XML in a Nutshell, 2nd Edition

By Elliotte Rusty Harold, W. Scott Means

Publisher	: O'Reilly
Pub Date	: June 2002
ISBN	: 0-596-00292-0
Pages	: 634

This powerful new edition provides developers with a comprehensive guide to the rapidly evolving XML space. Serious users of XML will find topics on just about everything they need, from fundamental syntax rules, to details of DTD and XML Schema creation, to XSLT transformations, to APIs used for processing XML documents. Simply put, this is the only reference of its kind among XML books.

EEn

777

Chapters 1, 2 and 3

Chapter 1. Introducing XML

XML, the Extensible Markup Language, is a W3C-endorsed standard for document markup. It defines a generic syntax used to mark up data with simple, human-readable tags. It provides a standard format for computer documents. This format is flexible enough to be customized for domains as diverse as web sites, electronic data interchange, vector graphics, genealogy, real-estate listings, object serialization, remote procedure calls, voice-mail systems, and more.

You can write your own programs that interact with, massage, and manipulate the data in XML documents. If you do, you'll have access to a wide range of free libraries in a variety of languages that can read and write XML so that you can focus on the unique needs of your program. Or you can use off-the-shelf software, such as web browsers and text editors, to work with XML documents. Some tools are able to work with any XML document. Others are customized to support a particular XML application in a particular domain, such as vector graphics, and may not be of much use outside that domain. But in all cases, the same underlying syntax is used, even if it's deliberately hidden by the more user-friendly tools or restricted to a single application.

1.1 The Benefits of XML

XML is a metamarkup language for text documents. Data is included in XML documents as strings of text. The data is surrounded by text markup that describes the data. XML's basic unit of data and markup is called an element. The XML specification defines the exact syntax this markup must follow: how elements are delimited by tags, what a tag looks like, what names are acceptable for elements, where attributes are placed, and so forth. Superficially, the markup in an XML document looks a lot like the markup in an HTML document, but there are some crucial differences.

Most importantly, XML is a metamarkup language. That means it doesn't have a fixed set of tags and elements that are supposed to work for everybody in all areas of interest for all time. Any attempt to create a finite set of such tags is doomed to failure. Instead, XML allows developers and writers to define the elements they need as they need them. Chemists can use elements that describe molecules, atoms, bonds, reactions, and other items encountered in chemistry. Real-estate agents can use elements that describe apartments, rents, commissions, locations, and other items needed for real estate. Musicians can use elements that describe quarter notes, half notes, G-clefs, lyrics, and other objects common in music. The X in XML stands for Extensible. Extensible means that the language can be extended and adapted to meet many different needs.

Although XML is quite flexible in the elements it allows to be defined, it is quite strict in many other respects. It provides a grammar for XML documents that says where tags may be placed, what they must look like, which element names are legal, how attributes are attached to elements, and so forth. This grammar is specific enough to allow the development of XML parsers that can read any XML document. Documents that satisfy this grammar are said to be well-formed. Documents that are not well-formed are not allowed, any more than a C program that contains a syntax error is allowed. XML processors will reject documents that contain well-formedness errors.

For reasons of interoperability, individuals or organizations may agree to use only certain tags. These tag sets are called XML applications. An XML application is not a software application that uses XML, such as Mozilla or Microsoft Word. Rather, it's an application of XML in a particular domain like vector graphics or cooking.

The markup in an XML document describes the structure of the document. It lets you see which elements are associated with which other elements. In a well-designed XML document, the markup also describes the document's semantics. For instance, the markup can indicate that an element is a date or a person or a bar code. In well-designed XML applications, the markup says nothing about how the document should be displayed. That is, it does not say that an element is bold or italicized or a list item. XML is a structural and semantic markup language, not a presentation language.[1]

The markup permitted in a particular XML application can be documented in a schema. Particular document instances can be compared to the schema. Documents that match the schema are said to be valid. Documents that do not match are invalid. Validity depends on the schema. That is, whether a document is valid or invalid depends on which schema you compare it to. Not all documents need to be valid. For many purposes it is enough that the document merely be well-formed.

There are many different XML schema languages, with different levels of expressivity. The most broadly supported schema language and the only one defined by the XML 1.0 specification itself is the document type definition (DTD). A DTD lists all the legal markup and specifies where and how it may be included in a document. DTDs are optional in XML. On the other hand, DTDs may not always be enough. The DTD syntax is quite limited and does not allow you to make many useful statements such as "This element contains a number" or "This string of text is a date between 1974 and 2032." The W3C XML Schema Language (which sometimes goes by the misleadingly generic label schemas) does allow you to express constraints of this nature. Besides these two, there are many other schema languages from which to choose, including RELAX NG, Schematron, Hook, and Examplotron, and this is hardly an exhaustive list.

All current schema languages are purely declarative. However, there are always some constraints that cannot be expressed in anything less than a Turing complete programming language. For example, given an XML document that represents an order, a Turing complete language is required to multiply the `price` of each `order_item` by its `quantity`, sum them all up, and verify that the sum equals the value of the `subtotal` element. Today's schema languages are also incapable of verifying extra-document constraints such as "Every `SKU` element matches the SKU field of a record in the products table of the inventory database." If you're writing programs to read XML documents, you can add code to verify statements like these, just as you would if you were writing code to read a tab-delimited text file. The difference is that XML parsers present you with the data in a much more convenient format and do more of the work for you before you have to resort to your own custom code.

1.1.1 What XML Is Not

XML is a markup language, and it is only a markup language. It's important to remember that. The XML hype has gotten so extreme that some people expect XML to do everything up to and including washing the family dog.

First of all, XML is not a programming language. There's no such thing as an XML compiler that reads XML files and produces executable code. You might perhaps define a scripting language that used a native XML format and was interpreted by a binary program, but even this application would be unusual.[2] XML can be used as a format for instructions to programs that do make things happen, just like a traditional program may read a text config file and take different action depending on what it sees there. Indeed, there's no reason a config file can't be XML instead of unstructured text. Some more recent programs are beginning to use XML config files; but in all cases it's the program taking action, not the XML document itself. An XML document by itself simply is. It does not do anything.

Secondly, XML is not a network transport protocol. XML won't send data across the network, any more than HTML will. Data sent across the network using HTTP, FTP, NFS, or some other protocol might happen to be encoded in an XML format, but again there has to be some software outside the XML document that actually does the sending.

Finally, to mention the example where the hype most often obscures the reality, XML is not a database. You're not going to replace an Oracle or MySQL server with XML. A database can contain XML data, either as a VARCHAR or a BLOB or as some custom XML data type, but the database itself is not an XML document. You can store XML data into a database on a server or retrieve data from a database in an XML format, but to do this, you need to be running software written in a real programming language such as C or Java. To store XML in a database, software on the client side will send the XML data to the server using an established network protocol such as TCP/IP. Software on the server side will receive the XML data, parse it, and store it in the database. To retrieve an XML document from a database, you'll generally pass through some middleware product like Enhydra that makes SQL queries against the database and formats the result set as XML before returning it to the client. Indeed, some databases may integrate this software code into their core server or provide plug-ins to do it such as the Oracle XSQL servlet. XML serves very well as a ubiquitous, platform-independent transport format in these scenarios. However, it is not the database, and it shouldn't be used as one.

1.2 Portable Data

XML offers the tantalizing possibility of truly cross-platform, long-term data formats. It's long been the case that a document written on one platform is not necessarily readable on a different platform, or by a different program on the same platform, or even by a future or past version of the same program on the same platform. When the document can be read, there's no guarantee that all the information will come across. Much of the data from the original moon landings in the late 1960s and early 1970s is now effectively lost. Even if you can find a tape drive that can read the now obsolete tapes, nobody knows in what format the data is stored on the tapes!

XML is an incredibly simple, well-documented, straightforward data format. XML documents are text and can be read with any tool that can read a text file. Not just the data, but also the markup is text, and it's present right there in the XML file as tags. You don't have to wonder whether every eighth byte is random padding, guess whether a four-byte quantity is a two's complement integer or an IEEE 754 floating point number, or try to decipher which integer codes map to which formatting properties. You can read the tag names directly to find out exactly what's in the document. Similarly, since element boundaries are defined by tags, you aren't likely to be tripped up by unexpected line-ending conventions or the number of spaces that are mapped to a tab. All the important details about the structure of the document are explicit. You don't have to reverse-engineer the format or rely on incomplete and often unavailable documentation.

A few software vendors may want to lock in their users with undocumented, proprietary, binary file formats. However, in the long term we're all better off if we can use the cleanly documented, well-understood, easy to parse, text-based formats that XML provides. XML lets documents and data be moved from one system to another with a reasonable hope that the receiving system will be able to make sense out of it. Furthermore, validation lets the receiving side check that what it gets is what it expects. Java promised portable code; XML delivers portable data. In many ways, XML is the most portable and flexible document format designed since the ASCII text file.

1.3 How XML Works

Example 1-1 shows a simple XML document. This particular XML document might be seen in an inventory-control system or a stock database. It marks up the data with tags and attributes describing the color, size, bar-code number, manufacturer, name of the product, and so on.

Example 1-1. An XML document

```
<?xml version="1.0"?>
<product barcode="2394287410">
  <manufacturer>Verbatim</manufacturer>
  <name>DataLife MF 2HD</name>
  <quantity>10</quantity>
  <size>3.5"</size>
  <color>black</color>
  <description>floppy disks</description>
</product>
```

This document is text and might well be stored in a text file. You can edit this file with any standard text editor such as BBEdit, jEdit, UltraEdit, Emacs, or vi. You do not need a special XML editor. Indeed, we find most general-purpose XML editors to be far more trouble than they're worth and much harder to use than simply editing documents in a text editor.

Programs that actually try to understand the contents of the XML document—that is, do more than merely treat it as any other text file—will use an XML parser to read the document. The parser is responsible for dividing the document into individual elements, attributes, and other pieces. It passes the contents of the XML document to an application piece by piece. If at any point the parser detects a violation of the well-formedness rules of XML, then it reports the error to the application and stops parsing. In some cases the parser may read further in the document, past the original error, so that it can detect and report other errors that occur later in the document. However, once it has detected the first well-formedness error, it will no longer pass along the contents of the elements and attributes it encounters.

Individual XML applications normally dictate more precise rules about exactly which elements and attributes are allowed where. For instance, you wouldn't expect to find a `G_Clef` element when reading a biology document. Some of these rules can be precisely specified with a schema written in any of several languages including the W3C XML Schema Language, RELAX NG, and DTDs. A document may contain a URI indicating where the schema can be found. Some XML parsers will notice this and compare the document to its schema as they read it to see if the document satisfies the constraints specified there. Such a parser is called a validating parser . A violation of those constraints is called a validity error , and the whole process of checking a document against a schema is called validation. If a validating parser finds a validity error, it will report it to the application on whose behalf it's parsing the document. This application can then decide whether it wishes to continue parsing the document. However, validity errors are not necessarily fatal (unlike well-formedness errors), and an application may choose to ignore them. Not all parsers are validating parsers. Some merely check for well-formedness.

The application that receives data from the parser may be:

- A web browser such as Netscape Navigator or Internet Explorer that displays the document to a reader
- A word processor such as StarOffice Writer that loads the XML document for editing
- A database such as Microsoft SQL Server that stores the XML data in a new record
- A drawing program such as Adobe Illustrator that interprets the XML as two-dimensional coordinates for the contents of a picture
- A spreadsheet such as Gnumeric that parses the XML to find numbers and functions used in a calculation
- A personal finance program such as Microsoft Money that sees the XML as a bank statement
- A syndication program that reads the XML document and extracts the headlines for today's news
- A program that you yourself wrote in Java, C, Python or some other language that does exactly what you want it to do
- Almost anything else

XML is an extremely flexible format for data. It is used for all of this and a lot more. These are real examples. In theory, any data that can be stored in a computer can be stored in XML format. In practice, XML is suitable for storing and exchanging any data that can plausibly be encoded as text. It's only really unsuitable for multimedia data such as photographs, recorded sound, video, and other very large bit sequences.

1.4 The Evolution of XML

XML is a descendant of SGML, the Standard Generalized Markup Language. The language that would eventually become SGML was invented by Charles F. Goldfarb, Ed Mosher, and Ray Lorie at IBM in the 1970s and developed by several hundred people around the world until its eventual adoption as ISO standard 8879 in 1986. SGML was intended to solve many of the same problems XML solves in much the same way XML solves them. It is a semantic and structural markup language for text documents. SGML is extremely powerful and achieved some success in the U.S. military and government, in the aerospace sector, and in other domains that needed ways of efficiently managing technical documents that were tens of thousands of pages long.

SGML's biggest success was HTML, which is an SGML application. However, HTML is just one SGML application. It does not have or offer anywhere near the full power of SGML itself. Since it restricts authors to a finite set of tags designed to describe web pages—and describes them in a fairly presentationally oriented way at that—it's really little more than a traditional markup language that has been adopted by web browsers. It doesn't lend itself to use beyond the single application of web-page design. You would not use HTML to exchange data between incompatible databases or to send updated product catalogs to retailer sites, for example. HTML does web pages, and it does them very well, but it only does web pages.

SGML was the obvious choice for other applications that took advantage of the Internet but were not simple web pages for humans to read. The problem was that SGML is complicated—very, very complicated. The official SGML specification is over 150 very technical pages. It covers many special cases and unlikely scenarios. It is so complex that almost no software has ever implemented it fully. Programs that implemented or relied on different subsets of SGML were often incompatible with each other. The special feature one program considered essential would be considered extraneous fluff and omitted by the next program.

In 1996, Jon Bosak, Tim Bray, C. M. Sperberg-McQueen, James Clark, and several others began work on a "lite" version of SGML that retained most of SGML's power while trimming a lot of the features that had proven redundant, too complicated to implement, confusing to end users, or simply not useful over the previous 20 years of experience with SGML. The result, in February of 1998, was XML 1.0, and it was an immediate success. Many developers who knew they needed a structural markup language but hadn't been able to bring themselves to accept SGML's complexity adopted XML whole-heartedly. It was used in domains ranging from legal court filings to hog farming.

However, XML 1.0 was just the beginning. The next standard out of the gate was Namespaces in XML, an effort to allow markup from different XML applications to be used in the same document without conflicting. Thus a web page about books could have a `title` element that referred to the title of the page and `title` elements that referred to the title of a book, and the two would not conflict.

Next up was the Extensible Stylesheet Language (XSL), an XML application for transforming XML documents into a form that could be viewed in web browsers. This soon split into XSL Transformations (XSLT) and XSL Formatting Objects (XSL-FO). XSLT has become a general-purpose language for transforming one XML document into another, whether for web-page display or some other purpose. XSL-FO is an XML application for describing the layout of both printed pages and web pages that rivals PostScript for its power and expressiveness.

However, XSL is not the only option for styling XML documents. The Cascading Style Sheets (CSS) language was already in use for HTML documents when XML was invented, and it proved to be a reasonable fit to XML as well. With the advent of CSS Level 2, the W3C made styling XML documents an explicit goal for CSS and gave it equal importance to HTML. The pre-existing Document Style Sheet and Semantics Language (DSSSL) was also adopted from its roots in the SGML world to style XML documents for print and the Web.

The Extensible Linking Language, XLink, began by defining more powerful linking constructs that could connect XML documents in a hypertext network that made HTML's `A` tag look like it is an abbreviation for "anemic." It also split into two separate standards: XLink for describing the connections between documents and XPointer for addressing the individual parts of an XML document. At this point, it was noticed that both XPointer and XSLT were developing fairly sophisticated yet incompatible syntaxes to do exactly the same thing: identify particular elements of an XML document. Consequently, the addressing parts of both specifications were split off and combined into a third specification, XPath.

Another piece of the puzzle was a uniform interface for accessing the contents of the XML document from inside a Java, JavaScript, or C++ program. The simplest API was merely to treat the document as an object that contained other objects. Indeed, work was already underway inside and outside the W3C to define such a Document Object Model (DOM) for HTML. Expanding this effort to cover XML was not hard.

Outside the W3C, David Megginson, Peter Murray-Rust, and other members of the xml-dev mailing list recognized that third party XML parsers, while all compatible in the documents they could parse, were incompatible in their APIs. This led to the development of the Simple API for

XML, SAX. In 2000, SAX2 was released to add greater configurability in parsing, namespace support, and a cleaner API.

One of the surprises during the evolution of XML was that developers used it more for data-oriented structures such as serialized objects and database records than for the narrative structures for which SGML had traditionally been used. DTDs worked very well for narrative structures, but had some limits when faced with the data-oriented structures developers were actually creating. In particular, the lack of data typing and the fact that DTDs were not themselves XML documents were perceived as major problems. A number of companies and individuals began working on schema languages that addressed these deficiencies. Many of these proposals were submitted to the W3C, which formed a working group to try to merge the best parts of all of these and come up with something greater than the sum of its parts. In 2001, this group released Version 1.0 of the W3C XML Schema Language. Unfortunately, they produced something overly complex and burdensome. Consequently, several developers went back to the drawing board to invent cleaner, simpler, more elegant schema languages, including RELAX NG and Schematron.

Eventually, it became apparent that XML 1.0, XPath, the W3C XML Schema Language, SAX, and DOM all had similar but subtly different conceptual models of the structure of an XML document. For instance, XPath and SAX don't consider CDATA sections to be anything more than syntax sugar, but DOM does treat them differently than plain-text nodes. Thus the W3C XML Core Working Group began work on an XML Information Set that all these standards could rely on and refer to.

Development of extensions to the core XML specification continues. Future directions include:

XML Query Language

> A fourth-generation language for extracting information that meets specified criteria from one or more XML documents

Canonical XML

> A standard algorithm for determining whether two XML documents are the same after insignificant details, such as whether single or double quotes delimit attribute values, are accounted for

XInclude

> A means of building a single XML document out of multiple well-formed, potentially valid XML documents and pieces thereof

XML Signatures

> A standard for digitally signing XML documents, embedding those signatures in XML documents, and authenticating the resulting documents

XML Encryption

> A standard XML syntax for encrypted digital content, including portions of XML documents

SAX 2.1

> A set of small extensions to SAX2 that provides extra information about an XML document recommended by the Infoset, including the XML declaration

DOM Level 3

> Many additional classes, interfaces, and methods that build on top of DOM2 to provide schema support, standard means of loading and saving XML documents, and many more additional capabilities

XFragment

> An effort to make sense out of pieces of XML documents that may not be well-formed documents when considered in isolation

Doubtless, many new extensions of XML remain to be invented. XML has proven itself a solid foundation for many diverse technologies.

[1] A few XML applications, such as XSL Formatting Objects, are designed to describe the presentation of text. However, these are exceptions that prove the rule. Although XSL-FO does describe presentation, you'd never write an XSL-FO document directly. Instead, you'd write a more semantically structured XML document, then use an XSL Transformations stylesheet to change the structure-oriented XML into presentation-oriented XML.

[2] At least one XML application, XSL Transformations, has been proven to be Turing complete by construction. See http://www.unidex.com/turing/utm.htm for one universal Turing machine written in XSLT.

Chapter 2. XML Fundamentals

This chapter shows you how to write simple XML documents. You'll see that an XML document is built from text content marked up with text tags such as `<SKU>`, `<Record_ID>`, and `<author>` that look superficially like HTML tags. However, in HTML you're limited to about a hundred predefined tags that describe web-page formatting. In XML you can create as many tags as you need. Furthermore, these tags will mostly describe the type of content they contain rather than formatting or layout information. In XML you don't say that something is italicized or indented or bold; you say that it's a book or a biography or a calendar.

Although XML is looser than HTML in regards to which tags it allows, it is much stricter about where those tags are placed and how they're written. In particular, all XML documents must be well-formed. Well-formedness rules specify constraints such as "Every start-tag must have a matching end-tag" and "Attribute values must be quoted." These rules are unbreakable, which makes parsing XML documents easy and writing them a little harder, but they still allow an almost unlimited flexibility of expression.

2.1 XML Documents and XML Files

An XML document contains text, never binary data. It can be opened with any program that knows how to read a text file. Example 2-1 is close to the simplest XML document imaginable. Nonetheless, t is a well-formed XML document. XML parsers can read it and understand it (at least as far as a computer program can be said to understand anything).

Example 2-1. A very simple yet complete XML document

```
<person>
  Alan Turing
</person>
```

In the most common scenario, this document would be the entire contents of a file named *person.xml*, or perhaps *2-1.xml*. However, XML is not picky about the filename. As far as the parser is concerned, this file could be called person.txt, person, or Hey you, there's some XML in this here file! Your operating system may or may not like these names, but an XML parser won't care. The document might not even be in a file at all. It could be a record or a field in a database. It could be generated on the fly by a CGI program in response to a browser query. It could even be stored in more than one file, though that's unlikely for such a simple document. If it is served by a web server, it will probably be assigned the MIME media type `application/xml` or `text/xml`.

However, specific XML applications may use more specific MIME media types such as `application/mathml+xml`, `application/XSLT+xml`, `image/svg+xml`, `text/vnd.wap.wml`, or even `text/html` (in very special cases).

> For generic XML documents, `application/xml` should be preferred to `text/xml`, although most web servers use `text/xml` by default. `text/xml` uses the ASCII character set as a default, which is incorrect for most XML documents.

2.2 Elements, Tags, and Character Data

The document in Example 2-1 is composed of a single element named person. The element is delimited by the start-tag `<person>` and the end-tag `</person>`. Everything between the start-tag and the end-tag of the element (exclusive) is called the element's content. The content of this element is the text string:

```
Alan Turing
```

The whitespace is part of the content, though many applications will choose to ignore it. `<person>` and `</person>` are markup. The string "Alan Turing" and its surrounding whitespace are character data. The tag is the most common form of markup in an XML document, but there are other kinds we'll discuss later.

2.2.1 Tag Syntax

XML tags look superficially like HTML tags. Start-tags begin with < and end-tags begin with `</`. Both of these are followed by the name of the element and are closed by >. However, unlike HTML tags, you are allowed to make up new XML tags as you go along. To describe a person, use `<person>` and `</person>` tags. To describe a calendar, use `<calendar>` and `</calendar>` tags. The names of the tags generally reflect the type of content inside the element, not how that content will be formatted.

2.2.1.1 Empty elements

There's also a special syntax for empty elements, i.e., elements that have no content. Such an element can be represented by a single empty-element tag that begins with < but ends with `/>`. For instance, in XHTML, an XMLized reformulation of standard HTML, the line-break and horizontal-rule elements are written as `<br />` and `<hr />` instead of `<br>` and `<hr>`. These are exactly equivalent to `<br></br>` and `<hr></hr>`, however. Which form you use for empty elements is completely up to you. However, what you cannot do in XML and XHTML (unlike HTML) is use only the start-tag—for instance `<br>` or `<hr>`—without using the matching the end-tag. That would be a well-formedness error.

2.2.1.2 Case sensitivity

XML, unlike HTML, is case sensitive. `<Person>` is not the same as `<PERSON>` is not the same as `<person>`. If you open an element with a `<person>` tag, you can't close it with a `</PERSON>` tag. You're free to use upper- or lowercase or both as you choose. You just have to be consistent within any one element.

2.2.2 XML Trees

Let's look at a slightly more complicated XML document. Example 2-2 is a `person` element that contains more information suitably marked up to show its meaning.

Example 2-2. A more complex XML document describing a person

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

2.2.2.1 Parents and children

This XML document is still composed of one `person` element. However, now this element doesn't merely contain undifferentiated character data. It contains four child elements: a `name` element and three `profession` elements. The `name` element contains two child elements of its own, `first_name` and `last_name`.

The `person` element is called the parent of the `name` element and the three `profession` elements. The `name` element is the `parent` of the `first_name` and `last_name` elements. The `name` element and the three `profession` elements are sometimes called each other's siblings. The `first_name` and `last_name` elements are also siblings.

As in human society, any one parent may have multiple children. However, unlike human society, XML gives each child exactly one parent, not two or more. Each element (with one exception I'll note shortly) has exactly one parent element. That is, it is completely enclosed by another element. If an element's start-tag is inside some element, then its end-tag must also be inside that element. Overlapping tags, as in `<strong><em>this common example from HTML</strong></em>`, are prohibited in XML. Since the `em` element begins inside the `strong` element, it must also finish inside the `strong` element.

2.2.2.2 The root element

Every XML document has one element that does not have a parent. This is the first element in the document and the element that contains all other elements. In Example 2-1 and Example 2-2, the `person` element filled this role. It is called the root element of the document . It is also sometimes called the document element. Every well-formed XML document has exactly one root element. Since elements may not overlap, and since all elements except the root have exactly one parent, XML documents form a data structure programmers call a tree. Figure 2-1 diagrams this relationship for Example 2-2. Each gray box represents an element. Each black box represents character data. Each arrow represents a containment relationship.

Figure 2-1. A tree diagram for Example 2-2

2.2.3 Mixed Content

In Example 2-2, the contents of the `first_name`, `last_name`, and `profession` elements were character data, that is, text that does not contain any tags. The contents of the `person` and `name` elements were child elements and some whitespace that most applications will ignore. This dichotomy between elements that contain only character data and elements that contain only child elements (and possibly a little whitespace) is common in documents that are data oriented. However, XML can also be used for more free-form, narrative documents such as business reports, magazine articles, student essays, short stories, web pages, and so forth, as shown by Example 2-3.

Example 2-3. A narrative-organized XML document

```
<biography>
  <name><first_name>Alan</first_name> <last_name>Turing</last_name>
  </name> was one of the first people to truly deserve the name
  <emphasize>computer scientist</emphasize>. Although his contributions
  to the field are too numerous to list, his best-known are the
  eponymous <emphasize>Turing Test</emphasize> and
  <emphasize>Turing Machine</emphasize>.

  <definition>The <term>Turing Test</term> is to this day the standard
  test for determining whether a computer is truly intelligent. This
  test has yet to be passed. </definition>

  <definition>The <term>Turing Machine</term> is an abstract finite
  state automaton with infinite memory that can be proven equivalent
  to any any other finite state automaton with arbitrarily large memory.
  Thus what is true for a Turing machine is true for all equivalent
  machines no matter how implemented.
  </definition>

  <name><last_name>Turing</last_name></name> was also an accomplished
  <profession>mathematician</profession> and
  <profession>cryptographer</profession>. His assistance
  was crucial in helping the Allies decode the German Enigma
  machine. He committed suicide on <date><month>June</month>
  <day>7</day>, <year>1954</year></date> after being
  convicted of homosexuality and forced to take female
  hormone injections.
</biography>
```

The root element of this document is `biography`. The `biography` contains `name`, `definition`, `profession`, and `emphasize` child elements. It also contains a lot of raw character data. Some of these elements such as `last_name` and `profession` only contain character data. Others such as `name` contain only child elements. Still others such as `definition` contain both character data and child elements. These elements are said to contain mixed content. Mixed content is common in XML documents containing articles, essays, stories, books, novels, reports, web pages, and anything else that's organized as a written narrative. Mixed content is less common and harder to work with in computer-generated and processed XML documents used for purposes such as database exchange, object serialization, persistent file formats, and so on. One of the strengths of XML is the ease with which it can be adapted to the very different requirements of human-authored and computer-generated documents.

2.3 Attributes

XML elements can have attributes. An attribute is a name-value pair attached to the element's start-tag. Names are separated from values by an equals sign and optional whitespace. Values are enclosed in single or double quotation marks. For example, this `person` element has a `born` attribute with the value `1912-06-23` and a `died` attribute with the value `1954-06-07`:

```
<person born="1912-06-23" died="1954-06-07">
  Alan Turing
</person>
```

This next element is exactly the same as far an XML parser is concerned. It simply uses single quotes instead of double quotes, puts some extra whitespace around the equals signs, and reorders the attributes.

```
<person died = '1954-06-07'  born = '1912-06-23' >
  Alan Turing
</person>
```

The whitespace around the equals signs is purely a matter of personal aesthetics. The single quotes may be useful in cases where the attribute value itself contains a double quote. Attribute order is not significant.

Example 2-4 shows how attributes might be used to encode much of the same information given in the data-oriented document of Example 2-2.

Example 2-4. An XML document that describes a person using attributes

```
<person>
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

This raises the question of when and whether one should use child elements or attributes to hold information. This is a subject of heated debate. Some informaticians maintain that attributes are for metadata about the element while elements are for the information itself. Others point out that it's not always so obvious what's data and what's metadata. Indeed, the answer may depend on where the information is put to use.

What's undisputed is that each element may have no more than one attribute with a given name. That's unlikely to be a problem for a birth date or a death date; it would be an issue for a profession, name, address, or anything else of which an element might plausibly have more than one. Furthermore, attributes are quite limited in structure. The value of the attribute is simply a text string. The division of a date into a year, month, and day with hyphens in the previous example is at the limits of the substructure that can reasonably be encoded in an attribute. Consequently, an element-based structure is a lot more flexible and extensible. Nonetheless, attributes are certainly more convenient in some applications. Ultimately, if you're designing your own XML vocabulary, it's up to you to decide when to use which.

Attributes are also useful in narrative documents, as Example 2-5 demonstrates. Here it's perhaps a little more obvious what belongs to elements and what to attributes. The raw text of the narrative is presented as character data inside elements. Additional information annotating that data is presented as attributes. This includes source references, image URLs, hyperlinks, and birth and death dates. Even here, however, there's more than one way to do it. For instance, the footnote numbers could be attributes of the `footnote` element rather than character data.

Example 2-5. A narrative XML document that uses attributes

```
<biography xmlns:xlink="http://www.w3.org/1999/xlink/namespace/">

  <image source="http://www.turing.org.uk/turing/pi1/bus.jpg"
  width="152" height="345"/>
  <person born='1912-06-23'
  died='1954-06-07'><first_name>Alan</first_name>
  <last_name>Turing</last_name> </person> was one of the first people
  to truly deserve the name <emphasize>computer scientist</emphasize>.
  Although his contributions to the field were too numerous to list,
  his best-known are the eponymous <emphasize xlink:type="simple"
  xlink:href="http://cogsci.ucsd.edu/~asaygin/tt/ttest.html">Turing
  Test</emphasize> and <emphasize  xlink:type="simple"
  xlink:href="http://mathworld.wolfram.com/TuringMachine.html">Turing
  Machine</emphasize>.

  <last_name>Turing</last_name> was also an accomplished
  <profession>mathematician</profession> and
  <profession>cryptographer</profession>. His assistance
  was crucial in helping the Allies decode the German Enigma
  machine.<footnote source="The Ultra Secret, F.W. Winterbotham,
  1974">1</footnote>

  He committed suicide on <date><month>June</month> <day>7</day>,
  <year>1954</year></date> after being convicted of homosexuality
  and forced to take female hormone injections.<footnote
  source="Alan Turing: the Enigma, Andrew Hodges, 1983">2</footnote>
</biography>
```

2.4 XML Names

The XML specification can be quite legalistic and picky at times. Nonetheless, it tries to be efficient where possible. One way it does that is by reusing the same rules for different items where possible. For example, the rules for XML element names are also the rules for XML attribute names, as well as for the names of several less common constructs. Generally, these are referred to simply as XML names.

Element and other XML names may contain essentially any alphanumeric character. This includes the standard English letters A through Z and a through z as well as the digits 0 through 9. XML names may also include non-English letters, numbers, and ideograms such as ö, ç, , and . They may also include these three punctuation characters:

_

the underscore

-

the hyphen

.

the period

XML names may not contain other punctuation characters such as quotation marks, apostrophes, dollar signs, carets, percent symbols, and semicolons. The colon is allowed, but its use is reserved for namespaces as discussed in Chapter 4. XML names may not contain whitespace of any kind, whether a space, a carriage return, a line feed, a nonbreaking space, and so forth. Finally, all names beginning with the string `XML` (in any combination of case) are reserved for standardization in W3C XML-related specifications.

XML names may only start with letters, ideograms, and the underscore character. They may not start with a number, hyphen, or period. There is no limit to the length of an element or other XML name. Thus these are all well-formed elements:

- `<Drivers_License_Number>98 NY 32</Drivers_License_Number>`
- `<month-day-year>7/23/2001</month-day-year>`
- `<first_name>Alan</first_name>`
- `<_4-lane>I-610</_4-lane>`
- `<téléphone>011 33 91 55 27 55 27</téléphone>`
-

These are not acceptable elements:

- `<Driver's_License_Number>98 NY 32</Driver's_License_Number>`
- `<month/day/year>7/23/2001</month/day/year>`
- `<first name>Alan</first name>`
- `<4-lane>I-610</4-lane>`

2.5 Entity References

The character data inside an element may not contain a raw unescaped opening angle bracket (<). This character is always interpreted as beginning a tag. If you need to use this character in your text, you can escape it using the `&lt;` entity reference. When a parser reads the document, it will replace the `&lt;` entity reference with the actual < character. However, it will not confuse `&lt;` with the start of a tag. For example:

```
<SCRIPT LANGUAGE="JavaScript">
  if (location.host.toLowerCase( ).indexOf("cafeconleche") &lt; 0) {
    location.href="http://www.cafeconleche.org/";
  }
```

```
</SCRIPT>
```

The character data inside an element may not contain a raw unescaped ampersand (`&`) either. This is always interpreted as beginning an entity or character reference. However, the ampersand may be escaped using the `&amp;` entity reference like this:

```
<publisher>O'Reilly &amp; Associates</publisher>
```

Entity references such as `&amp;` and `&lt;` are considered to be markup. When an application parses an XML document, it replaces this particular markup with the actual characters to which the entity reference refers.

XML predefines exactly five entity references. These are:

&lt;

The less-than sign; a.k.a. the opening angle bracket (<)

&amp;

The ampersand (&)

&gt;

The greater-than sign; a.k.a. the closing angle bracket (>)

&quot;

The straight, double quotation marks (")

&apos;

The apostrophe; a.k.a. the straight single quote (')

Only `&lt;` and `&amp;` must be used instead of the literal characters in element content. The others are optional. `&quot;` and `&apos;` are useful inside attribute values where a raw `"` or `'` might be misconstrued as ending the attribute value. For example, this image tag uses the `&apos;` entity reference to fill in the apostrophe in O'Reilly:

```
<image source='oreilly_koala3.gif' width='122' height='66'
       alt='Powered by O&apos;Reilly Books'
/>
```

Although there's no possibility of an unescaped greater-than sign (>) being misinterpreted as closing a tag it wasn't meant to close, `&gt;` is allowed mostly for symmetry with `&lt;`.

In addition to the five predefined entity references, you can define others in the document type definition. We'll discuss how to do this in Chapter 3.

2.6 CDATA Sections

When an XML document includes samples of XML or HTML source code, the < and & characters in those samples must be encoded as `&lt;` and `&amp;`. The more sections of literal code a document includes and the longer they are, the more tedious this encoding becomes. Instead you can enclose each sample of literal code in a CDATA section. A CDATA section is set off by a `<![CDATA[` and `]]>`. Everything between the `<![CDATA[` and the `]]>` is treated as raw character data. Less-than signs don't begin. Ampersands don't start entity references. Everything is simply character data, not markup.

For example, in a Scalable Vector Graphics (SVG) tutorial written in XHTML, you might see something like this:

```
<p>You can use a default <code>xmlns</code> attribute to avoid
having to add the svg prefix to all your elements:</p>
     <![CDATA[
       <svg xmlns="http://www.w3.org/2000/svg"
            width="12cm" height="10cm">
         <ellipse rx="110" ry="130" />
         <rect x="4cm" y="1cm" width="3cm" height="6cm" />
       </svg>
     ]]>
```

The SVG source code has been included directly in the XHTML file without carefully replacing each < with `&lt;`. The result will be a sample SVG document, not an embedded SVG picture, as might happen if this example were not placed inside a CDATA section.

The only thing that can not appear in a CDATA section is the CDATA section end delimiter `]]>`.

CDATA sections exist for the convenience of human authors, not for programs. Parsers are not required to tell you whether a particular block of text came from a CDATA section, from normal character data, or from character data that contained entity references such as `&lt;` and `&amp;`. By the time you get access to the data, these differences will have been washed away.

2.7 Comments

XML documents can be commented so that coauthors can leave notes for each other and themselves, documenting why they've done what they've done or items that remain to be done. XML comments are syntactically similar to HTML comments. Just as in HTML, they begin with `<!--` and end with the first occurrence of `-->`. For example:

```
<!-- I need to verify and update these links when I get a chance. -->
```

The double hyphen `--` should not appear anywhere inside the comment until the closing `-->`. In particular, a three hyphen close like `--->` is specifically forbidden.

Comments may appear anywhere in the character data of a document. They may also appear before or after the root element. (Comments are not elements, so this does not violate the tree structure or the one-root element rules for XML.) However, comments may not appear inside a tag or inside another comment.

Applications that read and process XML documents may or may not pass along information included in comments. They are certainly free to drop them out if they choose. Do not write documents or applications that depend on the contents of comments being available. Comments are strictly for making the raw source code of an XML document more legible to human readers. They are not intended for computer programs. For this purpose you should use a *processing instruction* instead.

2.8 Processing Instructions

In HTML, comments are sometimes abused to support nonstandard extensions. For instance, the contents of the `script` element are sometimes enclosed in a comment to protect it from display by a nonscript-aware browser. The Apache web server parses comments in *.shtml* files to recognize server side includes. Unfortunately, these documents may not survive being passed through various HTML editors and processors with their comments and associated semantics intact. Worse yet, it's possible for an innocent comment to be misconstrued as input to the application.

XML provides the processing instruction as an alternative means of passing information to particular applications that may read the document. A processing instruction begins with `<?` and ends with `?>`. Immediately following the `<?` is an XML name called the target, possibly the name of the application for which this processing instruction is intended or possibly just an identifier for this particular processing instruction. The rest of the processing instruction contains text in a format appropriate for the applications for which the instruction is intended.

For example, in HTML a robots `META` tag is used to tell search-engine and other robots whether and how they should index a page. The following processing instruction has been proposed as an equivalent for XML documents:

```
<?robots index="yes" follow="no"?>
```

The target of this processing instruction is `robots`. The syntax of this particular processing instruction is two pseudoattributes, one named `index` and one named `follow`, whose values are either `yes` or `no`. The semantics of this particular processing instruction are that if the `index` attribute has the value `yes`, then search-engine robots should index this page. If `index` has the value `no`, then it won't be. Similarly, if `follow` has the value `yes`, then links from this document will be followed.

Other processing instructions may have totally different syntaxes and semantics. For instance, processing instructions can contain an effectively unlimited amount of text. PHP includes large programs in processing instructions. For example:

```
<?php
  mysql_connect("database.unc.edu", "clerk", "password");
  $result = mysql("HR", "SELECT LastName, FirstName FROM Employees
    ORDER BY LastName, FirstName");
  $i = 0;
  while ($i < mysql_numrows ($result)) {
     $fields = mysql_fetch_row($result);
     echo "<person>$fields[1] $fields[0] </person>\r\n";
     $i++;
  }
  mysql_close( );
?>
```

Processing instructions are markup, but they're not elements. Consequently, like comments, processing instructions may appear anywhere in an XML document outside of a tag, including before or after the root element. The most common processing instruction, `xml-stylesheet`, is used to attach stylesheets to documents. It always appears before the root element, as Example 2-6 demonstrates. In this example, the `xml-stylesheet` processing instruction tells browsers to apply the CSS stylesheet *person.css* to this document before showing it to the reader.

Example 2-6. A very simple yet complete XML document

```
<?xml-stylesheet href="person.css" type="text/css"?>
<person>
  Alan Turing
</person>
```

The processing instruction names `xml`, `XML`, `XmL`, etc., in any combination of case, are forbidden to avoid confusion with the XML declaration. Otherwise, you're free to pick any legal XML name for your processing instructions.

2.9 The XML Declaration

XML documents should (but do not have to) begin with an XML declaration. The XML declaration looks like a processing instruction with the name `xml` and `version`, `standalone`, and `encoding` attributes. Technically, it's not a processing instruction though, just the XML declaration; nothing more, nothing less. Example 2-7 demonstrates.

Example 2-7. A very simple XML document with an XML declaration

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<person>
  Alan Turing
</person>
```

XML documents do not have to have an XML declaration. However, if an XML document does have an XML declaration, then that declaration must be the first thing in the document. It must not be preceded by any comments, whitespace, processing instructions, and so forth. The reason is that an XML parser uses the first five characters (`<?xml`) to make some reasonable guesses about the encoding, such as whether the document uses a single byte or multibyte character set. The only thing that may precede the XML declaration is an invisible Unicode byte-order mark. We'll discuss this further in Chapter 5.

2.9.1 encoding

So far we've been a little cavalier about encodings. We've said that XML documents are composed of pure text, but we haven't said what encoding that text uses. Is it ASCII? Latin-1? Unicode? Something else?

The short answer to this question is "Yes." The long answer is that by default XML documents are assumed to be encoded in the UTF-8 variable-length encoding of the Unicode character set. This is a strict superset of ASCII, so pure ASCII text files are also UTF-8 documents. However, most XML processors, especially those written in Java, can handle a much broader range of character sets. All you have to do is tell the parser which character encoding the document uses. Preferably this is done through metainformation, stored in the filesystem or provided by the server. However, not all systems provide character-set metadata so XML also allows documents to specify their own character set with an encoding declaration inside the XML declaration. Example 2-8 shows how you'd indicate that a document was written in the ISO-8859-1 (Latin-1) character set that includes letters like ö and ç needed for many non-English Western European languages.

Example 2-8. An XML document encoded in Latin-1

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<person>
  Erwin Schrödinger
</person>
```

The `encoding` attribute is optional in an XML declaration. If it is omitted and no metadata is available, then the Unicode character set is assumed. The parser may use the first several bytes of the file to try to guess which encoding of Unicode is in use. If metadata is available and it conflicts with the encoding declaration, then the encoding specified by the metadata wins. For example, if an HTTP header says a document is encoded in ASCII but the encoding declaration says it's encoded in UTF-8, then the parser will pick ASCII.

The different encodings and the proper handling of non-English XML documents will be discussed in greater detail in Chapter 5.

2.9.2 standalone

If the `standalone` attribute has the value `no`, then an application may be required to read an external DTD (that is a DTD in a file other than the one it's reading now) to determine the proper values for parts of the document. For instance, a DTD may provide default values for attributes that a parser is required to report even though they aren't actually present in the document.

Documents that do not have DTDs, like all the documents in this chapter, can have the value `yes` for the `standalone` attribute. Documents that do have DTDs can also have the value `yes` for the `standalone` attribute if the DTD doesn't in any way change the content of the document or if the DTD is purely internal. Details for documents with DTDs are covered in Chapter 3.

The `standalone` attribute is optional in an XML declaration. If it is omitted, then the value `no` is assumed.

2.10 Checking Documents for Well-Formedness

Every XML document, without exception, must be well-formed. This means it must adhere to a number of rules, including the following:

1. Every start-tag must have a matching end-tag.
2. Elements may nest, but may not overlap.
3. There must be exactly one root element.
4. Attribute values must be quoted.
5. An element may not have two attributes with the same name.
6. Comments and processing instructions may not appear inside tags.
7. No unescaped < or `&` signs may occur in the character data of an element or attribute.

This is not an exhaustive list. There are many, many ways a document can be malformed. You'll find a complete list in Chapter 20. Some of these involve constructs that we have not yet discussed such as DTDs. Others are extremely unlikely to occur if you follow the examples in this chapter (for example, including whitespace between the opening < and the element name in a tag).

Whether the error is small or large, likely or unlikely, an XML parser reading a document is required to report it. It may or may not report multiple well-formedness errors it detects in the document. However, the parser is not allowed to try to fix the document and make a best-faith effort of providing what it thinks the author really meant. It can't fill in missing quotes around attribute values, insert an omitted end-tag, or ignore the comment that's inside a start-tag. The parser is required to return an error. The objective here is to avoid the bug-for-bug compatibility wars that plagued early web browsers and continue to this day. Consequently, before you publish an XML document, whether that document is a web page, input to a database, or something else, you'll want to check it for well-formedness.

The simplest way to do this is by loading the document into a web browser that understands XML documents such as Mozilla. If the document is well-formed, the browser will display it. If it isn't, then it will show an error message.

Instead of loading the document into a web browser, you can use an XML parser directly. Most XML parsers are not intended for end users. They are class libraries designed to be embedded into an easier-to-use program such as Mozilla. They provide a minimal command-line interface, if that; that interface is often not particularly well documented. Nonetheless, it can sometimes be quicker to run a batch of files through a command-line interface than loading each of them into a web browser. Furthermore, once you learn about DTDs and schemas, you can use the same tools to validate documents, which most web browsers won't do.

There are many XML parsers available in a variety of languages. Here, we'll demonstrate checking for well-formedness with the Apache XML Project's Xerces-J 1.4, which you can download from http://xml.apache.org/xerces-j. This open source package is written in pure Java so it should run across all major platforms. The procedure should be similar for other parsers, though details will vary.

To use this parser, you'll first need a Java 1.1 or later compatible virtual machine. Virtual machines for Windows, Solaris, and Linux are available from http://java.sun.com/. To install Xerces-J 1.4.4, just add *xerces.jar* and *xercesSamples.jar* files to your Java class path. In Java 2 you can simply put those *.jar* files into your *jre/lib/ext* directory.

The class that actually checks files for well-formedness is called `sax.SAXCount`. It's run from a Unix shell or DOS prompt like any other standalone Java program. The command-line arguments are the URLs to or filenames of the documents you want to check. Here's the result of running *SAXCount* against an early version of Example 2-5. The very first line of output tells you where the first problem in the file is. The rest of the output is a more or less irrelevant stack trace.

```
D:\xian\examples\02>java sax.SAXCount 2-5.xml
[Fatal Error] 2-5.xml:3:30: The value of attribute "height" must not contain the
'<' character.
Stopping after fatal error: The value of attribute "height" must not contain the
'<' character.
at org.apache.xerces.framework.XMLParser.reportError(XMLParser.java:
1282)
at org.apache.xerces.framework.XMLDocumentScanner.reportFatalXMLError(
XMLDocumentScanner.java:644)
at org.apache.xerces.framework.XMLDocumentScanner.scanAttValue(
XMLDocumentScanner.java:519)
at org.apache.xerces.framework.XMLParser.scanAttValue(
XMLParser.java:1932)
at org.apache.xerces.framework.XMLDocumentScanner.scanElement(
XMLDocumentScanner.java:1800)
at org.apache.xerces.framework.XMLDocumentScanner$ContentDispatcher.
dispatch(XMLDocumentScanner.java:1223)
```

```
at org.apache.xerces.framework.XMLDocumentScanner.parseSome(
XMLDocumentScanner.java:381)
at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:1138)
at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:1177)
at sax.SAXCount.print(SAXCount.java:135)
at sax.SAXCount.main(SAXCount.java:331)
```

As you can see, it found an error. In this case the error message wasn't particularly helpful. The actual problem wasn't that an attribute value contained a < character. It was that the closing quote was missing from the `height` attribute value. Still, that was enough for us to locate and fix the problem. Despite the long list of output, *SAXCount* only reports the first error in the document, so you may have to run it multiple times until all the mistakes are found and fixed. Once we fixed Example 2-5 to make it well-formed, *SAXCount* simply reported how long it took to parse the document and what it saw when it did:

```
D:\xian\examples\02>java sax.SAXCount 2-5.xml
2-5.xml: 140 ms (17 elems, 12 attrs, 0 spaces, 564 chars)
```

Now that the document has been corrected to be well-formed, it can be passed to a web browser, a database, or whatever other program is waiting to receive it. Almost any nontrivial document crafted by hand will contain well-formedness mistakes. That makes it important to check your work before publishing it.

This example works with Xerces-J 1.0 through 1.4.4. The recently released Xerces-J 2.0 provides a similar program named `sax.Counter`.

Chapter 3. Document Type Definitions (DTDs)

While XML is extremely flexible, not all the programs that read particular XML documents are so flexible. Many programs can work with only some XML applications but not others. For example, Adobe Illustrator 10 can read and write Scalable Vector Graphics (SVG) files, but you wouldn't expect it to understand a Platform for Privacy Preferences (P3P) document. And within a particular XML application, it's often important to ensure that a given document indeed adheres to the rules of that XML application. For instance, in XHTML, `li` elements should only be children of `ul` or `ol` elements. Browsers may not know what to do with them, or may act inconsistently, if `li` elements appear in the middle of a `blockquote` or `p` element.

The solution to this dilemma is a *document type definition* (DTD). DTDs are written in a formal syntax that explains precisely which elements and entities may appear where in the document and what the elements' contents and attributes are. A DTD can make statements such as "A `ul` element only contains `li` elements" or "Every `employee` element must have a `social_security_number` attribute." Different XML applications can use different DTDs to specify what they do and do not allow.

A validating parser compares a document to its DTD and lists any places where the document differs from the constraints specified in the DTD.[1] The program can then decide what it wants to do about any violations. Some programs may reject the document. Others may try to fix the document or reject just the invalid element. Validation is an optional step in processing XML. A validity error is not necessarily a fatal error like a well-formedness error, though some applications may choose to treat it as one.

3.1 Validation

A valid document includes a document type declaration that identifies the DTD the document satisfies. The DTD lists all the elements, attributes, and entities the document uses and the contexts in which it uses them. The DTD may list items the document does not use as well. Validity operates on the principle that everything not permitted is forbidden. Everything in the document must match a declaration in the DTD. If a document has a document type declaration and the document satisfies the DTD that the document type declaration indicates, then the document is said to be valid. If it does not, it is said to be invalid.

There are many things the DTD does not say. In particular, it does not say the following:

- What the root element of the document is
- How many of instances of each kind of element appear in the document
- What the character data inside the elements looks like
- The semantic meaning of an element; for instance, whether it contains a date or a person's name

DTDs allow you to place some constraints on the form an XML document takes, but there can be quite a bit of flexibility within those limits. A DTD never says anything about the length, structure, meaning, allowed values, or other aspects of the text content of an element.

Validity is optional. A parser reading an XML document may or may not check for validity. If it does check for validity, the program receiving data from the parser may or may not care about validity errors. In some cases, such as feeding records into a database, a validity error may be quite serious, indicating that a required field is missing, for example. In other cases, rendering a web page perhaps, a validity error may not be so important, and you can work around it. Well-formedness is required of all XML documents; validity is not. Your documents and your programs can use it or not as you find needful.

3.1.1 A Simple DTD Example

Recall Example 2-2 from the last chapter; this described a person. The person had a name and three professions. The name had a first name and a last name. The particular person described in that example was Alan Turing. However, that's not relevant for DTDs. A DTD only describes the general type, not the specific instance. A DTD for person documents would say that a `person` element contains one `name` child element and zero or more `profession` child elements. It would further say that each `name` element contains a `first_name` child element and a `last_name` child element. Finally it would state that the `first_name`, `last_name`, and `profession` elements all contain text. Example 3-1 is a DTD that describes such a `person` element.

Example 3-1. A DTD for the person

```
<!ELEMENT person     (name, profession*)>
<!ELEMENT name       (first_name, last_name)>
<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name  (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
```

This DTD would probably be stored in a separate file from the documents it describes. This allows it to be easily referenced from multiple XML documents. However, it can be included inside the XML document if that's convenient, using the document type declaration we discuss later in this section. If it is stored in a separate file, then that file would most likely be named *person.dtd*, or something similar. The .dtd extension is fairly standard though not specifically required by the XML specification. If this file were served by a web server, it would be given the MIME media type `application/xml-dtd`.

Each line of Example 3-1 is an element declaration. The first line declares the `person` element; the second line declares the `name` element; the third line declares the `first_name` element; and so on. However, the line breaks aren't relevant except for legibility. Although it's customary to put only one declaration on each line, it's not required. Long declarations can even span multiple lines.

The first element declaration in Example 3-1 states that each `person` element must contain exactly one `name` child element followed by zero or more `profession` elements. The asterisk after profession stands for "zero or more." Thus, every person must have a name and may or may not have a profession or multiple professions. However, the name must come before all professions. For example, this `person` element is valid:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
</person>
```

However, this `person` element is not valid because it omits the name:

```
<person>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

This `person` element is not valid because a `profession` element comes before the `name`:

```
<person>
  <profession>computer scientist</profession>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

The `person` element may not contain any element except those listed in its declaration. The only extra character data it can contain is whitespace. For example, this is an invalid `person` element because it adds a `publication` element:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
  <publication>On Computable Numbers...</publication>
</person>
```

This is an invalid `person` element because it adds some text outside the allowed children:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  was a <profession>computer scientist</profession>,
  a <profession>mathematician</profession>, and a
  <profession>cryptographer</profession>
</person>
```

In all these examples of invalid elements, you could change the DTD to make these elements valid. All the examples are well-formed, after all. However, with the DTD in Example 3-1, they are not valid.

The name declaration says that each name element must contain exactly one first_name element followed by exactly one last_name element. All other variations are forbidden.

The remaining three declarations—first_name, last_name, and profession—all say that their elements must contain #PCDATA. This is a DTD keyword standing for parsed character data —that is, raw text possibly containing entity references such as &amp; and &lt;, but not containing any tags or child elements.

Example 3-1 placed the most complicated and highest-level declaration at the top. However, that's not required. For instance, Example 3-2 is an equivalent DTD that simply reorders the declarations. DTDs allow forward, backward, and circular references to other declarations.

Example 3-2. An alternate DTD for the person element

```
<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name  (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
<!ELEMENT name       (first_name, last_name)>
<!ELEMENT person     (name, profession*)>
```

3.1.2 The Document Type Declaration

A valid document includes a reference to the DTD to which it should be compared. This is given in the document's single document type declaration. A document type declaration looks like this:

```
<!DOCTYPE person SYSTEM "http://www.cafeconleche.org/dtds/person.dtd">
```

This says that the root element of the document is person and that the DTD for this document can be found at the URI http://www.cafeconleche.org/dtds/person.dtd.

URI stands for Uniform Resource Identifier. URIs are a superset of URLs . They include not only URLs but also Uniform Resource Names (URNs). A URN allows you to identify a resource such as the DTD for SVGs irrespective of its location. Indeed, the resource might exist at multiple locations, all equally authoritative. In practice, the only URIs in wide use today are URLs.

The document type declaration is included in the prolog of the XML document after the XML declaration but before the root element. (The prolog is everything in the XML document before the root element start-tag.) Example 3-3 demonstrates.

Example 3-3. A valid person document

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE person SYSTEM "http://www.cafeconleche.org/dtds/person.dtd">
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

If the document resides at the same base site as the DTD, you can use a relative URL instead of the absolute form. For example:

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

You can even use just the filename if the DTD is in the same directory as the document:

```
<!DOCTYPE person SYSTEM "person.dtd">
```

3.1.2.1 Public IDs

Standard DTDs may actually be stored at multiple URLs. For example, if you're drawing an SVG picture on your laptop at the beach, you probably want to validate your drawing without opening a network connection to the W3C's web site where the official SVG DTD resides. Such DTDs may be associated with public IDs. The name of the public ID uniquely identifies the XML application in use. At the same time, a backup URI is also included in case the validator does not recognize the public ID. To indicate that you're specifying a public ID, use the keyword `PUBLIC` in place of `SYSTEM`. For example, this document type declaration refers to the Rich Site Summary (RSS) DTD standardized by Netscape:

```
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
              "http://my.netscape.com/publish/formats/rss-0.91.dtd">
```

A local catalog server can be used to convert the public IDs into the most appropriate URLs for the local environment. The catalogs themselves can be written in XML, specifically, the OASIS XML catalog format (http://www.oasis-open.org/committees/entity/spec.html). In practice, however, PUBLIC IDs aren't used very much. Almost all validators rely on the URI to actually validate the document.

3.1.3 Internal DTD Subsets

When you're first developing a DTD, it's often useful to keep the DTD and the canonical example document in the same file so you can modify and check them simultaneously. Therefore, the document type declaration may actually contain the DTD between square brackets rather than referencing it at an external URI. Example 3-4 demonstrates.

Example 3-4. A valid person document with an internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE person [
  <!ELEMENT first_name (#PCDATA)>
  <!ELEMENT last_name  (#PCDATA)>
  <!ELEMENT profession (#PCDATA)>
  <!ELEMENT name       (first_name, last_name)>
  <!ELEMENT person     (name, profession*)>
]>
<person>
  <name>
```

```
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

Some document type declarations contain some declarations directly but link in others using a `SYSTEM` or `PUBLIC` identifier. For example, this document type declaration declares the `profession` and `person` elements itself but relies on the file *name.dtd* to contain the declaration of the name element:

```
<!DOCTYPE person SYSTEM "name.dtd" [
  <!ELEMENT profession (#PCDATA)>
  <!ELEMENT person (name, profession*)>
]>
```

The part of the DTD between the brackets is called the internal DTD subset. All the parts that come from outside this document are called the external DTD subset. Together they make up the complete DTD. As a general rule, the two different subsets must be compatible. Neither can override the element declarations the other makes. For example, if *name.dtd* also declared the `person` element, then there would be a problem. However, entity declarations can be overridden with some important consequences for DTD structure and design, which we'll see shortly when we discuss entities.

When you use an external DTD subset, you should give the `standalone` attribute of the XML declaration the value `no`. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

Actually, the XML specification includes four very detailed rules about exactly when the presence of an external DTD subset does and does not require the `standalone` attribute to have the value `no`. However, the net effect of these rules is that almost all XML documents that use external DTD subsets require `standalone` to have the value `no`. Since setting `standalone` to `no` is always permitted even when it's not required, it's simply not worth worrying about the uncommon cases.

A validating processor is required to read the external DTD subset. A nonvalidating processor may do so, but is not required to, even if `standalone` has the value `no`. This means that if the external subset makes declarations that have consequences for the content of a document (for instance, providing default values for attributes) then the content of the document depends on which parser you're using and how it's configured. This has led to no end of confusion. Although some of the earliest XML parsers did not resolve external entities, most of the parsers still being used can do so and generally will do so. You should read the external DTD subset unless efficiency is a major concern or you're very familiar with the structure of the documents you're parsing.

3.1.4 Validating a Document

As a general rule, web browsers do not validate documents but only check them for well-formedness. If you're writing your own programs to process XML, you can use the parser's API to validate documents. If you're writing documents by hand and you want to validate them, you can either use one of the online validators or run a local program to validate the document.

The online validators are probably the easiest way to validate your documents. There are two of note:

- The Brown University Scholarly Technology Group's XML Validation Form at http://www.stg.brown.edu/service/xmlvalid/
- Richard Tobin's XML well-formedness checker and validator at http://www.cogsci.ed.ac.uk/%7Erichard/xml-check.html

First, you have to place the document and associated DTDs on a publicly accessible web server. Next, load one of the previous URLs in a browser, and type the URL of the document you're checking into the online form. The validating server will retrieve your document and tell you what, if any, errors it found. Figure 3-1 shows the results of using the Brown validator on a simple invalid but well-formed document.

Figure 3-1. Validity errors detected by the Brown University online validator

Most XML parser class libraries include a simple program to validate documents you can use if you're comfortable installing and using command-line programs. In Xerces 1.x, that program is *sax.SAXCount*. (Xerces 2.x uses *sax.Counter* instead.) Use the `-v` flag to turn on validation. (By default, *SAXCount* only checks for well-formedness.) Then pass the URLs or filenames of the documents you wish to validate to the *SAXCount* program on the command line like this:

```
C:\>java sax.SAXCount -v invalid_fibonacci.xml
```

```
[Error] invalid_fibonacci.xml:8:10: Element type "title" must be declared.
[Error] invalid_fibonacci.xml:110:22: The content of element type
"Fibonacci_Numbers" must match "(fibonacci)*".
fibonacci.xml: 541 ms (103 elems, 101 attrs, 307 spaces, 1089 chars)
```

You can see from this output that the document *invalid_fibonacci.xml* has two validity errors that need to be fixed: the first in line 8 and the second in line 110.

There are also some simple GUI programs for validating XML documents, including the Topologi Schematron Validator for Windows (http://www.topologi.com) shown in Figure 3-2. Despite the name, this product can actually validate documents against schemas written in multiple languages, including DTDs and the W3C XML Schema Language, as well as Schematron.

Figure 3-2. Validity errors detected by the Topologi Schematron Validator

3.2 Element Declarations

Every element used in a valid document must be declared in the document's DTD with an element declaration. Element declarations have this basic form:

```
<!ELEMENT element_name content_specification>
```

The name of the element can be any legal XML name. The content specification specifies what children the element may or must have in what order. Content specifications can be quite complex. They can say, for example, that an element must have three child elements of a given type, or two children of one type followed by another element of a second type, or any elements chosen from seven different types interspersed with text.

3.2.1 #PCDATA

About the simplest content specification is one that says an element may only contain parsed character data, but may not contain any child elements of any type. In this case the content specification consists of the keyword `#PCDATA` inside parentheses. For example, this declaration says that a `phone_number` element may contain text, but may not contain elements:

```
<!ELEMENT phone_number (#PCDATA)>
```

3.2.2 Child Elements

Another simple content specification is one that says the element must have exactly one child of a given type. In this case, the content specification simply consists of the name of the child element inside parentheses. For example, this declaration says that a `fax` element must contain exactly one `phone_number` element:

```
<!ELEMENT fax (phone_number)>
```

A `fax` element may not contain anything else except the `phone_number` element, and it may not contain more or less than one of those.

3.2.3 Sequences

In practice, however, a content specification that lists exactly one child element is rare. Most elements contain either parsed character data or (at least potentially) multiple child elements. The simplest way to indicate multiple child elements is to separate them with commas. This is called a sequence. It indicates that the named elements must appear in the specified order. For example, this element declaration says that a `name` element must contain exactly one `first_name` child element followed by exactly one `last_name` child element:

```
<!ELEMENT name (first_name, last_name)>
```

Given this declaration, this `name` element is valid:

```
<name>
  <first_name>Madonna</first_name>
  <last_name>Ciconne</last_name>
</name>
```

However, this one is not valid because it flips the order of two elements:

```
<name>
  <last_name>Ciconne</last_name>
  <first_name>Madonna</first_name>
</name>
```

This element is invalid because it omits the `last_name` element:

```
<name>
  <first_name>Madonna</first_name>
```

```
</name>
```

This one is invalid because it adds a `middle_name` element:

```
<name>
  <first_name>Madonna</first_name>
  <middle_name>Louise</middle_name>
  <last_name>Ciconne</last_name>
</name>
```

3.2.4 The Number of Children

As the previous examples indicate, not all instances of a given element necessarily have exactly the same children. You can affix one of three suffixes to an element name in a content specification to indicate how many of that element are expected at that position. These suffixes are:

?

Zero or one of the element is allowed.

*

Zero or more of the element is allowed.

+

One or more of the element is required.

For example, this declaration says that a `name` element must contain a `first_name`, may or may not contain a `middle_name`, and may or may not contain a `last_name`:

```
<!ELEMENT name (first_name, middle_name?, last_name?)>
```

Given this declaration, all these `name` elements are valid:

```
<name>
  <first_name>Madonna</first_name>
  <last_name>Ciconne</last_name>
</name>
<name>
  <first_name>Madonna</first_name>
  <middle_name>Louise</middle_name>
  <last_name>Ciconne</last_name>
</name>
<name>
  <first_name>Madonna</first_name>
</name>
```

However, these are not valid:

```
<name>
  <first_name>George</first_name>
  <!-- only one middle name is allowed -->
  <middle_name>Herbert</middle_name>
  <middle_name>Walker</middle_name>
  <last_name>Bush</last_name>
</name>
<name>
  <!-- first name must precede last name -->
  <last_name>Ciconne</last_name>
  <first_name>Madonna</first_name>
</name>
```

You can allow for multiple middle names by placing an asterisk after the `middle_name`:

```
<!ELEMENT name (first_name, middle_name*, last_name?)>
```

If you wanted to require a `middle_name` to be included, but still allow for multiple middle names, you'd use a plus sign instead, like this:

```
<!ELEMENT name (first_name, middle_name+, last_name?)>
```

3.2.5 Choices

Sometimes one instance of an element may contain one kind of child, and another instance may contain a different child. This can be indicated with a choice. A choice is a list of element names separated by vertical bars. For example, this declaration says that a `methodResponse` element contains either a `params` child or a `fault` child:

```
<!ELEMENT methodResponse (params | fault)>
```

However, it cannot contain both at once. Each `methodResponse` element must contain one or the other.

Choices can be extended to an indefinite number of possible elements. For example, this declaration says that each `digit` element can contain exactly one of the child elements named `zero`, `one`, `two`, `three`, `four`, `five`, `six`, `seven`, `eight`, or `nine`:

```
<!ELEMENT digit
 (zero | one | two | three | four | five | six | seven | eight | nine)
>
```

3.2.6 Parentheses

Individually, choices, sequences, and suffixes are fairly limited. However, they can be combined in arbitrarily complex fashions to describe most reasonable content models. Either a choice or a sequence can be enclosed in parentheses. When so enclosed, the choice or sequence can be suffixed with a `?`, `*`, or `+`. Furthermore, the parenthesized item can be nested inside other choices or sequences.

For example, let's suppose you want to say that a `circle` element contains a `center` element and either a `radius` or a `diameter` element, but not both. This declaration does that:

```
<!ELEMENT circle (center, (radius | diameter))>
```

To continue with a geometry example, suppose a `center` element can either be defined in terms of Cartesian or polar coordinates. Then each center contains either an `x` and a `y` or an `r` and a   . We would declare this using two small sequences, each of which is parenthesized and combined in a choice:

```
<!ELEMENT center ((x, y) | (r,   ))>
```

Suppose you don't really care whether the `x` element comes before the `y` element or vice versa, nor do you care whether `r` comes before . Then you can expand the choice to cover all four possibilities:

```
<!ELEMENT center ((x, y) | (y, x) | (r,   ) | (  , r) )>
```

As the number of elements in the sequence grows, the number of permutations grows more than exponentially. Thus, this technique really isn't practical past two or three child elements. DTDs are not very good at saying you want n instances of A and m instances of B, but you don't really care which order they come in.

Suffixes can be applied to parenthesized elements too. For instance, let's suppose that a polygon is defined by individual coordinates for each vertex, given in order. For example, this is a right triangle:

What we want to say is that a polygon is composed of three or more pairs of x-y or r- coordinates. An `x` is always followed by a `y`, and an `r` is always followed by a . This declaration does that:

The plus sign is applied to `((x, y) | (r,   ))`.

To return to the name example, suppose you want to say that a name can contain just a first name, just a last name, or a first name and a last name with an indefinite number of middle names. This declaration achieves that:

```
<!ELEMENT name (last_name
               | (first_name, ( (middle_name+, last_name) | (last_name?) )
               ) >
```

3.2.7 Mixed Content

In narrative documents it's common for a single element to contain both child elements and un-marked up, nonwhitespace character data. For example, recall this `definition` element from Chapter 2:

```
<definition>The <term>Turing Machine</term> is an abstract finite
state automaton with infinite memory that can be proven equivalent
```

```
to any any other finite state automaton with arbitrarily large memory.
Thus what is true for a Turing machine is true for all equivalent
machines no matter how implemented.
</definition>
```

The `definition` element contains some nonwhitespace text and a `term` child. This is called mixed content. An element that contains mixed content is declared like this:

```
<!ELEMENT definition (#PCDATA | term)*>
```

This says that a `definition` element may contain parsed character data and `term` children. It does not specify in which order they appear, nor how many instances of each appear. This declaration allows a `definition` to have one `term` child, no `term` children, or twenty-three `term` children.

You can add any number of other child elements to the list of mixed content, though `#PCDATA` must always be the first child in the list. For example, this declaration says that a `paragraph` element may contain any number of `name`, `profession`, `footnote`, `emphasize`, and `date` elements in any order, interspersed with parsed character data:

```
<!ELEMENT paragraph
  (#PCDATA | name | profession | footnote | emphasize | date )*
>
```

This is the only way to indicate that an element contains mixed content. You cannot say, for example, that there must be exactly one `term` child of the `definition` element, as well as parsed character data. You cannot say that the parsed character data must all come after the `term` child. You cannot use parentheses around a mixed-content declaration to make it part of a larger grouping. You can only say that the element contains any number of any elements from a particular list in any order, as well as undifferentiated parsed character data.

3.2.8 Empty Elements

Some elements do not have any content at all. These are called empty elements and are sometimes written with a closing `/>`. For example:

```
<image source="bus.jpg" width="152" height="345"
       alt="Alan Turing standing in front of a bus"
/>
```

These elements are declared by using the keyword `EMPTY` for the content specification. For example:

```
<!ELEMENT image EMPTY>
```

This merely says that the `image` element must be empty, not that it must be written with an empty-element tag. Given this declaration, this is also a valid `image` element:

```
<image source="bus.jpg" width="152" height="345"
       alt="Alan Turing standing in front of a bus"></image>
```

If an element is empty, then it can contain nothing, not even whitespace. For instance, this is an invalid `image` element:

```
<image source="bus.jpg" width="152" height="345"
       alt="Alan Turing standing in front of a bus">
</image>
```

3.2.9 ANY

Very loose DTDs occasionally want to say that an element exists without making any assertions about what it may or may not contain. In this case you can specify the keyword `ANY` as the content specification. For example, this declaration says that a `page` element can contain any content including mixed content, child elements, and even other `page` elements:

```
<!ELEMENT page ANY>
```

The children that actually appear in the `page` elements' content in the document must still be declared in element declarations of their own. `ANY` does not allow you to use undeclared elements.

`ANY` is sometimes useful when you're just beginning to design the DTD and document structure and you don't yet have a clear picture of how everything fits together. However, it's extremely bad form to use `ANY` in finished DTDs. About the only time you'll see it used is when external DTD subsets and entities may change in uncontrollable ways. However, this is actually quite rare. You'd really only need this if you were writing a DTD for an application like XSLT or RDF that wraps content from arbitrary, unknown XML applications.

3.3 Attribute Declarations

As well as declaring its elements, a valid document must declare all the elements' attributes. This is done with `ATTLIST` declarations. A single `ATTLIST` can declare multiple attributes for a single element type. However, if the same attribute is repeated on multiple elements, then it must be declared separately for each element where it appears. (Later in this chapter you'll see how to use parameter entity references to make this repetition less burdensome.)

For example, this `ATTLIST` declaration declares the `source` attribute of the `image` element:

```
<!ATTLIST image source CDATA #REQUIRED>
```

It says that the `image` element has an attribute named `source`. The value of the `source` attribute is character data, and instances of the `image` element in the document are required to provide a value for the `source` attribute.

A single `ATTLIST` declaration can declare multiple attributes for the same element. For example, this `ATTLIST` declaration not only declares the `source` attribute of the `image` element, but also the `width`, `height`, and `alt` attributes:

```
<!ATTLIST image source CDATA #REQUIRED
                width  CDATA #REQUIRED
                height CDATA #REQUIRED
                alt    CDATA #IMPLIED
>
```

This declaration says the `source`, `width`, and `height` attributes are required. However, the `alt` attribute is optional and may be omitted from particular `image` elements. All four attributes are declared to contain character data, the most generic attribute type.

This declaration has the same effect and meaning as four separate ATTLIST declarations, one for each attribute. Whether to use one ATTLIST declaration per attribute is a matter of personal preference, but most experienced DTD designers prefer the multiple-attribute form. Given judicious application of whitespace, it's no less legible than the alternative.

3.3.1 Attribute Types

In merely well-formed XML, attribute values can be any string of text. The only restrictions are that any occurrences of < or & must be escaped as `&lt;` and `&amp;` and whichever kind of quotation mark, single or double, is used to delimit the value must also be escaped. However, a DTD allows you to make somewhat stronger statements about the content of an attribute value. Indeed, these are stronger statements than can be made about the contents of an element. For instance, you can say that an attribute value must be unique within the document, that it must be a legal XML name token, or that it must be chosen from a fixed list of values.

There are ten attribute types in XML. They are:

- CDATA
- NMTOKEN
- NMTOKENS
- Enumeration
- ENTITY
- ENTITIES
- ID
- IDREF
- IDREFS
- NOTATION

These are the only attribute types allowed. A DTD cannot say that an attribute value must be an integer or a date between 1966 and 2002, for example.

3.3.1.1 CDATA

A CDATA attribute value can contain any string of text acceptable in a well-formed XML attribute value. This is the most general attribute type. For example, you would use this type for an `alt` attribute of an `image` element because there's no particular form the text in such an attribute has to follow.

```
<!ATTLIST image alt CDATA #IMPLIED>
```

You would also use this for other kinds of data such as prices, URIs, email and snail mail addresses, citations, and other types that—while they have more structure than a simple string of text—don't match any of the other attribute types. For example:

```
<!ATTLIST sku
 list_price             CDATA #IMPLIED
 suggested_retail_price CDATA #IMPLIED
 actual_price           CDATA #IMPLIED
>
<!-- All three attributes should be in the form $XX.YY -->
```

3.3.1.2 NMTOKEN

An XML name token is very close to an XML name. It must consist of the same characters as an XML name, that is, alphanumeric and/or ideographic characters and the punctuation marks _, -, ., and :. Furthermore, like an XML name, an XML name token may not contain whitespace. However, a name token differs from an XML name in that any of the allowed characters can be the first character in a name token, while only letters, ideographs, and the underscore can be the first character of an XML name. Thus 12 and .cshrc are valid XML name tokens although they are not valid XML names. Every XML name is an XML name token, but not all XML name tokens are XML names.

The value of an attribute declared to have type `NMTOKEN` is an XML name token. For example, if you knew that the `year` attribute of a `journal` element should contain an integer such as 1990 or 2015, you might declare it to have `NMTOKEN` type, since all years are name tokens:

```
<!ATTLIST journal year NMTOKEN #REQUIRED>
```

This still doesn't prevent the document author from assigning the `year` attribute values like "99" or "March", but it at least eliminates some possible wrong values, especially those that contain whitespace such as "1990 C.E." or "Sally had a little lamb."

3.3.1.3 NMTOKENS

A `NMTOKENS` type attribute contains one or more XML name tokens separated by whitespace. For example, you might use this to describe the `dates` attribute of a `performances` element, if the dates were given in the form 08-26-2000, like this:

```
<performances dates="08-21-2001 08-23-2001 08-27-2001">
  Kat and the Kings
</performances>
```

The appropriate declaration is:

```
<!ATTLIST performances dates NMTOKENS #REQUIRED>
```

On the other hand, you could not use this for a list of dates in the form 08/27/2001 because the forward slash is not a legal name character.

3.3.1.4 Enumeration

An enumeration is the only attribute type that is not an XML keyword. Rather, it is a list of all possible values for the attribute, separated by vertical bars. Each possible value must be an XML name token. For example, the following declarations say that the value of the `month` attribute of a `date` element must be one of the twelve English month names, that the value of the `day` attribute must be a number between 1 and 31, and that the value of the `year` attribute must be an integer between 1970 and 2009:

```
<!ATTLIST date month (January | February | March | April | May | June
  | July | August | September | October | November | December) #REQUIRED
>
<!ATTLIST date day (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
  | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25
  | 26 | 27 | 28 | 29 | 30 | 31) #REQUIRED
>
<!ATTLIST date year (1970 | 1971 | 1972 | 1973 | 1974 | 1975 | 1976
  | 1977 | 1978 | 1979 | 1980 | 1981 | 1982 | 1983 | 1984 | 1985 | 1986
  | 1987 | 1988 | 1989 | 1990 | 1991 | 1992 | 1993 | 1994 | 1995 | 1996
  | 1997 | 1998 | 1999 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006
  | 2007 | 2008 | 2009 ) #REQUIRED
>
<!ELEMENT date EMPTY>
```

Given this DTD, this `date` element is valid:

```
<date month="January" day="22" year="2001"/>
```

However, these `date` elements are invalid:

```
<date month="01"      day="22" year="2001"/>
<date month="Jan"     day="22" year="2001"/>
<date month="January" day="02" year="2001"/>
<date month="January" day="2"  year="1969"/>
<date month="Janvier" day="22" year="2001"/>
```

This trick works here because all the desired values happen to be legal XML name tokens. However, we could not use the same trick if the possible values included whitespace or any punctuation besides the underscore, hyphen, colon, and period.

3.3.1.5 ID

An `ID` type attribute must contain an XML name (not a name token but a name) that is unique within the XML document. More precisely, no other `ID` type attribute in the document can have the same value. (Attributes of non-`ID` type are not considered.) Each element may have no more than one `ID` type attribute.

As the keyword suggests, `ID` type attributes assign unique identifiers to elements. `ID` type attributes do not have to have the name "ID" or "id", though they very commonly do. For example, this `ATTLIST` declaration says that every `employee` element must have a `social_security_number` ID attribute:

```
<!ATTLIST employee social_security_number ID #REQUIRED>
```

ID numbers are tricky because a number is not an XML name and therefore not a legal XML `ID`. The normal solution is to prefix the values with an underscore or a common letter. For example:

```
<employee social_security_label="_078-05-1120"/>
```

3.3.1.6 IDREF

An `IDREF` type attribute refers to the `ID` type attribute of some element in the document. Thus, it must be an XML name. `IDREF` attributes are commonly used to establish relationships between elements when simple containment won't suffice.

For example, imagine an XML document that contains a list of `project` elements and `employee` elements. Every `project` has a `project_id` ID type attribute, and every `employee` has a `social_security_number` ID type attribute. Furthermore, each `project` has `team_member` child elements that identify who's working on the project, and each `employee` element has `assignment` children that indicate to which projects that employee is assigned. Since each project is assigned to multiple employees and some employees are assigned to more than one project, it's not possible to make the employees children of the projects or the projects children of the employees. The solution is to use IDREF type attributes like this:

```
<project project_id="p1">
  <goal>Develop Strategic Plan</goal>
  <team_member person="ss078-05-1120"/>
  <team_member person="ss987-65-4320"/>
</project>
<project project_id="p2">
  <goal>Deploy Linux</goal>
  <team_member person="ss078-05-1120"/>
  <team_member person="ss9876-12-3456"/>
</project>
<employee social_security_label="ss078-05-1120">
  <name>Fred Smith</name>
  <assignment project_id="p1"/>
  <assignment project_id="p2"/>
</employee>
<employee social_security_label="ss987-65-4320">
  <name>Jill Jones</name>
  <assignment project_id="p1"/>
</employee>
<employee social_security_label="ss9876-12-3456">
  <name>Sydney Lee</name>
  <assignment project_id="p2"/>
</employee>
```

In this example, the project_id attribute of the project element and the social_security_number attribute of the employee element would be declared to have type ID. The person attribute of the team_member element and the project_id attribute of the assignment element would have type IDREF. The relevant ATTLIST declarations look like this:

```
<!ATTLIST employee social_security_number ID    #REQUIRED>
<!ATTLIST project  project_id             ID    #REQUIRED>
<!ATTLIST team_member person              IDREF #REQUIRED>
<!ATTLIST assignment  project_id          IDREF #REQUIRED>
```

These declarations constrain the person attribute of the team_member element and the project_id attribute of the assignment element to match the ID of something in the document. However, they do not constrain the person attribute of the team_member element to match only employee IDs or constrain the project_id attribute of the assignment element to match only project IDs. It would be valid (though not necessarily correct) for a team_member to hold the ID of another project or even the same project.

3.3.1.7 IDREFS

An IDREFS type attribute contains a whitespace-separated list of XML names, each of which must be the ID of an element in the document. This is used when one element needs to refer to multiple other elements. For instance, the previous project example could be rewritten so that the assignment children of the employee element were replaced by a single assignments attribute. Similarly, the team_member children of the project element could be replaced by a team attribute like this:

```
<project project_id="p1" team="ss078-05-1120 ss987-65-4320">
  <goal>Develop Strategic Plan</goal>
</project>
```

```
<project project_id="p2" team="ss078-05-1120 ss9876-12-3456">
  <goal>Deploy Linux</goal>
</project>
<employee social_security_label="ss078-05-1120" assignments="p1 p2">
  <name>Fred Smith</name>
</employee>
<employee social_security_label="ss987-65-4320" assignments="p1">
  <name>Jill Jones</name>
</employee>
<employee social_security_label="ss9876-12-3456" assignments="p2">
  <name>Sydney Lee</name>
</employee>
```

The appropriate declarations are:

```
<!ATTLIST employee social_security_number ID     #REQUIRED
                   assignments            IDREFS #REQUIRED>
<!ATTLIST project  project_id             ID     #REQUIRED
                   team                   IDREFS #REQUIRED>
```

3.3.1.8 ENTITY

An `ENTITY` type attribute contains the name of an unparsed entity declared elsewhere in the DTD. For instance, a `movie` element might have an entity attribute identifying the MPEG or QuickTime file to play when the movie was activated:

```
<!ATTLIST movie source ENTITY #REQUIRED>
```

If the DTD declared an unparsed entity named `X-Men-trailer`, then this `movie` element might be used to embed that video file in the XML document:

```
<movie source="X-Men-trailer"/>
```

We'll discuss unparsed entities in more detail later in this chapter.

3.3.1.9 ENTITIES

An `ENTITIES` type attribute contains the names of one or more unparsed entities declared elsewhere in the DTD, separated by whitespace. For instance, a `slide_show` element might have an `ENTITIES` attribute identifying the JPEG files to show and in which order they were to be shown:

```
<!ATTLIST slide_show slides ENTITIES #REQUIRED>
```

If the DTD declared unparsed entities named `slide1`, `slide2`, `slide3`, and so on through `slide10`, then this `slide_show` element might be used to embed the show in the XML document:

```
<slide_show slides="slide1 slide2 slide3 slide4 slide5 slide6
                    slide7 slide8 slide9 slide10"/>
```

3.3.1.10 NOTATION

A `NOTATION` type attribute contains the name of a notation declared in the document's DTD. This is perhaps the rarest attribute type and isn't much used in practice. In theory, it could be used to associate types with particular elements, as well as limiting the types associated with the element.

For example, these declarations define four notations for different image types and then specify that each `image` element must have a `type` attribute that selects exactly one of them:

```
<!NOTATION gif  SYSTEM "image/gif">
<!NOTATION tiff SYSTEM "image/tiff">
<!NOTATION jpeg SYSTEM "image/jpeg">
<!NOTATION png  SYSTEM "image/png">
<!ATTLIST  image type NOTATION (gif | tiff | jpeg | png) #REQUIRED>
```

The `type` attribute of each `image` element can have one of the four values `gif`, `tiff`, `jpeg`, or `png` but not any other value. This has a slight advantage over the enumerated type in that the actual MIME media type of the notation is available, whereas an enumerated type could not specify `image/png` or `image/gif` as an allowed value because the forward slash is not a legal character in XML names.

3.3.2 Attribute Defaults

As well as providing a data type, each `ATTLIST` declaration includes a default declaration for that attribute. There are four possibilities for this default:

#IMPLIED

> The attribute is optional. Each instance of the element may or may not provide a value for the attribute. No default value is provided.

#REQUIRED

> The attribute is required. Each instance of the element must provide a value for the attribute. No default value is provided.

#FIXED

> The attribute value is constant and immutable. This attribute has the specified value regardless of whether the attribute is explicitly noted on an individual instance of the element. If it is included, though, it must have the specified value.

Literal

> The actual default value is given as a quoted string.

For example, this `ATTLIST` declaration says that `person` elements can have but do not have to have `born` and `died` attributes:

```
<!ATTLIST person born CDATA #IMPLIED
                 died CDATA #IMPLIED
>
```

This `ATTLIST` declaration says that every `circle` element must have `center_x`, `center_y`, and `radius` attributes:

```
<!ATTLIST circle center_x NMTOKEN #REQUIRED
                 center_y NMTOKEN #REQUIRED
                 radius   NMTOKEN #REQUIRED
```

```
>
```

This `ATTLIST` declaration says that every `biography` element has an `xmlns:xlink` attribute and that the value of that attribute is `http://www.w3.org/1999/xlink`, even if the start-tag of the element does not explicitly include an `xmlns:xlink` attribute.

```
<!ATTLIST biography
   xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink">
```

This `ATTLIST` declaration says that every `web_page` element has a `protocol` attribute. If a particular `web_page` element doesn't have an explicit `protocol` attribute, then the parser will supply one with the value `http`:

```
<!ATTLIST web_page protocol NMTOKEN "http">
```

3.4 General Entity Declarations

As you learned in Chapter 2, XML predefines five entities for your convenience:

&lt;

The less-than sign; a.k.a. the opening angle bracket (<)

&amp;

The ampersand (&)

&gt;

The greater-than sign; a.k.a. the closing angle bracket (>)

&quot;

The straight, double quotation marks (")

&apos;

The apostrophe; a.k.a. the straight single quote (')

The DTD can define many more. This is useful not just in valid documents, but even in documents you don't plan to validate.

Entity references are defined with an `ENTITY` declaration in the DTD. This gives the name of the entity, which must be an XML name, and the replacement text of the entity. For example, this entity declaration defines `&super;` as an abbreviation for supercalifragilisticexpialidocious:

```
<!ENTITY super "supercalifragilisticexpialidocious">
```

Once that's done, you can use `&super;` anywhere you'd normally have to type the entire word (and probably misspell it).

Entities can contain markup as well as text. For example, this declaration defines `&footer;` as an abbreviation for a standard web-page footer that will be repeated on many pages:

```
<!ENTITY footer '<hr size="1" noshade="true"/>
<font CLASS="footer">
<a href="index.html">O&apos;Reilly Home</a> |
<a href="sales/bookstores/">O&apos;Reilly Bookstores</a> |
<a href="order_new/">How to Order</a> |
<a href="oreilly/contact.html">O&apos;Reilly Contacts</a><br>
<a href="http://international.oreilly.com/">International</a> |
<a href="oreilly/about.html">About O&apos;Reilly</a> |
<a href="affiliates.html">Affiliated Companies</a>
</font>
<p>
<font CLASS="copy">
Copyright 2000, O&apos;Reilly &amp; Associates, Inc.<br/>
<a href="mailto:webmaster@oreilly.com">webmaster@oreilly.com</a>
</font>
</p>
'>
```

The entity replacement text must be well-formed. For instance, you cannot put a start-tag in one entity and the corresponding end-tag in another entity.

The other thing you have to be careful about is that you need to use different quote marks inside the replacement text from the ones that delimit it. Here we've chosen single quotes to surround the replacement text and double quotes internally. However, we did have to change the single quote in "O'Reilly" to the predefined general entity reference `&apos;`. Replacement text may itself contain entity references that are resolved before the text is replaced. However, self-referential and circular references are forbidden.

General entities insert replacement text into the body of an XML document. They can also be used inside the DTD in places where they will eventually be included in the body of an XML document, for instance in an attribute default value or in the replacement text of another entity. However, they cannot be used to provide the text of the DTD itself. For instance, this is illegal:

Shortly, we'll see how to use a different kind of entity—the parameter entity—to achieve the desired result.

3.5 External Parsed General Entities

The footer example is about at the limits of what you can comfortably fit in a DTD. In practice, web sites prefer to store repeated content like this in external files and load it into their pages using PHP, server-side includes, or some similar mechanism. XML supports this technique through external general entity references, though in this case the client, rather than the server, is responsible for integrating the different pieces of the document into a coherent whole.

An external parsed general entity reference is declared in the DTD using an `ENTITY` declaration. However, instead of the actual replacement text, the `SYSTEM` keyword and a URI to the replacement text is given. For example:

```
<!ENTITY footer SYSTEM "http://www.oreilly.com/boilerplate/footer.xml">
```

Of course, a relative URL will often be used instead. For example:

```
<!ENTITY footer SYSTEM "/boilerplate/footer.xml">
```

In either case when the general entity reference `&footer;` is seen in the character data of an element, the parser may replace it with the document found at http://www.oreilly.com/boilerplate/footer.xml. References to external parsed entities are not allowed in attribute values. Most of the time, this shouldn't be too big a hassle because attribute values tend to be small enough to be easily included in internal entities.

Notice we wrote that the parser may replace the entity reference with the document at the URL, not that it must. This is an area where parsers have some leeway in just how much of the XML specification they wish to implement. A validating parser must retrieve such an external entity. However, a nonvalidating parser may or may not choose to retrieve the entity.

Furthermore, not all text files can serve as external entities. In order to be loaded in by a general entity reference, the document must be potentially well-formed when inserted into an existing document. This does not mean the external entity itself must be well-formed. In particular, the external entity might not have a single root element. However, if such a root element were wrapped around the external entity, then the resulting document should be well-formed. This means, for example, that all elements that start inside the entity must finish inside the same entity. They cannot finish inside some other entity. Furthermore, the external entity does not have a prolog and, therefore, cannot have an XML declaration or a document type declaration.

3.5.1 Text Declarations

Instead of an XML declaration, an external entity may have a text declaration; this looks a lot like an XML declaration. The main difference is that in a text declaration the encoding declaration is required, while the version info is optional. Furthermore, there is no standalone declaration. The main purpose of the text declaration is to warn the parser if the external entity uses a different text encoding than the including document. For example, this is a common text declaration:

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

However, you could also use this text declaration with no version attribute:

```
<?xml encoding="MacRoman"?>
```

Example 3-5 is a well-formed external entity that could be included from another document using an external general entity reference.

Example 3-5. An external parsed entity

```
<?xml encoding="ISO-8859-1"?>
<hr size="1" noshade="true"/>
<font CLASS="footer">
  <a href="index.html">O'Reilly Home</a> |
  <a href="sales/bookstores/">O'Reilly Bookstores</a> |
  <a href="order_new/">How to Order</a> |
  <a href="oreilly/contact.html">O'Reilly Contacts</a><br>
  <a href="http://international.oreilly.com/">International</a> |
  <a href="oreilly/about.html">About O'Reilly</a> |
  <a href="affiliates.html">Affiliated Companies</a>
</font>
<p>
  <font CLASS="copy">
    Copyright 2000, O'Reilly &amp; Associates, Inc.<br/>
    <a href="mailto:webmaster@oreilly.com">webmaster@oreilly.com</a>
  </font>
</p>
```

3.6 External Unparsed Entities and Notations

Not all data is XML. There are a lot of ASCII text files in the world that don't give two cents about escaping < as `&lt;` or adhering to the other constraints by which an XML document is limited. There are probably even more JPEG photographs, GIF line art, QuickTime movies, MIDI sound files, and so on. None of these are well-formed XML, yet all of them are necessary components of many documents.

The mechanism that XML suggests for embedding these things in your documents is the external unparsed entity. The DTD specifies a name and a URI for the entity containing the non-XML data. For example, this `ENTITY` declaration associates the name `turing_getting_off_bus` with the JPEG image at http://www.turing.org.uk/turing/pi1/bus.jpg:

```
<!ENTITY turing_getting_off_bus
         SYSTEM "http://www.turing.org.uk/turing/pi1/bus.jpg"
         NDATA jpeg>
```

3.6.1 Notations

Since the data is not in XML format, the `NDATA` declaration specifies the type of the data. Here the name `jpeg` is used. XML does not recognize this as meaning an image in a format defined by the Joint Photographs Experts Group. Rather this is the name of a notation declared elsewhere in the DTD using a `NOTATION` declaration like this:

```
<!NOTATION jpeg SYSTEM "image/jpeg">
```

Here we've used the MIME media type image/jpeg as the external identifier for the notation. However, there is absolutely no standard or even a suggestion for exactly what this identifier should be. Individual applications must define their own requirements for the contents and meaning of notations.

3.6.2 Embedding Unparsed Entities in Documents

The DTD only declares the existence, location, and type of the unparsed entity. To actually include the entity in the document at one or more locations, you insert an element with an `ENTITY` type attribute whose value is the name of an unparsed entity declared in the DTD. You do not use an entity reference like `&turing_getting_off_bus;`. Entity references can only refer to parsed entities.

Suppose the `image` element and its `source` attribute are declared like this:

```
<!ELEMENT image EMPTY>
<!ATTLIST image source ENTITY #REQUIRED>
```

Then, this `image` element would refer to the photograph at http://www.turing.org.uk/turing/pi1/bus.jpg:

```
<image source="turing_getting_off_bus"/>
```

We should warn you that XML doesn't guarantee any particular behavior from an application that encounters this type of unparsed entity. It very well may not display the image to the user. Indeed, the parser may be running in an environment where there's no user to display the image to. It may not even understand that this is an image. The parser may not load or make any sort of connection with the server where the actual image resides. At most, it will tell the application on whose behalf it's parsing that there is an unparsed entity at a particular URI with a particular notation and let the application decide what, if anything, it wants to do with that information.

> Unparsed general entities are not the only plausible way to embed non-XML content in XML documents. In particular, a simple URL, possibly associated with an XLink, does a fine job for many purposes, just as it does in HTML (which gets along just fine without any unparsed entities). Including all the necessary information in a single empty element like `<image source = "http://www.turing.org.uk/turing/pi1/bus.jpg" />` is arguably preferable to splitting the same information between the element where it's used and the DTD of the document in which it's used. The only thing an unparsed entity really adds is the notation, but that's too nonstandard to be of much use.
>
> In fact, many experienced XML developers, including the authors of this book, feel strongly that unparsed entities are a complicated, confusing mistake that should never have been included in XML in the first place. Nonetheless, they are a part of the specification, so we describe them here.

3.6.3 Notations for Processing Instruction Targets

Notations can also be used to identify the exact target of a processing instruction. A processing instruction target must be an XML name, which means it can't be a full path like */usr/local/bin/tex*. A notation can identify a short XML name like `tex` with a more complete specification of the external program for which the processing instruction is intended. For example, this notation associates the target `tex` with the more complete path */usr/local/bin/tex*:

```
<!NOTATION tex SYSTEM "/usr/local/bin/tex">
```

In practice, this technique isn't much used or needed. Most applications that read XML files and pay attention to particular processing instructions simply recognize a particular target string like `php` or `robots`.

3.7 Parameter Entities

It is not uncommon for multiple elements to share all or part of the same attribute lists and content specifications. For instance, any element that's a simple XLink will have `xlink:type` and `xlink:href` attributes, and perhaps `xlink:show` and `xlink:actuate` attributes. In XHTML, a `th` element and a `td` element contain more or less the same content. Repeating the same content specifications or attribute lists in multiple element declarations is tedious and error-prone. It's entirely possible to add a newly defined child element to the declaration of some of the elements but forget to include it in others.

For example, consider an XML application for residential real-estate listings that provides separate elements for apartments, sublets, coops for sale, condos for sale, and houses for sale. The element declarations might look like this:

```
<!ELEMENT apartment (address, footage, rooms, baths, rent)>
<!ELEMENT sublet    (address, footage, rooms, baths, rent)>
<!ELEMENT coop      (address, footage, rooms, baths, price)>
<!ELEMENT condo     (address, footage, rooms, baths, price)>
<!ELEMENT house     (address, footage, rooms, baths, price)>
```

There's a lot of overlap between the declarations, i.e., a lot of repeated text. And if you later decide you need to add an additional element, `available_date` for instance, then you need to add it to all five declarations. It would be preferable to define a constant that can hold the common parts of the content specification for all five kinds of listings and refer to that constant from inside the content specification of each element. Then to add or delete something from all the listings, you'd only need to change the definition of the constant.

An entity reference is the obvious candidate here. However, general entity references are not allowed to provide replacement text for a content specification or attribute list, only for parts of the DTD that will be included in the XML document itself. Instead, XML provides a new construct exclusively for use inside DTDs, the parameter entity, which is referred to by a parameter entity reference. Parameter entities behave like and are declared almost exactly like a general entity. However, they use a `%` instead of an `&`, and they can only be used in a DTD while general entities can only be used in the document content.

3.7.1 Parameter Entity Syntax

A parameter entity reference is declared much like a general entity reference. However, an extra percent sign is placed between the `<!ENTITY` and the name of the entity. For example:

```
<!ENTITY % residential_content "address, footage, rooms, baths">
<!ENTITY % rental_content      "rent">
<!ENTITY % purchase_content    "price">
```

Parameter entities are dereferenced in the same way as a general entity reference, only with a percent sign instead of an ampersand:

```
<!ELEMENT apartment (%residential_content;, %rental_content;)>
<!ELEMENT sublet    (%residential_content;, %rental_content;)>
<!ELEMENT coop      (%residential_content;, %purchase_content;)>
<!ELEMENT condo     (%residential_content;, %purchase_content;)>
<!ELEMENT house     (%residential_content;, %purchase_content;)>
```

When the parser reads these declarations, it substitutes the entity's replacement text for the entity reference. Now all you have to do to add an `available_date` element to the content specification of all five listing types is add it to the `residential_content` entity like this:

```
<!ENTITY % residential_content "address, footage, rooms,
                                baths, available_date">
```

The same technique works equally well for attribute types and element names. You'll see several examples of this in the next chapter on namespaces and in Chapter 9.

This trick is limited to external DTDs. Internal DTD subsets do not allow parameter entity references to be only part of a markup declaration. However, parameter entity references can be used in internal DTD subsets to insert one or more entire markup declarations, typically through external parameter entities.

3.7.2 Redefining Parameter Entities

What makes parameter entity references particularly powerful is that they can be redefined. If a document uses both internal and external DTD subsets, then the internal DTD subset can specify new replacement text for the entities. If `ELEMENT` and `ATTLIST` declarations in the external DTD subset are written indirectly with parameter entity references instead of directly with literal element names, the internal DTD subset can change the DTD for the document. For instance, a single document could add a `bedrooms` child element to the listings by redefining the `residential_content` entity like this:

```
<!ENTITY % residential_content "address, footage, rooms,
                                bedrooms, baths, available_date">
```

In the event of conflicting entity declarations, the first one encountered takes precedence. The parser reads the internal DTD subset first. Thus the internal definition of the `residential_content` entity is used. When the parser reads the external DTD subset, every declaration that uses the `residential_content` entity will contain a `bedrooms` child element it wouldn't otherwise have.

Modular XHTML, which we'll discuss in Chapter 7, makes heavy use of this technique to allow particular documents to select only the subset of HTML that they actually need.

3.7.3 External DTD Subsets

Real-world DTDs can be quite complex. The SVG DTD is over 1,000 lines long. The XHTML 1.0 strict DTD (the smallest of the three XHTML DTDs) is more than 1,500 lines long. And these are only medium-sized DTDs. The DocBook XML DTD is over 11,000 lines long. It can be hard to work with, comprehend, and modify such a large DTD when it's stored in a single monolithic file.

Fortunately, DTDs can be broken up into independent pieces. For instance, the DocBook DTD is distributed in 28 separate pieces covering different parts of the spec: one for tables, one for notations, one for entity declarations, and so on. These different pieces are then combined at validation time using external parameter entity references.

An external parameter entity is declared using a normal `ENTITY` declaration with a % sign just like a normal parameter entity. However, rather than including the replacement text directly, the declaration contains the `SYSTEM` keyword followed by a URI to the DTD piece it wants to include. For example, the following `ENTITY` declaration defines an external entity called names whose content is taken from the file at the relative URI names.dtd. Then the parameter entity reference `%names;` inserts the contents of that file into the current DTD.

```
<!ENTITY % names SYSTEM "names.dtd">
%names;
```

You can use either relative or absolute URIs as convenient. In most situations, relative URIs are more practical.

3.8 Conditional Inclusion

XML offers the `IGNORE` directive for the purpose of "commenting out" a section of declarations. For example, a parser will ignore the following declaration of a `production_note` element, as if it weren't in the DTD at all:

```
<![IGNORE[
  <!ELEMENT production_note (#PCDATA)>
]]>
```

This may not seem particularly useful. After all, you could always simply use an XML comment to comment out the declarations you want to remove temporarily from the DTD. If you feel that way, the `INCLUDE` directive is going to seem even more pointless. Its purpose is to indicate that the given declarations are actually used in the DTD. For example:

```
<![INCLUDE[
  <!ELEMENT production_note (#PCDATA)>
]]>
```

This has exactly the same effect and meaning as if the `INCLUDE` directive were not present. However, now consider what happens if we don't use `INCLUDE` and `IGNORE` directly. Instead, suppose we define a parameter entity like this:

```
<!ENTITY % notes_allowed "INCLUDE">
```

Then we use a parameter entity reference instead of the keyword:

```
<![%notes_allowed;[
  <!ELEMENT production_note (#PCDATA)>
]]>
```

The `notes_allowed` parameter entity can be redefined from outside this DTD. In particular, it can be redefined in the internal DTD subset of a document. This provides a switch individual documents can use to turn the `production_note` declaration on or off. This technique allows document authors to select only the functionality they need from the DTD.

3.9 Two DTD Examples

Some of the best techniques for DTD design only become apparent when you look at larger documents. In this section, we'll develop DTDs that cover the two different document formats for describing people that were presented in Example 2-4 and Example 2-5 of the last chapter.

3.9.1 Data-Oriented DTDs

Data- oriented DTDs are very straightforward. They make heavy use of sequences, occasional use of choices, and almost no use of mixed content. Example 3-6 shows such a DTD. Since this is a small example, and since it's easier to understand when both the document and the DTD are on the same page, we've made this an internal DTD included in the document. However, it would be easy to take it out and put it in a separate file.

Example 3-6. A flexible yet data-oriented DTD describing people

```
<?xml version="1.0"?>
<!DOCTYPE person  [
  <!ELEMENT person (name+, profession*)>
  <!ELEMENT name EMPTY>
  <!ATTLIST name first CDATA #REQUIRED
                 last  CDATA #REQUIRED>
  <!-- The first and last attributes are required to be present
       but they may be empty. For example,
       <name first="Cher" last=""> -->
  <!ELEMENT profession EMPTY>
  <!ATTLIST profession value CDATA #REQUIRED>
]>
<person>
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

The DTD here is contained completely inside the internal DTD subset. First a person `ELEMENT` declaration states that each `person` must have one or more `name` children, and zero or more `profession` children, in that order. This allows for the possibility that a person changes his name or uses aliases. It assumes that each person has at least one name but may not have a profession.

This declaration also requires that all `name` elements precede all `profession` elements. Here the DTD is less flexible than it ideally would be. There's no particular reason that the names have to come first. However, if we were to allow more random ordering, it would be hard to say that there must be at least one `name`. One of the weaknesses of DTDs is that it occasionally forces extra sequence order on you when all you really need is a constraint on the number of some element. Schemas are more flexible in this regard.

Both `name` and `profession` elements are empty so their declarations are very simple. The attribute declarations are a little more complex. In all three cases the form of the attribute is open, so all three attributes are declared to have type `CDATA`. All three are also required. However, note the use of comments to suggest a solution for edge cases such as celebrities with no last names. Comments are an essential tool for making sense of otherwise obfuscated DTDs.

3.9.2 Narrative-Oriented DTDs

Narrative-oriented DTDs tend be a lot looser and make much heavier use of mixed content than do DTDs that describe more database-like documents. Consequently, they tend to be written from the bottom up, starting with the smallest elements and building up to the largest. They also tend to use parameter entities to group together similar content specifications and attribute lists.

Example 3-7 is a standalone DTD for biographies like the one shown in Example 2-5 of the last chapter. Notice that not everything it declares is actually present in Example 2-5. That's often the case with narrative documents. For instance, not all web pages contain unordered lists, but the XHTML DTD still needs to declare the `ul` element for those XHTML documents that do include them. Also, notice that a few attributes present in Example 2-5 have been made into fixed defaults here. Thus, they could be omitted from the document itself, once it is attached to this DTD.

Example 3-7. A narrative-oriented DTD for biographies

```
<!ATTLIST biography xmlns:xlink CDATA #FIXED
                                       "http://www.w3.org/1999/xlink">

<!ELEMENT person (first_name, last_name)>
<!-- Birth and death dates are given in the form yyyy/mm/dd -->
<!ATTLIST person born CDATA #IMPLIED
                 died CDATA #IMPLIED>

<!ELEMENT date   (month, day, year)>
<!ELEMENT month  (#PCDATA)>
<!ELEMENT day    (#PCDATA)>
<!ELEMENT year   (#PCDATA)>

<!-- xlink:href must contain a URI.-->
<!ATTLIST emphasize xlink:type (simple) #IMPLIED
                    xlink:href CDATA   #IMPLIED>

<!ELEMENT profession (#PCDATA)>
<!ELEMENT footnote   (#PCDATA)>

<!-- The source is given according to the Chicago Manual of Style
     citation conventions -->
<!ATTLIST footnote source CDATA #REQUIRED>

<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name  (#PCDATA)>

<!ELEMENT image EMPTY>
<!ATTLIST image source CDATA   #REQUIRED
                width  NMTOKEN #REQUIRED
                height NMTOKEN #REQUIRED
                ALT    CDATA   #IMPLIED
>
<!ENTITY % top_level "( #PCDATA | image | paragraph | definition
                      | person | profession | emphasize | last_name
                      | first_name | footnote | date )*">

<!ELEMENT paragraph  %top_level; >
<!ELEMENT definition %top_level; >
<!ELEMENT emphasize  %top_level; >
<!ELEMENT biography  %top_level; >
```

The root `biography` element has a classic mixed-content declaration. Since there are several elements that can contain other elements in a fairly unpredictable fashion, we group all the possible top-level elements (elements that appear as immediate children of the root element) in a single `top_level` entity reference. Then we can make all of them potential children of each other in a straightforward way. This also makes it much easier to add new elements in the future. That's important since this one small example is almost certainly not broad enough to cover all possible biographies.

3.10 Locating Standard DTDs

DTDs and validity are most important when you're exchanging data with others ; they let you verify that you're sending what the receiver expects and vice versa. Of course, this works best if both ends of a conversation agree on which DTD and vocabulary they will use. There are many standard DTDs for different professions and disciplines and more are created every day. It is often better to use an established DTD and vocabulary than to design your own.

However, there is no agreed-upon, central repository that documents and links to such efforts. We know of at least three attempts to create such a central repository. These are:

- James Tauber's schema.net at http://www.schema.net/
- OASIS's xml.org at http://www.xml.org/xml/registry.jsp
- Microsoft's Biztalk initiative at http://www.biztalk.org (registration required)

However, none of these has succeeded in establishing itself as the standard place to list DTDs and none of them cover more than a minority of the existing public DTDs. Indeed, probably the largest list of DTDs online does not even attempt to be a general repository, instead being simply the result of collecting XML and SGML news for several years. This is Robin Cover's list of XML applications at http://www.oasis-open.org/cover/siteIndex.html#toc-applications.

The W3C is one of the most prolific producers of standard XML DTDs. It has moved almost all of its future development to XML including SVG, the Platform for Internet Content Selection (PICS), the Resource Description Framework (RDF), the Mathematical Markup Language (MathML), and even HTML itself. DTDs for these XML applications are generally published as appendixes to the specifications for the applications. The specifications are all found at http://www.w3.org/TR/.

However, XML isn't just for the Web, and far more activity is going on outside the W3C than inside it. Generally, within any one field, you should look to that field's standards bodies for DTDs relating to that area of interest. For example, the American Institute of Certified Public Accountants has published a DTD for XFRML, the Extensible Financial Reporting Markup Language. The Object Management Group (OMG) has published a DTD for describing Unified Modeling Language (UML) diagrams in XML. The Society of Automotive Engineers has published an XML application for emissions information as required by the 1990 U.S. Clean Air Act. Chances are that in any industry that makes heavy use of information technology, some group or groups, either formal or informal, are already working on DTDs that cover parts of that industry.

[1] The document type *declaration* and the document type *definition* are two different things. The abbreviation *DTD* is properly used only to refer to the document type *definition*.