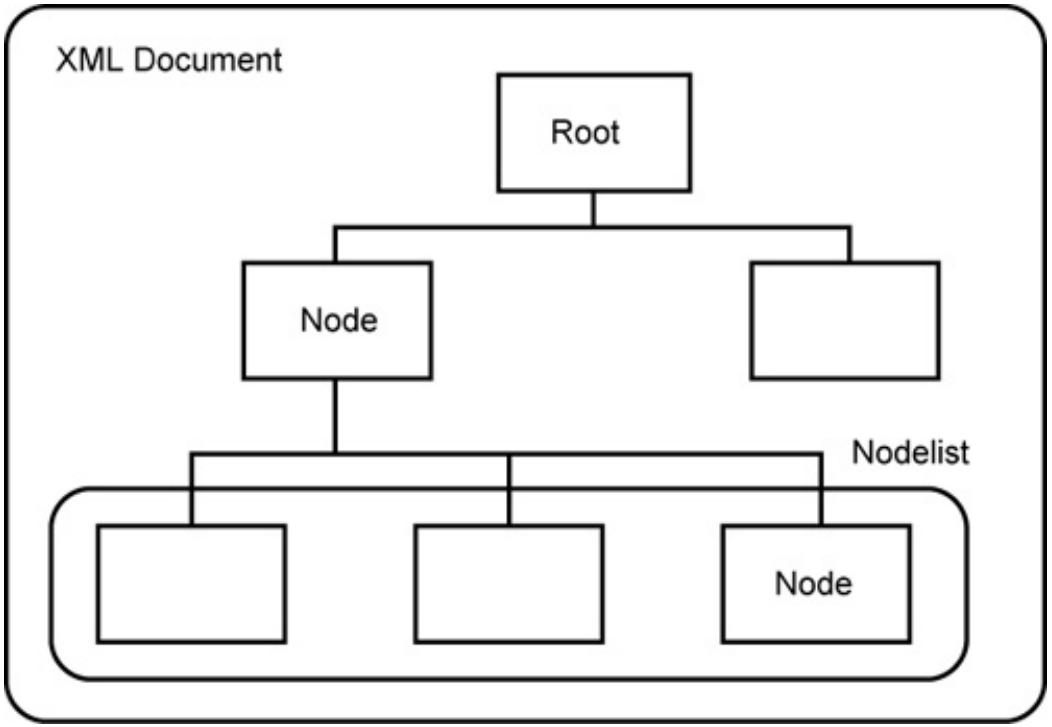
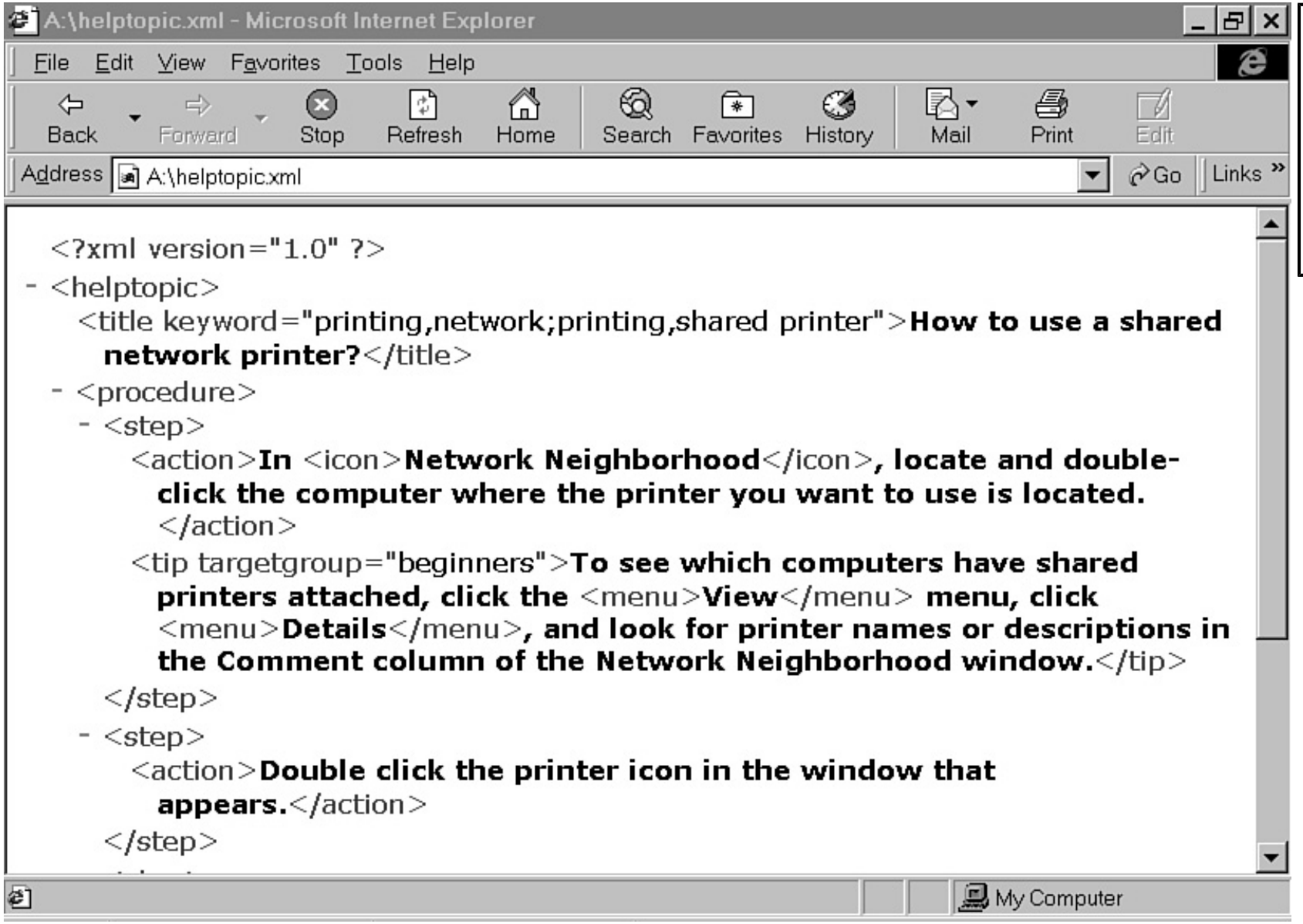


Figure 12.1 *The result of data binding in IE4.*

We now have the principle working. We were able to read information from our XML file, but the result is hardly exciting.

We just read information on the first musician that appears in our XML file.

When a data set (XML file) is loaded, the information is split into different records, only one of which is available. It is named the current record.



After loading of the XML file, the first record is normally identified as the current record.

Of course we want information on all our musicians. We can use two techniques to obtain this information:

- We can add a facility for sequentially loading the data of each musician—in other words, to navigate through our data set.
- We could use an element that can handle more than one value, which is the HTML TABLE element.

Let’s first investigate how we can navigate through the data set.

We’ll use the code in Listing 12.6.

Listing 12.6 Adding Buttons to Navigate Through the Data Set

```
1:  <input type=button value="<<"
2:      onclick='xmldso.recordset.movefirst();'>
3:  <input type=button value="<"
4:      onclick='xmldso.recordset.moveprevious();>
5:  <input type=button value=">"
6:      onclick='xmldso.recordset.movenext();'>
7:  <input type=button value=">>"
8:      onclick='xmldso.recordset.movelast();'>
```

Four buttons (INPUT type="button") have been added with semantics:

- Go to first (<<)
- Go to last (>>)
- Go to previous (<)
- Go to next (>)

The buttons are related to the four available methods to navigate the record set generated by our XML data source object.

But if you open this in your browser you'll see that it doesn't really work as intended.

For the next case some testing needs to be added to check if the end of file (EOF) is reached. And the same goes for previous for the beginning of the file (BOF), as done in Listing 12.7.

Listing 12.7 Checking the Position in the Data Set

```
1:   <input type=button value="<<"
2:       onclick='xmldso.recordset.movefirst();'>
3:   <input type=button value="<"
4:       onclick='xmldso.recordset.moveprevious();
5:           if (xmldso.recordset.BOF)
6:           xmldso.recordset.movefirst();'>
7:   <input type=button value=">"
8:       onclick='xmldso.recordset.movenext();
9:           if (xmldso.recordset.EOF)
```

```
10:          xmldso.recordset.movelast();'>
11:  <input type=button value=">>"
12:      onclick='xmldso.recordset.movelast();'>
```

Figure 12.2 shows our HTML file with navigation buttons loaded into Internet Explorer 4.0.

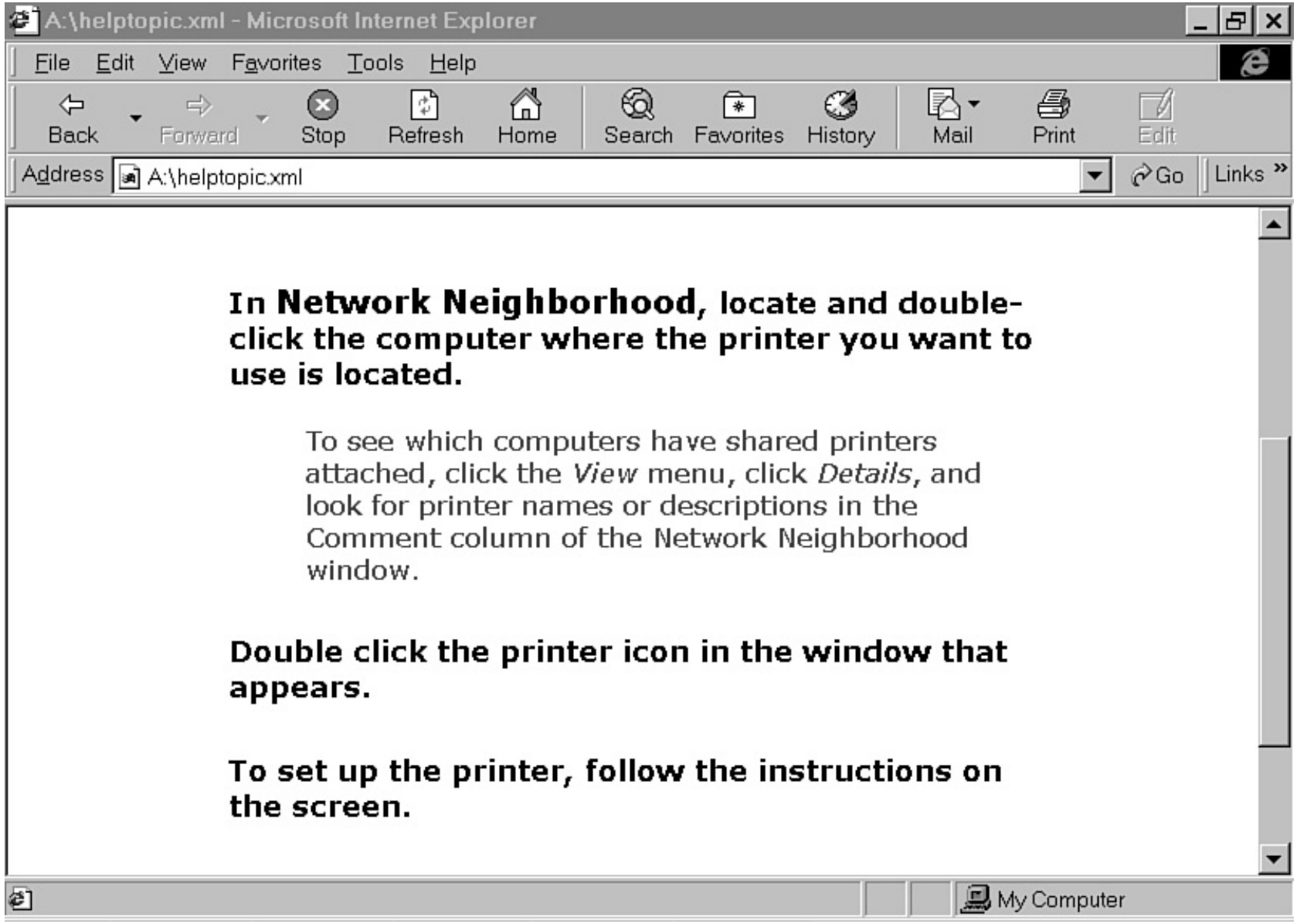


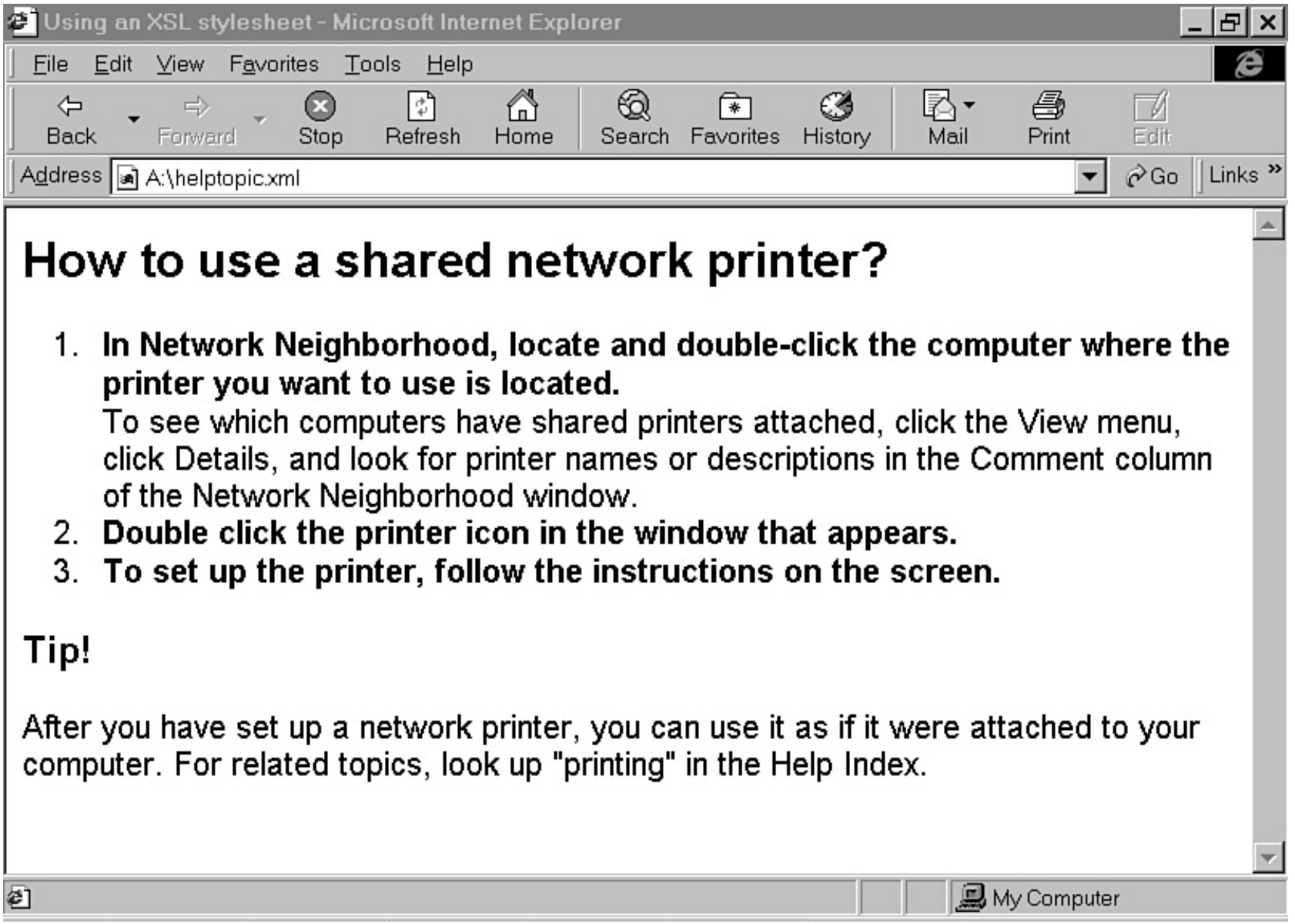
Figure 12.2 *Navigation buttons added.*

We can also use a second solution, the overview in tabular format. Let's bind a TABLE to the data. See Listing 12.8.

Listing 12.8 Using a Table

```
1:  <TABLE DATASRC="#xmldso">
2:  <THEAD>
3:  <TH>Musician</TH>
4:  <TH>Instrument</TH>
5:  </THEAD>
6:  <TR>
7:      <TD><SPAN DATAFLD="name"></SPAN></TD>
8:      <TD><SPAN DATAFLD="instrument"></SPAN></TD>
9:  </TR>
10: </TABLE>
```

The table is set bound to the data source. Inside the table are single-valued elements (SPAN) that refer to the different XML elements. See Listing 12.9.



Listing 12.9 The Complete HTML File

```
1:   <HTML>
2:   <HEAD>
3:   <TITLE>Overview of musicians</TITLE>
```

```
4:   </HEAD>
5:   <BODY>
6:   <H1>An overview of my favorite musicians</H1>
7:   <HR>
8:   <P>My favorite musicians are:</P>
9:   <APPLET
10:      code=”com.ms.xml.dso.XMLDSO.class”
11:      id=”xmldso”
12:      width=”0”
13:      height=”0”
14:      mayscript=”true”>
15:  <PARAM NAME=”URL” VALUE=”musicians.xml”>
16:  </APPLET>
17:  <TABLE BORDER=”2” CELLPADDING=”3” CELLSPACING=”2” width=”40%”
⇒DATASRC=”#xmldso”>
18:  <THEAD>
19:  <TH>Musician</TH>
20:  <TH>Instrument</TH>
21:  </THEAD>
22:  <TR>
23:      <TD><SPAN DATAFLD=”name”></SPAN></TD>
24:      <TD><SPAN DATAFLD=”instrument”></SPAN></TD>
25:  </TR>
26:  </TABLE>
27:  <HR>
28:  <!-- some other stuff -- >
29:  </BODY>
30:  </HTML>
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Figure 12.3 shows our musicians data in tabular format in Internet Explorer 4.0.

Figure 12.3 *Our data in tabular format in IE4.*

You have seen that the idea of bringing data from the server to the client is a very powerful one. A lot of filtering, sorting, and computation can be done on the client side without consuming bandwidth.

> Be careful, however. This data binding feature is part of Microsoft's implementation of Dynamic HTML, which is violently incompatible with Netscape's implementation.
>
> In other words, this mechanism only works with Internet Explorer.

Viewing XML Using the XML Object API

This section shows you how to use JavaScript to access the XML Object Model (API) to transform the XML to HTML.

The object model used in IE4 has been superseded by the Document Object Model (DOM) defined by the W3 organization, which is implemented in IE5, beta 2.

An XML document is modeled as a tree structure. It starts with the top-level element (its root) and branches out to its descendants.

It uses three classes of objects:

- XML Document object
- XML Element object
- XML Element Collection object

Figure 12.4 gives a graphical representation of this model.

Figure 12.4 *The XML Object Model in IE4.*

The Document object represents the XML source document. It holds the element tree and document information such as the character set used, the document type, the file size, and so on. Once this object is created you can access its information by calling its properties and manipulate it by calling its methods.

Examples of available XML Document object properties are shown in Table 12.2.

Table 12.2 XML Document Object Properties

Name	***Returns***
Root	The root element of the document
Charset	A string specifying the character set used by the input document
Version	The version of the XML specification being used
FileSize	The file size of the XML document

The full list of available properties and methods can be found at the following URL: http://www.microsoft.com/xml/articles/xmlmodel.aspA document is modeled as a tree of nodes, where the root element contains other elements (nodes) containing other nodes... until you reach the leafs of the tree, which are empty elements, text, comments, and processing instructions.

In Internet Explorer 4 the XML Object Model distinguishes five types of element objects, of which three are important:

- ELEMENT type—If the node is a container or an empty element
- TEXT type—If the leaf node contains PCDATA or CDATA
- COMMENT typeFor comments

Once an element object is created you can access its information by calling its properties and manipulate it by calling its methods.

The most useful XML Element object properties are shown in Table 12.3.

Table 12.3 XML Element Object Properties

Name	***Returns***
Type	0 for type element, 1 for type text, and so on for the other types
Tag	Name, an uppercase string that is the name of the tag
Text	The text content of an element or comment
Children	An enumeration of the child elements of the specified element

Useful XML Element object methods are shown in Table 12.4.

Table 12.4 XML Element Object Methods

Name	***Function***
setAttribute()	Set an attribute value
removeAttribute()	Remove an attribute
addChild()	Add a child element
removeChild()	Remove a child element

The Element Collection object is used for manipulating the children of an element. Its main use is to iterate over the child elements.

The only XML Element Collection object property is explained in Table 12.5.

Table 12.5 XML Element Collection Object Property

Name	***Returns***

length	The length of a collection (the number of nodes)

The only XML Element Collection object method is explained in Table 12.6.

Table 12.6 XML Element Collection Object Method

Name	***Function***
item(index)	The requested item at the position (number) specified by the index parameter In other words, item(0) returns the first child element
item(elementname)	A collection of all elements with that specific name
item(elementname, index)	The element at the position (number) specified by the index parameter in the collection of the elements with that specific name In other words, item(ITEM, 3) returns the fourth ITEM element

We will be using a lot of those properties and methods in an example shortly.

This object model is language neutral and can be accessed from JavaScript, VBScript, C++, or Java.

Using a sample JavaScript, we will read our musicians.xml file and convert it to HTML by calling the properties and methods defined for the three different objects (Document object, Element object, Element Collection object).

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

The result will be a table for all the musicians. It will have a table row for each individual musician that will include the values for name, instrument, and number of recordings in separate cells.

The file musicians.xml is shown in Listing 12.1. We have our file with structured data. The next thing to do is to create a document object.

```
var xml = new ActiveXObject("msxml");
```

Then you need to specify the XML data file to be loaded by the XML document object

```
xml.URL = "file:///c|/xmlex/musicians.xml";
```

Let's think for a minute what the rest of the program needs to do. Every node that's entered has to be checked to see if:

- It is an element
- It is text

If it is text, we simply want to output it. If it's an element, we need to know if the tag name is:

- musicians—We have to start the table.
- musician—For every musician, we have to start a new row.
- something else—The others (name, instrument, NrOfRecordings) start a table cell.

We implemented this logic into JavaScript as shown in Listing 12.10.

Listing 12.10 JavaScript Function output_doc(elem)

```
1:   function output_doc(elem)
2:   {
3:   if (elem.type == 0) //Check if the node type is an element
```

```
4:           {
5:               if (elem.tagName == “MUSICIANS”) //tagnames are returned
⇒in uppercase
6:               {
7:                   document.write(“<TABLE BORDER=’1’ CELLPADDING=’5’>”);
⇒//begin table
8:                   traverse(elem); //to be defined yet; for traversing
⇒the children
9:                   document.write(“</TABLE>”);
10:                      }
11:              else if (elem.tagName == “MUSICIAN”)
12:              {
13:                  document.write(“<TR>”); //begin row
14:                  traverse(elem); //to be defined yet; for traversing
⇒the children
15:                  document.write(“</TR>”);
16:                      }
17:              else
18:              {
19:                  document.write(“<TD>”); //begin cell
20:                  traverse(elem); //to be defined yet; for traversing
⇒the children
21:                  document.write(“</TD>”);
22:              }
23:
24:            }
25:  else if (elem.type==1) //Check if the node type is text
26:          document.write(elem.text);
27:      else
28:          alert(“Unknown type encountered”);
29:    }
```

The children for every element need to be processed in the same way: the same tests shown in Listing 12.10 have to be carried out—a clear case for recursion (a program/function that calls itself).

In Listing 12.11 we declare a function that iterates over all children elements, calling each time the already defined output_doc() function.

Listing 12.11 JavaScript Function traverse(elem)

```
1:   function traverse(elem)
2:      {var i;
3:       if (elem.children != null)//if there are child elements
4:          {
5:          for (i=0; i < elem.children.length; i++) //repeat over all
⇒children
6:              output_doc(elem.children.item(i)); //do the tests and the
⇒output for every child
7:          }
8:          }
```

Now we want to start to traverse the tree. We need to define the root element to accomplish this:

```
var docroot = xml.root;
```

And for this root element we call our function output_doc.

```
output_doc(docroot);
```

The complete picture can be seen in Listing 12.12.

Listing 12.12 HTML File Using the Two JavaScript Functions

```
<HTML>
<HEAD><TITLE>XML Object Model in Explorer 4.0</TITLE>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JScript" FOR="window" EVENT="onload">
  document.write("<HTML><HEAD><TITLE>My favorite
⇒musicians</TITLE></HEAD>\n");
  document.write("<BODY><H2>My favorite musicians</H2><HR>\n")    ;
  var xml = new ActiveXObject("msxml");
  xml.URL = "file:///c|/xmlex/musicians.xml";
  var docroot = xml.root;
  output_doc(docroot);

  function traverse(elem)
    {var i;
        if (elem.children != null)
            {
            for (i=0; i < elem.children.length; i++)
            output_doc(elem.children.item(i));
            }
     }

  function output_doc(elem)
  {
        if (elem.type == 0)
            {
                if (elem.tagName == "MUSICIANS")
                    {
                        document.write("<TABLE BORDER='1'
⇒CELLPADDING='5'>");
                        traverse(elem);
                        document.write("</TABLE>");
                    }
                else if (elem.tagName == "MUSICIAN")
                    {
                        document.write("<TR>");
                        traverse(elem);
                        document.write("</TR>");
                    }
                else
                    {
                        document.write("<TD>");
                        traverse(elem);
                        document.write("</TD>");
                    }

            }
          else if (elem.type==1)
                document.write(elem.text);
          else
                alert("Unknown type encountered");
```

```
50:              }
51:  </SCRIPT>
52:  </BODY>
53:  </HTML>
```

The result of loading this page in Internet Explorer 4 is shown in Figure 12.5.

Figure 12.5 *The result of our JavaScript in IE4.*

You can see that using this approach is a lot of hassle to simply make XML viewable by Internet Explorer 4.0. However, if you really need to do computing based on XML data, or need to execute transformations of XML data, this API is simple and powerful.

Viewing XML via MS XSL Processor

XSL stands for *Extensible Style sheet Language*. The first XSL proposal was jointly made by Microsoft, Inso Corporation, and ArborText. Microsoft has also made available an XSL processor for prototyping and testing. This processor allows XML data to be transformed into HTML.

> Please note that a conversion to HTML takes place. It is not the XML that is rendered directly.

The MS XSL Processor was available in two packages:

- The Microsoft XSL Command-line utility
- The Microsoft XSL ActiveX control

> The XSL syntax to be used with the MS XSL Processor is completely outdated and superseded by a new XSL draft.
>
> The result of this is that the XSL Command-line utility has been withdrawn from Microsoft's Web site.

Let us explain the principle of working with an XSL style sheet.

On one hand we need an XML file, such as the musicians.xml file shown in Listing 12.1.

On the other hand we need an XSL style sheet. Let's name it musicians.xsl. The content is shown in Listing 12.13.

> Don't spend too much time trying to understand the syntax because the XSL development work by the W3C is still ongoing and the syntax has gone through substantial changes already.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Listing 12.13 musicians.xsl

```
1:   <xsl>
2:   <rule>
3:     <root/>
4:     <HTML>
5:       <HEAD>
6:           <TITLE>My favorite musicians</TITLE>
7:       </HEAD>
8:       <BODY>
9:           <H1>My favorite musicians</H1><HR/>
10:          <TABLE BORDER=”2” CELLPADDING=”5”>
11:          <children/>
12:          </TABLE>
13:      </BODY>
14:    </HTML>
15:  </rule>
16:  <rule>
17:      <target-element type=”musician”/>
18:      <TR>
19:      <children/>
20:      </TR>
21:  </rule>
22:  <rule>
23:      <target-element type=”name”/>
24:      <TD>
25:      <children/>
26:      </TD>
27:  </rule>
28:  <rule>
29:      <target-element type=”instrument”/>
30:      <TD>
31:      <children/>
```

```
32:      </TD>
33:  </rule>
34:  <rule>
35:      <target-element type=”NrOfRecordings”/>
36:      <TD>
37:      <children/>
38:      </TD>
39:  </rule>
40:  </xsl>
```

To use the XSL ActiveX control, place the following OBJECT element as shown in Listing 12.14 in your HTML page.

Listing 12.14 OBJECT Element Referring to the XSL Control

```
1:  <OBJECT ID=”XSLControl”
2:  CLASSID=”CLSID:2BD0D2F2-52EC-11D1-8C69-0E16BC000000”
3:  codebase=”http://www.microsoft.com/xml/xsl/msxsl.cab”
4:  style=”display:none”>
5:  </OBJECT>
```

This object automatically downloads and installs the XSL ActiveX control.

Next, point the control to the XML document and XSL style sheet as shown in Listing 12.15.

Listing 12.15 OBJECT Element Referring to the XML Document and XSL Style Sheet

---

```
1:  <OBJECT ID=”XSLControl”
2:  CLASSID=”CLSID:2BD0D2F2-52EC-11D1-8C69-0E16BC000000”
3:  codebase=”http://www.microsoft.com/xml/xsl/msxsl.cab”
4:  style=”display:none”>
5:    <PARAM NAME=”documentURL” VALUE=”musicians.xml”>
6:    <PARAM NAME=”styleURL” VALUE=”musicians.xsl”>
7:  </OBJECT>
```

This object generates an HTML string from the XML document and the accompanying style sheet.

This HTML string has to be put somewhere in our page. See Listing 12.16.

Listing 12.16 Script for Loading the Generated HTML

```
1:  <SCRIPT FOR=window EVENT=onload>
2:    var xslHTML = XSLControl.htmlText;
3:    document.all.item(“xslTarget”).innerHTML = xslHTML;
4:  </SCRIPT>
```

This script first captures the HTML in a variable and places it in the element with ID “xslTarget”.

So we need to add an element with this ID on our page.

```
<DIV id=xslTarget></DIV>
```

The complete picture is shown in Listing 12.17.

Listing 12.17 Loading Generated HTML

```
1:  <HTML>
2:    <HEAD>
3:      <TITLE>My favorite musicians</TITLE>
4:        <SCRIPT FOR=window EVENT=onload>
5:        var xslHTML = XSLControl.htmlText;
6:        document.all.item("xslTarget").innerHTML = xslHTML;
7:      </SCRIPT>
8:    </HEAD>
9:    <BODY>
10:          <OBJECT ID="XSLControl"
11:                  CLASSID="CLSID:2BD0D2F2-52EC-11D1-8C69-0E16BC000000"
12:                  CODEBASE="http://www.microsoft.com/xml/xsl/msxsl.cab"
13:                  STYLE="display:none">
14:             <PARAM NAME="documentURL" VALUE="musicians.xml">
15:               <PARAM NAME="styleURL" VALUE="musicians.xsl">
16:       </OBJECT>
17:         <DIV id="xslTarget"></DIV>
18:   </BODY>
19:   </HTML>
```

Viewing XML in Internet Explorer 5

You can use XML with IE5 in different ways. We'll explore these ways more profoundly now.

> The information here is based on Internet Explorer Developer Release beta 2.

Overview of XML Support in Internet Explorer 5

In addition to the already existing XML support in Internet Explorer 4, expect to see the following in Release 5:

- Full conformance to the XML 1.0 specification
- Direct viewing of XML by using XSL or CSS (Cascading Style Sheets)
- A mechanism for embedding XML in HTML called "data islands"
- Namespace support
- Better performance
- Improved robustness

Viewing XML Using the XML Data Source Object

Internet Explorer 5 beta 2 ships with a new C++ Data Source object, giving you better performance and the ability to bind directly to an XML data island (see below) without the need to use an APPLET or OBJECT tag, as shown in Listing 12.18.

Listing 12.18 Using the XML Data Source Object in Internet Explorer 5 Beta 2

```
1:   <HTML>
2:   <HEAD>
3:   <TITLE>Overview of musicians</TITLE>
4:   </HEAD>
5:   <BODY>
6:   <H1>An overview of my favorite musicians</H1>
7:   <HR>
8:   <P>My favorite musicians are:</P>
9:   <XML ID=”xmldso” src=”musicians.xml”></XML>
10:  <TABLE BORDER=”2” CELLPADDING=”3” CELLSPACING=”2” width=”40%”
⇒DATASRC=”#xmldso”>
11:  <THEAD>
12:  <TH>Musician</TH>
13:  <TH>Instrument</TH>
14:  </THEAD>
15:  <TR>
16:      <TD><SPAN DATAFLD=”name”></SPAN></TD>
17:      <TD><SPAN DATAFLD=”instrument”></SPAN></TD>
18:  </TR>
19:  </TABLE>
20:  <HR>
21:  <!-- some other stuff -- >
22:  </BODY>
23:  </HTML>
```

Viewing XML Using the XML Object API

A lot of thinking is going on in the W3C to define a standardized Document Object Model.

See http://www.w3.org/DOM/.

Microsoft tries to keep in sync with these new developments. The result is that the Document Object Model used in Internet Explorer 5 has changed and been extended.

Microsoft’s object model in IE5 uses four base classes of objects:

- Document object
- Node object
- NodeList object
- NamedNodeMap object

Figure 12.6 gives a graphical representation of this model.

Figure 12.6 *The XML Object Model in Internet Explorer 5.*

Note the difference in names used with the previous version: Node instead of Element, and NodeList instead of Element Collection

Each of these objects contains a number of properties and methods, enabling you to access information about those objects and manipulate them.

The list of properties and methods has been extended compared to the previous version.

The Document object represents the top node of the tree.

Once this object is created you can access its information and manipulate it by calling its properties and

methods.

An example of a Document object property is shown in Table 12.7.

Table 12.7 IDOMDocument Object Properties

Name	***Returns***
DocumentElement	The root element of the XML file

Previous | Table of Contents | Next

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

An example of a Document object method is shown in Table 12.8.

Table 12.8 IDOMDocument Object Method

Name	*Function*
createElement(tagName)	Returns an Element with name tagName

The full description of all available properties and methods can be found at: http://www.microsoft.com/workshop/c-frame.htm?913723996859#/xml/default.asp.

The Node object represents a node in an XML document.

Internet Explorer 5's XML Object Model distinguishes 12 types of nodes:

- ELEMENT type
- ATTRIBUTE type
- TEXT type
- CDATA_SECTION type
- ENTITY_REFERENCE type
- ENTITY type
- PROCESSING_INSTRUCTION type
- COMMENT type
- DOCUMENT type
- DOCUMENT_TYPE type
- DOCUMENT_FRAGMENT type
- NOTATION type

Once this Node object is created you can access its information and manipulate it by calling its properties and methods.

The most useful Node object properties are shown in Table 12.9.

Table 12.9 IDOMNode Object Properties

Name	*Function*
NodeType	Returns 0 for type element, 1 for type attribute, and so on for the other types.
NodeName	Returns or sets the name of a node.
NodeValue	Returns or sets the value (text) of a node.
ChildNodes	Returns an enumeration of the child nodes of the specified node.

Some Node object methods are shown in Table 12.10.

Table 12.10 IDOMNode Object Methods

Name	*Function*
insertBefore(newChild, refChild)	To insert a child node defined by parameter newChild to the left of the specified node refChild or at the end of the list
removeChild(oldChild)	To remove the specified childnode oldChild

The NodeList object is used for manipulating the children of a node, and its main use is to iterate over the child nodes.

The only NodeList object property is shown in Table 12.11.

Table 12.11 IDOMNodeList Object Property

Name	*Returns*
length	The length of a nodelist (the number of nodes)

You'll find an example of a NodeList object method in Table 12.12.

Table 12.12 IDOMNodeList Object Methods

Name	*Function*
item()	Returns the requested item from the enumerated nodes

We will be using a lot of those properties and methods in an example shortly.

This object model is language neutral and can be accessed equally well from JavaScript, VBScript, C++, or

Java.

Due to the changes in the object model between versions 4.0 and 5.0, we need to rewrite the program listed in Listing 12.12. The new version is shown in Listing 12.19.

Listing 12.19 Using the Document Object Model in Internet Explorer 5 Beta 2

```
1:   <HTML>
2:   <HEAD><TITLE>XML Object Model in Explorer 5.0</TITLE>
3:   </HEAD>
4:   <BODY>
5:   <SCRIPT LANGUAGE=”JScript” FOR=”window” EVENT=”onload”>
6:     document.write(“<HTML><HEAD><TITLE>My favorite
⇒musicians</TITLE></HEAD>\n”);
7:     document.write(“<BODY><H2>My favorite musicians</H2><HR>\n”)    ;
8:     var xml = new ActiveXObject(“microsoft.xmldom”);
9:     xml.load(musicians.xml);
10:    var docroot = xml.documentElement;
11:    output_doc(docroot);
12:
13:    function traverse(node)
14:      {var i;
15:          if (node.childNodes != null)
16:              {
17:              for (i=0; i < node.childNodes.length; i++)
18:              output_doc(node.childNodes.item(i));
19:              }
20:       }
21:
22:    function output_doc(node)
23:    {
24:          if (node.nodeType == 0)
25:              {
26:                  if (node.nodeName == “musicians”)
27:                      {
28:                          document.write(“<TABLE BORDER=’1’
⇒CELLPADDING=’5’>”);
29:                          traverse(node);
30:                          document.write(“</TABLE>”);
31:                      }
32:                  else if (node.nodeName == “musician”)
33:                      {
34:                          document.write(“<TR>”);
35:                          traverse(node);
36:                          document.write(“</TR>”);
37:                      }
38:                  else
39:                      {
40:                          document.write(“<TD>”);
41:                          traverse(node);
42:                          document.write(“</TD>”);
43:                      }
44:
45:              }
46:           else if (node.nodeType==1)
47:                 document.write(node.nodeValue);
48:           else
49:                 alert(“Unknown type encountered”);
```

```
50:              }
51:  </SCRIPT>
52:  </BODY>
53:  </HTML>
```

Let’s have a look at the differences in code between the new (for IE5) and the older version (for IE4):

- The ActiveX object that generates the XML document object has the progID of microsoft.xmldom
- To specify the XML data file to be loaded by the XML document object we need to call the load method and pass it the URL of the XML file to be loaded
- To identify the root element we use the documentElement property
- In the script itself we use the properties and methods of the node and the nodelist objects
- The tag names tested for musicians and musician are now in lowercase

In the implementation of version 4 the tagName property of the Element object returned the name in uppercase. This isn’t the case for the nodeName property of a Node object (version 5).

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Viewing Embedded XML

With the newest release of Internet Explorer it will be possible to embed islands of data (called *data islands*) inside HTML pages. These data islands can be marked up using XML.

You'll do this by using the XML tag inside your HTML page. For getting in the XML data itself, you have two possibilities:

- Include your XML data inside the XML element.
- Use a URL to refer to the XML data.

Listing 12.20 shows an example of the XML data included inside the XML element.

Listing 12.20 Embedding XML in an HTML Page

```
1:   <HTML>
2:   <HEAD><TITLE>XML Object Model in Explorer 5.0</TITLE>
3:   </HEAD>
4:   <BODY>
5:   <H2>My favorite musicians</H2>
6:   <HR>
7:   <XML id="myfavs">
8:   <musicians>
9:   <musician>
10:  <name>Joey Baron
11:  </name>
12:  <instrument>drums
```

```
13:  </instrument>
14:  <NrOfRecordings>1
15:  </NrOfRecordings>
16:  </musician>
17:  <musician>
18:  <name>Bill Frisell
19:  </name>
20:  <instrument>guitar
21:  </instrument>
22:  <NrOfRecordings>3
23:  </NrOfRecordings>
24:  </musician>
25:  <musician>
26:  <name>Don Byron
27:  </name>
28:  <instrument>clarinet
29:  </instrument>
30:  <NrOfRecordings>2
31:  </NrOfRecordings>
32:  </musician>
33:  <musician>
34:  <name>Dave Douglas
35:  </name>
36:  <instrument>trumpet
37:  </instrument>
38:  <NrOfRecordings>1
39:  </NrOfRecordings>
40:  </musician>
41:  </musicians>
42:  </XML>
43:  </BODY>
44:  </HTML>
```

Listing 12.21 shows an example of how to refer an XML file in an HTML page.

Listing 12.21 Referring an XML File in an HTML Page

```
1:   <HTML>
2:   <HEAD><TITLE>XML Object Model in Explorer 5.0</TITLE>
3:   </HEAD>
4:   <BODY>
5:   <H2>My favorite musicians</H2>
6:   <HR>
7:   <XML ID=“myfavs” SRC=“file:///C|xmlex/musicians.xml”>
8:   </XML>
9:   </BODY>
10:  </HTML>
```

If we open those files in our browser, nothing appears.

Once again, you need to use the Document Object API to access the XML data. First, though, you have to work with the HTML Document Object Model because the XML tag is in fact an HTML tag. You can easily refer to the XML element by using its id, which is "myfavs".

Having found the XML tag, we need to find the root element of our XML data tree.

```
myfavs.documentElement
```

And from here on we can use the code we already know, as used in Listing 12.22.

Listing 12.22 Viewing 'Referred to' XML in a HTML Page

```
1:   <HTML>
2:   <HEAD><TITLE>XML Object Model in Explorer 5.0</TITLE>
3:   </HEAD>
4:   <SCRIPT>
5:    function traverse(node)
6:      {var i;
7:          if (node.childNodes != null)
8:              {
9:              for (i=0; i < node.childNodes.length; i++)
10:              output_doc(node.childNodes.item(i));
11:              }
12:       }
13:
14:    function output_doc(node)
15:    {
16:          if (node.nodeType == 0)
17:              {
18:                  if (node.nodeName == "musicians")
19:                      {
20:                          document.write("<TABLE BORDER='1'
⇒CELLPADDING='5'>");
21:                          traverse(node);
22:                          document.write("</TABLE>");
23:                      }
24:                  else if (node.nodeName == "musician")
25:                      {
26:                          document.write("<TR>");
27:                          traverse(node);
28:                          document.write("</TR>");
29:                      }
30:                  else
31:                      {
32:                          document.write("<TD>");
33:                          traverse(node);
34:                          document.write("</TD>");
35:                      }
36:
37:              }
38:           else if (node.nodeType==1)
39:                  document.write(node.nodeValue);
40:           else
41:                  alert("Unknown type encountered");
```

```
42:              }
43:
44:  </SCRIPT>
45:  <BODY>
46:  <H2>My favorite musicians</H2>
47: <XML ID="myfavs" src="musicians.xml"></xml>
48:  <SCRIPT>
49:  var root = myfavs.documentElement;
50:  output_doc(root);
51:  </SCRIPT>
52:  <HR>
53:  </BODY>
54:  </HTML>
```

The following two lines are equivalent:

```
var root = myfavs.documentElement;

var root = document.all("myfavs").XMLDocument;
```

Viewing XML Directly

It is possible to open any XML file directly in IE5. A default XSL style sheet will be applied in this case.

For example, see the file shown in Listing 12.23.

Listing 12.23 helptopic.xml

```
1:  <?xml version="1.0" ?>
2:  <helptopic>
3:   <title keyword="printing,network;printing,shared printer">
4:   How to use a shared network printer?</title>
5:  <procedure>
6:   <step><action>In <icon>Network Neighborhood</icon>,
7:    locate and double-click the computer where the printer
8:   you want to use is located. </action>
9:   <tip targetgroup="beginners">To see which computers have
10:  shared printers attached, click the <menu>View</menu> menu,
11:  click <menu>Details</menu>, and look for printer names or
12:  descriptions in the Comment column of the Network Neighborhood
⇒window.</tip>
13:  </step>
14:  <step>
15:  <action>Double click the printer icon in the window that
⇒appears.</action>
16:  </step>
17:  <step>
18:  <action>
19:  To set up the printer, follow the instructions on the screen.
20:  </action></step>
21:  </procedure>
```

```
22:  <tip>
23:   After you have set up a network printer, you can use it as if
24:   it were attached to your computer. For related topics,
25:   look up &amp;quot;printing&amp;quot; in the Help Index.
26:  </tip>
27:  </helptopic>
```

Figure 12.7 shows you the result of loading this XML file without a specific style sheet attached.

Figure 12.7 *Viewing an XML file directly in IE5.*

Of course if you want to present your XML file in a more exciting way, you need to supply a style sheet to render the XML data. This can be either an XSL or CSS style sheet.

Viewing XML with CSS

You specify the style sheet to be used by inserting a processing instruction of the following form:

```
<?xml:stylesheet type="text/css" href="helptopic.css" ?>
```

> The notation to be used is described in the W3C note, http://www.w3.org/TR/NOTE-xml-stylesheet.

Let us use the XML file from Listing 12.22. The first lines are as in Listing 12.23.

Listing 12.23 helptopic.xml

```
1:   <?xml version="1.0" ?>
2:   <?xml:stylesheet type="text/css" href="helptopic.css" ?>
3:   <helptopic>
4:   <title keyword="printing,network;printing,shared printer">
5:....
```

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

The content of our heltopic.css file could be as in Listing 12.24.

Listing 12.24 helptopic.css

```
1:   helptopic {    display: block;
2:       margin-top:3cm;
3:       margin-left:2cm;
4:       margin-right:2cm;
5:       margin-bottom:6cm;
6:       font-family:Verdana, Arial;
7:       font-size:11pt;
8:       padding:20pt; }
9:
10:  title {display: block;
11:      font-size:20pt;
12:      color:blue;
13:      font-weight:bold;
14:      text-align:center;
15:      margin-bottom:30pt;
16:      text-decoration:underline;}
17:
18:  procedure {display:block;
19:      margin-bottom:30pt}
20:
21:   step {display:block;
22:      margin-bottom:18pt}
23:
24:  action {display:block;
25:      font-weight:bold;}
26:
27:  tip {display:block;
```

```
28:      font-size:10pt;
29:      margin-left:+1cm;
30:      margin-top:12pt;
31:      color:blue;}
32:
33:  icon {display:inline;
34:      font-size:12pt;}
35:
36:  todo {display:inline;
37:      color:red;}
38:
39:  menu {display:inline;
40:      font-style:italic;}
```

Figure 12.8 shows you the result of loading this XML file with a CSS style sheet attached.

Figure 12.8 *Viewing an XML file with a CSS style sheet in IE5.*

Viewing XML with XSL

You relate an XML document with an XSL style sheet by inserting a processing instruction of the following form:

```
<?xml:stylesheet type="text/xsl" href="helptopic.xsl" ?>
```

The first lines are as shown in Listing 12.25.

Listing 12.25 helptopic.xml

```
1:   <?xml version="1.0" ?>
2:   <?xml:stylesheet type="text/css" href="helptopic.css" ?>
3:   <helptopic>
4:   <title keyword="printing,network;printing,shared printer">
5:....
```

A sample heltopic.xsl file is shown in Listing 12.26.

> Please note that the XSL syntax for use with IE5 isn't completely in sync with the latest draft of the W3C, which is itself still a moving target.

Listing 12.26 helptopic.xsl

```
1:  <?xml version="1.0"?>
2:  <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
3:
4:  <!-- default behaviour, thanks to Ken Holman -->
```

```
5:
6:  <xsl:template><xsl:apply-templates/></xsl:template>
7:  <xsl:template match="textnode()"><xsl:value-of/></xsl:template>
8:
9:  <!-- specific behaviour -->
10:
11:  <xsl:template match="/">
12:    <html>
13:      <head>
14:        <title>Using an XSL stylesheet </title>
15:      </head>
16:      <body bgcolor="#FFFFFF">
17:        <xsl:apply-templates/>
18:       </body>
19:    </html>
20:  </xsl:template>
21:
22:  <xsl:template match="title">
23:   <H2>
24:     <xsl:apply-templates/>
25:   </H2>
26:  </xsl:template>
27:
28:  <xsl:template match="procedure">
29:    <OL>
30:      <xsl:apply-templates/>
31:    </OL>
32:  </xsl:template>
33:
34:  <xsl:template match="step">
35:   <LI>
36:     <xsl:apply-templates/>
37:   </LI>
38:  </xsl:template>
39:
40:  <xsl:template match="action">
41:  <B>
42:    <xsl:apply-templates/>
43:  </B><BR/>
44:  </xsl:template>
45:
46:  <xsl:template match="helptopic/tip">
47:   <H3>Tip!</H3>
48:    <xsl:apply-templates/>
49:  </xsl:template>
50:
51: </xsl:stylesheet>
```

Figure 12.9 shows you the result of loading this XML file with a XSL style sheet attached.

Figure 12.9 *Viewing an XML file with a XSL style sheet in IE5.*

Summary

In this chapter we have seen how XML data can be used with:

- Internet Explorer 4
- Internet Explorer 5, beta 2

For each version we explored the different possibilities:

- The XML Data Source object
- The DOM API
- CSS style sheets
- XSL style sheets

Q&A

Q Should I use the DOM API, CSS, or XSL?
A From a standards view, both DOM and CSS are stable standards. This cannot be said of XSL, which is still on the drawing board. From an implementation point of view, the support of CSS in IE5, as it stands now, is far from perfect.
Q I receive white pages in IE5 when opening XML files.
A Chances are great that corruption was introduced during installation. Uninstall completely and then reinstall.

Exercises

Bring the following file into Internet Explorer 4.0 using the XML Data Source object. The data is restricted and should appear in the following order:

- Author
- Title
- ISBN number

Then, using the same XML file, bring it into Internet Explorer using the XML Object Model API to display only a list of authors.

```
1:   <?xml version=“1.0”?>
2:   <books>
3:       <book>
4:           <title>Sam’s Teach Yourself C++ in 21 Days, Second Edition
5:           </title>
6:           <author>Jesse Liberty
7:           </author>
8:           <description>This book teaches you the basics of object-
⇒oriented programming with C++ and is completely revised to
⇒ANSI standards. It can be used with any C++ compiler.
9:           </description>
10:          <ISBN>0-672-31070-8
11:          </ISBN>
12:          <pages>700
13:          </pages>
14:          <targetgroup>Beginning - Intermediate
15:          </targetgroup>
16:          <price unit=“USA”>29.99
17:          </price>
18:      </book>
19:         <book>
20:          <title>Maximum Java 1.1
21:          </title>
22:          <author>Glenn Vanderburg et al.
23:          </author>
```

```
24:          <description>Written by JAVA experts, this book explores
⇒the JAVA 1.1 language, tools, and core JAVA API without
⇒reviewing fundamentals or basic techniques.
25:          </description>
26:          <ISBN>1-57521-290-0
27:          </ISBN>
28:          <pages>900
29:          </pages>
30:          <targetgroup>Expert
31:          </targetgroup>
32:          <price unit="USA">49.99
33:          </price>
34:      </book>
35:      <book>
36:          <title>JavaScript Unleashed, Second Edition
37:          </title>
38:          <author>Richard Wagner et al.
39:          </author>
40:         <description>This book helps you thoroughly understand
⇒and apply JavaScript.
41:          </description>
42:          <ISBN>1-57521-306-0
43:          </ISBN>
44:          <pages>1000
45:          </pages>
46:          <targetgroup>Casual - Experienced
47:          </targetgroup>
48:          <price>49.99
49:          </price>
50:      </book>
51:      <book>
52:          <title>Sam's Teach Yourself Visual C++ 5 in 21 Days, Fourth
⇒Edition
53:          </title>
54:          <author>Nathan and Ori Gurewich
55:          </author>
56:          <description>This book merges the power of the best-selling
⇒"Teach Yourself" series with the knowledge of Nathan and
⇒Ori Gurewich, renowned experts in code, creating the most
⇒efficient way to learn Visual C++.
57:          </description>
58:          <ISBN>0-672-31014-7
59:          </ISBN>
60:          <pages>832
61:          </pages>
62:          <targetgroup>New - Casual
63:          </targetgroup>
64:          <price>35.00
65:          </price>
66:      </book>
67:  </books>
```

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 13
Viewing XML in Other Browsers

On Day 12 we covered how XML can be used with Microsoft Internet Explorer version 4 and 5.

Today we will discuss how you can view XML in other browsers such as:

- Netscape Navigator/Mozilla/Gecko
- DocZilla
- Browsers based on the Inso/Synex Viewport engine

Viewing/Browsing XML in Netscape Navigator/Mozilla/Gecko

In this part we will discuss how XML is handled by Navigator/Mozilla/Gecko.

Netscape's Vision for XML

Netscape sees use of XML for three types of information: documents, data, and metadata.

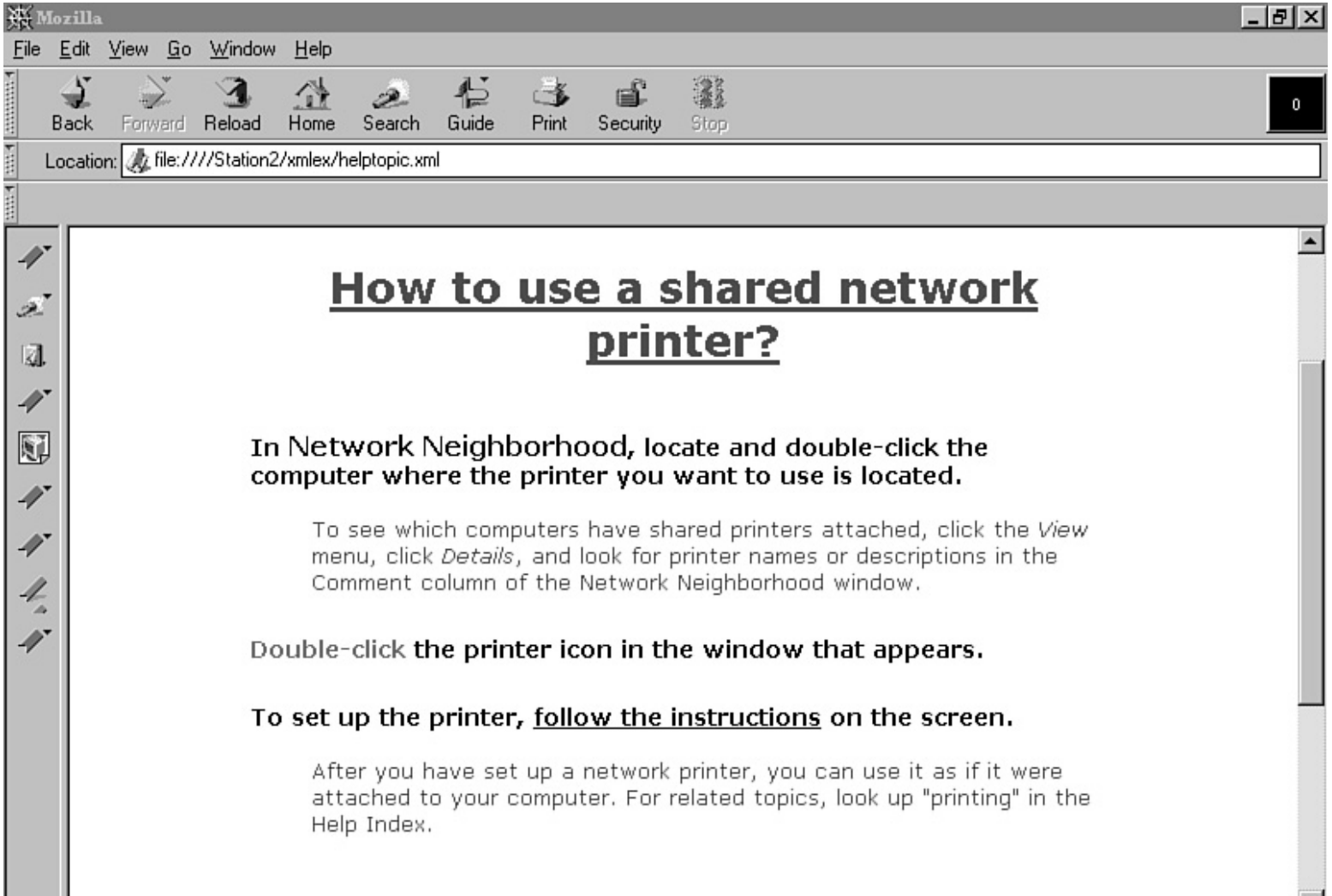
For documents, Netscape uses XML with *CSS* (Cascading Style Sheets). As far as XSL is concerned, Netscape has adopted a wait and see approach, because XSL isn't a stable standard yet.

For data and metadata, Netscape favors the *RDF* (Resource Description Framework).

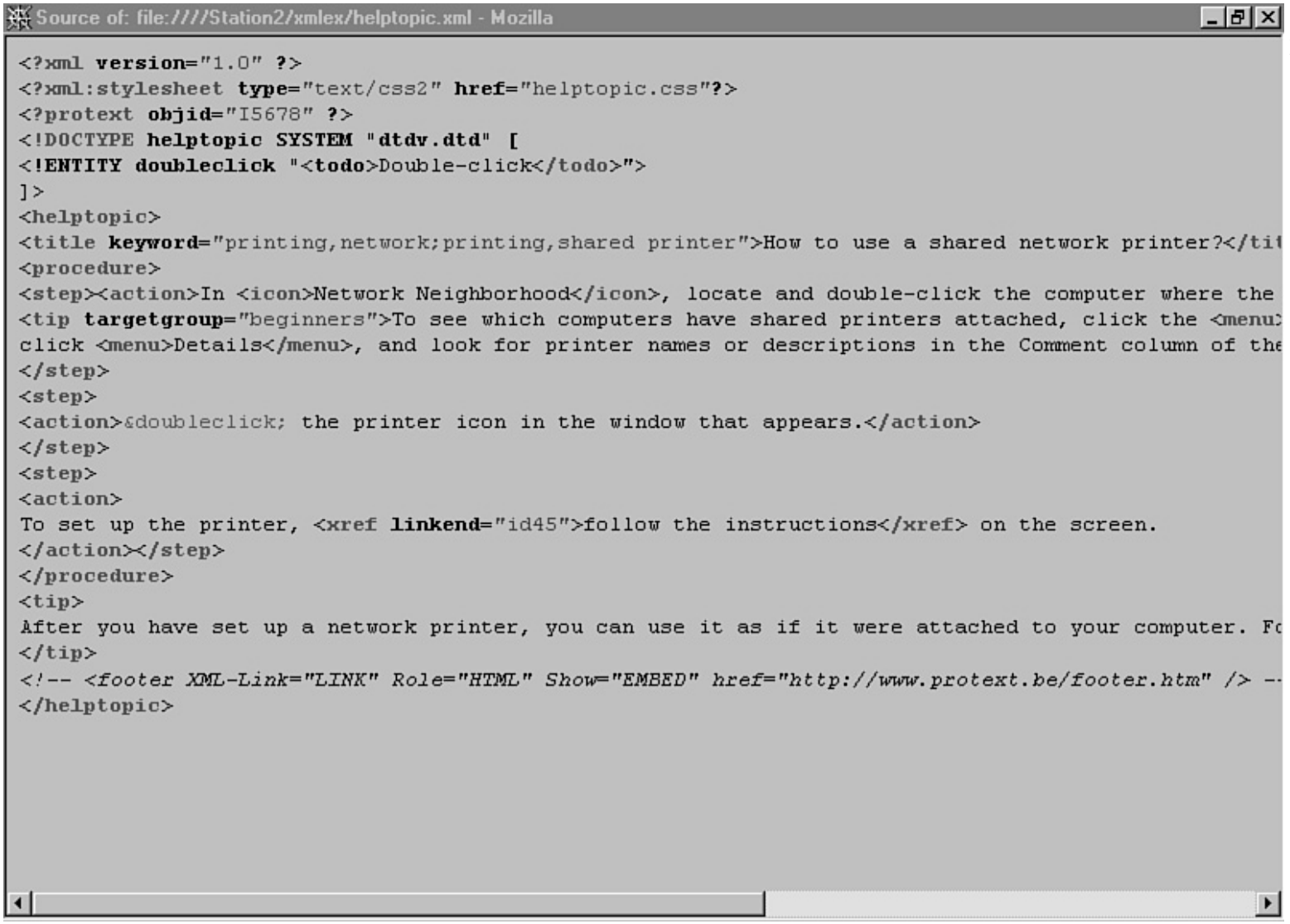
RDF is a framework for describing and interchanging metadata. It is a model for describing resources; resources that have properties that take certain values.

*http://www.macmillan.com/hyb/bio.htm

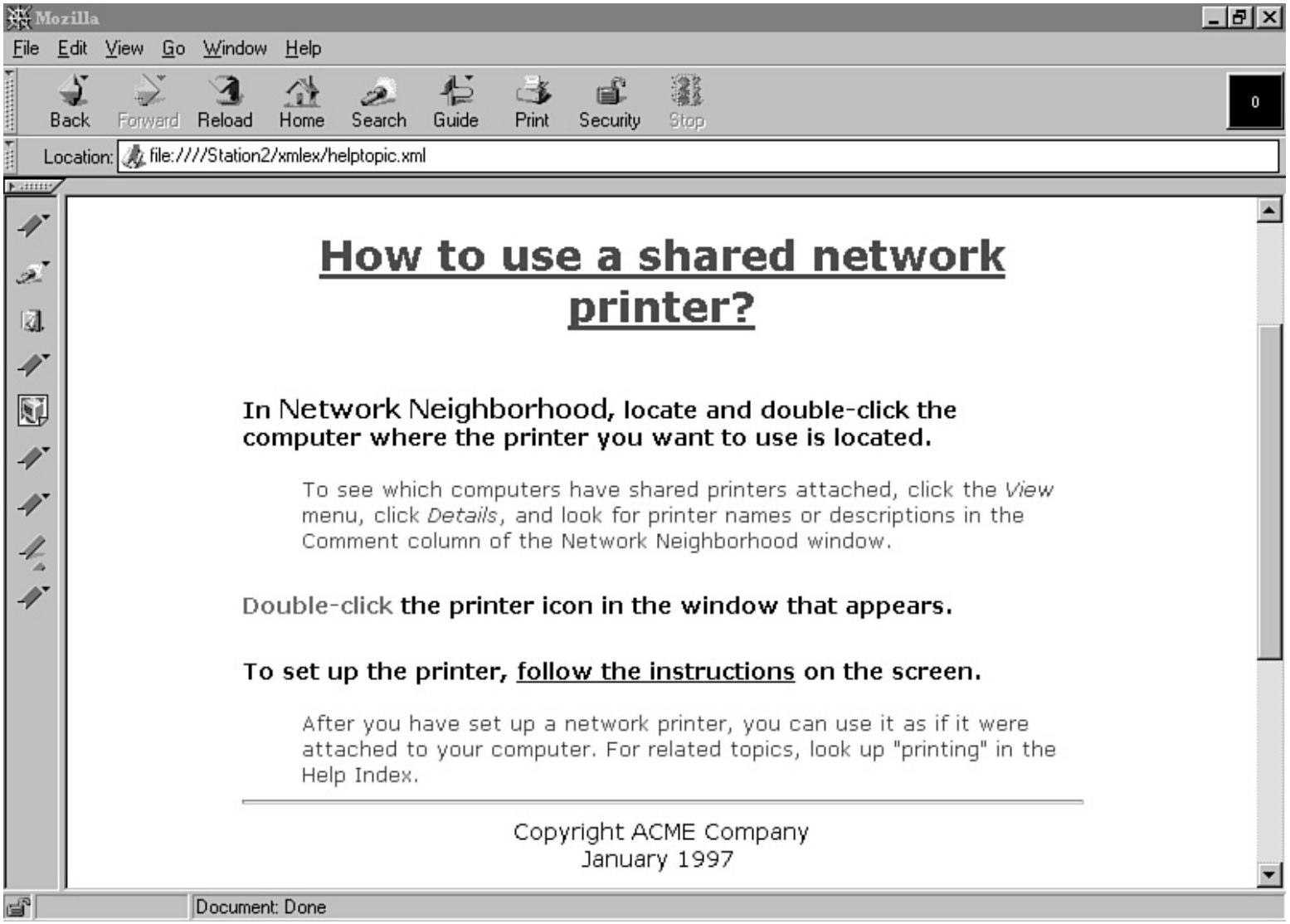
A *resource* is any object that can have a URI (Uniform Resource Identifier).



PropertyType is the name of the property.



Value is the value of the property.

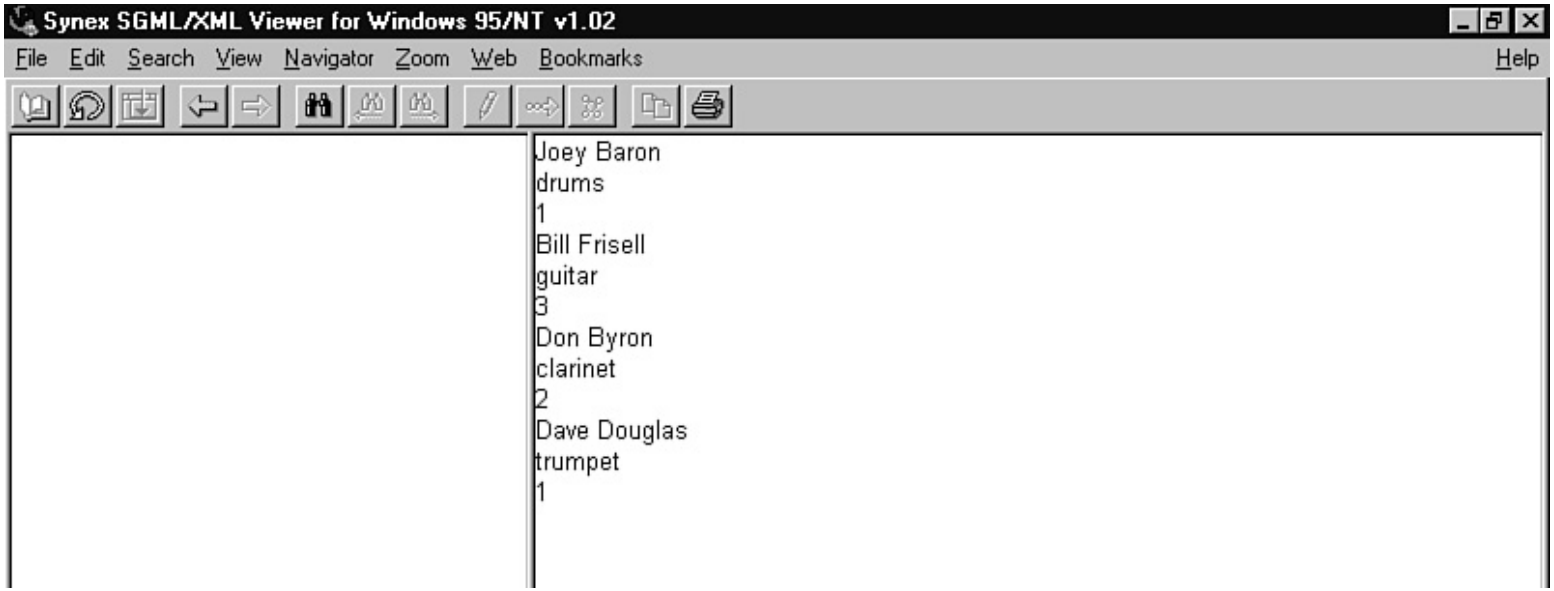


PropertyTypes and Values can be resources on their own.

An example of these concepts:

- Resource = http://www.macmillan.com/history/history.htm
- PropertyType = lastupdated
- Value = 2/12/1998

The relationships between resources, PropertyTypes, and Values are represented by a directed labeled graph, as shown in Figure 13.1.

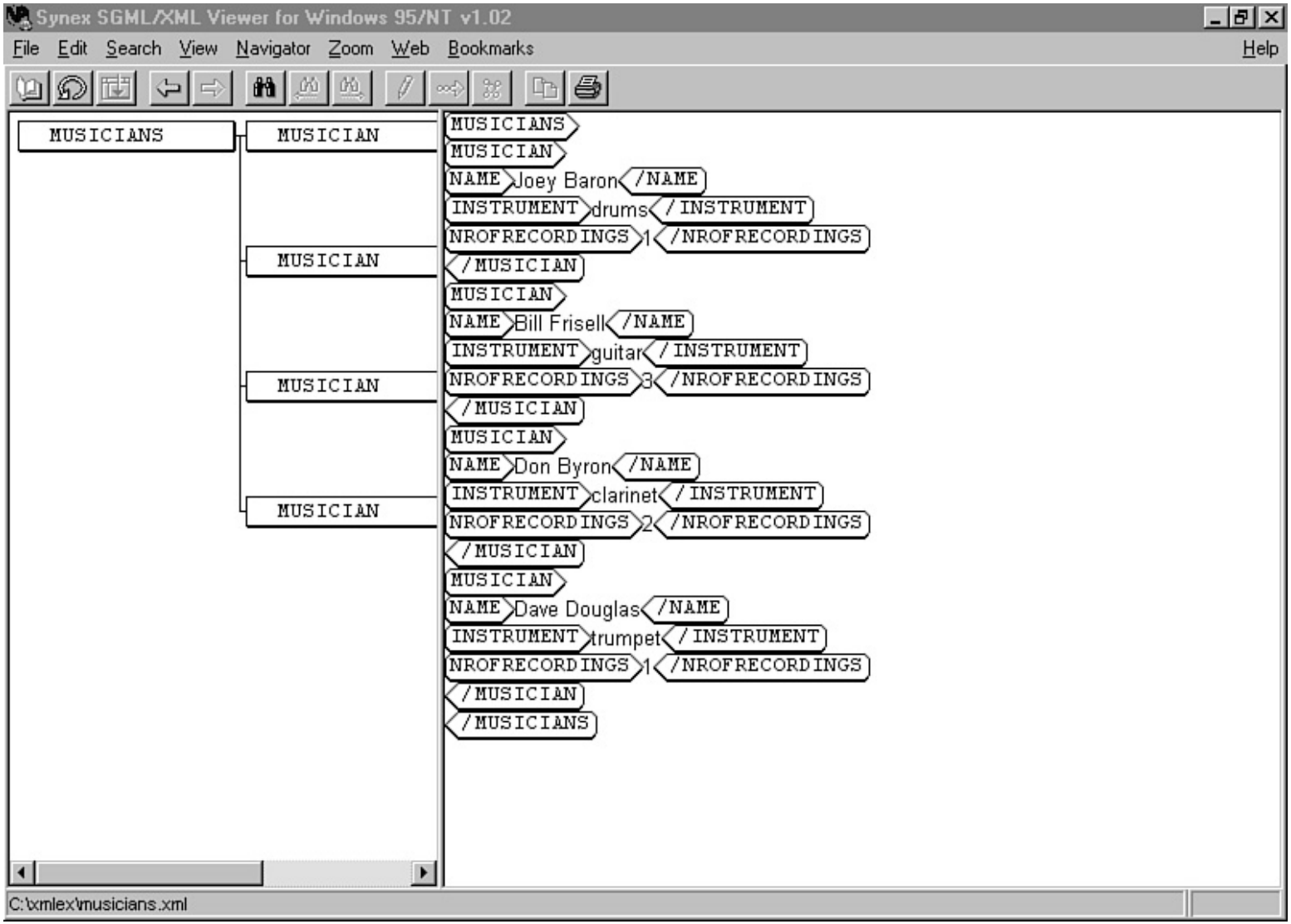


A *graph* is a collection of points in which some pair of points are connected by lines. In a *directed graph* the lines have arrowheads, indicating a travel direction.

Figure 13.1 *Our example as a directed labeled graph.*

XML is expected to become the preferred format for the interchange of these (meta)data.

To learn more about RDF, visit http://www.w3.org/TR/WD-rdf-syntax/.



Viewing XML in Netscape Navigator 4

Netscape Navigator 4 does not support XML.

If you want to use XML with Netscape Navigator 4, you first need to convert (outside the Netscape environment) to HTML or another format that Navigator 4 can handle natively.

Viewing XML in Mozilla 5/Gecko

Netscape has put the source code of its browser in the public domain, where it is being developed further by the Free Software Community, under the name Mozilla.

Information on the Mozilla project can be found at the following Web sites:

- General information is at http://www.mozilla.org/.
- The FAQ about the Mozilla project is located at http://junior.apk.net/~qc/dok/mozilla_faq.
- The compiled binaries can be retrieved from http://www.mozilla.org/binaries/html.

Gecko is the name of just one part (module) of the Mozilla development effort: the new layout engine. It is aimed at being an embeddable object. It will be used by any application developer to add Web browsing functionality to their application.

You'll find the Gecko homepage at http://developer.netscape.com/software/communicator/ngl/index.html.

The XML support of Mozilla:

- Includes James Clark's expat parser
- Implements XML+CSS (Cascading Style Sheet) support at the document level
- Offers an API to the XML Document Object Model (DOM) via JavaScript and any plug-in
- Supports XLink
- Supports elements from the HTML namespace

From the standards viewpoint, Mozilla supports parts of CSS 2.0 and fully supports:

- The XML 1.0 recommendation
- The DOM 1.0 recommendation
- The CSS 1.0 recommendation

To view an XML file in Mozilla you need to define a CSS style sheet with formatting specifications. These are kept in a separate file. You relate your XML file to the CSS file by including a processing instruction inside your XML file.

Listing 13.1 is an XML file describing a help topic for an online help system.

Listing 13.1 helptopic.xml

```
1:   <?xml version="1.0" ?>
2:   <?protext objid="I5678" ?>
3:   <!DOCTYPE helptopic SYSTEM "dtdv.dtd" [
4:   <!ENTITY doubleclick "<todo>Double-click</todo>">
5:   ]>
6:   <helptopic>
7:   <title keyword="printing,network;printing,shared printer">
⇒How to use a shared network printer?</title>
8:   <procedure>
9:   <step><action>In <icon>Network Neighborhood</icon>,
⇒locate and double-click the computer where the printer
```

```
⇒you want to use is located. </action>
10:  <tip targetgroup=“beginners”>To see which computers
⇒have shared printers attached, click the <menu>View</menu> menu,
⇒click <menu>Details</menu>, and look for printer names or descriptions
⇒in the Comment column of the Network Neighborhood window.</tip>
11:  </step>
12:  <step>
13:  <action>&doubleclick; the printer icon in the window that
⇒appears.</action>
14:  </step>
15:  <step>
16:  <action>
17:  To set up the printer, <xref linkend=”id45”>follow the
⇒instructions</xref> on the screen.
18:  </action></step>
19:  </procedure>
20:  <tip>
21:  After you have set up a network printer, you can use it as if it were
⇒attached to your computer. For related topics, look up
⇒&quot;printing&quot; in the Help Index.
22:  </tip>
23:  </helptopic>
```

We have to relate our XML file with a style sheet, helptopic.css. This is done by adding the following processing instruction (PI) to our XML file.

```
<?xml-stylesheet type=“text/css2” href=“helptopic.css”?>
```

Our XML file now looks as shown in Listing 13.2:

Listing 13.2 helptopic.xml with Processing Instruction Added

```
1:  <?xml version=”1.0” ?>
2:  <?xml-stylesheet type=“text/css2” href=“helptopic.css”?>
3:  <?protext objid=“I5678” ?>
4:  <!DOCTYPE helptopic SYSTEM “dtdv.dtd” [
5:  <!ENTITY doubleclick “<todo>Double-click</todo>”>
6:  ]>
7:  <helptopic>
8:  ...
```

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

In this PI, we refer to a CSS file, helptopic.css.

Because there are no fixed semantics (how an element needs to be presented or how it needs to behave) associated with our XML elements, the first thing we need to indicate is whether an element is *inline-level* (does not cause line breaks) or *block-level* (causes line breaks).

An example of a block-level element is the action element that needs to be captured in CSS as follows:

```
action {display:block}
```

An example of an inline-level element is the todo element which, translated to CSS syntax, is:

```
todo [display:inline}
```

The rest of the style sheet is completed with traditional formatting specifications. See Listing 13.3.

Listing 13.3 helptopic.css

```
helptopic {display: block;
    margin-top:3cm;
    margin-left:2cm;
    margin-right:2cm;
    margin-bottom:6cm;
    font-family:Verdana, Arial;
    font-size:11pt;
    padding:20pt;}

title {display: block;
    font-size:20pt;
    color:blue;
    font-weight:bold;
    text-align:center;
    margin-bottom:30pt;
    text-decoration:underline;}

procedure {display:block;
    margin-bottom:30pt}

step {display:block;
    margin-bottom:18pt}

action {display:block;
    font-weight:bold;}

tip {display:block;
    font-size:10pt;
    margin-left:+1cm;
    margin-top:12pt;
    color:blue;}

icon {display:inline;
    font-size:12pt;}

todo {display:inline;
    color:red;}

menu {display:inline;
    font-style:italic;}

xref {display:inline;
```

```
43:      text-decoration:underline;}
```

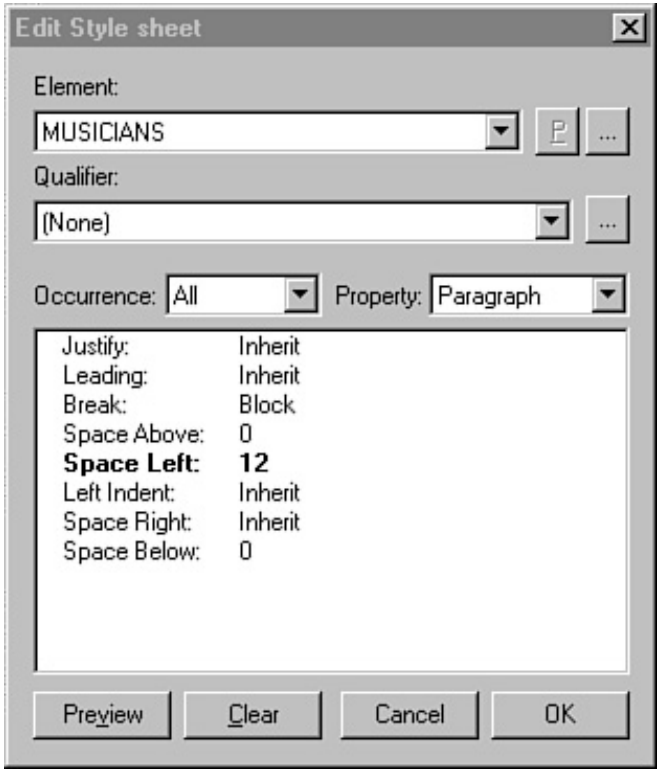
The result, in a Mozilla 5 browser, is shown in Figure 13.2.

Figure 13.2 *The XML file in Mozilla 5 using CSS.*

Figure 13.3 shows the source code.

This figure shows as source the XML file itself, not an HTML converted version of it.

At the time of writing, not much information was available about working with CSS and XML in Mozilla 5. It isn't clear, for example, which features of CSS Level 2 will or won't be implemented.



New documentation on the CSS support in Mozilla is announced at this address: http://www.mozilla.org/rdf/doc/xml.html.

Although documentation is lacking and all CSS Level 2 features may not be available in the near future, XML+CSS is the way to go for displaying XML documents.

Figure 13.3 *The source file of the Web page.*

The XML data is directly rendered to HTML without any intermediate conversion or translation phase.

Mozilla 5 provides 100% support of the DOM 1.0 recommendation.

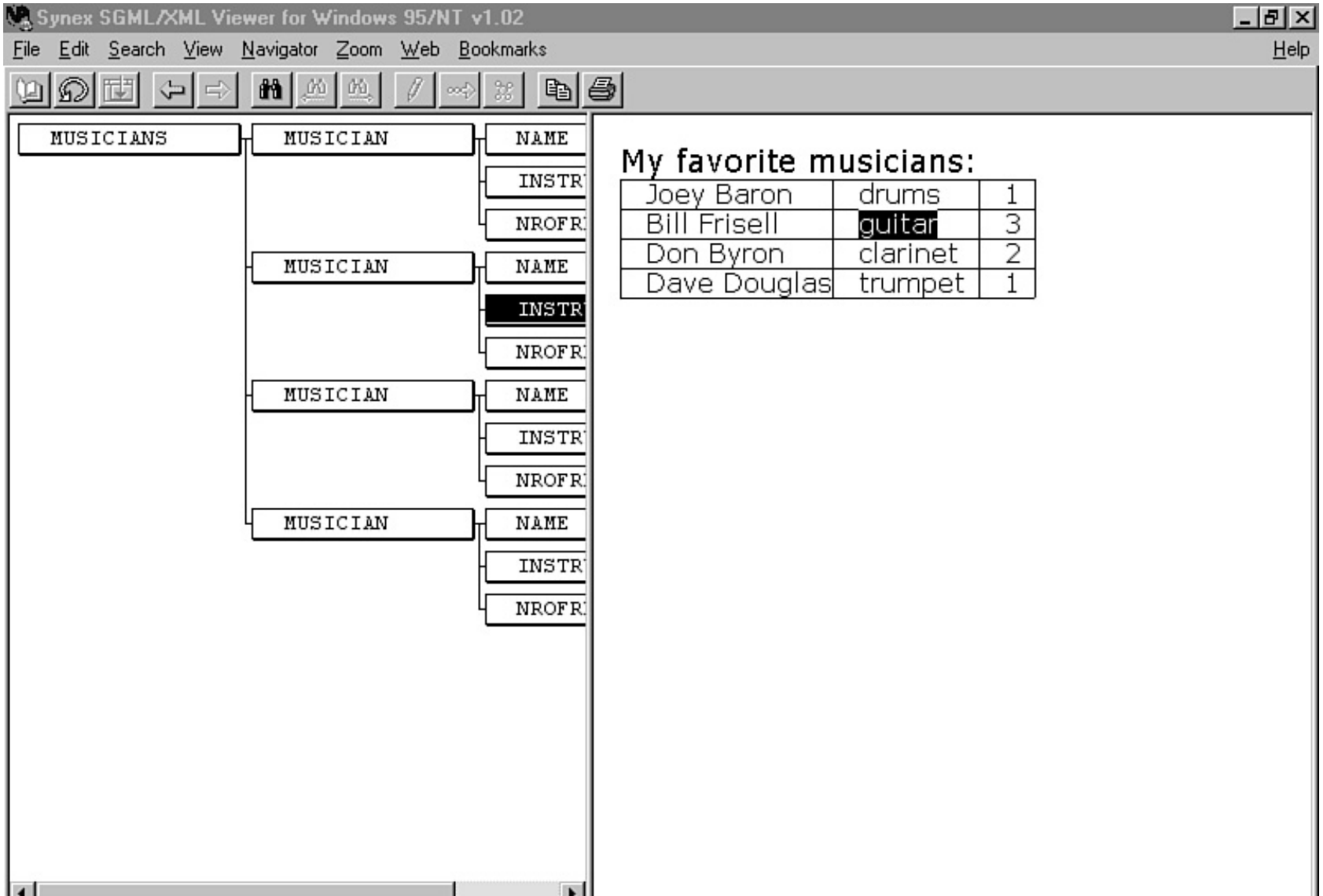
The following example is JavaScript source code. It uses the DOM API to replace a price in U.S. dollars with a price in Canadian dollars.

Listing 13.4 books2.xml

```
1:   <?xml version="1.0"?>
2:   <?xml-stylesheet type="text/css" href="books.css"?>
3:   <books xmlns:html="http://www.w3.org/TR/REC-html40">
4:        <html:form>
5:             <html:h3>Convert to Canadian dollars:</html:h3>
6:             <html:input type="button" onclick="convert()" value="CAN"/>
7:        </html:form>
8:        <book>
9:             <title>Sam's Teach Yourself C++ in 21 Days, Second Edition
10:            </title>
11:            <author>Jesse Liberty
12:            </author>
13:            <description>This book teaches you the basics of
⇒object-oriented programming with C++ and is completely revised to ANSI
⇒standards. It can be used with any C++ compiler.
14:            </description>
15:            <ISBN>0-672-31070-8
16:            </ISBN>
17:            <pages>700
18:            </pages>
19:            <targetgroup>Beginning - Intermediate
20:            </targetgroup>
21:            <price unit="USA">29.99
22:            </price>
23:      </book>
24:      <book>
25:            <title>Maximum Java 1.1
26:            </title>
27:            <author>Glenn Vanderburg et al.
```

```
28:             </author>
29:             <description>Written by JAVA experts, this book explores
⇒the JAVA 1.1 language, tools, and core JAVA API without reviewing
⇒fundamentals or basic techniques.
30:             </description>
31:             <ISBN>1-57521-290-0
32:             </ISBN>
33:             <pages>900
34:             </pages>
35:             <targetgroup>Expert
36:             </targetgroup>
37:             <price unit="USA">49.99
38:             </price>
39:      </book>
40:      <book>
41:             <title>JavaScript Unleashed, Second Edition
42:             </title>
43:             <author>Richard Wagner et al.
44:             </author>
45:             <description>This book helps you thoroughly understand and
⇒apply JavaScript.
46:             </description>
47:             <ISBN>1-57521-306-0
48:             </ISBN>
49:             <pages>1000
50:             </pages>
51:             <targetgroup>Casual - Experienced
52:             </targetgroup>
53:             <price>49.99
54:             </price>
55:      </book>
56:      <book>
57:             <title>Sam's Teach Yourself Visual C++ 5 in 21 Days,
⇒Fourth Edition
58:             </title>
59:             <author>Nathan and Ori Gurewich
60:             </author>
61:             <description>This book merges the power of the best-
⇒selling "Teach Yourself" series with the knowledge of Nathan and Ori
⇒Gurewich, renowned experts in code, creating the most efficient way to
⇒learn Visual C++.
62:             </description>
63:             <ISBN>0-672-31014-7
```

```
64:             </ISBN>
65:             <pages>832
66:             </pages>
67:             <targetgroup>New - Casual
68:             </targetgroup>
69:             <price>35.00
70:             </price>
71:       </book>
72:       <html:script src="convert.js" />
73:     </books>
```



Line 2 contains the processing instruction to use the style sheet defined in the books.css file.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

On line 3 (the element books), the html namespace has been declared because HTML elements will be used within the file.

You'll find the HTML elements form, h3, and input on lines 4 through 7, and the script element on line 72.

By declaring this namespace the browser knows that the element h3 from namespace html receives the semantics (formatting and behavior) defined for this namespace.

The input element (line 6) refers to a JavaScript function defined in the convert.js file, referred to on line 72, and listed in Listing 13.5.

Listing 13.5 convert.js

```
1:  function convert() {
2:  var priceElements = document.getElementsByTagName(“price”);
```

```
3:  //returns a NodeList of all elements with the tag price
4:  var i;
5:  for (i=0;i < priceElements.length;i++) {
⇒// looping over all those price elements
6:       var USAprice =
⇒parseFloat(priceElements.item(i).firstChild.nodeValue);
7:       //the firstChild property returns the Text node
8:       //of this Text node we take the nodeValue, being the text itself
9:       //we convert this text to a number (floating)
10:      priceElements.item(i).setAttribute(“unit”,“Canadian dollar”);
11:      //we set the attribute “unit” to value “Canadian dollar”
12:      var convertedprice = USAprice * 1.4;
13:      //we convert the price to Canadian dollar
14:      var newprice = document.createTextNode(convertedprice);
15:      //we create a new Text node containing the new price
16:      priceElements.item(i).replaceChild(newprice,priceElements.
⇒item(i).firstChild);
17:      //we replace the old Text node with the new one
18:      }
19:  }
```

First we create a NodeList containing all the element nodes with name price (line 2).

We loop over all those element nodes (line 5).

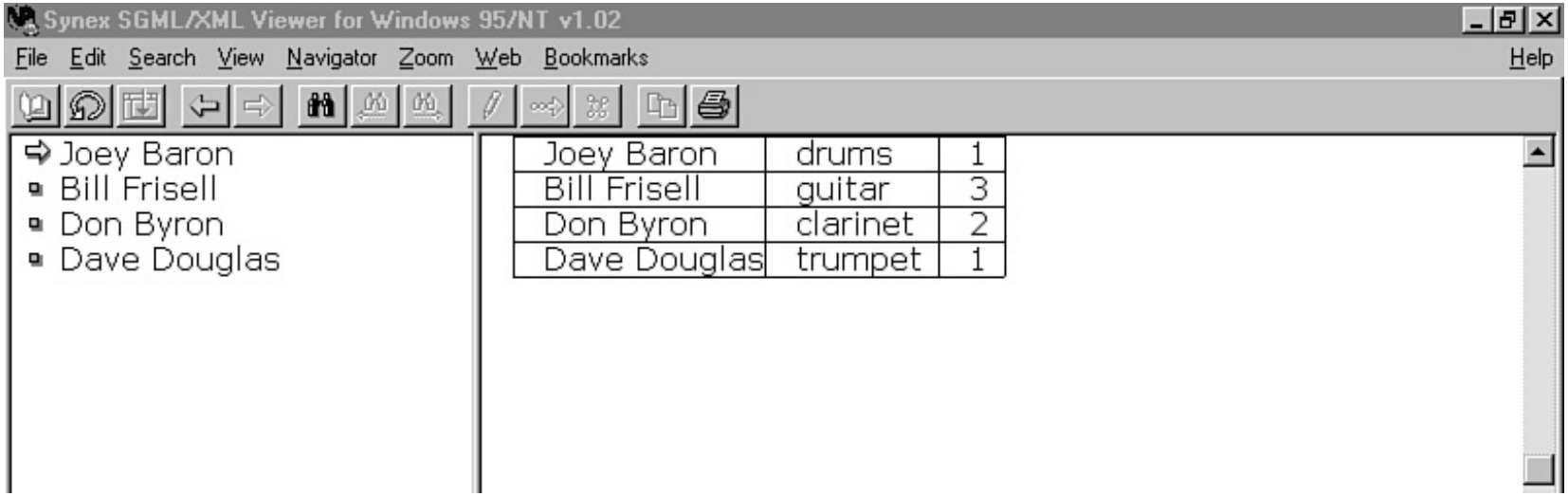
For each price element we first use the firstChild property, which returns a node of type Text. We take the content from this text node by using the nodeValue property. This string needs to be converted to a number, hence the use of the parseFloat() function.

This number is converted on line 12.

With this value we create a new TextNode (line 14), which is then used to replace the old value (line 16).

Mozilla supports transclusions using Xlink syntax.

Transclusions are (portions of) other documents that are included at the place of reference as if they occurred locally.



In other words, you refer to another document and the contents of that document are placed at the point of reference.

The following is an example of a transclusion.

```
<footer xml:link="simple" role="HTML" show="embed" href="footer.htm"/>
```

The XML element footer is a link that retrieves the content of footer.htm and embeds it at its position in the XML file.

The content of footer.htm could be as shown in Listing 13.6.

Listing 13.6 footer.htm

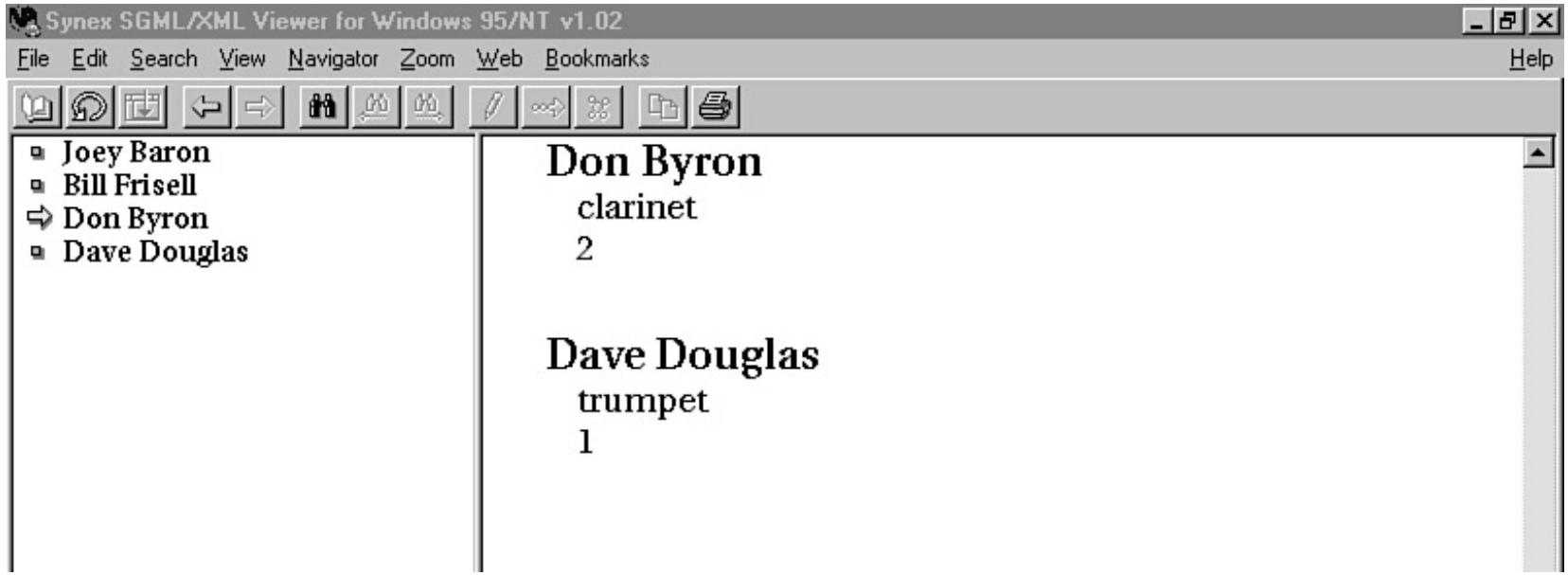
```
1:  <hr style="margin-top:2cm">
2:  <div class="footer" align="center">
3:  <small>
4:  Copyright ACME Company<BR>
5:  January 1997
6:  </small>
```

```
7:  </div>
```

This gives the result in the browser as shown in Figure 13.4.

Figure 13.4 *Transclusion in Mozilla 5.*

Using transclusions is a fine alternative for today's practice of server-specific server-side include statements. Now it is possible to define client-side inclusions.



Microsoft seems to be working on transclusion as well, but it's uncertain at the moment if their version will be compatible with other browsers.

Viewing XML with DocZilla

The Finnish company Citec Information Technologies is using the Mozilla 5 code to develop a high-end browser. This high-end browser will be a functional XML, SGML, and HTML browser with the following features:

- The ability to display documents with CSS
- Full DOM and ECMAscript support
- On-the-fly generation of TOCs
- A structured indexing and search mechanism

More information about Citec's browser can be found at http://www.citec.fi/multidoczilla.

Viewing XML with Browsers Based on Inso's Viewport Engine

For years the Inso/Synex Viewport engine has been the market leader for browsing native SGML files.

The engine has been used in products such as

- Panorama (Interleaf, previously SoftQuad)
- SplitVision (Sšrman Information)
- Multidoc Pro (Citec Information Technology)

The latest version of the engine is also able to render XML files.

Features of the Viewport Engine

Noteworthy features of the Viewport environment include the use of style sheets, navigators, webs, and other hypermedia support.

You can define one or more *style sheets* for every document. This can be done by using a graphical style sheet editor.

Unfortunately, the language used to define the style specification is Viewport specific. But in upcoming releases XSL will either complement or supersede the native style sheet format.

A *navigator* serves as an active and dynamic table of contents view.

Any document element can be extracted and assembled into a navigator. The elements maintain their original order and hierarchy in the document. This makes it possible to easily define a table of contents, a table of graphics, a table of tables, and so on.

A *web* is an independent file containing anchors and links.

Anchors can be attached to textual spans or to a region of a graphic. They can contain an annotation and can serve as a bookmark.

Links connect two anchors.

You can mount and unmount webs. If a web is mounted, anchors are attached to open documents and are displayed as clickable icons. This leads to very powerful possibilities. In one of our actual projects, we deliver the same document with two views: an "actual" view and a "change" view. In the actual view the user sees the information as it is now. In the change view another style sheet is opened, showing new information in red, changed information in blue with change bars, and deleted information as strikethrough. This change information is kept in the document in a revision attribute. Also in change view a web is opened containing a list of anchors referring to all the new and changed topics. This allows the user to go over all the changes very quickly. Furthermore, the reasons for the changes are explained in the associated annotation.

The same concept of link information kept outside the document itself is also part of the XLink proposal.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

How it Works

In addition to your XML file, you need accompanying style sheets and navigators. Both use specific markups defined in an SGML DTD. To create a style sheet, you need to create a file with an .ssh extension. The content of this file needs to be that shown in Listing 13.7.

Listing 13.7 A Viewport Style Sheet Definition

```
1:  <!doctype stylesheet PUBLIC “-//Synex Information AB//DTD stylesheet
⇒V2.00//EN>
2:  <stylesheet>
3:  </stylesheet>
```

The additional content of the style specifications will be automatically supplied when you use the graphical style sheet editor.

To create a navigator you need to create a file with a .nav extension. The content of this file needs to be that shown in Listing 13.8.

Listing 13.8 A Viewport Navigator Definition

```
1:  <!doctype toc-def PUBLIC “-//SYNEX Information AB//DTD Navigator
⇒v2.00//EN”>
2:  <toc-def>
3:  </toc-def>
```

The additional content of this file is automatically supplied when you use the browser to define navigators.

Now that we have our style sheet and navigator defined we need to establish a relationship between our XML file and these supporting files.

This can be accomplished in two ways:

- They can be related to the document itself.
- If a DTD having a public identifier is used, they can be related to this DTD.

In case of a relationship with the document itself, you have to add to the document the following processing instruction:

```
<?stylespec “menuname” “filename”?>
```

for the style specification, where:

- menuname is the name that will be used on the View menu of the browser.
- filename is the name of the style sheet file (*.ssh).

and

```
<?navigator “menuname” “filename”?>
```

for the navigator, where:

- menuname is the name that will be used on the Navigator menu of the browser.
- filename is the name of the navigator file (*.nav).

If the DTD has a public identifier, you need to edit the ENTITYRSC file in the Viewport environment. In this ENTITYRSC file you can define a relationship between style sheets and navigators on one hand and public identifiers on the other. See Listing 13.9.

Listing 13.9 Defining Relationships Between a DTD and Viewport Style Sheets and Navigators

```
1:  PUBLIC “-//Pro Text//DTD online help//EN”
2:      STYLESPEC “standard”     “standard.ssh”
3:      STYLESPEC “large”     “large.ssh”
4:      NAVIGATOR “topics”      “topics.nav”
5:      NAVIGATOR “figures”        “figures.nav”
```

Let’s show an example starting with the XML file, as shown in Listing 13.10.

Listing 13.10 musicians.xml

```
1:   <?xml version=“1.0”?>
2:   <!DOCTYPE musicians [
```

```
3:   <!ELEMENT musicians  (musician)+ >
4:   <!ELEMENT musician  (name, instrument, NrOfRecordings)>
5:   <!ELEMENT (name, instrument, NrOfRecordings)  (#PCDATA)>
6:   ]>
7:   <musicians>
8:     <musician>
9:   <name>Joey Baron
10:  </name>
11:  <instrument>drums
12:  </instrument>
13:  <NrOfRecordings>1
14:  </NrOfRecordings>
15:    </musician>
16:    <musician>
17:  <name>Bill Frisell
18:  </name>
19:  <instrument>guitar
20:  </instrument>
21:  <NrOfRecordings>3
22:  </NrOfRecordings>
23:    </musician>
24:    <musician>
25:  <name>Don Byron
26:  </name>
27:  <instrument>clarinet
28:  </instrument>
29:  <NrOfRecordings>2
30:  </NrOfRecordings>
31:    </musician>
32:    <musician>
33:  <name>Dave Douglas
34:  </name>
35:  <instrument>trumpet
36:  </instrument>
37:  <NrOfRecordings>1
38:  </NrOfRecordings>
```

```
39:    </musician>
40:  </musicians>
```

Now add two style sheets and one navigator.

Files musicians.ssh and musician2.ssh contain the following:

```
<!doctype stylesheet PUBLIC “-//Synex Information AB//DTD stylesheet V2.00//EN>
<stylesheet>
</stylesheet>
```

File musician.nav contains the following:

```
<!doctype toc-def PUBLIC “-//SYNEX Information AB//DTD Navigator v2.00//EN”>
<toc-def>
</toc-def>
```

Now let’s bring the files together by including processing instructions in our XML file, shown in Listing 13.11.

Listing 13.11 musicians.xml with Processing Instructions

```
1:  <?xml version=“1.0”?>
2:  <!DOCTYPE musicians [
3:  <!ELEMENT musicians  (musician)+ >
4:  <!ELEMENT musician  (name, instrument, NrOfRecordings)>
5:  <!ELEMENT name  (#PCDATA)>
6:  <!ELEMENT instrument  (#PCDATA)>
7:  <!ELEMENT NrOfRecordings (#PCDATA)>
8:  ]>
9:  <?stylespec “table” “musicians.ssh”?>
10:  <?stylespec “indent” “musician2.ssh”?>
11:  <?navigator “toc” “musicians.nav”?>
12:  <musicians>
13:  ...
14:  </musicians>
```

---

table and indent will be added to the View menu now and toc to the Navigator menu.

Open the XML file with the Synex Viewport–based browser. See Figure 13.5.

Figure 13.5 *The XML file opened for the first time.*

We don't see much for the moment, but that's because we didn't add information to our style sheets and navigators yet.

We change our view (using the View menu) to include tags and the tree structure of our document. See Figure 13.6.

Figure 13.6 *Tag names and tree structure are visible.*

By clicking on a tag or a tree node with the right mouse button, you see the option to open the style sheet editor. See Figure 13.7.

Figure 13.7 *The style sheet editor opened.*

Figure 13.8 shows a possible result of our style sheet editing.

Figure 13.8 *A finished style sheet applied.*

The next thing to do is to define a navigator. For this file it would be good to have an overview with the musicians' names.

To open the navigator editor, use the right mouse button to click on an element name that you want to include in your navigator. See Figure 13.9.

Clicking the mouse button displays the results shown in Figure 13.10.

Figure 13.9 *The navigator editor.*

Figure 13.10 *The defined navigator.*

You'll see the same file but with another style sheet attached, as shown in Figure 13.11.

Figure 13.11 *Another style sheet attached.*

The style sheet used in this latest example is shown in Listing 13.12.

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Listing 13.12 A Sample Style Sheet

```
1:   <!DOCTYPE STYLESHEET PUBLIC “-//SYNEX INFORMATION AB//DTD STYLESHEET
⇒V2.00//EN”[
2:   <!ELEMENT sheet (margins?)>
3:   <!ELEMENT margins EMPTY>
4:   <!ATTLIST margins
5:        left CDATA #IMPLIED
6:        right CDATA #IMPLIED
7:        top CDATA #IMPLIED
8:        bottom CDATA #IMPLIED>
9:
10:  <!ELEMENT font-shadow EMPTY>
11:  <!ATTLIST font-shadow
12:       v NUMBER #REQUIRED>
13:
14:  <!ELEMENT shadow-color EMPTY>
15:  <!ATTLIST shadow-color
16:       v CDATA #REQUIRED>
```

```
17:
18:  ]>
19:
20:  <STYLESHEET>
21:
22:  <STYLE TAG=“NROFRECORDINGS”>
23:  <SPC-LEFT V=“36”>
24:  <BREAK-BEFORE>
25:  <BREAK-AFTER>
26:  </STYLE>
27:
28:  <STYLE TAG=“MUSICIANS”>
29:  <FONT-FAMILY V=“Utopia”>
30:  <FONT-SIZE V=14>
31:  <SPC-ABOVE V=“12”>
32:  <SPC-LEFT V=“24”>
33:  <SPC-BELOW V=“12”>
34:  <BREAK-BEFORE>
35:  <BREAK-AFTER>
36:  </STYLE>
37:
38:  <STYLE TAG=“NAME”>
39:  <FONT-WEIGHT V=Bold>
40:  <FONT-SCALE V=“120”>
41:  <SPC-ABOVE V=“24”>
42:  <BREAK-BEFORE>
43:  <BREAK-AFTER>
44:  </STYLE>
45:
46:  <STYLE TAG=“INSTRUMENT”>
47:  <SPC-LEFT V=“36”>
48:  <BREAK-BEFORE>
49:  <BREAK-AFTER>
50:  </STYLE>
```

```
51:

52:  </STYLESHEET>
```

The navigator used is shown in Listing 13.13.

Listing 13.13 A Sample Navigator

```
1:  <!DOCTYPE TOC-DEF PUBLIC “-//SYNEX INFORMATION AB//DTD NAVIGATOR

⇒V2.00//EN”>

2:

3:  <TOC-DEF SCALE=70>

4:  <TOC BODY=“MUSICIAN” TITLE=”NAME”>

5:  </TOC-DEF>
```

The Synex Viewport engine is the state of the art for viewing native SGML files. That it also directly supports XML documents can only be applauded.

But there are also some drawbacks:

- The style sheet mechanism is proprietary.
- The browser doesn’t come for free.
- The product is only known inside the SGML community, not in the much broader HTML community.

Summary

In this chapter we discussed other browsers you can use to browse/view XML. Although Mozilla 5 is still under development, it shows a lot of promise for directly displaying XML files with associated CSS style sheets.

Furthermore it attempts to consistently implement Web standards and recommendations. This alone needs to be heavily applauded.

The DocZilla and Synex Viewport were discussed as well.

Q&A

Q I downloaded Gecko and it doesn’t seem to work well. What’s happening?
A The Gecko development is continuously going on. Every day a new build of Gecko is made available. If the build of one day isn’t stable enough, chances are great that the next build is already better.
Q I don’t see formatted text in my Synex browser.
A Check if your XML file is in one way or another associated with a style sheet.

Exercises

Using Listing 13.14,

- Create a CSS (Cascading Style Sheet) file and relate the XML file
- Open the XML file with Mozilla/Gecko

Listing 13.14 books.xml

```
1:  <?xml version=“1.0”?>
2:  <books>
3:      <book>
4:          <title>Sam’s Teach Yourself C++ in 21 Days, Second Edition
5:          </title>
6:          <author>Jesse Liberty
7:          </author>
8:          <description>This book teaches you the basics of
⇒object-oriented programming with C++ and is completely revised
⇒to ANSI standards. It can be used with any C++ compiler.
9:          </description>
10:          <ISBN>0-672-31070-8
11:          </ISBN>
12:          <pages>700
13:          </pages>
14:          <targetgroup>Beginning - Intermediate
15:          </targetgroup>
16:          <price unit=“USA”>29.99
17:          </price>
18:      </book>
19:      <book>
20:          <title>Maximum Java 1.1
21:          </title>
22:          <author>Glenn Vanderburg et al.
23:          </author>
24:          <description>Written by JAVA experts, this book
⇒explores the JAVA 1.1 language, tools, and core JAVA API without
⇒reviewing fundamentals or basic techniques.
25:          </description>
```

```
26:          <ISBN>1-57521-290-0
27:          </ISBN>
28:          <pages>900
29:          </pages>
30:          <targetgroup>Expert
31:          </targetgroup>
32:          <price unit=“USA”>49.99
33:          </price>
34:      </book>
35:      <book>
36:          <title>JavaScript Unleashed, Second Edition
37:          </title>
38:          <author>Richard Wagner et al.
39:          </author>
40:          <description>This book helps you thoroughly understand and
⇒apply JavaScript.
41:          </description>
42:          <ISBN>1-57521-306-0
43:          </ISBN>
44:          <pages>1000
45:          </pages>
46:          <targetgroup>Casual - Experienced
47:          </targetgroup>
48:          <price>49.99
49:          </price>
50:      </book>
51:      <book>
52:          <title>Sam’s Teach Yourself Visual C++ 5 in 21 Days, Fourth
⇒Edition
53:          </title>
54:          <author>Nathan and Ori Gurewich
55:          </author>
56:          <description>This book merges the power of the best-selling
⇒“Teach Yourself” series with the knowledge of Nathan and Ori Gurewich,
```

```
⇒renowned experts in code, creating the most efficient way to learn Visual
⇒C++.
57:          </description>
58:          <ISBN>0-672-31014-7
59:          </ISBN>
60:          <pages>832
61:          </pages>
62:          <targetgroup>New - Casual
63:          </targetgroup>
64:          <price>35.00
65:          </price>
66:      </book>
67:  </books>
```

Previous | Table of Contents | Next

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Chapter 14
Processing XML

Today you'll learn why and how XML data can be used for further processing. We will also discuss the two main programming paradigms used in XML processing:

- The event-driven approach
- The tree-based approach

Reasons for Processing XML

By using XML, you've already made the decision to mark up your data in the best possible way. You have also seen that some browsers can display your XML files natively. Why would you want to write programs to process your XML, then?

Well, the answer is limited only by your imagination, but let's enumerate some possible reasons:

- Delivery to multiple media
- Delivery to multiple target groups
- Adding, removing, and restructuring information
- Database loading
- Reporting

Delivery to Multiple Media

Information needs to be published via different media: on the Web, in hard copy, on CD-ROM, as help files, and so on.

For delivery on the Web, HTML will still be in place for awhile. But which type of HTML: version 3.2, version 4, DHTML Microsoft flavor or DHTML Netscape flavor?

Until now you had these choices:

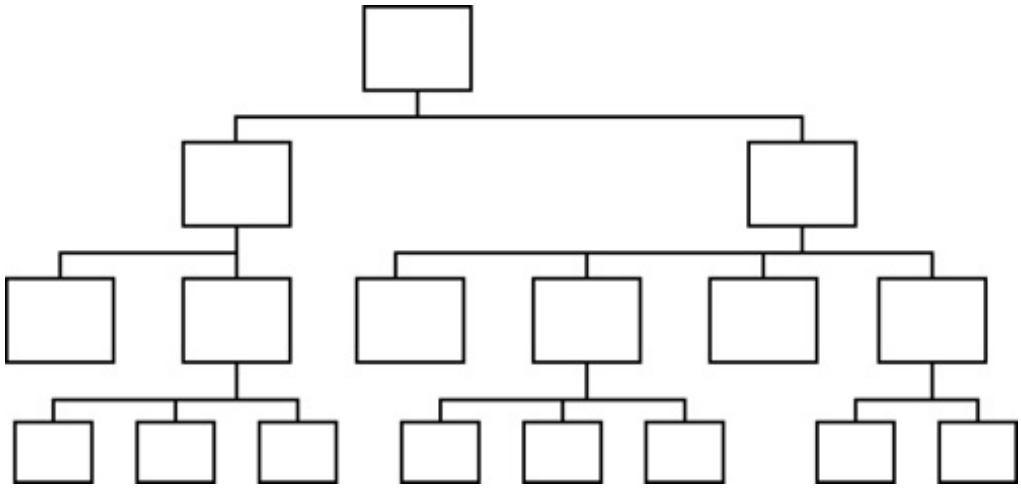
- Use only those HTML tags that are supported by all browsers (the common denominator approach).
- Make browser-specific optimized HTML pages.
- Include scripts in your HTML pages to generate browser-specific code.

With XML, you can use another tactic. You can check which browser is asking for information and transform your XML data on-the-fly to the most appropriate version of HTML.

The big advantage of this approach is that you have to manage just one source file. All formatting and processing is kept separate from your data. This means that if a new flavor of HTML arrives, you need to write a new conversion but your data doesn't need to be modified.

For getting to hard copy, you can convert your XML data to RTF (Rich Text Format) for use with Microsoft Word, or to another markup language used in other text processing tools (for example, MIF for FrameMaker).

The XML file in Listing 14.1 can be used as source code for generating RTF output.

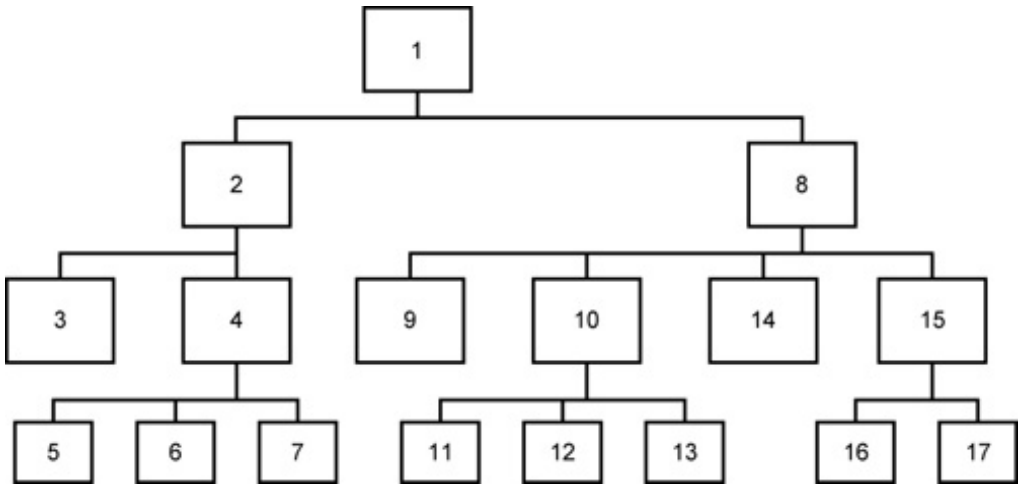


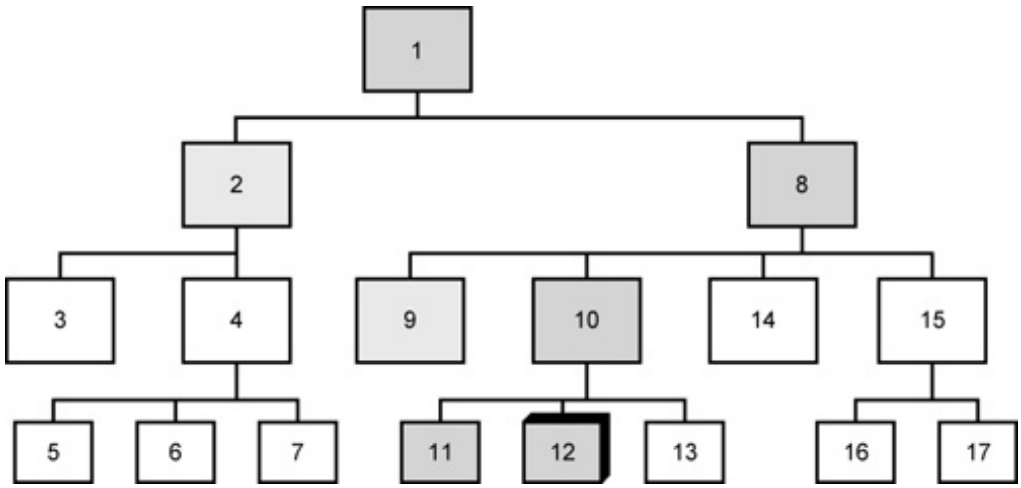
Listing 14.1 musicians.xml—Your XML File to Convert

```
 1: <?xml version="1.0"?>
 2: <!DOCTYPE musicians [
 3: <!ELEMENT musicians  (musician)+ >
 4: <!ELEMENT musician  (name, instrument, NrOfRecordings)>
 5: <!ELEMENT (name, instrument, NrOfRecordings)  (#PCDATA)>
 6: ]>
 7: <musicians>
 8:   <musician>
 9: <name>Joey Baron
10: </name>
11: <instrument>drums
12: </instrument>
13: <NrOfRecordings>1
14: </NrOfRecordings>
15:   </musician>
16:   <musician>
17: <name>Bill Frisell
18: </name>
19: <instrument>guitar
20: </instrument>
21: <NrOfRecordings>3
22: </NrOfRecordings>
23:   </musician>
24:   <musician>
25: <name>Don Byron
26: </name>
27: <instrument>clarinet
28: </instrument>
29: <NrOfRecordings>2
30: </NrOfRecordings>
```

```
31:   </musician>
32:   <musician>
33: <name>Dave Douglas
34: </name>
35: <instrument>trumpet
36: </instrument>
37: <NrOfRecordings>1
38: </NrOfRecordings>
39:   </musician>
40: </musicians>
```

This could become the RTF file in Listing 14.2.





This is just one example of how XML can be converted to another markup language. How to do it will be discussed during the next few days.

Listing 14.2 musicians.rtf—The RTF File After Conversion

```
 1: {\rtf1\ansi\ansicpg1252\uc1
 2:
 3: ... //lots of rtf code deleted
 4:
 5: \pard\plain \s18\li1440\sb320\widctlpar
 6: \adjustright \b\f15\fs28\lang2057\cgrid {Joey Baron
 7: \par }\pard\plain \s19\li2880\sb120\widctlpar
 8: \adjustright \f15\fs20\lang2057\cgrid {drums
 9: \par }\pard\plain \s20\li3600\sb120\sa400\widctlpar
10: \adjustright \i\f15\fs20\lang2057\cgrid {1
11: \par }\pard\plain \s18\li1440\sb320\widctlpar
12: \adjustright \b\f15\fs28\lang2057\cgrid {Bill Frisell
13: \par }\pard\plain \s19\li2880\sb120\widctlpar
14: \adjustright \f15\fs20\lang2057\cgrid {guitar
15: \par }\pard\plain \s20\li3600\sb120\sa400\widctlpar
16: \adjustright \i\f15\fs20\lang2057\cgrid {3
```

```
17: \par }\pard\plain \s18\li1440\sb320\widctlpar
18: \adjustright \b\f15\fs28\lang2057\cgrid {Don Byron
19: \par }\pard\plain \s19\li2880\sb120\widctlpar
20: \adjustright \f15\fs20\lang2057\cgrid {clarinet
21: \par }\pard\plain \s20\li3600\sb120\sa400\widctlpar
22: \adjustright \i\f15\fs20\lang2057\cgrid {2
23: \par }\pard\plain \s18\li1440\sb320\widctlpar
24: \adjustright \b\f15\fs28\lang2057\cgrid {Dave Douglas
25: \par }\pard\plain \s19\li2880\sb120\widctlpar
26: \adjustright \f15\fs20\lang2057\cgrid {trumpet
27: \par }\pard\plain \s20\li3600\sb120\sa400\widctlpar
28: \adjustright \i\f15\fs20\lang2057\cgrid {1}{\f2\lang2067\cgrid0
29: \par }\pard\plain \widctlpar\adjustright \fs20\lang2057\cgrid {
30: \par }}
```

This file can be read by Microsoft Word and most other word processing software.

Delivery to Multiple Target Groups

Not only do you want to publish to different media, you probably also want to deliver different information to different types of groups. Some examples:

- Beginners, advanced users, experts
- Members, non-members
- Onshore, offshore

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

The XML file in Listing 14.3 could serve as source code for generating specific output according to user experience.

Listing 14.3 delivery.xml—An XML File Used for Selective Delivery

```
 1: <?xml version=“1.0”?>
 2: <procedure type=“disassemble”>
 3:     <object>
 4: <type>Mismatcher</type>
 5: <seriesnr>21568</seriesnr>
 6:        </object>
 7:     <title>How to disassemble Mismatcher 21568</title>
 8:     <steps experience=“firsttime”>
 9:         <step>
10:             <action>Press ....
11:             </action>
12:             <result>The door ....
13:             </result>
14:         </step>
15:         <step>
16:             <action>Push ....
17:             </action>
18:             <result>The back ...
19:             </result>
20:         </step>
21:         <step>
22:             <action>Drill ....
23:             </action>
24:             <result>Part x123 ...
25:             </result>
26:         </step>
```

```
27:     </steps>
28:     <steps experience="donebefore">
29:         <step>
30:             <action>Throw it on the ground.
31:             </action>
32:         </step>
33:     </steps>
34:     <tip>Take a bottle of ....
35:     </tip>
36: </procedure>
```

This data allows you to customize the steps according to the experience of your reader. If your audience consists of first-time users, the element steps, with an attribute with the name experience and a value of donebefore (lines 28–33), can be stripped away.

You may also decide that the tip (lines 34 and 35) is only relevant for the first-timers and filter it for the target group donebefore.

Adding, Removing, and Restructuring Information

If you define the reports to be generated from your relational databases, you can arrange the data as you like: leave data out, compute new data based on existing ones (for example, an average), and so on.

The same goes for XML coded data. You can add, remove, and rearrange your data.

For example, take the data in Listing 14.4.

Listing 14.4 books.xml—An XML File for Restructuring Ideas

```
 1: <?xml version="1.0"?>
 2: <books>
 3:     <book>
 4:         <title>Sams Teach Yourself C++ in 21 Days, Second Edition
 5:         </title>
 6:         <author>Jesse Liberty
 7:         </author>
 8:         <description>This book teaches you the basics of
⇒object-oriented programming with C++ and is completely revised
⇒to ANSI standards. It can be used with any C++ compiler.
9:         </description>
10:         <ISBN>0-672-31070-8
11:         </ISBN>
12:         <pages>700
13:         </pages>
14:         <targetgroup>Beginning - Intermediate
15:         </targetgroup>
16:         <price unit="USA">29.99
17:         </price>
18:     </book>
19:     <book>
20:         ...
21:     </book>
22: </books>
```

You can do the following things with this listing:

- Leave out the targetgroup information
- Add a second price element in "Canadian dollar"
- Add a list with all book titles or with all authors
- Put the ISBN number after the price

Database Loading

Refer to Listing 14.1. It would be a good idea to have this information in a database or a spreadsheet. In fact, this is how XML is mainly perceived by Microsoft: as a delivery and exchange format for structured data. And structured data normally resides in relational databases.

In the foreseeable future, you will have XML writers and XML readers for every database. However, you can write some code yourself to generate a comma-separated values file, as shown in Listing 14.5.

Listing 14.5 musicians.csv—A Comma-Separated Values File

```
1: Name,Instrument,NrOfRecordings
2: Joey Baron,drums,1
3: Bill Frisell,guitar,3
4: Don Byron,clarinet,2
5: Dave Douglas,trumpet,1
```

Or you can write some SQL code, as in Listing 14.6.

Listing 14.6 Some SQL Code

```
1: create table musicians (
2:     name text not null primary key,
3:     instrument text,
4:     nrofrecordings integer
5: );
6: insert into musicians values ('Joey Baron','drums',1);
7: insert into musicians values ('Bill Frisell','guitar',3);
8: insert into musicians values ('Don Byron','clarinet',2);
9: insert into musicians values ('Dave Douglas','trumpet',1);
```

Reporting

Processing the XML files can easily answer questions such as the following:

- How many musicians are there in the file?
- How many of them play guitar?
- What is the total number of recordings?

Now that you have turned your documents into real text databases, you can process them in many ways, generating output as needed.

Three Processing Paradigms

XML documents can be viewed as

- A special type of text file
- A sequence of events
- A hierarchy

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

An XML Document as a Text File

An XML file can be viewed as a text file, in which you find a mixture of the real data and markup language. You can find a lot of free tools on the market that excel in text manipulation using regular expressions: awk, grep, perl, and python.

If you want to know how many musicians you have in your musicians.xml file, you can use the following grep command:

```
grep -c “<musician>” musicians.xml
```

Here’s a breakdown of this command:

- -c is the option to print only the number of lines matched.
- Within quotes, the text (or pattern) to be matched, in this case the start tag of element musician.
- The file to be searched, for example, musicians.xml.

This example will return 4.

An XML Document as a Series of Events

Everyone knows how to work with a graphical user interface such as Windows, Mac OS, or Motif. In these environments, the OS captures all kinds of events: mouse movements and clicks, keyboard input, and so on. Subsequently, the OS sends

messages to the program indicating which event has occurred. It is the responsibility of the programmer of the application to code the responses to those events.

An analogy can be drawn for XML files. When you read an XML file in a sequential way, all kinds of events happen. You encounter the start of a document, you read start and end tags, you encounter comments and processing instructions, and so on. All of these can be looked at as data events.

It is the responsibility of the programmer to code the responses to these events. These responses are called *event handlers*.

An *event handler* is the code that is executed when an event has occurred.

For the file musicians.xml (Listing 14.1), an event-based processor will generate the events listed in Listing 14.7.

Listing 14.7 Events Generated by the musicians.xml File

```
 1: Start document
 2: Start element: musicians
 3: Start element: musician
 4: Start element: name
 5: Characters: Joey Baron
 6: End element: name
 7: Start element: instrument
 8: Characters: drums
 9: End element: instrument
10: Start element: NrOfRecordings
11: Characters: 1
12: End element: NrOfRecordings
13: End element: musician
14: ...
15: End element: musicians
16: End document
```

It is important to see that the traversal of the XML document happens in a hierarchically sequential, left-to-right fashion.

Consider the XML structure shown in Figure 14.1.

Figure 14.1 *The XML tree.*

Figure 14.2 indicates the order in which the different nodes of the tree are traversed.

Figure 14.2 *The order of traversal.*

During traversal, most of the event-based processors store information about already-traversed nodes.

In event-driven mode, Balise from the company AIS keeps track of element types and attributes for ancestor nodes up to the root and for the first-left siblings of these ancestor nodes as indicated in Figure 14.3.

Figure 14.3 *The context information that is kept.*

Figure 14.3 shows which other nodes are being tracked for node number 12.

The context information that is kept can differ from product to product. You can expect that at least the information on previous siblings within the same parent element and on all ancestors is stored.

Thanks to this cataloging of some context (however limited), you can write event handlers using this context:

- Start element: para, first after title.
  In this way, you can define a different response if your para element comes first after title.
- Start element: li, with an ancestor of ol and an attribute of type with a value of i.
  Here also, you can write a specific response if this condition occurs.

On the other hand, you cannot answer questions that require looking further in the data stream. For example:

- Is an element the last child element of its parent element?
  You can know this only after having processed the element, not when you receive the start element event.
- Does this element have an element in it that has an attribute with the name experience and the value firsttime?
  You can only know this after processing the element.

There are solutions, of course, but they require a second pass.

Advantages of this event-based processing are

- It’s simple.
- It works fast.
- It doesn’t consume a lot of memory.

Disadvantages are

- It’s impossible to look ahead.

Two implementations are mentioned here:

- Omnimark
- SAX

Omnimark is the market leader for doing heavy conversions in the SGML community. Recently it has been XML-enabled, and a free (although restricted) version called Omnimark LE is available on the Web at http://www.omnimark.com/develop/omle40/index.html.

Omnimark will be covered in more depth in Day 15, “Event-Driven Programming.”

SAX stands for a Simple API for XML. It came about after Peter Murray-Rust, one of the early adopters of XML, made a complaint on the XML developers’ mailing list. He said that when he wanted to change the parser coupled to his XML browser, JUMBO, he had to rewrite code because the APIs of the different parsers differed. The question raised was, “While waiting for the API defined by the W3C (DOM), can we agree on a simple event-based API?”

Previous | Table of Contents | Next

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

This was publicly discussed on the XML-DEV list, and lots of people contributed. It was David Megginson who wrote the SAX proposal, together with its implementation in Java.

SAX can be found at http://www.megginson.com/SAX/index.html. It's also subject of the study on Day 15.

XML as a Hierarchy/Tree

A tree-based processor translates the XML document into an internal tree structure and allows an application to navigate that tree.

In the case of your musicians.xml file, a possible resulting tree structure is shown in Figure 14.4.

Trees can be more or less extensive depending on whether attributes, entities, and so on need to be stored as separate nodes.

Once the tree is constructed in memory, it can be navigated. Please note that you have two passes:

- Pass 1: Parsing and tree-building
- Pass 2: The data processing itself

In this way, it becomes possible to answer your questions that require a look ahead:

- Is an element the last child element of its parent element?
- Does this element have an element below it that has an attribute with the name experience and the value firsttime?

Because you have access to the full document (the complete tree), you have access to all information

required.

Figure 14.4 *A tree starting from* musicians.xml.

The advantage of this approach is that it gives access to the whole document, so it's easy to look ahead. The disadvantages are that it's more difficult to first build a tree and then navigate it, it requires more memory, and it's slower, requiring two passes.

The World Wide Web consortium (W3C) developed a standard tree-based API for XML and HTML. It is called the Document Object Model (DOM) and is a W3C recommendation as of October 1, 1998. The specification can be found at http://www.w3.org/TR/ REC-DOM-Level-1/.

The DOM will be implemented in version 5 of both Internet Explorer and Mozilla (Netscape). It's covered extensively on Day 16, "Programming with the Document Object Model."

There is also software that brings you both worlds, mixing the event-driven approach and the tree-based approach. Balise, from the company AIS, is well known for this in the SGML community and includes a non-validating XML parser in its latest version. Unfortunately, no free version is available.

Summary

Today you saw many reasons why you would want to process your XML files. This processing can be done in different ways, depending on how you look at your XML data:

- Plain text files
- A series of events happening
- A tree that can be navigated and manipulated

In the next few days, you will explore these viewpoints and their implementations much more deeply.

Q&A

Q Is it easier to process XML than HTML?
A Yes, for two reasons.
The first is that you normally model your data better with XML than with HTML. You can better capture the hierarchical structure, the semantics, and the meta-data of your information without bothering too much with formatting.
The second reason is that XML files are well formed. You have a much better idea what comes in the data stream. HTML files, on the other hand, can take many forms and appearances.
Q Which processing approach do you recommend?
A It depends on the problem that you want to tackle.
For example, if you want to count the number of times the element tool is used in an XML document, it is not very efficient to first build a tree representation of your document and then traverse it. When you have to deal with complex structural objects such as tables that require a lot of look-ahead, the tree-based approach simplifies the processing task.
Also of importance is that the DOM, which is tree-based, is an official W3C recommendation now, and a lot of tools will support it as the standard API.

Exercise

See the following XML file:

```
 1: <?xml version="1.0"?>
 2: <memo>
 3:     <meta>
 4:         <from>P. Hermans</from>
 5:         <to>S. North</to>
 6:         <regarding>deadlines</regarding>
 7:     </meta>
 8:     <body>
 9:         <dear>Simon</dear>
10:         <p>I will <verystrong>not</verystrong> be able to finish
11: all chapters before leaving on 11..holidays.</p>
12:         <p>Please advise what to do.</p>
13:         <close>Paul</close>
14:     </body>
15: </memo>
```

Using this file:

- Write down the sequential list of events.
- Draw the tree structure.
- Indicate in which order the tree is navigated.
- For the last p element in the tree, indicate which other nodes still are accessible in the event-driven approach.
- For the body element in the tree, indicate which other nodes are still accessible in the tree-based approach.

Sams Teach Yourself XML in 21 Days
(Publisher: Macmillan Computer Publishing)
Author(s): Simon North
ISBN: 1575213966
Publication Date: 04/13/99

Part III

Chapter 15
Event-Driven Programming

In the previous chapter we outlined the differences between event-driven and tree-based processing of XML files.

Today we will dig deeper into the subject of event driven processing by using two event-driven processing environments:

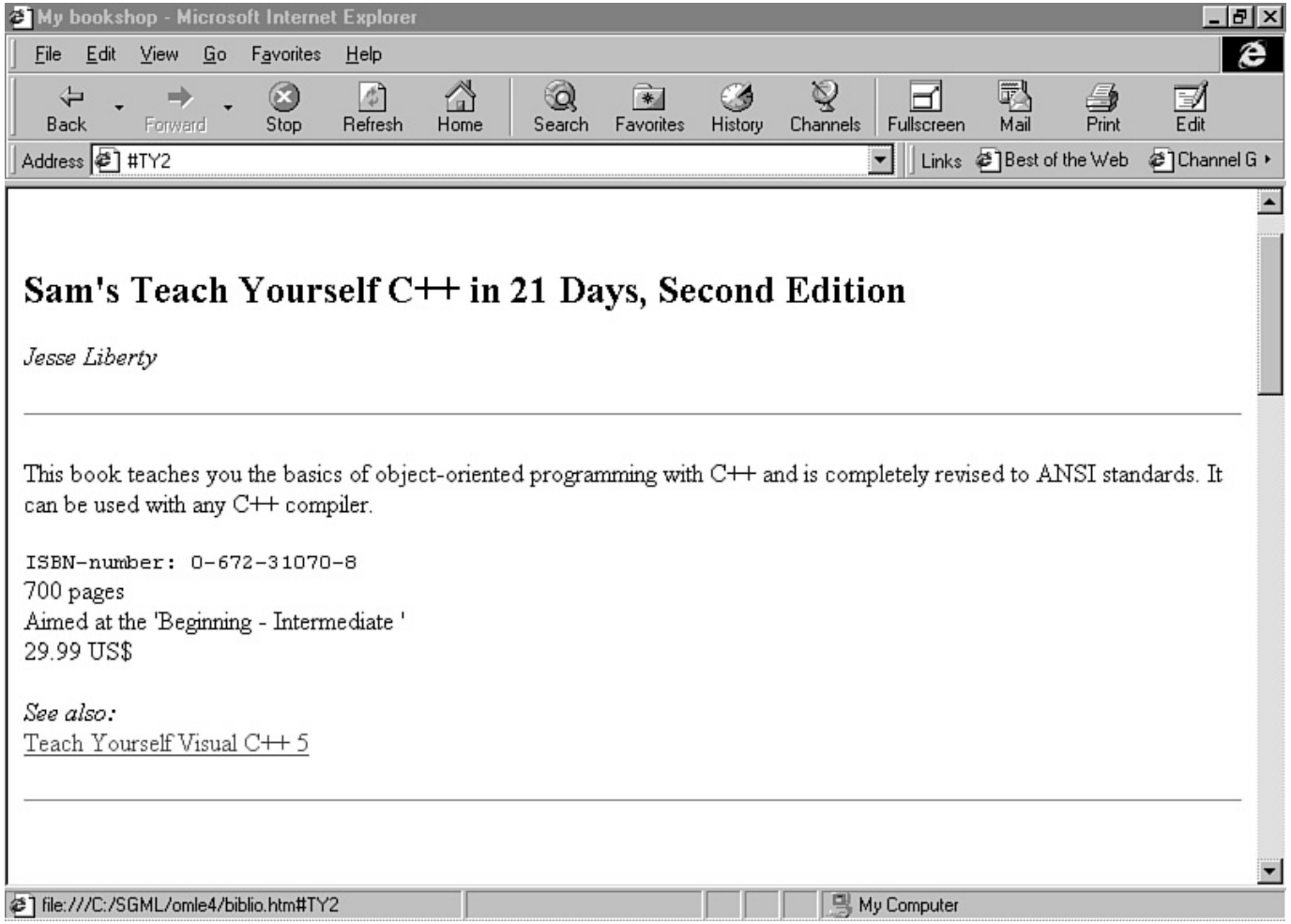
- Omnimark LE
- SAX

Omnimark LE

In this part we will cover the programming language Omnimark of Omnimark Technologies.

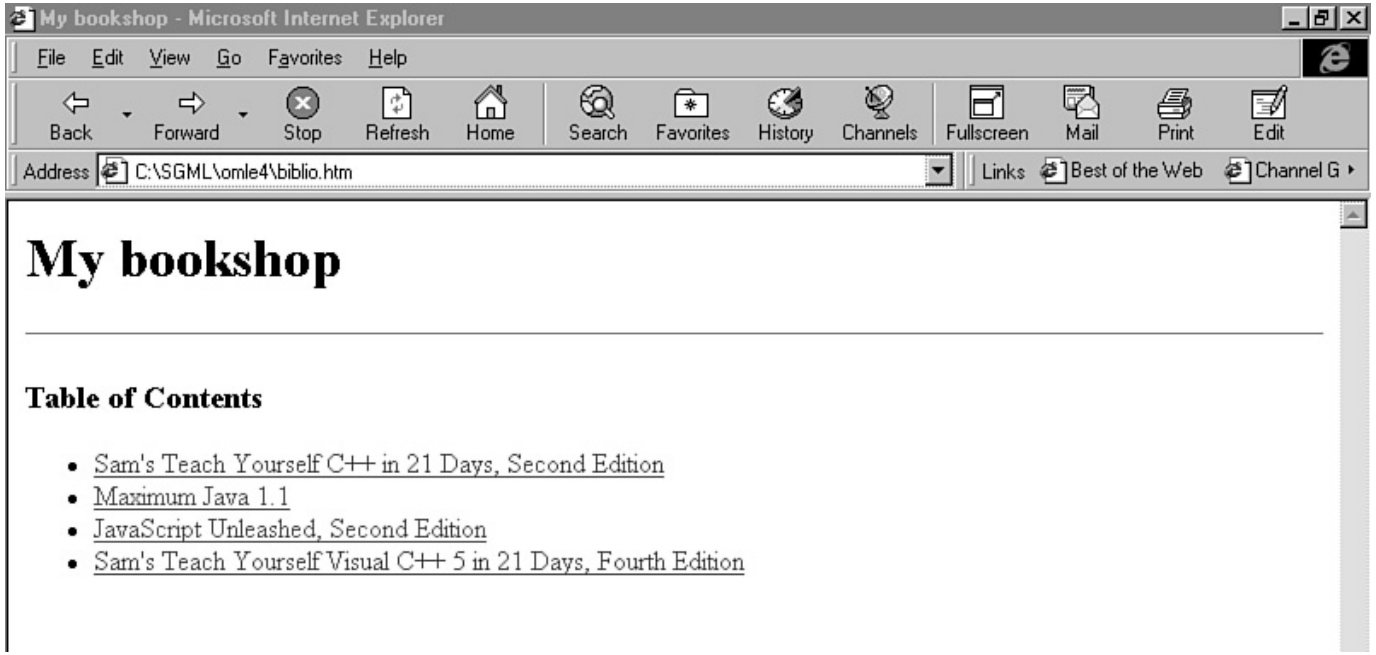
What Is Omnimark LE?

Omnimark LE is the light and free version of Omnimark, which is an event-driven programming language for processing data streams. The current version is 4.01 and is XML enabled.



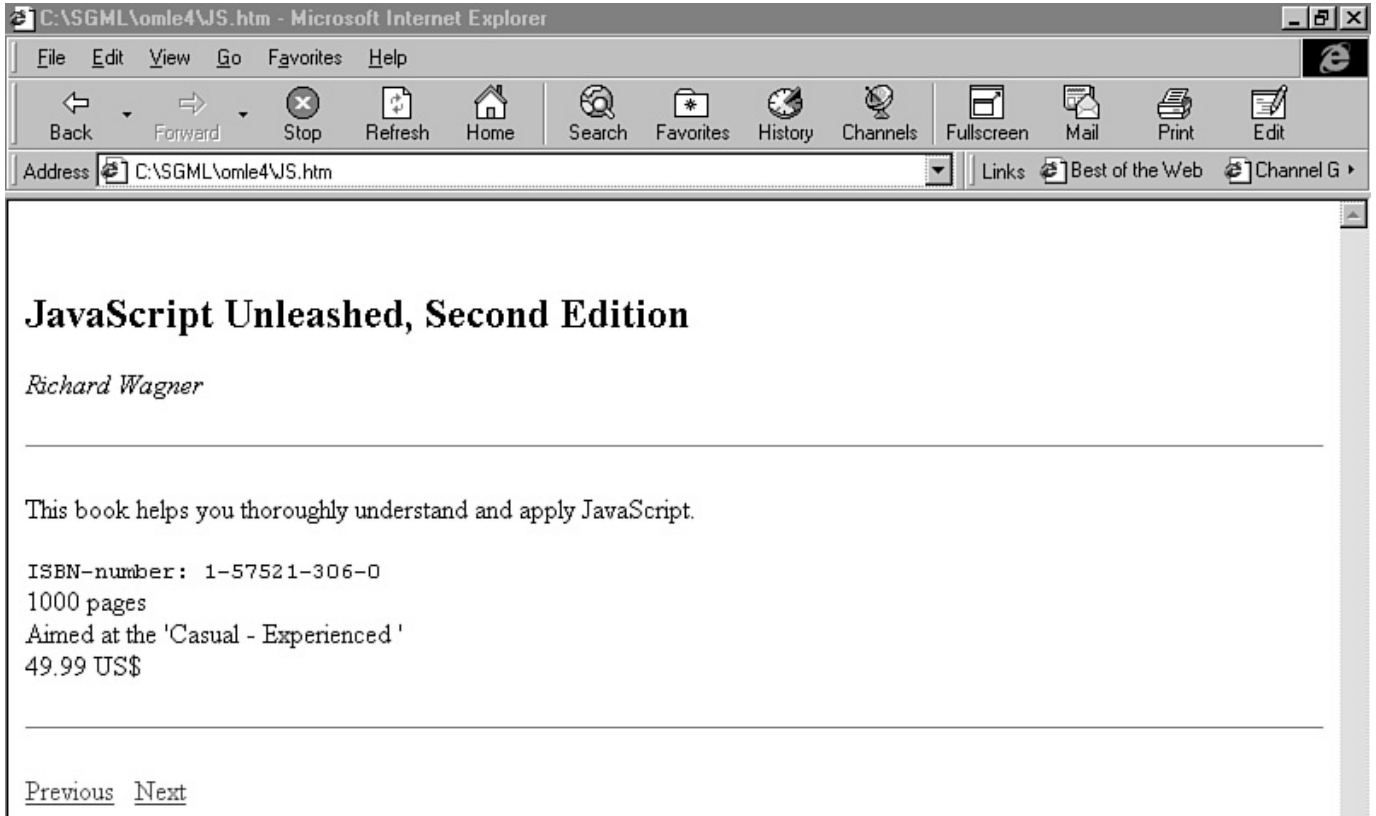
The Omnimark XML parser is a validating parser that requires a valid XML DTD to process XML documents.

This free version has a limitation that you cannot compile and execute programs that contain more than 200 countable actions in the program source. A *countable action* is a statement that Omnimark executes.



What counts as a countable action is explained at the following URL: http://www.omnimark.com/develop/omle40/sampcode.html.

Most of the time the free version is perfectly suited for writing small to medium conversions.



If you are running against the countable action limit, consider splitting your script into different parts and running them sequentially or applying the tips and techniques mentioned at http://www.sesha.com/omlette/.

Finding and Installing Omnimark LE

Omnimark LE can be found at the following URL: http://www.omnimark.com/develop/omle40/index.html. The file omle40.exe is 1955 Kb large.

To install, double-click this file. An InstallShield wizard will take over to guide you through the installation.

How Omnimark Works

In the event-driven paradigm, pieces of code are executed when certain events occur during the processing of the input document.

Therefore we need to formulate rules. Rules define and associate both the actions to be executed and the events that cause these actions to be triggered.

This is reflected in the structure of a rule, which has two parts:

- A rule header
- A rule body

The rule header is used to define the event and has a structure of its own. It defines the event and, optionally, additional conditions that need to be satisfied before the actions are fired.

In the rule body are the actions that will be executed.

An Omnimark script/program is a collection of such rules.

Running Omnimark LE

You run Omnimark from the command line, which takes the following form:

```
Omle –s script.xom input.xml –of output.htm
```

Where:

- omle invokes the program
- The -s parameter refers to the script to use (such as script.xom)
- The input file comes next, which in our case is an XML file (such as input.xml)
- -of specifies the file toward which output is written (such as output.htm)

Basic Events in the Omnimark Language

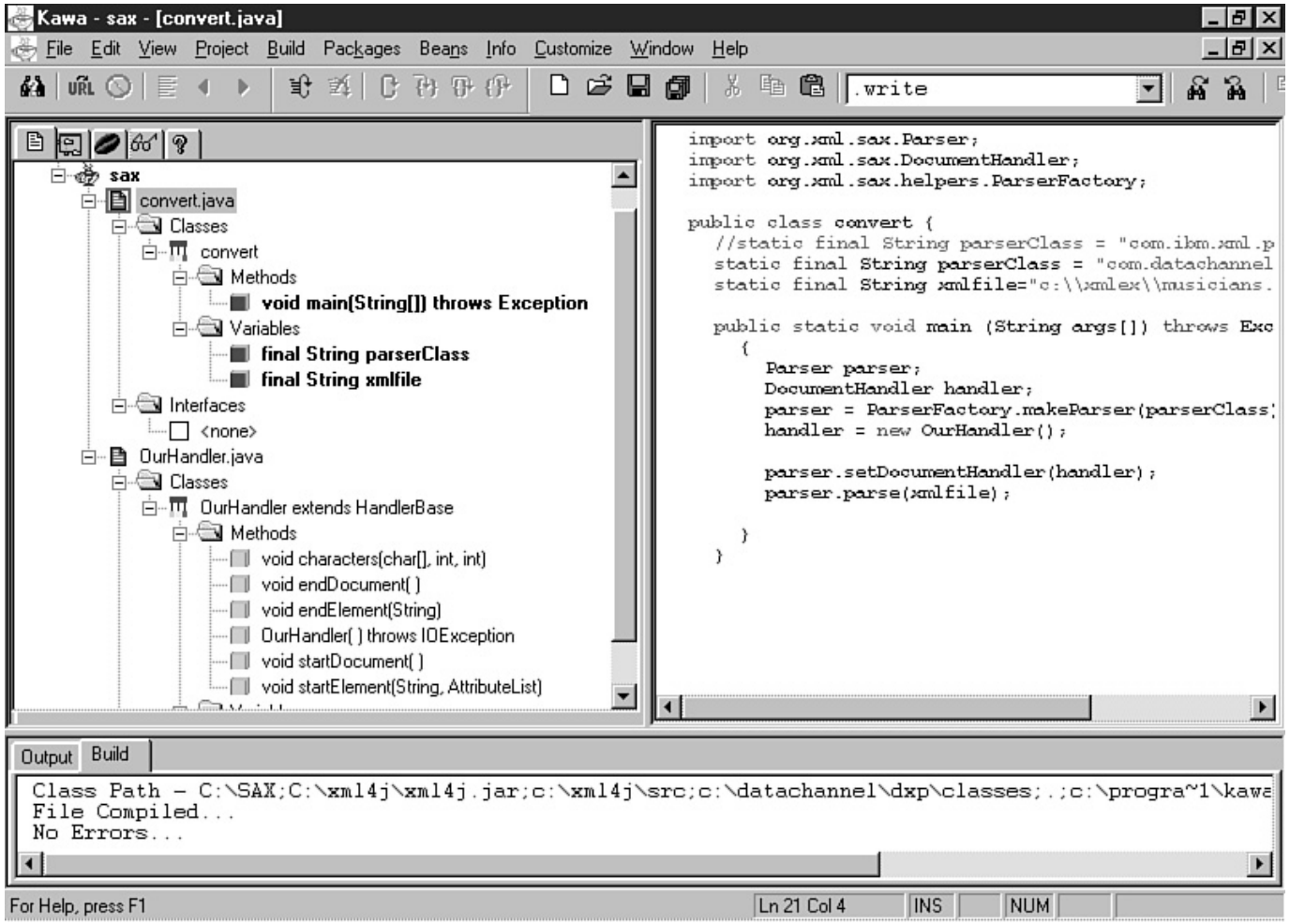
The following are some of the basic events Omnimark detects:

- document-start
- document-end
- element
- processing-instruction

document-start allows you to do processing and produce output before the beginning of an XML file, as shown in Listing 15.1.

Listing 15.1 A document-start Rule

```
1:  document-start  ;event
2:       output “<HTML>%n”     ;our action
3:       || “<HEAD>%n”
4:       || “<TITLE>Our first example</TITLE>%n”
5:       || “</HEAD>%n<BODY>”
```



The rule header on line 1 defines the event, which is document-start.

The rule body (lines 2–5) lists the actions to be executed. When the document starts (event), we write a string with HTML tags to the output (action).

%n stands for “end of line”
|| concatenates two strings
; starts a comment in your code

The result of this rule follows:

```
<HTML>
<HEAD>
<TITLE>Our first example</TITLE>
</HEAD>
<BODY>
```

document-end allows you to do processing and produce output after the end of an XML file, as shown in Listing 15.2.

Listing 15.2 A document-end Rule

```
1:  document-end ;event
2:       output “</BODY>%n</HTML>%n”
```

When the document ends (event), we write a string with HTML tags to the output (action).

The result of this rule follows:

```
</BODY>
</HTML>
```

Line 3 being empty is the result of the %n end-of-line after the HTML end tag on line 2 of Listing 15.2.

When an element occurs you have to deal with three things:

- The start of the element
- Its content
- The end of the element

When dealing with the content of an element, parsing ceases in Omnimark to give you the ability to decide how and when to treat the content. You need explicitly get parsing going again. This can be done by using the following:

- The parse continuation operator %c as used in Listing 15.3.
- suppress, which continues parsing but suppresses the output of the parsed content as shown in Listing 15.4.

Listing 15.3 Using the Parse Continuation Operator

```
1:  element musician   ; event, when the element musician is encountered
2:       output “<HR>%n” ; at the start of the element
3:       output “%c”     ; continue parsing
```

```
4:       output “<HR>%n” ; at the end of the element
```

---

or:

Listing 15.4 Using the Suppress Action

```
1:  element meta   ; event, when the element meta is encountered
2:      suppress ; continue parsing but suppress output
```

Previous | Table of Contents | Next