Just enough XML

Introductory Discussion

- Elements
- Character set
- Entities
- Markup
- Document types
In this chapter we will explore the fundamental concepts of XML documents and XML systems. If XML were a great work of literature then this chapter would be the Cliff notes. The chapter will introduce the ideas that define the language but will avoid the nitty-gritty details (the syntax) behind the constructs. As a result, some concepts may remain slightly fuzzy because you will not be able to work with them “hands on”. Later chapters will provide that opportunity.

This early presentation of these ideas will allow you to see XML’s “big picture”. We will do this by walking through the design process for an XML-like language. Hopefully by the end of the process, you will understand each of the design decisions and XML’s overall architecture.

Our objective is to equip you with “just enough” XML to appreciate the application scenarios and tool descriptions in the following parts of the book, but being over-achievers we may go a little too far. Feel free to leave at any time to read about XML in the real world.
First we should summarize what we are trying to achieve. In short, “What is XML used for?” XML is for the digital representation of documents. You probably have an intuitive feel for what a document is. We will work from your intuition.

Documents can be large and small. Both a multi-volume encyclopedia and a memo can be thought of as documents. A particular volume of the encyclopedia can also be called a document. XML allows you to think of the encyclopedia whichever way will allow you to get your job done most efficiently. You’ll notice that XML will give you these sorts of options in many places. XML also allows us to think of an email message as a document. XML can even represent the message from a police department’s server to a police officer’s handheld computer that reports that you have unpaid parking tickets.

When we say that we want to digitally represent documents we mean that we want to put them in some kind of computer-readable notation so that a computer can help us store, process, search, transmit, display and print them. In order for a computer to do useful things with a document, we are going to have to tell it about the structure of the document. This is our simple goal: to represent the documents in a way that the computer can “understand”, insofar as computers can understand anything.

XML documents can include pictures, movies and other multimedia, but we will not actually represent the multimedia components as XML. If you think of representation as a translation process, similar to language translation, then the multimedia components are the parts that we will leave in their “native language” because they have no simple translation into the “target language” (XML). We will just include them in their native formats as you might include a French or Latin phrase in an English text without explicit translation. Most pictures on the Web are files in formats called GIF or JPEG and most movies are in a format called MPEG. An XML document would just refer to those files in their native GIF, JPEG or MPEG formats. If you were transcribing an existing print document into XML, you would most likely represent the character-text parts as XML and the graphical parts in these other formats.

1. Sorry about that.
2.2 | Elements: The logical structure

Before we can describe exactly how we are going to represent documents, we must have a model in our heads of how a document is structured. Most documents (for example books and magazines) can be broken down into components (chapters and articles). These can also be broken down into components (titles, paragraphs, figures and so forth). It turns out that just about every document can be viewed this way.

In XML, these components are called elements. Each element represents a logical component of a document. Elements can contain other elements and can also contain the words and sentences that you would usually think of as the text of the document. XML calls this text the document’s character data. This hierarchical view of XML documents is demonstrated in Figure 2-1.

Markup professionals call this the tree structure of the document. The element that contains all of the others (e.g. book, report or memo) is known as the root element. This name captures the fact that it is the only element that does not “hang” off of some other element.

The elements that are contained in the root are called its subelements. They may contain subelements themselves. If they do, we will call them branches. If they do not, we will call them leaves.

Thus, the chapter and article elements are branches (because they have subelements), but the paragraph and title elements are leaves (because they only contain character data). The root element is also referred to as the document element because it holds the entire logical document within it. The terms root element and document element are interchangeable.

Elements can also have extra information attached to them called attributes. Attributes describe properties of elements. For instance a CIA-record element might have a security attribute that gives the security rating for that element. A CIA database might only release certain records to certain people depending on their security rating. It will not always be clear which aspects of a document should be represented with elements and which should be represented with attributes, but we will give some guidelines in Chapter 54, “Creating a document type definition”, on page 748.

1. This arboreal metaphor is firmly rooted in computer science. However, markup experts have recently extended it with the term “grove”. This term recognizes that a single document may best be viewed as multiple trees.
Figure 2-1  Hierarchical views of documents
Real-world documents do not always fit this tree model perfectly. They often have non-hierarchical features such as cross-references or hypertext links from one section of the tree to another. XML can represent these structures too. In fact, XML goes beyond the powerful links provided by HTML. More on this in 2.8, “Hyperlinking and addressing”, on page 36.

2.3 | Unicode: The character set

Texts are made up of characters. If we are going to represent texts, then we must represent the characters that comprise them. So we must decide how we are going to represent characters at the bits and bytes level. This is called the character encoding. We must also decide what characters we are going to allow in our documents. This is the character set. A particularly restrictive character set might allow only upper-case characters. A very large character set might allow Eastern ideographs and Arabic characters.

If you are a native English speaker you may only need the fifty-two upper- and lower-case characters, some punctuation and a few accented characters. The pervasive 7 bit ASCII character set caters to this market. It has just enough characters (128) for all of the letters, symbols, some accented characters and some other oddments. ASCII is both a character set and a character encoding. It defines what set of characters are available and how they are to be encoded in terms of bits and bytes.

XML’s character set is Unicode, a sort of ASCII on steroids. Unicode includes thousands of useful characters from languages around the world. However the first 128 characters of Unicode are compatible with ASCII and there is a character encoding of Unicode, UTF-8 that is compatible with 7 bit ASCII. This means that at the bits and bytes level, the first 128 characters of UTF-8 Unicode and 7 bit ASCII are the same. This feature of Unicode allows authors to use standard plain-text editors to create XML immediately.

1. It also includes some not-so-useful characters – there is an entire section dedicated to “dingbats” and there is a proposal to include “Klingon”, the artificial language from Star Trek™.
2.4 | Entities: The physical structure

An XML document is defined as a series of characters. An XML processor starts at the beginning and works to the end. XML provides a mechanism for allowing text to be organized non-linearly and potentially in multiple pieces. The parser reorganizes it into the linear structure.

The “piece-of-text” construct is called an entity. An entity could be as small as a single character or as large as all the characters of a book.

Entities have names. Somewhere in your document, you insert an entity reference to make use of an entity. The processor replaces the entity reference with the entity itself, which is called the replacement text. It works somewhat like a word processor macro.

For instance an entity named “sigma”, might contain the name of a Greek character. You would use a reference to the entity whenever you wanted to insert the sigma character. An entity could also be called “introduction-chapter” and be a chapter in a book. You would refer to the entity at the point where you wanted the chapter to appear.

One of the ideas that excited Ted Nelson, the man who coined the word hypertext, was the idea that text could be reused in many different contexts automatically. An update in one place would propagate across all uses of the text. The feature of XML that allows text reuse is called the external entity. External entities are often referred to merely as entities, but the meaning is usually clear from context. An XML document can be broken up into many files on a hard disk or objects in a database and each of them is called an entity in XML terminology. Entities could even be spread across the Internet. Whereas XML elements describe the document’s logical structure, entities keep track of the location of the chunks of bytes that make up an XML document. We call this the physical structure of the document.

Note The unit of XML text that we will typically talk about is the entity. You may be accustomed to thinking about files, but entities do not have to be stored as files.
For instance, entities could be stored in databases or generated on the fly by a computer program. Some file formats (e.g., a zip file) even allow multiple entities to reside in the same file at once. The term that covers all of these possibilities is entity, not file. Still, on most Web sites each entity will reside in a single file so in those cases external entities and files will functionally be the same. This setup is simple and efficient, but will not be sufficient for very large sites.

Entities’ bread and butter occupation is less sexy than reusing bits of text across the Internet. But it is just as important: entities help to break up large files to make them editable, searchable, downloadable and otherwise usable on the ordinary computer systems that real people use. Entities allow authors to break their documents into workable chunks that can fit into memory for editing, can be downloaded across a slow modem and so forth.

Without entities, authors would have to break their documents unnaturally into smaller documents with only weak links between them (as is commonly done with HTML). This complicates document management and maintenance. If you have ever tried to print out one of these HTML documents broken into a hundred HTML files then you know the problem. Entities allow documents to be broken up into chunks without forgetting that they actually represent a single coherent document that can be printed, edited and searched as a unit when that makes sense.

Non-XML objects are referenced in much the same way and are called unparsed entities. We think of them as “data entities” because there is no XML markup in them that will be noticed by the XML processor. Data entities include graphics, movies, audio, raw text, PDF and anything else you can think of that is not XML (including HTML and other forms of SGML).¹ Each data entity has an associated notation that is simply a statement declaring whether the entity is a GIF, JPEG, MPEG, PDF and so forth.

Entities are described in all of their glorious (occasionally gory) detail in Chapter 55, “Entities: Breaking up is easy to do”, on page 780.

---

¹ Actually, a data entity could even contain XML, but it wouldn’t be treated as part of the main XML document.
2.5 | Markup

We have discussed XML’s conceptual model, the tree of elements, its strategy for encoding characters, Unicode, and its mechanism for managing the size and complexity of documents, entities. We have not yet discussed how to represent the logical structure of the document and link together all of the physical entities.

Although there are XML word processors, one of the design goals of XML was that it should be possible to create XML documents in standard text editors. Some people are not comfortable with word processors and even those who are may depend on text editors to “debug” their document if the word processor makes a mistake, or allows the user to make a mistake. The only way to allow authors convenient access to both the structure and data of the document in standard text editors is to put the two right beside each other, “cheek to cheek”.

As we discussed in the introduction, the stuff that represents the logical structure and connects the entities is called markup. An XML document is made up exclusively of markup and character data. Both are in Unicode. Collectively they are termed XML text.

This last point is important! Unless the context unambiguously refers to data, as in “textual data”, when we say “XML text”, we mean the markup and the data.

Caution The term XML text refers to the combination of character data and markup, not character data alone. Character data + markup = text.

Markup is differentiated from character data by special characters called delimiters. Informally, text between a less-than (“<”) and a greater-than (“>”) character or between an ampersand (“&”) and a semicolon (“;”) character is markup. Those four characters are the most common delimiters. This rule will become more concrete in later chapters. In the meantime, Example 2-1 is an example of a small document to give you a taste of XML markup.
The markup between the less-than and greater-than is called a *tag*.
You may be familiar with other languages that use similar syntax. These include HTML and other SGML-based languages.

### 2.6 | Document types

The concept of a document type is fairly intuitive. You are well aware that letters, novels and telephone books are quite different, and you are probably comfortable recognizing documents that conform to one of these categories. No matter what its title or binding, you would call a book that listed names and phone numbers a phone book. So, a document type is defined by its elements. If two documents have radically different elements or allow elements to be combined in very different ways then they probably do not conform to the same document type.

#### 2.6.1 Document type definitions

This notion of a document type can be formalized in XML. A *document type definition* (or *DTD*) is a series of definitions for element types, attributes, entities and notations. It declares which of these are legal within the document and in what places they are legal. A document can claim to conform to a particular DTD in its *document type declaration*.

---

**Example 2-1. A small XML document**

```xml
<?xml version="1.0"?>
<!DOCTYPE Q-AND-A SYSTEM "http://www.q.and.a.com/faq.dtd">
<Q-AND-A>
<QUESTION>I'm having trouble loading a WurdWriter 2.0 file into WurdPurformertWriter 7.0. Any suggestions?</QUESTION>

<ANSWER>Why don't you use XML?</ANSWER>

<QUESTION>What's XML?</QUESTION>

<ANSWER>It's a long story, but there is a book I can recommend...</ANSWER>
</Q-AND-A>
```

The markup between the less-than and greater-than is called a *tag*.
DTDs are powerful tools for organizational standardization in much the same way that forms, templates and style-guides are. A very rigid DTD that only allows one element type in a particular place is like a form: “Just fill in the blanks!”. A more flexible DTD is like a style-guide in that it can, for instance, require every list to have two or more items, every report to have an abstract and could restrict footnotes from appearing within footnotes.

DTDs are critical for organizational standardization, but they are just as important for allowing robust processing of documents by software. For example, a letter document with a chapter in the middle of it would be most unexpected and unlikely to be very useful. Letter printing software would not reliably be able to print such a document because it is not well defined what a chapter in a letter looks like. Even worse is a situation where a document is missing an element expected by the software that processes it. If your mail program used XML as its storage format, you might expect it to be able to search all of the incoming email addresses for a particular person’s address. Let us presume that each message stores this address in a from element. What do we do about letters without from elements when we are searching them? Programmers could write special code to “work around” the problem, but these kinds of workarounds make code difficult to write.

### 2.6.2 HTML: A cautionary tale

HTML serves as a useful cautionary tale. It actually has a fairly rigorous structure, defined in SGML, and available from the World Wide Web Consortium. But everybody tends to treat the rules as if they actually came from the World Wrestling Federation – they ignore them.

The programmers that maintain HTML browsers spend a huge amount of time incorporating support for all of the incorrect ways people combine the HTML elements in their documents. Although HTML has an SGML DTD, very few people use it, and the browser vendors have unofficially sanctioned the practice of ignoring it. Programming workarounds is expensive, time consuming, boring and frustrating, but the worst problem is that there is no good definition of what these illegal constructs mean. Some

---

1. The document type declaration is usually abbreviated “DOCTYPE”, because the obvious abbreviation would be the same as that for document type definition!
incorrect constructs will actually make HTML browsers crash, but others will merely make them display confusing or random results.

In HTML, the title element is used to display the document’s name at the top of the browser window (on the title bar). But what should a browser do if there are two titles? Use the first? Use the last? Use both? Pick one at random? Since the HTML standard does not allow this construct it certainly does not specify a behavior. Believe it or not, an early version of Netscape’s browser showed each title sequentially over time, creating a primitive sort of text animation. That behavior disappeared quickly when Netscape realized that authors were actually creating invalid HTML specifically to get this effect! Since authors cannot depend on nonsensical documents to work across browsers, or even across browser versions, there must be a formal definition of a valid, reasonable document of a particular type. In XML, the DTD provides a formal definition of the element types, attributes and entities allowed in a document of a specified type.

There is also a more subtle, related issue. If you do not stop and think carefully about the structure of your documents, you may accidently slip back into specifying them in terms of their formatting rather than their abstract structure. We are accustomed to thinking of documents in terms of their rendition. That is because, prior to GML, there was no practical way to create a document without creating a rendition. The process of creating a DTD gives us an opportunity to rethink our documents in terms of their structure, as abstractions.

### 2.6.3 Declaring a DTD

Example 2-2 shows examples of some of the declarations that are used to express a DTD:

#### Example 2-2. Markup declarations

```xml
<!ELEMENT Q-AND-A (QUESTION, ANSWER)+>
<!-- This allows: question, answer, question, answer ... -->

<!ELEMENT QUESTION (#PCDATA)>
<!-- Questions are just made up of text -->

<!ELEMENT ANSWER (#PCDATA)>
<!-- Answers are just made up of text -->
```
Some XML documents do not have a document type declaration. That does not mean that they do not conform to a document type. It merely means that they do not claim to conform to some formally defined document type definition.

If the document is to be useful as an XML document, it must still have some structure, expressed through elements, attributes and so forth. When you create a stylesheet for a document you will depend on it having certain elements, on the element type names having certain meanings, and on the elements appearing in certain places. However it manifests itself, that set of things that you depend on is the document type.

You can formalize that structure in a DTD. In addition to or instead of a formal computer-readable DTD, you can also write out a prose description. You might consider the many HTML books in existence to be prose definitions of HTML. Finally, you can just keep the document type in your head and maintain conformance through careful discipline. If you can achieve this for large, complex documents, your powers of concentration are astounding! Which is our way of saying: we do not advise it. We will discuss DTDs more in Chapter 54, “Creating a document type definition”, on page 748.

### 2.7 | Well-formedness and validity

Every language has rules about what is or is not valid in the language. In human languages that takes many forms: words have a particular correct pronunciation (or range of pronunciations) and they can be combined in certain ways to make valid sentences (grammar). Similarly XML has two different notions of “correct”. The first is merely that the markup is intelligible: the XML equivalent of “getting the pronunciation right”. A document with intelligible markup is called a *well-formed* document. One

---

**Caution** A DTD is a concept; markup declarations are the means of expressing it. The distinction is important because other means of expressing DTDs are being proposed. However, most people, even ourselves, don’t make the distinction in normal parlance. We just talk about the declarations as though they are the DTD that they describe.
The important goal of XML was that these basic rules should be simple so that they could be strictly adhered to.

The experience of the HTML market provided a cautionary tale that guided the development of XML. Much of the HTML on the Web does not conform to even the simplest rules in the HTML specifications. This makes automated processing of HTML quite difficult.

Because Web browsers will display ill-formed documents, authors continue to create them. In designing XML, we decided that XML processors should actually be prohibited from trying to recover from a well-formedness error in an XML document. This was a controversial decision because there were many who felt that it was inappropriate to restrict XML implementors from deciding the best error recovery policy for their application.

The XML equivalent of “using the right words in the right place” is called validity and is related to the notion of document types. A document is valid if it declares conformance to a DTD in a document type declaration and actually conforms to the DTD.

Documents that do not have a document type declaration are not really invalid – they do not violate their DTD – but they are not valid either, because they cannot be validated against a DTD.

If HTML documents with multiple titles were changed over to use XML syntax, they would be well-formed and invalid (presuming the HTML DTD was also converted to XML syntax). If we remove the document type declaration, so that they no longer claim to conform to the HTML DTD, then they would become merely well-formed but neither valid nor invalid.

**Caution** For most of us, the word “invalid” means something that breaks the rules. It is an easy jump from there to concluding that an XML document that does not conform to a DTD is free to break any rules at all. So for clarity, we may sometimes say “type-valid” and “non-type-valid”, rather than “valid” and “invalid”.

You should think carefully before you decide to make a document that is well-formed but not valid. If the document is one-of-a-kind and is small, then making it well-formed is probably sufficient. But if it is to be part of any kind of information system (even a small one) or if it is a large document, then you should write a DTD for it and validate whenever you revise...
it. When you decide to build or extend your information system, the fact that the document is guaranteed to be consistent will make your programming or stylesheet writing many times easier and your results much more reliable.

2.8 | Hyperlinking and addressing

If you have used the Web, then you probably do not need to be convinced of the importance of hyperlinking. One thing you might not know, however, is that the Web’s notions of hyperlink are fairly tame compared to what is available in the best academic and commercial hypertext systems. XML alone does not correct this, but it has an associated standard called XLink that goes a long way towards making the Web a more advanced hypertext environment.

The first deficiency of today’s Web links is that there are no standardized mechanisms for making links that are external to the documents that they are linking from. Let’s imagine, for example that you stumble upon a Web page for your favorite music group. You read it, enjoy it and move on. Imagine next week you stumble upon a Web page with all of the lyrics for all of their songs (with appropriate copyrights, of course!). You think: there should be a link between these two pages. Someone visiting one might want to know about the other and vice versa.

What you want to do is make an external link. You want to make a link on your computer that appears on both of the other computers. But of course you do not have the ability to edit those two documents. XLink will allow this external linking. It provides a representation for external links, but it does not provide the technology to automatically publish those links to the world. That would take some kind of link database that would track all of the links from people around the world. Needless to say this is a big job and though there are prototypes, there is no standardized system yet.

You may wonder how all of these links will be displayed, how readers will select link sheets and annotations, how browsers will talk to databases and so forth. The simple answer is: “nobody knows yet.”

Before the first Web browser was developed there was no way to know that we would develop a convention of using colored, underlined text to represent links (and even today some browsers use other conventions). There was also no way to know that browsers would typically have “back” buttons and “history lists”. These are just conventions that arose and browser features that became popular.

This same process will now occur with external links. Some user interface (perhaps a menu) will be provided to apply external link sheets, and there will probably be some mechanism for searching for link sheets related to a document on the Web. Eventually these will stabilize into standards that will be ubiquitous and transparent (we hope!). In the meantime, things are confused, but that is the price for living on the cutting edge. XLink moves us a notch further ahead by providing a notation for representing the links.

Another interesting feature of XML extended links is that they can point to more than one resource. For instance instead of making a link from a word to its definition, you might choose to link to definitions in several different dictionaries. The browser might represent this as a popup menu, a tiny window with the choices listed, or might even open one window for each. The same disclaimer applies: the XML Link specification does not tell browsers exactly what they must do. Each is free to try to make the most intuitive, powerful user interface for links. XML brings many interesting hypertext ideas from university research labs and high tech companies “to the masses.” We still have to work out exactly how that will look and who will use them for what. We live in interesting times!

2.9 | Stylesheets

To a certain extent, the concerns described above are endemic to generalized markup. Because it describes structure, and not formatting, it allows variations in display and processing that can sometimes disturb people.

However, as the Web has evolved, people have become less and less tolerant of having browser vendors control the “look and feel” of their documents. An important part of all communication, but especially modern business communication, is the idea of style. Stylesheets allow us to attach our own visual style to documents without destroying the virtue of generalized markup. Because the style is described in a separate entity, the stylesheet, software that is not interested in style can ignore it.
For instance most search engines would not care if your corporate color is blue or green, so they will just ignore those declarations in the stylesheet. Similarly, software that reads documents aloud to the sight-impaired would ignore font sizes and colors and concentrate on the abstractions – paragraphs, sections, titles and so forth.

The Web has a very simple stylesheet language called Cascading Style Sheets (CSS), which arose out of the early battles between formatting and generalized markup in HTML. Like any other specification, CSS is a product of its environment, and so is not powerful enough to describe the formatting of documents types that are radically different in structure from HTML.

Because CSS is not sufficient, the World Wide Web Consortium is working on a complementary alternative called the Extensible Stylesheet Language (XSL). XSL will have many features from CSS, but will also borrow some major ideas from ISO’s DSSSL stylesheet language. XSL will be extensible, just as XML is, so that it will be appropriate for all document types and not just for HTML. Like the linking specification, XSL is still under development so its exact shape is not known. Nevertheless, there is a general design that we will review later on.

### 2.10 | Programming interfaces and models

This subject may seem intimidating if you are not a programmer – possibly even if you are! But we are just going to take a high-level view of a few constructs that will be helpful in understanding the chapters that follow. We’ll cover the XML geek-speak Top Term List: Parsing, APIs, DOM, and SAX.

#### 2.10.1 Parsing

Great as XML is for representing data, eventually that data has to be processed, which requires the use of one or more programs. One of the nice things about writing XML applications is that there is an abundance of reusable component and utility software available to help.

All great programmers try to reduce their work! If every programmer reinvented the wheel when it came to basic processing of XML, no pro-
grammer would ever get around to building applications that *use* XML. Instead of implementing basic XML processing over and over again, programmers tend to download or buy packages that implement various types of XML services.

The most basic reusable service is parsing. Parsing is about ripping apart the textual representation of a document and turning it into a set of conceptual objects.

For example, a parser looking at the document in Example 2-1 would recognize the characters `<QUESTION>` to be a start-tag, and would know that they signaled the start of a `QUESTION` element. The tag is part of the representation; the element is the conceptual object.

If the parser were also validating the document according to the DTD in Example 2-2, it would make sure that an `ANSWER` element followed the `QUESTION` element.

As a human being, you do parsing subconsciously. Because you’ve learned about elements and attributes, when you look at XML text you can think about the document in those conceptual terms.

But without an XML *parser*, a computer program can only see the characters. It’s sort of the opposite of not seeing the forest for the trees. Without some form of parsing, an XML application cannot see the tree because of all of the characters!

### 2.10.2 APIs

There are many good XML parsers out there for use with many different programming languages. There are so many that it is hard to choose. A software developer would hate to pick one and be wedded to it forever. The programmer might want to change some day to a faster or cheaper one, or from a non-validating parser to a validating one.

Switching parsers (often also called *processors*) is easy if the two “look” the same to the programmer. You can plug in different brands and types of light bulbs into the same socket because of the standardization of the socket. The equivalent concept in software components is the standardization of *Application Processing Interfaces (APIs)*.
2.10.2.1 The DOM

The World Wide Web Consortium has standardized an API for working with XML. It is called the Document Object Model and it is available in Version 5 Web browsers. If you write code for Microsoft’s DOM implementation, it should be relatively easy to make that code also work on Netscape’s DOM.

But the DOM is not only for use in browsers. It can also be used on the server side. You can use the DOM to read, write and transmit XML on your Web server. DOM-based programs can talk to some XML content management systems. The DOM is very popular for general XML processing. It has been implemented, for example, for use with Python and Perl scripts and with the C++ and Java™ programming languages, among others. In fact, Microsoft’s DOM implementation is a built-in part of Windows 2000 itself.

2.10.2.2 SAX

The DOM is popular and useful but it is not the be-all and end-all of XML parsing APIs. It is a little bit like putting a plane on automatic pilot. You point your DOM-building processor at an XML document and it returns you an object tree based on the structure of the document.

But if the document is five hundred megabytes of text and resides on the “other side” of the Internet, your program will just wait. And wait. And wait. When you finally get the data it will fill your computer’s memory and some of its disk space. If you are having a bad day it might fill up everything and then crash the computer.

In a situation like this, you would rather just get tiny bits of the data as they come in. An event-based parser allows this mode of operation. Event-based parsers let your application work on the bit of the data that the parser finds at each “event” in the document.

For example, each XML start-tag corresponds to a “start element” event. Each end-tag corresponds to an “end element” event. Characters and other constructs have their own events. The event-based XML parser tells the application what it sees in the document as if through a peep-hole. It does not try to describe the larger picture to the application.1

The most popular event-based API is the Simple API for XML. SAX was developed by XML processor users and developers in an open discussion
group called XML-DEV. Despite the name, SAX is not actually any simpler than the DOM. It is much more efficient and low-level, however. The price for efficiency is convenience. The processor only provides you with a peephole view, so if your application needs more than that, you’ll need to write your own code to understand the “big picture” of the parsed document.

These two APIs are pervasive in the XML processing world. There are many other services that we could envision for XML handling: link management, searching and so forth. It is likely that these will be built either on top of or as extensions to these two popular APIs.

### 2.11 | Conclusion

There are a lot of new ideas here to absorb, but we’ll be repeating and reemphasizing them as we move along. At this point, though, we’re ready to look at where XML is going and the ways that it is being used in the real world.

---

1. If you concluded from this description that a DOM processor in effect uses an event-based parser as it constructs the DOM, you are right.