Creating Self-Describing Data

2nd Edition
Covers Schemas

Learning XML

O'REILLY®

Erik T. Ray

O'REILLY®

Learning XML

XML (Extensible Markup Language) has become ubiquitous. It is used as the base format for everything from configuration files to document management to messages sent between computers. While XML has evolved into a complex collection of specifications, a smaller core of XML remains the foundation. *Learning XML* explains this foundation and its capabilities succinctly and professionally, with references to real-life projects and cogent examples.

XML's intense hype wave has cooled, and developers are now using XML on a daily rather than experimental basis. This second edition of *Learning XML* explains how to work with XML in a wide variety of contexts. Anyone who needs to get close to XML and XML vocabularies—creating documents, defining schemas, transforming between formats, presenting information, or writing programs that deal directly with documents—will find this book a guide to the core technologies they need to build with XML.

Learning XML starts with coverage of XML's foundations and then explores technologies that address more specific needs. For writers producing XML documents, the book demystifies the process of creating documents with the appropriate structure and format. It also discusses stylesheets for viewing documents in the next generation of browsers and other devices. Designers will learn what parts of XML are most helpful to their team and will get started on creating schemas. For programmers, the book explains how to define and validate document structures and begin programming XML-oriented applications.

Topics covered include:

- The basic concepts and core syntax of XML
- Transforming XML using XPath and XSLT
- Using stylesheets, including CSS and XSL-FO, for formatting documents
- Document modeling with DTDs, W3C XML Schema, RELAX NG, and Schematron
- An introduction to internationalization using Unicode
- An introduction to programming with XML, using SAX, DOM, and pull

Learning XML provides a solid base on which you'll be able to build.

www.oreilly.com

US $39.95 CAN $61.95

ISBN-10: 0-596-00420-6

ISBN-13: 978-0-596-00420-0

9 780596 004200 53995

Learning XML

Other XML resources from O'Reilly

Related titles

XML in a Nutshell
XML Pocket Reference
XSLT
XSLT Cookbook
XML Schema
Web Services Essentials
SVG Essentials
Programming Web Services with SOAP
Programming Web Services with XML-RPC
XPath and XPointer
XSL-FO
Perl and XML
Python and XML
Java and XML
Java and XML Data Binding
Java and XSLT

XML Books Resource Center

xml.oreilly.com is a complete catalog of O'Reilly's books on XML and related technologies, including sample chapters and code examples.

XML.com helps you discover XML and learn how this Internet technology can solve real-world problems in information management and electronic commerce.

Conferences

O'Reilly & Associates brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

SECOND EDITION

Learning XML

Erik T. Ray

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Learning XML, Second Edition
by Erik T. Ray

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Simon St.Laurent
Production Editor:	Philip Dangler
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato

Printing History:

January 2001:	First Edition.
September 2003:	Second Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-00420-6
ISBN-13: 978-0-596-00420-0
[M] [02/07]

Table of Contents

Foreword

In 1976, two landers named Viking set down on Mars and turned their dish-shaped antennae toward earth. A few hours later, delighted scientists and engineers received the first pictures from the surface of another planet. Over the next few years, the Viking mission continued to collect thousands of images, instrument readings, and engineering data—enough to keep researchers busy for decades and making it one of the most successful science projects in history.

Of critical importance were the results of experiments designed to detect signs of life in the Martian soil. At the time, most researchers considered the readings conclusive evidence against the prospect of living organisms on Mars. A few, however, held that the readings could be interpreted in a more positive light. In the late 1990's, when researchers claimed to have found tiny fossils in a piece of Martian rock from Antarctica, they felt it was time to revisit the Viking experiment and asked NASA to republish the results.

NASA staff retrieved the microfilm from storage and found it to be largely intact and readable. They then began scanning the data, intending to publish it on CD-ROM. This seemed like a simple task at first—all they had to do was sort out the desired experiment data from the other information sent back from the space probes. But therein lay the problem: how could they extract specific pieces from a huge stream of munged information? All of the telemetry from the landers came in a single stream and was stored the same way. The soil sampling readings were a tiny fraction of information among countless megabytes of diagnostics, engineering data, and other stuff. It was like finding the proverbial needle in a haystack.

To comb through all this data and extract the particular information of interest would have been immensely expensive and time-consuming. It would require detailed knowledge of the probe's data communication specifications which were buried in documents that were tucked away in storage or perhaps only lived in the heads of a few engineers, long since retired. Someone might have to write software to split the mess into parallel streams of data from different instruments. All the information was there. It was just nearly useless without a lot of work to decipher it.

Luckily, none of this ever had to happen. Someone with a good deal of practical sense got in touch with the principal investigator of the soil sampling experiment. He happened to have a yellowing copy of the computer printout with analysis and digested results, ready for researchers to use. NASA only had to scan this information in and republish it as it was, without the dreaded interpretation of aging microfilm.

This story demonstrates that data is only as good as the way it's packaged. Information is a valuable asset, but its value depends on its longevity, flexibility, and accessibility. Can you get to your data easily? Is it clearly labeled? Can you repackage it in any form you need? Can you provide it to others without a hassle? These are the questions that the Extensible Markup Language (XML) was designed to answer.

Preface

Since its introduction in the late 90s, Extensible Markup Language (XML) has unleashed a torrent of new acronyms, standards, and rules that have left some in the Internet community wondering whether it is all really necessary. After all, HTML has been around for years and has fostered the creation of an entirely new economy and culture, so why change a good thing? XML isn't here to replace what's already on the Web, but to create a more solid and flexible foundation. It's an unprecedented effort by a consortium of organizations and companies to create an information framework for the 21st century that HTML only hinted at.

To understand the magnitude of this effort, we need to clear away some myths. First, in spite of its name, XML is not a markup language; rather, it's a toolkit for creating, shaping, and using markup languages. This fact also takes care of the second misconception, that XML will replace HTML. Actually, HTML is taking advantage of XML by becoming a cleaner version of itself, called XHTML. And that's just the beginning. XML will make it possible to create hundreds of new markup languages to cover every application and document type.

The standards process will figure prominently in the growth of this information revolution. XML itself is an attempt to rein in the uncontrolled development of competing technologies and proprietary languages that threatens to splinter the Web. XML creates a playground where structured information can play nicely with applications, maximizing accessibility without sacrificing richness of expression.

XML's enthusiastic acceptance by the Internet community has opened the door for many sister standards. XML's new playmates include stylesheets for display and transformation, strong methods for linking resources, tools for data manipulation and querying, error checking and structure enforcement tools, and a plethora of development environments. As a result of these new applications, XML is assured a long and fruitful career as the structured information toolkit of choice.

Of course, XML is still young, and many of its siblings aren't quite out of the playpen yet. Many XML specifications are mere speculation about how best to solve problems. Nevertheless, it's always good to get into the game as early as possible

rather than be taken by surprise later. If you're at all involved in information management or web development, then you need to know about XML.

This book is intended to give you a birds-eye view of the XML landscape that is now taking shape. To get the most out of this book, you should have some familiarity with structured markup, such as HTML or TEX, and with World Wide Web concepts such as hypertext linking and data representation. You don't need to be a developer to understand XML concepts, however. We'll concentrate on the theory and practice of document authoring without going into much detail about writing applications or acquiring software tools. The intricacies of programming for XML are left to other books, while the rapid changes in the industry ensure that we could never hope to keep up with the latest XML software. Nevertheless, the information presented here will give you a decent starting point for jumping in any direction you want to go with XML.

What's Inside

The book is organized into the following chapters:

Chapter 1, *Introduction*, is an overview of XML and some of its common uses. It's a springboard to the rest of the book, introducing the main concepts that will be explained in detail in following chapters.

Chapter 2, *Markup and Core Concepts*, describes the basic syntax of XML, laying the foundation for understanding XML applications and technologies.

Chapter 3, *Modeling Information*, delves into the concepts of data modeling, showing how to encode information with XML from simple software preferences to complex narrative documents.

Chapter 4, *Quality Control with Schemas*, shows how to use DTDs and various types of schemas to describe your document structures and validate documents against those descriptions.

Chapter 5, *Presentation Part I: CSS*, explores Cascading Style Sheets (CSS), a technology for presenting your XML documents in web browsers.

Chapter 6, *XPath and XPointer*, explains XPath, a vocabulary for addressing parts of XML documents that is useful both for transformations and programming, as well as its extensions into XPointer.

Chapter 7, *Transformation with XSLT*, applies XPath, demonstrating how to use Extensible Stylesheet Language Transformations (XSLT) to transform XML documents into other XML documents.

Chapter 8, *Presentation Part II: XSL-FO*, describes and demonstrates the use of Extensible Stylesheet Language Formatting Objects (XSL-FO) to create print representations of XML documents.

Chapter 9, *Internationalization*, examines internationalization issues with XML, including character encoding issues, language specification, and the use of MIME media type identifiers.

Chapter 10, *Programming*, describes various approaches to processing XML documents and creating programs around XML.

Appendix A, *Resources*, lists resources which may be useful in your further exploration of XML.

Appendix B, *A Taxonomy of Standards*, provides a list of the many standards at the heart of XML.

Style Conventions

Items appearing in this book are sometimes given a special appearance to set them apart from the regular text. Here's how they look:

Italic
: Used for commands, email addresses, URIs, filenames, emphasized text, first references to terms, and citations of books and articles.

`Constant width`
: Used for literals, constant values, code listings, and XML markup.

`Constant width italic`
: Used for replaceable parameter and variable names.

`Constant width bold`
: Used to highlight the portion of a code listing being discussed.

Examples

The examples from this book are freely downloadable from the book's web site at *http://www.oreilly.com/catalog/learnxml2*.

Comments and Questions

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/learnxml2

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

You can sign up for one or more of our mailing lists at:

http://elists.oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

You may also write to the author directly at *etr@ravelgrane.com*.

Acknowledgments

I wish to thank my reviewers, Jeni Tennison, Mike Fitzgerald, and Jeff Maggard, for their excellent advice and enthusiasm. Thanks to the open source community for making software development as fun as it is useful. And thanks to my friends and family for putting up with all this geeky stuff when I really should have been out working on my tan.

CHAPTER 1
Introduction

Anywhere there is information, you'll find XML, or at least hear it scratching at the door. XML has grown into a huge topic, inspiring many technologies and branching into new areas. So priority number one is to get a broad view, and ask the big questions, so that you can find your way through the dense jungle of standards and concepts.

A few questions come to mind. What is XML? We will attack this from different angles. It's more than the next generation of HTML. It's a general-purpose information storage system. It's a markup language toolkit. It's an open standard. It's a collection of standards. It's a lot of things, as you'll see.

Where did XML come from? It's good to have a historical perspective. You'll see how XML evolved out of earlier efforts like SGML, HTML, and the earliest presentational markup.

What can I do with XML? A practical question, again with several answers: you can store and retrieve data, ensure document integrity, format documents, and support many cultural localizations. And what can't I do with XML? You need to know about the limitations, as it may not be a good fit with your problem.

How do I get started? Without any hesitation, I hope. I'll describe the tools you need to get going with XML and test the examples in this book. From authoring, validating, checking well-formedness, transforming, formatting, and writing programs, you'll have a lot to play with.

So now let us dive into the big questions. By the end of this chapter, you should know enough to decide where to go from here. Future chapters will describe topics in more detail, such as core markup, quality control, style and presentation, programming interfaces, and internationalization.

What Is XML?

XML is a lot like the ubiquitous plastic containers of Tupperware®. There is really no better way to keep your food fresh than with those colorful, airtight little boxes. They come in different sizes and shapes so you can choose the one that fits best. They lock tight so you know nothing is leaking out and germs can't get in. You can tell items apart based on the container's color, or even scribble on it with magic marker. They're stackable and can be nested in larger containers (in case you want to take them with you on a picnic). Now, if you think of information as a precious commodity like food, then you can see the need for a containment system like Tupperware®.

An Information Container

XML contains, shapes, labels, structures, and protects information. It does this with symbols embedded in the text, called *markup*. Markup enhances the meaning of information in certain ways, identifying the parts and how they relate to each other. For example, when you read a newspaper, you can tell articles apart by their spacing and position on the page and the use of different fonts for titles and headings. Markup works in a similar way, except that instead of spaces and lines, it uses symbols.

Markup is important to electronic documents because they are processed by computer programs. If a document has no labels or boundaries, then a program will not know how to distinguish a piece of text from any other piece. Essentially, the program would have to work with the entire document as a unit, severely limiting the interesting things you can do with the content. A newspaper with no space between articles and only one text style would be a huge, uninteresting blob of text. You could probably figure out where one article ends and another starts, but it would be a lot of work. A computer program wouldn't be able to do even that, since it lacks all but the most rudimentary pattern-matching skills.

XML's markup divides a document into separate information containers called *elements*. Like Tupperware® containers, they seal up the data completely, label it, and provide a convenient package for computer processing. Like boxes, elements nest inside other elements. One big element may contain a whole bunch of elements, which in turn contain other elements, and so on down to the data. This creates an unambiguous hierarchical structure that preserves all kinds of ancillary information: sequence, ownership, position, description, association. An XML *document* consists of one outermost element that contains all the other elements, plus some optional administrative information at the top.

Example 1-1 is a typical XML document containing a short telegram. Take a moment to dissect it with your eyes and then we'll walk through it together.

Example 1-1. An XML document

```
<?xml version="1.0"?>
<!DOCTYPE telegram SYSTEM "/xml-resources/dtds/telegram.dtd">
<telegram pri="important">
  <to>Sarah Bellum</to>
  <from>Colonel Timeslip</from>
  <subject>Robot-sitting instructions</subject>
  <graphic fileref="figs/me.eps"/>
  <message>Thanks for watching my robot pal
    <name>Zonky</name> while I'm away.
    He needs to be recharged <emphasis>twice a
    day</emphasis> and if he starts to get cranky,
    give him a quart of oil. I'll be back soon,
    after I've tracked down that evil
    mastermind <villain>Dr. Indigo Riceway</villain>.
  </message>
</telegram>
```

Can you tell the difference between the markup and the data? The markup symbols are delineated by angle brackets (<>). `<to>` and `</villain>` are two such symbols, called *tags*. The data, or *content*, fills the space between these tags. As you get used to looking at XML, you'll use the tags as signposts to navigate visually through documents.

At the top of the document is the XML declaration, `<?xml version="1.0"?>`. This helps an XML-processing program identify the version of XML, and what kind of character encoding it has, helping the XML processor to get started on the document. It is optional, but a good thing to include in a document.

After that comes the document type declaration, containing a reference to a grammar-describing document, located on the system in the file */xml-resources/dtds/telegram.dtd*. This is known as a document type definition (DTD). `<!DOCTYPE...>` is one example of a type of markup called a *declaration*. Declarations are used to constrain grammar and declare pieces of text or resources to be included in the document. This line isn't required unless you want a parser to validate your document's structure against a set of rules you provide in the DTD.

Next, we see the `<telegram>` tag. This is the start of an element. We say that the element's name or type (not to be confused with a data type) is "telegram," or you could just call it a "telegram element." The end of the element is at the bottom and is represented by the tag `</telegram>` (note the slash at the beginning). This element contains all of the contents of the document. No wonder, then, that we call it the *document element*. (It is also sometimes called the *root element*.) Inside, you'll see more elements with start tags and end tags following a similar pattern.

There is one exception here, the empty tag `<graphic.../>`, which represents an empty element. Rather than containing data, this element references some other information that should be used in its place, in this case a graphic to be displayed. Empty elements do not mark boundaries around text and other elements the way

container elements do, but they still may convey positional information. For example, you might place the `graphic` inside a mixed-content element, such as the `message` element in the example, to place the graphic at that position in the text.

Every element that contains data has to have both a start tag and an end tag or the empty form used for `graphic`. (It's okay to use a start tag immediately followed by an end tag for an empty element; the empty tag is effectively an abbreviation of that.) The names in start and end tags have to match exactly, even down to the case of the letters. XML is very picky about details like this. This pickiness ensures that the structure is unambiguous and the data is airtight. If start tags or end tags were optional, the computer (or even a human reader) wouldn't know where one element ended and another began, causing problems with parsing.

From this example, you can see a pattern: some tags function as bookends, marking the beginning and ending of regions, while others mark a place in the text. Even the simple document here contains quite a lot of information:

Boundaries
: A piece of text starts in one place and ends in another. The tags `<telegram>` and `</telegram>` define the start and end of a collection of text and markup.

Roles
: What is a region of text doing in the document? Here, the tags `<name>` and `</name>` give an obvious purpose to the content of the element: a name, as opposed to any other kind of inline text such as a date or emphasis.

Positions
: Elements preserve the order of their contents, which is especially important in prose documents like this.

Containment
: The nesting of elements is taken into account by XML-processing software, which may treat content differently depending on where it appears. For example, a title might have a different font size depending on whether it's the title of a newspaper or an article.

Relationships
: A piece of text can be linked to a resource somewhere else. For instance, the tag `<graphic.../>` creates a relationship (link) between the XML fragment and a file named *me.eps*. The intent is to import the graphic data from the file and display it in this fragment.

An important XML term to understand is *document*. When you hear that word, you probably think of a sequence of words partitioned into paragraphs, sections, and chapters, comprising a human-readable record such as a book, article, or essay. But in XML, a document is even more general: it's the basic unit of XML information, composed of elements and other markup in an orderly package. It can contain text such as a story or article, but it doesn't have to. Instead, it might consist of a

database of numbers, or some abstract structure representing a molecule or equation. In fact, one of the most promising applications of XML is as a format for application-to-application data exchange. Keep in mind that an XML document can have a much wider definition than what you might think of as a traditional document. The following are short examples of documents.

The Mathematics Markup Language (MathML) encodes equations. A well-known equation among physicists is Newton's Law of Gravitation: $F = GMm / r^2$. The document in Example 1-2 represents that equation.

Example 1-2. A MathML document

```
<?xml version="1.0"?>
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mi>F</mi>
  <mo>=</mo>
  <mi>G</mi>
  <mo>&InvisibleTimes;</mo>
  <mfrac>
    <mrow>
      <mi>M</mi>
      <mo>&InvisibleTimes;</mo>
      <mi>m</mi>
    </mrow>
    <apply>
      <power/>
      <ci>r</ci>
      <cn>2</cn>
    </apply>
  </mfrac>
</math>
```

While one application might use this input to display the equation, another might use it to solve the equation with a series of values. That's a sign of XML's power.

You can also store graphics in XML documents. The Scalable Vector Graphics (SVG) language is used to draw resizable line art. The document in Example 1-3 defines a picture with three shapes (a rectangle, a circle, and a polygon).

Example 1-3. An SVG document

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg
   PUBLIC "-//W3C//DTD SVG 20001102//EN"
   "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">
<svg>
  <desc>Three shapes</desc>
  <rect fill="green" x="1cm" y="1cm" width="3cm" height="3cm"/>
  <circle fill="red" cx="3cm" cy="2cm" r="4cm"/>
  <polygon fill="blue" points="110,160 50,300 180,290"/>
</svg>
```

It's also worth noting that a document is not necessarily the same as a file. A *file* is a package of data treated as a contiguous unit by the computer's operating system. This is called a *physical* structure. An XML document can exist in one file or in many files, some of which may be on another system. It may not be in a file at all, but generated in a stream from a program. XML uses special markup to integrate the contents of different files to create a single entity, which we describe as a *logical* structure. By keeping a document independent of the restrictions of a file, XML facilitates a linked web of document parts which can reside anywhere.

That's XML markup in a nutshell. The whole of the next chapter is devoted to this topic. There, we'll go into deeper detail about the picky rules and describe some new components you haven't seen yet. You'll then be able to tear apart any XML document and know what all the pieces are for, and put together documents of your own.

A Markup Language Toolkit

Strictly speaking, XML is not a markup language. A language has a fixed vocabulary and grammar, but XML doesn't actually define any elements. Instead, it lays down a foundation of syntactic constraints on which you can build your own language. So a more apt description might be to call XML a markup language toolkit. When you need a markup language, you can build one quickly using XML's rules, and you'll be comfortable knowing that it will automatically be compatible with all the generic XML tools out there.

The telegram in Example 1-1 is marked up in a language I invented for fun. I chose a bunch of element names that I thought were important to represent a typical telegram, and a couple that were gratuitously silly, like `villain`. This is okay, because the language is for my use and I can do whatever I want with it. Perhaps I have something in mind for the `villain` element, like printing it in a different color to stand out. The point is that XML gives me the ability to tailor a markup language any way I want, which is a very powerful feature.

Well-formedness

Because XML doesn't have a predetermined vocabulary, it's possible to invent a markup language as you go along. Perhaps in a future telegram I want to identify a new kind of thing with an element I've never used before. Say I wrote to a friend inviting her to a party, and I enclosed the date in an element called, appropriately, `date`. Free-form XML, as I like to call it, is perfectly legal as long as it's well-formed. In other words, as long as you spell tags correctly, use both start tags and end tags, and obey all the other minimal rules, it's good XML.

Documents that follow the syntax rules of XML are *well-formed* XML documents. This piece of text would fail the test on three counts:

```
<equation<a < b<equation>
```

Can you find all the problems? First, the start tag is spelled incorrectly, because it has two left brackets instead of a left and a right. Second, there is a left bracket in the content of the element, which is illegal. Third, the end tag of the element is missing a slash. This is not well-formed XML. Any program that parses it should stop at the first error and refuse to have anything more to do with it.

Well-formedness is XML's "purity test." What does this get us? Compatibility. It allows you to write a program or a library of routines that know nothing about the incoming data except that it will be well-formed XML. An XML editor could be used to edit any XML document, a browser to view any document, and so on. Programs are more robust and less complex when the data is more consistent.

Validity

Some programs are not so general-purpose, however. They may perform complex operations on highly specific data. In this case, you may need to concretize your markup language so that a user doesn't slip in an unexpected element type and confuse the program. What you need is a formal *document model*. A document model is the blueprint for an *instance* of a markup language. It gives you an even stricter test than well-formedness, so you can say that Document X is not just well-formed XML, but it's also an instance of the Mathematics Markup Language, for example.

When a document instance matches a document model, we say that it is *valid*. You may hear it phrased as, "this is valid XHTML" or "valid SVG." The markup languages (e.g., XHTML and SVG) are *applications* of XML. Today, there are hundreds of XML applications for encoding everything from plays to chemical formulae. If you're in the market for a markup language, chances are you'll find one that meets your needs. If not, you can always make your own. That's the power of XML.

There are several ways to define a markup language formally. The two most common are document type definitions (DTDs) and schemas. Each has its strong points and weak points.

Document type definitions (DTDs)

DTDs are built into the XML 1.0 specification. They are usually separate documents that your document can refer to, although parts of DTDs can also reside inside your document. A DTD is a collection of rules, or *declarations*, describing elements and other markup objects. An element declaration adds a new element type to the vocabulary and defines its *content model*, what the element can contain and in which order. Any element type not declared in the DTD is illegal. Any element containing something not declared in the DTD is also illegal. The DTD doesn't restrict what kind of data can go inside elements, which is the primary flaw of this kind of document model.

Schemas

Schemas are a later invention, offering more flexibility and a way to specify patterns for data, which is absent from DTDs. For example, in a schema you could declare an element called `date` and then require that it contains a legal date in the format YYYY-MM-DD. With DTDs the best you could do is say whether the element can contain characters or elements. Unfortunately, there is a lot of controversy around schemas because different groups have put forth competing proposals. Perhaps there will always be different types of schemas, which is fine with me.

An Open Standard

As Andrew Tanenbaum, a famous networking researcher, once said, "The wonderful thing about standards is that there are so many of them." We've all felt a little bewildered by all the new standards that support the information infrastructure. But the truth is, standards work, and without them the world would be a much more confusing place. From Eli Whitney's interchangeable gun parts to standard railroad gauges, the industrial revolution couldn't have happened without them.

The best kind of standard is one that is open. An open standard is not owned by any single company or individual. It is designed and maintained based on input from the community to fit real needs, not to satisfy a marketing agenda. Therefore, it isn't subject to change without notice, nor is it tied to the fortune of a company that could disappear in the next market downturn. There are no licensing fees, nondisclosure agreements, partnerships, or intellectual property disputes. It's free, public, and completely transparent.

The Internet was largely built upon open standards. IP, TCP, ASCII, email, HTML, Telnet, FTP—they are all open, even if they were funded by private and government organizations. Developers like open standards because they can have a say in how they are designed. They are free to use what works for them, rather than be tied to a proprietary package. And history shows that they work remarkably well.

XML is an open standard. It was designed by a group of companies, organizations, and individuals called the World Wide Web Consortium (W3C). The current recommendation was published in 1998, with a second edition published in 2000, although a new version (1.1, which modifies the list of allowable characters) is currently in the draft stage. The specification is free to the public, on the web at *http://www.w3.org/TR/REC-xml*. As a recommendation, it isn't strictly binding. There is no certification process, but developers are motivated to comply as closely as possible to attract customers and community approval.

In one sense, a loosely binding recommendation is useful, in that standards enforcement takes time and resources that no one in the consortium wants to spend. It also allows developers to create their own extensions, or to make partially working implementations that do a pretty good job. The downside, however, is that there's no

guarantee anyone will do a really good job. For example, the Cascading Style Sheets standard has languished for years because browser manufacturers couldn't be bothered to fully implement it. Nevertheless, the standards process is generally a democratic and public-focused process, which is a Good Thing.

A Constellation of Standards

Many people agree that spending money is generally more fun than saving it. Sure, you can get a little thrill looking at your bank statement and seeing the dividend from the 3% interest on your savings account, but it isn't as exciting as buying a new plasma screen television. So it is with XML. It contains information like a safe holds money, but the real fun comes from using that information. Whether you're publishing an XHTML document to the Web or generating an image from SVG, the results are much more gratifying than staring at markup.

XML's extended family provides many ways to squeeze usefulness out of XML documents. They are extensions and applications of XML that build bridges to other formats or make it easier to work with data. All the names and acronyms may be a little overwhelming at first, but it's worth getting to know this growing family.

Let's look at these categories in more detail.

Core syntax
: These are the minimal standards required to understand XML. They include the core recommendation and its extension, Namespaces in XML. The latter piece allows you to classify markup in different groups. One use of this is to combine markup from different XML applications in the same document. The core syntax of XML will be covered thoroughly in Chapter 2.

Human documents
: This category has markup languages for documents you'll actually read, as opposed to raw data. XHTML, the XML-friendly upgrade to the Hypertext Markup Language, is used to encode web pages. DocBook is for technical manuals which are heavy in technical terms and complex structures like tables, lists, and sidebars. The Wireless Markup Language (WML) is somewhat like XHTML but specializes in packaging documents for tiny screens on cellular phones. We will discuss this narrative style of document in Chapter 3.

Modeling
: In this group are all the technologies developed to create models of documents that formalize a markup language and can be used to test document instances against standard grammars. These include DTDs (part of the core XML 1.0 recommendation), the W3C's XML Schema, RELAX NG, and Schematron, all of which will be covered in Chapter 4.

The W3C and the Standards Process

The W3C has taken on the role of the unofficial smithy of the Web. Founded in 1994 by a number of organizations and companies around the world with a vested interest in the Web, their long-term goal is to research and foster accessible and superior web technology with responsible application. They help to banish the chaos of competing, half-baked technologies by issuing technical documents and recommendations to software vendors and end users alike.

Every recommendation that goes up on the W3C's web site must endure a long, tortuous process of proposals and revisions before it's finally ratified by the organization's Advisory Committee. A recommendation begins as a project, or *activity*, when somebody sends the W3C Director a formal proposal called a *briefing package*. If approved, the activity gets its own working group with a charter to start development work. The group quickly nails down details such as filling leadership positions, creating the meeting schedule, and setting up necessary mailing lists and web pages.

At regular intervals, the group issues reports of its progress, posted to a publicly accessible web page. Such a *working draft* does not necessarily represent a finished work or consensus among the members, but is rather a progress report on the project. People in the community are welcome to review it and make comments. Developers start to implement parts of the proposed technology to test it out, finding problems in the process. Software vendors press for more features. All this feedback is important to ensure work is going in the right direction and nothing important has been left out particularly when the *last call working draft* is out.

The draft then becomes a *candidate recommendation*. At this stage, the working group members are satisfied that the ideas are essentially sound and no major changes will be needed. Experts will continue to weigh in with their insights, mostly addressing details and small mistakes. The deadline for comments finally arrives and the working group goes back to work, making revisions and changes.

Satisfied that the group has something valuable to contribute to the world, the Director takes the candidate recommendation and blesses it into a *proposed recommendation*. It must then survive the scrutiny of the Advisory Committee and perhaps be revised a little more before it finally graduates into a recommendation.

The whole process can take years to complete, and until the final recommendation is released, you shouldn't accept anything as gospel. Everything can change overnight as the next draft is posted, and many a developer has been burned by implementing the sketchy details in a working draft, only to find that the actual recommendation is a completely different beast. If you're an end user, you should also be careful. You may believe that the feature you need is coming, only to find it was cut from the feature list at the last minute.

It's a good idea to visit the W3C's web site (*http://www.w3.org*) every now and then. You'll find news and information about evolving standards, links to tutorials, and pointers to software tools. It's listed, along with some other favorite resources, in Appendix B.

Locating and linking

Data is only as useful as it is easy to access it. That's why there is a slew of protocols available for getting to data deep inside documents. XPath provides a language for specifying the path to a piece of data. XPointer and XLink use these paths to create a link from one document to another. XInclude imports files into a document. The XML Query Language (XQuery), still in drafts, creates an XML interface for non-XML data sources, essentially turning databases into XML documents. We will explore XPath and XPointer in Chapter 6.

Presentation

XML isn't very pretty to look at directly. If you want to make it presentable, you need to use a stylesheet. The two most popular are Cascading Style Sheets (CSS) and the Extensible Style Language (XSL). The former is very simple and fine for most online documents. The latter is highly detailed and better for print-quality documents. CSS is the topic of Chapter 5. We will take two chapters to talk about XSL: Chapter 7 and Chapter 8.

Media

Not all data is meant to be read. The Scalable Vector Graphics language (SVG) creates images and animations. The Synchronized Multimedia Integration Language (SMIL) scripts graphic, sound, and text events in a timeline-based multimedia presentation. VoiceML describes how to turn text into speech and script interactions with humans.

Science

Scientific applications have been early adopters of XML. The Chemical Markup Language (CML) represents molecules in XML, while MathML builds equations. Software turns instances of these markup languages into the nicely rendered visual representations that scientists are accustomed to viewing.

Resource description

With so many documents now online, we need ways to sort through them all to find just the information we need. Resource description is a way of summarizing and showing relationships between documents. The Resource Description Framework (RDF) is a language for describing resources.

Communication

XML is an excellent way for different systems to communicate with each other. Interhost system calls are being standardized through applications like XML-RPC, SOAP, WSDL, and UDDI. XML Signatures ensures security in identification by encoding unique, verifiable signatures for documents of any kind. SyncML is a way to transfer data from a personal computer to a smaller device like a cellular phone, giving you a fast and dependable way to update address lists and calendars.

Transformation
: Converting between one format and another is a necessary fact of life. If you've ever had to import a document from one software application into another, you know that it can sometimes be a messy task. Extensible Style Language Transformations (XSLT) can automate the task for you. It turns one form of XML into another in a process called *transformation*. It is essentially a programming language, but optimized for traversing and building XML trees. Transformation is the topic of Chapter 7.

Development
: When all else fails, you can always fall back on programming. Most programming languages have support for parsing and navigating XML. They frequently make use of two standard interfaces. The Simple API for XML (SAX) is very popular for its simplicity and efficiency. The Document Object Model (DOM) outlines an interface for moving around an object tree of a document for more complex processing. Programming with XML will be the last topic visited in this book, in Chapter 10.

This list demonstrates that XML has worked well as a basis for information exchange and application in a variety of fields.

Where Did XML Come From?

XML is the result of a long evolution of data packaging reaching back to the days of punched cards. It is useful to trace this path to see what mistakes and discoveries influenced the design decisions.

History

Early electronic formats were more concerned with describing how things should look (presentation) than with document structure and meaning. troff and TEX, two early formatting languages, did a fantastic job of formatting printed documents, but lacked any sense of structure. Consequently, documents were limited to being viewed on screen or printed as hard copies. You couldn't easily write programs to search for and siphon out information, cross-reference information electronically, or repurpose documents for different applications.

Generic coding, which uses descriptive tags rather than formatting codes, eventually solved this problem. The first organization to seriously explore this idea was the Graphic Communications Association (GCA). In the late 1960s, the GenCode project developed ways to encode different document types with generic tags and to assemble documents from multiple pieces.

The next major advance was Generalized Markup Language (GML), a project by IBM. GML's designers, Charles Goldfarb, Edward Mosher, and Raymond Lorie,* intended it as a solution to the problem of encoding documents for use with multiple information subsystems. Documents coded in this markup language could be edited, formatted, and searched by different programs because of its content-based tags. IBM, a huge publisher of technical manuals, has made extensive use of GML, proving the viability of generic coding.

Inspired by the success of GML, the American National Standards Institute (ANSI) Committee on Information Processing assembled a team, with Goldfarb as project leader, to develop a standard text-description language based upon GML. The GCA GenCode committee contributed their expertise as well. Throughout the late 1970s and early 1980s, the team published working drafts and eventually created a candidate for an industry standard (GCA 101-1983) called the Standard Generalized Markup Language (SGML). This was quickly adopted by both the U.S. Department of Defense and the U.S. Internal Revenue Service.

In the years that followed, SGML really began to take off. The International SGML Users' Group started meeting in the United Kingdom in 1985. Together with the GCA, they spread the gospel of SGML around Europe and North America. Extending SGML into broader realms, the Electronic Manuscript Project of the Association of American Publishers (AAP) fostered the use of SGML to encode general-purpose documents such as books and journals. The U.S. Department of Defense developed applications for SGML in its Computer-Aided Acquisition and Logistic Support (CALS) group, including a popular table formatting document type called CALS Tables. And then, capping off this successful start, the International Standards Organization (ISO) ratified a standard for SGML (ISO 8879:1986).

SGML was designed to be a flexible and all-encompassing coding scheme. Like XML, it is basically a toolkit for developing specialized markup languages. But SGML is much bigger than XML, with a more flexible syntax and lots of esoteric parameters. It's so flexible that software built to process it is complex and generally expensive, and its usefulness is limited to large organizations that can afford both the software and the cost of maintaining SGML environments.

The public revolution in generic coding came about in the early 1990s, when Hypertext Markup Language (HTML) was developed by Tim Berners-Lee and Anders Berglund, employees of the European particle physics lab CERN. CERN had been involved in the SGML effort since the early 1980s, when Berglund developed a publishing system to test SGML. Berners-Lee and Berglund created an SGML document type for hypertext documents that was compact and efficient. It was easy to write

* Cute fact: the acronym GML also happens to be the initials of the three inventors.

software for this markup language, and even easier to encode documents. HTML escaped from the lab and went on to take over the world.

However, HTML was in some ways a step backward. To achieve the simplicity necessary to be truly useful, some principles of generic coding had to be sacrificed. For example, one document type was used for all purposes, forcing people to overload tags rather than define specific-purpose tags. Second, many of the tags are purely presentational. The simplistic structure made it hard to tell where one section began and another ended. Many HTML-encoded documents today are so reliant on pure formatting that they can't be easily repurposed. Nevertheless, HTML was a brilliant step for the Web and a giant leap for markup languages, because it got the world interested in electronic documentation and linking.

To return to the ideals of generic coding, some people tried to adapt SGML for the Web—or rather, to adapt the Web to SGML. This proved too difficult. SGML was too big to squeeze into a little web browser. A smaller language that still retained the generality of SGML was required, and thus was born the Extensible Markup Language (XML).

The Goals of XML

Dissatisfied with the existing formats, a group of companies and organizations began work in the mid-1990s at the World Wide Web Consortium (W3C) on a markup language that combined the flexibility of SGML with the simplicity of HTML. Their philosophy in creating XML is embodied by several important tenets:

Form should follow function

In other words, markup languages need to fit their data snugly. Rather than invent a single, generic language to cover all document types (badly), let there be many languages, each specific to its data. Users can choose element names and decide how they should be arranged in a document. The result will better labeling of data, richer formatting possibilities, and enhanced searching capability.

A document should be unambiguous

A document should be marked up in such a way that there is only one way to interpret the names, order, and hierarchy of the elements. Consider this example from old-style HTML:

```
<html>
  <body>
    <p>Here is a paragraph.
    <p>And here is another.
  </body>
</html>
```

Before XML, this was acceptable markup. Every browser knows that the beginning of a `<p>` signals the end of an open `p` element preceding it as well as the

beginning of a new p element. This prior knowledge about a markup language is something we don't have in XML, where the number of possible elements is infinite. Therefore, it's an ambiguous situation. Look at this example; does the first element contain the other, or are they adjacent?

```
<flooby>an element
<flooby>another element
```

You can't possibly know, and neither can an XML parser. It could guess, but it might guess incorrectly. That's why XML rules about syntax are so strict. It reduces errors by making it more obvious when a document has mis-coded markup. It also reduces the complexity of software, since programs won't have to make an educated guess or try to fix syntax mistakes to recover. It may make it harder to write XML, since the user has to pay attention to details, but this is a small price to pay for robust performance.

Separate markup from presentation

For your document to have maximum flexibility for output format, you should strive to keep the style information out of the document and stored externally. Documents that rely on stylistic markup are difficult to repurpose or convert into new forms. For example, imagine a document that contains foreign phrases that are marked up to be italic, and emphatic phrases marked up the same way, like this:

```
<example>Goethe once said, <i>Lieben ist wie
Sauerkraut</i>. I <i>really</i> agree with that
statement.</example>
```

Now, if you wanted to make all emphatic phrases bold but leave foreign phrases italic, you'd have to manually change all the <i> tags that represent emphatic text. A better idea is to tag things based on their meaning, like this:

```
<example>Goethe once said, <foreignphrase>Lieben
ist wie Sauerkraut</foreignphrase>. I <emphasis>really</emphasis>
agree with that statement.</example>
```

Instead of being incorporated in the tag, the style information is defined in another place, a document called a *stylesheet*. Stylesheets map appearance settings to elements, acting as look-up tables for a formatting program. They make things much easier for you. You can tinker with the presentation in one place rather than doing a global search and replace operation in the XML. If you don't like one stylesheet, you can swap it for another. And you can use the same stylesheet for multiple documents.

Keeping style out of the document enhances your presentation possibilities, since you are not tied to a single style vocabulary. Because you can apply any number of stylesheets to your document, you can create different versions on the fly. The same document can be viewed on a desktop computer, printed, viewed on a handheld device, or even read aloud by a speech synthesizer, and you never have to touch the original document source—simply apply a different stylesheet. (It is of course possible to create presentation vocabularies in XML—XSL-FO is

an excellent example. In XSL-FO's case, however, its creators expect developers to create XSL-FO through XSLT stylesheets, not directly.)

Keep it simple

For XML to gain widespread acceptance, it had to be simple. People don't want to learn a complicated system just to author a document. XML 1.0 is intuitive, easy to read, and elegant. It allows you to devise your own markup language that conforms to some logical rules. It's a narrow subset of SGML, throwing out a lot of stuff that most people don't need.

Simplicity also benefits application development. If it's easy to write programs that process XML files, there will be more and cheaper programs available to the public. XML's rules are strict, but they make the burden of parsing and processing files more predictable and therefore much easier.

It should enforce maximum error checking

Some markup languages are so lenient about syntax that errors go undiscovered. When errors build up in a file, it no longer behaves the way you want it to: its appearance in a browser is unpredictable, information may be lost, and programs may act strangely and possibly crash when trying to open the file.

The XML specification says that a file is not well-formed unless it meets a set of minimum syntax requirements. Your XML parser is a faithful guard dog, keeping out errors that will affect your document. It checks the spelling of element names, makes sure the boundaries are airtight, tells you when an object is out of place, and reports broken links. You may carp about the strictness, and perhaps struggle to bring your document up to standard, but it will be worth it when you're done. The document's durability and usefulness will be assured.

It should be culture-agnostic

There's no good reason to confine markup in a narrow cultural space such as the Latin alphabet and English language. And yet, earlier markup languages do just that. Irked by this limitation, XML's designers selected Unicode as the character set, opening it up to thousands of letters, ideographs, and symbols.

What Can I Do with XML?

Let me tackle that question by sorting the kinds of problems for which you would use XML.

Store and Retrieve Data

Just about every software application needs to store some data. There are look-up tables, work files, preference settings, and so on. XML makes it very easy to do this. Say, for example, you've created a calendar program and you need a way to store holidays. You could hardcode them, of course, but that's kind of a hassle since you'd

have to recompile the program if you need to add to the list. So you decide to save this data in a separate file using XML. Example 1-4 shows how it might look.

Example 1-4. Calendar data file

```
<caldata>
  <holiday type="international">
    <name>New Year's Day</name>
    <date><month>January</month><day>1</day></date>
  </holiday>
  <holiday type="personal">
    <name>Erik's birthday</name>
    <date><month>April</month><day>23</day></date>
  </holiday>
  <holiday type="national">
    <name>Independence Day</name>
    <date><month>July</month><day>4</day></date>
  </holiday>
  <holiday type="religious">
    <name>Christmas</name>
    <date><month>December</month><day>25</day></date>
  </holiday>
</caldata>
```

Now all your program needs to do is read in the XML file and convert the markup into some convenient data structure using an *XML parser*. This software component reads and digests XML into a more usable form. There are lots of libraries that will do this, as well as standalone programs. Outputting XML is just as easy as reading it. Again, there are modules and libraries people have written that you can incorporate in any program.

XML is a very good choice for storing data in many cases. It's easy to parse and write, and it's open for users to edit themselves. Parsers have mechanisms to verify syntax and completeness, so you can protect your program from corrupted data. XML works best for small data files or for data that is not meant to be searched randomly. A novel is a good example of a document that is not randomly accessed (unless you are one of those people who peek at the ending of a novel before finishing), whereas a telephone directory *is* randomly accessed and therefore may not be the best choice to put in a single, enormous XML document.

If you want to store huge amounts of data and need to retrieve it quickly, you probably don't want to use XML. It's a sequential storage medium, meaning that any search would have to go through most of the document. A database program like Oracle or MySQL would scale much better, caching frequently used data and using a hash table to zero in on records with lightning speed.

Format Documents

I mentioned before that a large class of XML documents are narrative, meaning they are for human consumption. But we don't expect people to actually read text with XML markup. Rather, the XML must be processed to put the data in a presentable form. XML has a number of strategies and tools for turning the unappealing mishmash of marked-up plain text into eye-pleasing views suitable for web pages, magazines, or whatever you like.

Most XML markup languages focus on the task of how to organize information semantically. That is, they describe the data for what it is, not in terms of how it should look. Example 1-2 encodes a mathematical equation, but it does not look like something you'd write on a blackboard or see in a textbook. How you get from the raw data to the finished product is called *formatting*.

CSS

There are a number of different strategies for formatting. The simplest is to apply a Cascading Style Sheet (CSS) to it. This is a separate document (not itself XML) that contains mappings from element names to presentation details (font style, color, margins, and so on). A formatting XML processor such as a web browser, reads the XML data file and the stylesheet, then produces a formatted page by applying the stylesheet's instructions to each element. Example 1-5 shows a typical example of a CSS stylesheet.

Example 1-5. A CSS stylesheet

```
telegram {
  display: block;
  background-color: tan;
  color: black;
  font-family: monospace;
  padding: 1em;
}
message {
  display: block;
  margin: .5em;
  padding: .5em;
  border: thin solid brown;
  background-color: wheat;
  whitespace: normal;
}
to:before {
  display: block;
  color: black;
  content: "To: ";
}
from:before {
  display: block;
  color: black;
```

Example 1-5. A CSS stylesheet (continued)

```
  content: "From: ";
}
subject:before {
  color: black;
  content: "Subject: ";
}
to, from, subject {
  display: block;
  color: blue;
  font-size: large;
}
emphasis {
  font-style: italic;
}
name {
  font-weight: bold;
}
villain {
  color: red;
  font-weight: bold;
}
```

To apply this stylesheet, you need to add a special instruction to the source document. It looks like this:

```
<?xml-stylesheet type="text/css" href="ex2_memo.css"?>
```

This is a *processing instruction*, not an element. It will be ignored by any XML processing software that doesn't handle CSS stylesheets.

To see the result, you can open the document in a web browser that accepts XML and can format with CSS. Figure 1-1 shows a screenshot of how it looks in Safari version 1.0 for Mac OS X.

CSS is limited to cases where the output text will be in the same order as the input data. It would not be so useful if you wanted to show only an excerpt of the data, or if you wanted it to appear in a different order from the data. For example, suppose you collected a lot of phone numbers in an XML file and then wanted to generate a telephone directory from that. With CSS, there is no way to sort the listings in alphabetical order, so you'd have to do the sorting in the XML file first.

Transformation to presentational formats

A more powerful technique is to transform the XML. *Transformation* is a process that breaks apart an XML document and builds a new one. The new document may or may not use the same markup language (in fact, XML is only one option; you can transform XML into any kind of text). With transformation, you can sort elements, throw out parts you don't want, and even generate new data such as headers and footers for pages. Transformation in XML is typically done with the language XSLT,

To:
Sarah Bellum
From:
Colonel Timeslip
Subject:
Robot-sitting instructions

Thanks for watching my robot pal **Zonky** while I'm away. He needs to be recharged *twice a day* and if he starts to get cranky, give him a quart of oil. I'll be back soon, after I've tracked down that evil mastermind Dr. Indigo Riceway.

Figure 1-1. Screenshot of a CSS-formatted document

essentially a programming language optimized for transforming XML. It requires a transformation instruction which happens to be called a stylesheet (not to be confused with a CSS stylesheet). The process looks like the diagram in Figure 1-2.

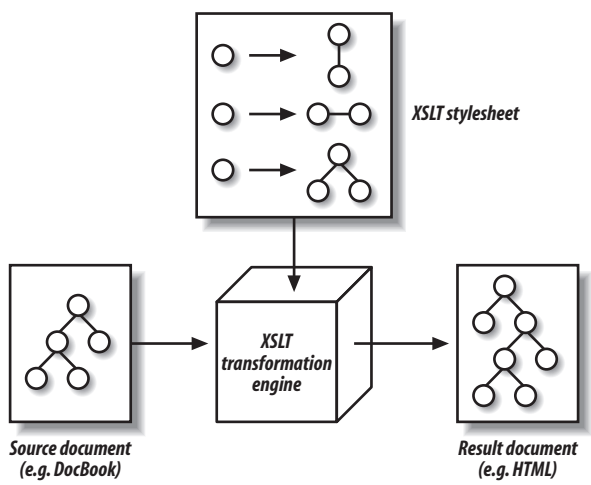

Figure 1-2. The transformation process

A popular use of transformations is to change a non-presentation XML data file into a format that combines data with presentational information. Typically, this format will throw away semantic information in favor of device-specific and highly presentational descriptions. For example, elements that distinguish between filenames and emphasized text would be replaced with tags that turn on italic formatting. Once you

lose the semantic information, it is much harder to transform the document back to the original data-specific format. That is okay, because what we get from presentational formats is the ability to render a pleasing view on screen or printed page.

There are many presentational formats. Public domain varieties include the venerable troff, which dates back to the first Unix system, and TEX, which is still popular in universities. Adobe's PostScript and PDF and Microsoft's Rich Text Format (RTF) are also good candidates for presentational formats. There are even some XML formats that can be included in this domain. XHTML is rather generic and presentational for narrative documents. SVG, a graphics description language, is another format you could transform to from a more semantic language.

Example 1-6 shows an XSLT stylesheet that changes any `telegram` document into HTML. Notice that XSLT is itself an XML application, using namespaces (an XML syntax for grouping elements by adding a name prefix) to distinguish between XSLT commands and the markup to be output. For every element type in the source document's markup language, there is a corresponding rule in the stylesheet describing how to handle it. I don't expect you to understand this code right now. There is a whole chapter on XSLT (Chapter 7) after which it will make more sense to you.

Example 1-6. An XSLT script for telegram documents

```
<xsl:transform
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="telegram">
  <html>
    <head><title>telegram</title></head>
    <body>
      <div style="background-color: wheat; padding=1em; ">
        <h1>telegram</h1>
        <xsl:apply-templates/>
      </div>
    </body>
  </html>
</xsl:template>

<xsl:template match="from">
  <h2><xsl:text>from: </xsl:text><xsl:apply-templates/></h2>
</xsl:template>

<xsl:template match="to">
  <h2><xsl:text>to: </xsl:text><xsl:apply-templates/></h2>
</xsl:template>

<xsl:template match="subject">
  <h2><xsl:text>subj: </xsl:text><xsl:apply-templates/></h2>
</xsl:template>

<xsl:template match="message">
```

Example 1-6. An XSLT script for telegram documents (continued)

```
  <blockquote>
    <font style="font-family: monospace">
      <xsl:apply-templates/>
    </font>
  </blockquote>
</xsl:template>

<xsl:template match="emphasis">
  <i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="name">
  <font color="blue"><xsl:apply-templates/></font>
</xsl:template>

<xsl:template match="villain">
  <font color="red"><xsl:apply-templates/></font>
</xsl:template>

<xsl:template match="graphic">
  <img width="100">
    <xsl:attribute name="src">
      <xsl:value-of select="@fileref"/>
    </xsl:attribute>
  </img>
</xsl:template>

</xsl:transform>
```

When applied against the document in Example 1-1, this script produces the following HTML. Figure 1-3 shows how it looks in a browser.

```
<html>
<head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
<title>telegram</title>
</head>
<body><div style="background-color: wheat; padding=1em; ">
<h1>telegram</h1>
  <h2>to: Sarah Bellum</h2>
  <h2>from: Colonel Timeslip</h2>
  <h2>subj: Robot-sitting instructions</h2>
  <blockquote><font style="font-family: monospace">Thanks for watching
  my robot pal
    <font color="blue">Zonky</font> while I'm away.
    He needs to be recharged <i>twice a
    day</i> and if he starts to get cranky,
    give him a quart of oil. I'll be back soon,
    after I've tracked down that evil
    mastermind <font color="red">Dr. Indigo Riceway</font>.
  </font></blockquote>
</div></body>
</html>
```

Figure 1-3. Transformation result

Transformation and formatting objects

Transforming XML into HTML is fine for online viewing. It is not so good for print media, however. HTML was never designed to handle the complex formatting of printed documents, with headers and footers, multiple columns, and page breaks. For that, you would want to transform into a richer format such as PDF. A direct transformation into PDF is not so easy to do, however. It requires extensive knowledge of the PDF specification which is huge and difficult, and much of the content is compressed.

A better solution is to transform your XML into an intermediate format, one that is generic and easy for humans to understand. This is XSL-FO, the style language for formatting objects. A *formatting object* is an abstract representation for a portion of a formatted page. You use XSLT to map elements to formatting objects, and an XSL formatter turns the formatting objects into pages, paragraphs, graphics, and other presentational components. The process is illustrated in Figure 1-4.

The source document on the left is first transformed, using an XSLT stylesheet and XSLT processor, into a formatting object tree using XSLT. This intermediate file is then fed into the XSL formatter which processes it into a presentational format, such as PDF. The beauty of this system is that it is modular. You can use any compliant XSLT processor and XSL formatter. You don't need to know anything about the

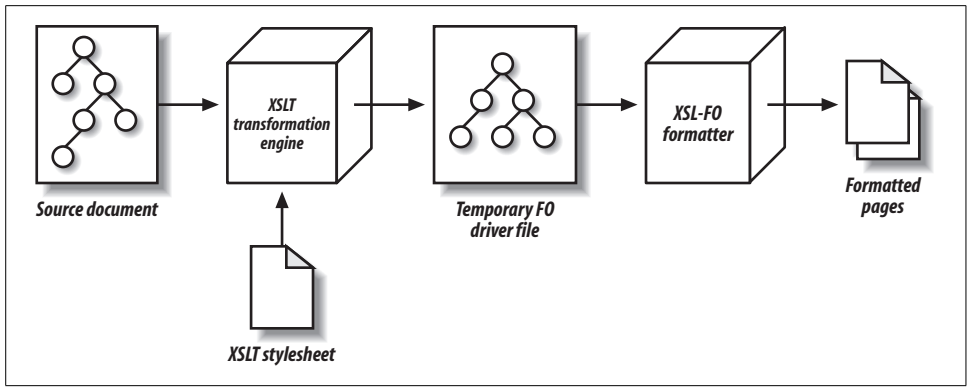

Figure 1-4. How XSL works

presentational format because XSL is so generic, describing layout and style attributes in the most declarative form. I will describe XSL in more detail in Chapter 8.

Programming solutions

Finally, if stylesheets do not fit the bill, which may be the case if your source data is just too raw for direct transformation, then you may find a programming solution to be to your liking. Although XSLT has much to offer in transformation, it tends to be rather weak in some areas, such as processing character data. I often find that, despite my best efforts to stay inside the XSLT paradigm, I sometimes have to resort to writing a program that preprocesses my XML data before a transformation. Or I may have to write a program that does the whole processing from source to presentational format. That option is always available, and we will see it in detail in Chapter 10.

Ensure Data Integrity

Trust is important for data—trust that it hasn't been corrupted, truncated, mistyped or left incomplete. Broken documents can confuse software, format as gibberish, and result in erroneous calculations. Documents submitted for publishing need to be complete and use only the markup that you specify. Transmitting and converting documents always entails risk that some information may be lost.

XML gives you the ability to guarantee a minimal level of trust in data. There are several mechanisms. First, there is well-formedness. Every XML parser is required to report syntax errors in markup. Missing tags, malformed tags, illegal characters, and other problems should be immediately reported to you. Consider this simple document with a few errors in it:

```
<announcement<
  <TEXT>Hello, world! I'm using XML & it's a lot of fun.</Text>
</anouncement>
```

When I run an XML well-formedness checker on it, here is what I get:

```
> xwf t.xml
t.xml:2: error: xmlParseEntityRef: no name
  <TEXT>Hello, world! I'm using XML & it's a lot of fun.</Text>
                                     ^
t.xml:2: error: Opening and ending tag mismatch: TEXT and Text
  <TEXT>Hello, world! I'm using XML & it's a lot of fun.</Text>
                                                          ^
t.xml:3: error: Opening and ending tag mismatch: announcement and
anouncement
</anouncement>
             ^
```

It caught two mismatched tags and an illegal character. And not only did it tell me what was wrong, it showed me where the errors were, so I can go back and correct them more easily. Checking if a document is well-formed can pick up a lot of problems:

- Mismatched tags, a common occurrence if you are typing in the XML by hand. The start and end tags have to match exactly in case and spelling.
- Truncated documents, which would be missing at least part of the outermost document (both start and end tags must be present).
- Illegal characters, including reserved markup delimiters like `<`, `>`, and `&`. There is a special syntax for complex or reserved characters which looks like `<` for `<`. If any part of that is missing, the parser will get suspicious. Parsers should also warn you if characters in a particular encoding are not correctly formed, which may indicate that the document was altered in a recent transmission. For example, transferring a file through FTP as ASCII text can sometimes strip out the high bit characters.

The well-formedness check has its limits. The parser doesn't know if you are using the right elements in the right places. For example, you might have an XHTML document with a `p` element inside the `head`, which is illegal. To catch this kind of problem, you need to test if the document is a valid instance of XHTML. The tool for this is a *validating parser*.

A validating parser works by comparing a document against a set of rules called a *document model*. One kind of document model is a *document type definition* (DTD). It declares all the elements that are allowed in a document and describes in detail what kind of elements they can contain. Example 1-7 is a small DTD for telegrams.

Example 1-7. A telegram DTD

```
<!ELEMENT telegram (from,to,subject,graphic?,message)>
<!ATTLIST telegram pri CDATA #IMPLIED>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT graphic EMPTY>
```

Example 1-7. A telegram DTD (continued)

```
<!ATTLIST graphic fileref CDATA #REQUIRED>
<!ELEMENT message (#PCDATA|emphasis|name|villain)*>
<!ELEMENT emphasis (#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

Before submitting the telegram document to a parser, I need to add this line to the top:

```
<!DOCTYPE telegram SYSTEM "/location/of/dtd">
```

Where "/location..." is the path to the DTD file on my system. Now I can run a validating parser on the telegram document. Here's the output I get:

```
> xval ex1_memo.xml
ex1_memo.xml:13: validity error: No declaration for element villain
    mastermind <villain>Dr. Indigo Riceway</villain>.
                                           ^
ex1_memo.xml:15: validity error: Element telegram content doesn't
follow the DTD
</telegram>
          ^
```

Oops! I forgot to declare the `villain` element, so I'm not allowed to use it in a telegram. No problem; it's easy to add new elements. This shows how you can detect problems with structure and grammar in a document.

The most important benefit to using a DTD is that it allows you to enforce and formalize a markup language. You can make your DTD public by posting it on the web, which is what organizations like the W3C do. For instance, you can look at the DTD for "strict" XHTML version 1.0 at *http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd*. It's a compact and portable specification, though a little dense to read.

One limitation of DTDs is that they don't do much checking of text content. You can declare an element to contain text (called PCDATA in XML), or not, and that's as far as you can go. You could not check whether an element that should be filled out is empty, or if it follows the wrong pattern. Say, for example, I wanted to make sure that the `to` element in the telegram isn't empty, so I have at least someone to give it to. With a DTD, there is no way to test that.

An alternative document modeling scheme provides the solution. XML Schemas provide much more detailed control over a document, including the ability to compare text with a pattern you define. Example 1-8 shows a schema that will test a telegram for completely filled-out elements.

Example 1-8. A schema for telegrams

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="telegram" type="telegramtype" />

  <xs:complexType name="telegramtype">
```