XML Security

Blake Dournaee

Copyright © 2002 by The McGraw-Hill Companies

Disclaimer and Limitation of Liability:
The content of this file is copyrighted material of McGraw-Hill. McGraw-Hill makes no representations or warranties as to the accuracy of any information contained in the McGraw-Hill Material, including any warranties of merchantability or fitness for a particular purpose. In no event shall McGraw-Hill have any liability to any party for special, incidental, tort, or consequential damages arising out of or in connection with the McGraw-Hill Material, even if McGraw-Hill has been advised of the possibility of such damages.

CHAPTER 3

XML Primer

The world of Extensible Markup Language (XML) is an endless ocean of standards and technologies that mimics a living being, constantly changing and ever evolving. Trying to capture the breadth and depth of XML in a single chapter is a hopeless task with no end. The goal here, then, is to focus on a few anchors in the XML ocean that represent *fundamental concepts*. There is little point in repeating details that can be found in the text of a published standard. Instead, the reader will find an explanation of fundamental XML concepts bolstered with real-world examples. Special focus is on building block technologies that provide the groundwork for the sea of XML.

This chapter is divided into two broad topics, an introduction to basic XML syntax and a preliminary discussion of XML processing. The division between syntax and processing is a theme that will be revisited throughout this book. The core XML *syntax* topics discussed include the basics of well-formed documents, markup concepts, some information about namespaces, and numerous examples. XML *processing* is discussed with the presentation of two topics: the Document Object Model (DOM) and the XPath data model. The seasoned reader with previous experience with XML might find this chapter a bit repetitive, but XML syntax and processing is a necessary building block for XML Security.

What Is XML?

This question reminds me of an assignment once given in high school where the task was to answer the broad question: "What is History?" As youngsters we were quick to respond with simple answers such as "History is what happened in the past," or for those of us who were really lazy, we would provide the proctor with Webster's definition (and no doubt receive a poor grade).

The question "What is XML?" is a loaded question that has no simple answer. It almost seems like any simple answer given would be akin to answering the "What is History" question with the same naiveté of someone in high school. For XML, the answer to this question often depends on the audience. There are books on XML for managers, software developers, sales representatives, and marketing people. Here we will take the viewpoint of a software engineer or computer scientist as we attempt to create yet another definition of this technology.

Meta-Language and Paradigm Shift

The topic of XML Security requires viewing XML from a slightly different angle compared to other technologies that leverage XML to accomplish its goals. XML Security is devoid (thankfully) of *presentation semantics*. That is, the current XML Security specifications don't focus on rendering or displaying an XML Signature or encrypted XML element. In this respect, XML Security technologies are more closely related to existing security technologies such as the Public Key Cryptography Standards (PKCS), discussed in Chapter 2. Don't look for any HTML or JavaScript code in this book, because it is simply not the focus of the XML Security specifications.

Two vocabulary words that are especially useful in defining XML from an XML Security standpoint are *meta-language* and *paradigm shift*. The noun *meta-language* best describes the *what* part of XML Security and the verb *paradigm-shift* helps describe the *where* part of XML Security. Thankfully, the *how* part is left to the dedicated authors of the XML Security Recommendations and Drafts.

Meta-Language

XML is not a language. Despite the name Extensible Markup Language, it is easiest to understand XML as a meta-language. What exactly does

this mean? A *meta-language* is a language used to describe other languages. Some would call this a true language. The prefix "meta" used in this sense has its roots in analytic philosophy and loosely means one level up or above and beyond.

So what does this mean for you? In simple terms, XML is a *syntax* used to describe other markup languages. The XML 1.0 Recommendation as released by the W3C defines no tag set or language keywords. The skeptical reader can check for himself or herself, but simply put, only a syntax and grammar is defined by the XML 1.0 Recommendation. The term *syntax* here refers to a set of constraints about how and where tags can be placed, and the acceptable range of characters that are legal, as well as the rules for markup in general. Moreover, the basic rules and syntax of XML 1.0 are deceptively simple and can be learned in the better part of an hour.

At this point, no examples have been provided, so the assumption is that the reader is as lost as ever. The next question is: "If XML is a syntax used to describe other markup languages, what are these *other markup languages?*" The answer to this question is the sea of XML-related standards and, more importantly, the XML Security Standards. For example, the XML Signature Recommendation defines a *markup language* used to represent a digital signature; the XML Encryption Draft defines a *markup language* used to represent encrypted elements. Similarly, markup languages such as MathML or DocBook are also *other* markup languages that are defined in accordance with the syntax put forth by the XML 1.0 Recommendation. MathML is a markup language for representing mathematics and DocBook is a markup language used for representing articles or books.

The final part of our XML definition relates to how exactly these other markup languages are defined: XML is a syntax used to describe other markup languages *using markup*. This seems like a circular definition—of course markup languages use *markup*. A short example of arbitrary markup is given in Listing 3-1.

The preceding listing is arbitrary data with tags around it. For example, we have a piece of data, `Samuel Adams`, that is marked up with the tag `<Good_Beer>`. The important point here isn't what the tags say or even what they mean; instead, the focus should be on *what the tags can do*. Tags provide *markup* and *markup* can accomplish many different things with regards to *data*. In fact, the list of things that markup can accomplish is so important that it belongs in a box.

Listing 3-1

Example of arbitrary markup

```
<Food>
  <FrenchFries> Curly Fries </FrenchFries>
  <Beers>
    <Good_Beer> Samuel Adams </Good_Beer>
    <Good_Beer> Guinness </Good_Beer>
    <Bad_Beer>  Budweiser </Bad_Beer>
    <Bad_Beer>  Fosters </Bad_Beer>
  </Beers>
</Food>
```

The Roles of Markup

- Markup can add semantics to data.
- Markup can demarcate data.
- Markup can define roles for data.
- Markup can define containment.
- Markup can define relationships.

With the careful use of *tags* around *arbitrary textual data,* we can accomplish almost any sort of semantics desired. We can invent tags and structures, give various roles to data, and define relationships between tags. The power of *markup* is nearly limitless for carving up any sort of data. This property of markup is especially powerful because the basic concept behind markup is simple. It doesn't take years of practice to begin using markup, nor is the syntax complicated and difficult to understand. It is this combined simplicity and power that makes XML an exciting technology.

Because XML is a meta-language, every use of XML and markup to add semantics to data results in the creation of a markup language. For example, if someone were to look at Listing 3-1 and ask the question: "What language is that?" the correct answer is: "It's a fictional markup language that uses the syntax of XML." In short, Listing 3-1 actually defines its own markup language. It uses the tags `<Food>`, `<FrenchFries>`, `<Beers>`, `<Good_Beer>`, and `<Bad_Beer>`. While the tag set is small and rather useless, it is a valid markup language. Before we move into the specifics of XML syntax, we will examine the other high-level defining component of XML as it relates to XML Security: *paradigm-shift*.

Paradigm-Shift

XML Security represents a clear paradigm-shift from ASN.1-based, binary standards toward more portable, text-based XML solutions. Most of the entities in the security world that relate to *cryptography* and public-key infrastructure (PKI) use ASN.1 types to encode various entities. When use the term *cryptography,* we are really referring to *applied cryptography* that takes a standards-based approach—real cryptographers probably do much of their work with pencil and paper.

Examples of these binary format security standards include X.509 certificates or most of the PKCS standards—all of these use ASN.1 types to encode their pieces and parts. Examples of these formats have already been given in Chapter 2, but the paradigm shift that XML Security promises is a shift from BER encoded ASN.1 "objects" to the analogous XML structures. A clear example of this shift is seen with the way a verification key (usually a public key) is represented in an XML Signature. As discussed in Chapter 4 and Chapter 8, the `<KeyValue>` tag (just more markup) is used to represent a raw public key that can be used for decryption. This is shown in Listing 3-2. This is contrasted with the X.509 `SubjectPublicKeyInfo` introduced in Chapter 2 and discussed again in Chapter 8, shown again in Listing 3-3.

Listing 3-2

The `<KeyValue>` element from an XML signature

```
<KeyValue>
  <RSAKeyValue>
    <Modulus>
      s3mkTQbzxuNFPFDtWd/9jvs8tF5ynBLilbG/sT24OglEol
      1PBvRe+VUJU0eI2SRhN/KtZv4iD2jwT0Sko0eeJw==
    </Modulus>
      <Exponent>EQ==</Exponent>
  </RSAKeyValue>
</KeyValue>
```

Listing 3-3

A binary `SubjectPublic-KeyInfo` interpreted with an ASN.1 parser

```
SEQUENCE {
  SEQUENCE {
    OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
    NULL
    }
  BIT STRING 0 unused bits
    30 46 02 41 00 B3 79 A4 4D 06 F3 C6 E3 45 3C 50
    ED 59 DF FD 8E FB 3C B4 5E 72 9C 12 E2 95 B1 BF
    B1 3D B8 38 69 44 A2 5D 4F 06 F4 5E F9 55 09 53
    47 88 D9 24 61 37 F2 AD 66 FE 22 0F 68 F0 4F 44
    A4 A3 47 9E 27 02 01 11
  }
```

Both Listings 3-2 and 3-3 are showing us the same *datatype,* an RSA public key. Both structures clearly demarcate the type of key as well as the modulus and the exponent. Listing 3-3 does this a bit more covertly, as the modulus and exponent are encoded inside the BIT STRING. Listing 3-2 uses XML syntax and *markup* while Listing 3-3 uses ASN.1 and BER. Both encoding schemes are intended to be extensible and both encoding schemes have tradeoffs. The XML version is certainly user-friendly and one might argue that it is easier for an application to parse text data with XML markup. The binary version, however, is far more compact (once encoded in binary, half as small or smaller than the equivalent XML version), but, as some would argue, harder for an application to parse. Some would even be appalled at the space wasted by the XML version. The difference between Listings 3-2 and 3-3 is the paradigm shift. For better or for worse, emerging XML Security standards have a strong "ASN.1 hate factor" and instead opt for cryptographic objects to take form as semantically clean, easy to read, and easy to parse XML variations.

Now that the reader has some basic high-level notions about what XML is (perhaps fuzzy notions), we can begin our descent from abstract high-level ideas to more concrete, tedious details.

Elements, Attributes, and Documents

Three important terms for describing basic XML syntax are *elements, attributes,* and *documents.* All three of these terms are special and important; they encompass a large portion of the conceptual playing field for XML and provide the foundation for the remainder of this chapter as well as the entire book.

Elements and Attributes

We have spent some previous discussion throwing around the term *markup.* A small example was given, and the reader should have seen some *tags* with stuff inside, but little else should be evident besides the fact that markup is useful for the intellectual carving of data.

Markup is often divided into two separate vocabulary words when we are talking about XML: *elements* and *attributes.* An element is a *start tag* and an *end tag* including the stuff inside of it and an attribute is a simple name-value pair where the value is in single or double quotes. An

attribute cannot stand by itself and must be inside the start tag of a given element. Two short examples follow:

```
<Food> Ice Cream </Food>
<Food Flavor="Chocolate"> Ice Cream </Food>
```

The first line in the previous example is an element called `Food` that has `Ice Cream` as its element content. The second line in the previous example is the same element with a name-value pair added to it (this is an attribute). The name is `Flavor` and the value is `Chocolate`. Elements may also be empty, having no element content. This is shown in the example that follows:

```
<HealthyFood></HealthyFood>
<HealthyFood/>
```

The first line in the previous example is an element called `<HealthyFood>` that has nothing inside of it. The second line is *shorthand* for the same empty element. Take note of this shorthand notation, because it is used pervasively in many XML documents. This notation can be confusing at first, but in all cases, it simply means an empty element.

Attributes may be used arbitrarily within start tags to add more meaning to the data. In fact, there was a great deal of contention over the inclusion of attributes within the XML syntax. The reason is because any data that can be modeled with elements alone can also be modeled with an attribute-centric approach and vice versa. Consider the following short example. This example contains the same data as Listing 3-1, but it is modeled almost entirely with attributes.

```
<Food FrenchFries = "CurlyFries"
      Good_Beer1  = "Samuel Adams"
      Good_Beer2  = "Guinness"
      Bad_Beer1   = "Budweiser"
      Bad_Beer2   = "Fosters"
</Food>
```

The markup used in the preceding example is certainly clumsier than Listing 3-1, but the point here is that we are *roughly* modeling the same data, but using attributes instead. There are five attributes inside the `Food` element and they provide us with the same basic information as shown in Listing 3-1. So which one is better? Both are legal XML documents; the answer to this question is really an answer to a much more complicated *data-modeling question*. The convention, however, is to use

attributes more sparingly than elements. Elements and their contents usually represent concrete information that will be displayed or rendered, while attributes usually represent information required for processing. This dichotomy, however, isn't strict or formal and there isn't anything written down that says that this is how it must be. This is just convention. XML Security-based standards use attributes heavily for algorithm information and data sources while elements are used for concrete cryptographic objects, such as keys or signature values. For example, consider the short example that follows:

```
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

This `<DigestMethod>` empty element uses an attribute called `Algorithm` with the rather long value `http://www.w3.org/2000/09/xmldsig#sha1`. This element is commonly seen in an XML Signature, and the meaning of the attribute value is intended to be the SHA-1 hash function. Notice that even though the element is empty, it still communicates information via the attribute. There is no requirement that all useful elements must have content—this use of an empty element with an attribute is frequently seen in the XML Security world. This shouldn't mean much to the reader at this point; these details will be hashed out in Chapter 4 and Chapter 5.

XML Documents

Throughout this book and throughout many of the XML Security standards, reference is made to something called an *XML document*. This term has a specific meaning and carries with it some implicit properties, the most notable of which is the *well-formed* property. This property is the most basic set of constraints that can be put on data represented using XML; it defines simple syntax rules for the legal positioning of elements and attributes. The reader is now asking, "So, what does *well-formed* mean? What are these constraints?" In short, the list of constraints for a well-formed document follows—again placed in a box because of their importance.

There is a bit of hidden detail here, but not much. Let's examine these four constraints and go through some examples.

Data represented in XML is well-formed if . . .

- There is exactly one *root element*.
- Every *start tag* has a matching *end tag*.
- No tag overlaps another tag.
- All elements and attributes must obey the naming constraints.

Root Element

The root element constraint is perhaps the easiest to see and understand. Simply put, any data that wants to be well formed must have *exactly* one root element. This means there must be one (and only one) *parent* element. The root element has a synonym called *document element*. Sometimes we use the term *root element* and sometimes we use the term *document element*. These refer to exactly the same thing; sometimes one just sounds better! Listings 3-4 and 3-5 are examples of data that do *not* have a single root element, while Listing 3-6 and Listing 3-7 correct these examples to make them well formed. The additional root elements have been added in bold.

Listing 3-4

Sample XML data without a root element (not well formed)

```
<Dark_Chocolate>
  <Brand1>Hersheys</Brand1>
  <Brand2>Ghiradelli</Brand2>
</Dark_Chocolate>
<Ice Cream>
  <Brand1>Ben and Jerry</Brand1>
  <Brand2>Dryers</Brand2>
</Ice Cream>
```

Listing 3-5

Sample XML data without a root element (not well formed)

```
<Student> Joe </Student>
<Student> Bob </Student>
<Student> Mary </Student>
```

Listing 3-6

Sample XML document (well-formed data)

```
<FatteningFoods>
  <Dark_Chocolate>
    <Brand1>Hersheys</Brand1>
    <Brand2>Ghiradelli</Brand2>
  </Dark_Chocolate>
  <Ice Cream>
    <Brand1>Ben and Jerry</Brand1>
    <Brand2>Dryers</Brand2>
  </Ice Cream>
</FatteningFoods>
```

Listing 3-7

Sample XML document (well-formed data)

```
<Students>
  <Student> Joe </Student>
  <Student> Bob </Student>
  <Student> Mary </Student>
</Students>
```

Start Tags and End Tags

The next constraints, start tags and end tags, are also simple and easy to see. For every start tag, there must be an associated end tag. Listing 3-8 shows the incorrect data and Listing 3-9 corrects this data by adding the proper end tags.

Again, this constraint is quite simple. It is easy to see by inspection if end tags or start tags are missing—the syntax so far is just not complicated. Let's move along to the last two constraints.

Overlapping Tags

Tags cannot overlap each other in such a way that one tag is closed before another tag. This constraint is difficult to describe with clarity, but sufficiently easy to see in an example. Consider the next small example:

```
<Student>
  <SSN>123-45-6789</Student>
</SSN>
```

The previous example is in a lot of syntactic trouble—not only is there no clear root element, but the `<Student>` tag is closed before the `<SSN>` tag. To fix this, you have to be sure that the elements do not overlap. The

Listing 3-8

Sample XML data missing some end-tags (not well formed)

```
<Candy>
  <Good_Candy>
    Milk Chocolate
  <Bad_Candy>
    Dark Chocolate
</Candy>
```

Listing 3-9

Sample XML document (well-formed data)

```
<Candy>
  <Good_Candy>
    Milk Chocolate
  </Good_Candy>
  <Bad_Candy>
    Dark Chocolate
  </Good_Candy>
</Candy>
```

following example shows how you can fix this to make it an XML document that is well formed.

```
<Student>
  <SSN>123-45-6789</SSN>
</Student>
```

Naming Constraints

Most of the tedious details of the well-formed property are contained within the naming constraints for elements and attributes. These constraints are much more broad than the previous constraints because they limit the range of acceptable characters for elements as well as some details on white space. The least you need to know is contained in the box that follows.

> ### *Naming Constraints*
>
> - Element names must begin with a letter or underscore.
> - Element names cannot contain embedded spaces.
> - Element names are case sensitive.
> - Attribute names must be unique per element (start tag).
> - Attribute values must use single or double quotes.
> - Attribute values cannot contain a < character.

Again, these naming constraints are still not that complicated. Listing 3-10 shows the gratuitous violation of every rule listed in the previous box. The idea is to get the reader in tune with what the simplest legal XML documents look like, and one way of reinforcing this is to look at the illegal use of XML data.

The reader is challenged to scan Listing 3-10 and make an attempt to find six naming constraint violations. The idea here is that well-formed XML documents are not complex; the syntax is simple to learn and quite intuitive.

If the reader has grasped the previous concepts with regard to XML documents, the well-formed property and markup (elements and attributes), most of the battle is already won. At this point, the reader should have the conceptual tools to build the *simplest possible legal XML documents*. We have made it this far with only three well-defined terms: documents, elements, and attributes. Once the reader can create and use XML documents, it is possible to begin playing with and understanding the pieces that comprise XML Security.

Aside from arbitrary binary data, any sort of XML data that is signed or encrypted must be, at the very least, an XML document. Another point that should be made here is that XML documents can be created using only the simplest of tools; a text editor is all that is required. No browsers or fancy tools are necessary. What we are looking at here is pure data, carved up with markup and limited only by the well-formed constraint.

As a final exercise, consider Listing 3-11 that shows an XML Signature. Using only the basic rules of XML syntax, the reader should have enough information to determine if Listing 3-11 is a legal XML document, despite the fact that the reader should have little or no knowledge of the meaning of any of the elements or attributes used.

Listing 3-10

Sample XML data that violates almost every naming constraint (not well formed)

```
<4Root_Element>
  <Message> This is an invalid element </message>
  <Another Message> This is also an invalid element </Another Message>
  <Note color="blue" color="green"> These are colors </Note>
  <Note color=red> I forgot this color </Note>
  <Note lt="<"> Less than sign </Note>
</4Root_Element>
```

Listing 3-11

XML data that is not well formed

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
     Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <Reference URI="http://www.rsasecurity.com">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>szlrBmSpQUJCO/ykyhS126/xMOM=<DigestValue>
    </Reference>
  </SignedInfo>
<SignatureValue>
  UOmRz+EiYhy5LEsZ+fXBKnHlzWpJ+HFCOQlhWdb
  I/DVlv7Szt11BEfn8fpC4bxG19UDd7MbpRedi
  3qUeVP+GSvMElrPo8u++KsHMKGsaPoqeUOoUI
  bW7biuW1rMSYpESdUeWbmy2p2/P8sulMouHrT
  q+Jv92GQ+itjLimhmHLTs=
</SignatureValue>
</Signature>
```

The solution is, of course, the erroneous `<DigestValue>` element. There are two opening tags for this element but no corresponding closing tag.

The URI

One potential confusing aspect of XML documents is the pervasive use of URIs. A URI is a *Uniform Resource Identifier* and is a short string value intended to identify something on the Web—whether it is a file, a service, or a person. In fact, a URI can identify *any* resource that has identity. An example of a URI is the string value `http://www.rsasecurity.com`.

Listing 3-11 includes five URIs within the markup. At this point, it's not clear what they are used for (other than the fact that they are *attribute* values), and they tend to clutter the markup because of their length.

Most readers have encountered URIs that identify Web resources such as Web pages that use the HTTP *scheme*. This is the same string that is pasted into a browser and used to visit your favorite Web site, and in this case data is retrieved from the location (Web pages, graphics, and so forth). The URIs used in Listing 3-11 differ in that not all of these URIs are used as a *data source*. That is, in the context of Listing 3-11, some of

the URI values are not meant to be retrieved, but instead are meant to be used as *identifiers*. This may seem odd because most URIs seen outside of XML documents are meant to be de-referenced and used as a data source.

As we examine various XML technologies we will see the distinction between those URIs used mainly as identifiers versus those URIs that are simply meant to be sources of data. We will see that any sort of URI can be used as a pure identifier, even if it happens to point to some real data. The first example of URIs used as identifiers occurs in the following section on *Namespaces in XML*.

Namespaces in XML

Another conceptual hurdle is the topic of *namespaces* in XML. The purpose of a namespace when used in an XML document is to prevent the collision of semantically different elements and attributes that happen to have the same name. For example, suppose that an author of an XML document wants to use an element called `<Fans>`. This particular element can refer to the noun *fan* as in a ceiling fan, or it can refer to the noun *fan* as related to an attendee at a sporting event. Still more problems occur if this element, ostensibly created by two different authors, is merged into a single XML document. Clearly, the meaning of the element is ambiguous and it is unreasonable to expect any sort of application to be able to make the distinction between the different elements without some other qualifying information.

This problem is solved with the use of a *namespace*. A namespace in the context of XML is simply a collection of element and attribute names identified by a *URI Reference*. The intention of an XML namespace is to provide a globally unique name for an element or attribute. The previous sentences should mean precious little to the reader without some examples. Before the examples, a point of clarification must be made. The term *URI Reference* may be confusing —what is a URI Reference? A URI Reference is a URI that is used as a *string identifier*; it has no purpose beyond this. Some may argue that this term is redundant and confusing, but it is the nature of the URI in these examples; simply an identifier.

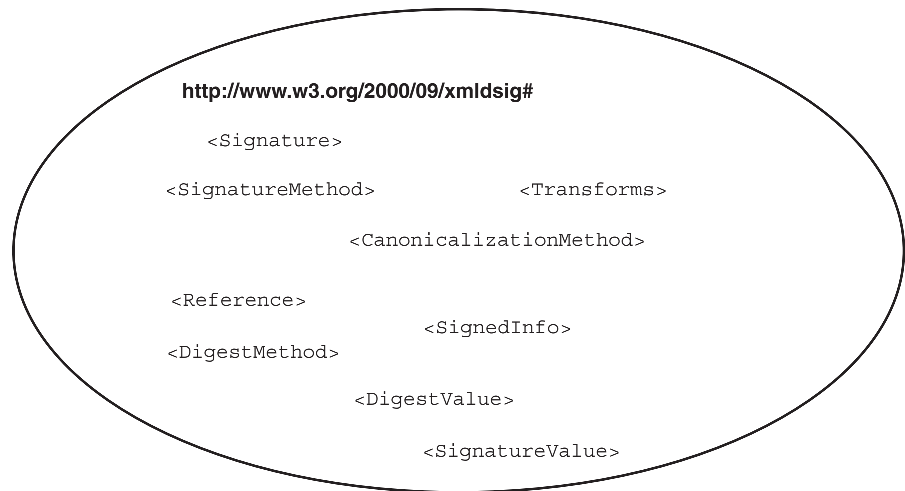
A pertinent example is the namespace used for an XML Signature. The URI Reference is `http://www.w3.org/2000/09/xmldsig#` and the collection of element and attribute names that are associated with this string identifier include the elements and attributes that help define an XML Signature. The specifics of the elements used in an XML Signature

are given in Chapter 4 and Chapter 5. The conceptual picture looks something like Figure 3-1.

The idea behind the namespace is quite simple, but the syntax to use a namespace inside an XML document can get quite confusing because there are multiple ways to accomplish the same thing. Let's work through some short examples. First consider Listing 3-12, which shows an XML document using elements from the XML Signature namespace *without* any sort of namespace qualification. Let's not worry about the contents of

Figure 3-1

The XML Signature namespace and its related elements



Listing 3-12

An XML document using elements from the XML Signature namespace *without* an explicit namespace qualification

```
<Signature>
  <SignedInfo>
    ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

the elements right now; this is not essential to understanding how the namespace declarations work.

Listing 3-12 is ambiguous as far as the namespace is concerned. This XML document uses elements called `<Signature>`, `<SignedInfo>`, and `<SignatureValue>`, but doesn't tell us where these element names came from. It is possible at this point that these element names belong to another XML document (not an XML signature). Listing 3-12 is akin to using the `<Fans>` element indiscriminately, leaving the semantics ambiguous. An XML namespace is declared with some use of the `xmlns` attribute. This is shown in Listing 3-13.

The syntax shown in bold in Listing 3-13 declares a *default namespace* for all of the elements in the XML document, including the root element. This means that all of the elements in this document (unless otherwise noted) all belong to the `http://www.w3.org/2000/09/xmldsig#` namespace. This syntax for namespaces is perhaps the most straightforward and easiest to see; it shows how to associate a single namespace within an XML document. We can throw a wrench in the example to see how the syntax becomes more complicated. Consider Listing 3-14.

Listing 3-14 poses a problem because it uses two elements, `<Fans>` and `<CeilingFans>`, that are not part of the XML Signature namespace. We need some way of marking these elements as part of a different name-

Listing 3-13

An XML document using elements from the XML Signature namespace using a *default namespace declaration*

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

Listing 3-14

An XML document with an improper element for the default namespace

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
  <Fans>
    <CeilingFans> 4 </CeilingFans>
  </Fans>
</Signature>
```

space, such that a processing application can make the proper distinction. This is where the syntax gets a bit more complicated.

What we need to do is declare an additional *namespace prefix*. The namespace prefix is an arbitrary string (usually short) associated with a given namespace. This prefix is declared as another attribute value inside the parent element for which the prefix is to be valid. This is shown in Listing 3-15.

The *namespace prefix* chosen for Listing 3-15 is the short string `foo`. By using the namespace prefix and the colon separator, we can associate a particular element with a given namespace inside the element itself. This is shown in Listing 3-15 when we declare the `<Fans>` element is a member of `http://fans.com` by naming the element `<foo:Fans>`. This idea is difficult to describe with much clarity, but it is easy to see with an example. The string `foo:Fans` is called the *qualified name* and consists of the *namespace prefix* (`foo`) and the *local part* (`Fans`).

XML documents that combine multiple technologies (such as a document that uses elements from the XML Signature syntax and XML Encryption syntax) will have to deal with declaring the appropriate namespaces and qualifying any elements used. For most of the discussion and examples in this book, namespaces are not shown because they add to the syntactic clutter and can cloud the understanding of some of the basic concepts. That being said, they are an *absolutely essential* part of XML and XML Security because without them there is no way to separate different technologies and elements used within a given context. Further, there is still much more to learn about namespaces! The previous discussion is only a primer and gives the reader just enough ammunition to understand the examples and usage within this book. The reader is urged to visit the references section for places to go to read more about namespaces.

Listing 3-15

One way of using two namespaces in an XML document

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#"
           xmlns:foo="http://fans.com">
  <SignedInfo>
    ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
  <Object>
    <foo:Fans>
      <foo:CeilingFans> 4 </foo:CeilingFans>
    </foo:Fans>
  </Object>
</Signature>
```

More Markup

Elements, attributes, and namespaces represent the most fundamental types of XML markup. Additional markup constructs that should be briefly mentioned are *comments, processing instructions,* and *character data sections.* While not too terribly important for the scope of this book, they are fundamental parts of an XML document and are used in the discussion of the Document Object Model (DOM), which is discussed in the second half of this chapter.

Comments

The idea of a comment for a programming language is pervasive for anyone who has compiled even the simplest program. The idea is exactly the same for XML documents; there is a way for document authors to inform others of what is going on inside their twisted minds. The syntax is simple and a brief example follows. The text between the `<!--` and `-->` delimeters is a comment:

```
<!--   Comment on This!   -->
```

Processing Instructions

A processing instruction is intended to be a customized instruction to the processing application. Processing instructions are currently not widely used in the XML Security standards and can be safely ignored for the most part. The syntax of a processing instruction is the two character delimeter `<?` followed by a *target* and then an *arbitrary data string* and finally the closing delimeter `?>` The key idea about processing instructions is that they are predefined and application specific. For example, if you need to enumerate a certain section of markup as significant for a custom application, you might denote this with a processing instruction. An example of a processing instruction follows:

```
<? application_processor_1 do_task ?>
```

The reason processing instructions are even mentioned here is because they show up in the DOM, which is discussed in the second half of this chapter.

Character Data Sections

XML has some restrictions on certain text characters; for example, you can't create an element with the following markup characters:

```
<Expression> 4 < 5 </Expression>
```

The obvious reason for this is because the processing application that parses the XML document can't determine where elements begin and end. With markup delimeters in the markup itself, it can't match up the < and > characters properly. To remedy this, XML defines what are called *predefined character entities* that are used to represent markup characters used in element and attribute names. These are shown in Table 3-1.

While these are useful, they can become cumbersome in practice. To remedy this, the CDATA section is used to add unparsed text to an XML document. The identifier CDATA stands for *character data*. The syntax looks sort of weird, but the idea here is to be able to add arbitrary data to the XML document without having to worry about using the predefined character entities. A CDATA section looks like this:

```
<Expression> <![ CDATA [ 4 < 5 ]] > </Expression
```

The idea is that you place the text in between the second [(left bracket) and first] (right bracket). This example is semantically equivalent to the first. The CDATA section doesn't have to be used for *just* unparsed text. It can be used for any arbitrary text that the parser should ignore.

More Semantics: The Document Prolog

By now the reader should have a fairly good grasp of XML basics. Some familiarity with namespaces as well as XML document basics allows the

Table 3-1
Predefined Character Entities

Predefined Character Entity Name	Value
>	>
<	<
"	"
&	&
'	'

reader to understand most of the content of any given XML document. There are, unfortunately, more details that must be hashed out. Most of the following details will not be pursued much further in this book, but they are *essential* to understanding and using XML documents outside the scope of this book.

Most of these details come to us as part of something called the *document prolog*. The term used is quite amusing; one definition for the term *prolog* is an introduction to a play or a novel. Unfortunately, XML documents are never as exciting as live entertainment or a good read, but the term is accurate. The document prolog is an optional set of declarations used to add semantics to the current XML document. The document prolog always precedes the root element in an XML document and carries with it some specific syntax constraints. To provide some motivation for the document prolog, let's look at some examples. Consider Listing 3-16.

Listing 3-16 is a portion of an XML Signature document. The names of the elements are unimportant; the only important thing is the structure of the elements and the attributes and attribute values. Notice that Listing 3-16 is littered with URI strings. This doesn't represent anything too odd—most XML documents use URIs as attribute values. The URI strings are valid attribute values and don't do anything sinister other than make Listing 3-16 more difficult to read.

What if we had a way to remove these URI strings from the markup to make the document more readable, but retain the URI values somehow? A properly constructed document prolog with the necessary declarations allows us to accomplish this. Let's give the solution first and then talk about it. See Listing 3-17.

Some might argue that Listing 3-17 is *more* difficult to read than Listing 3-16. This is only because we haven't discussed all the funny syntax yet. Listing 3-16 adds *general entity declarations* for the URI values—simply put, replacement text. To see this, consider the isolated entity declaration shown next:

```
<!ENTITY  dsig  "http://www.w3.org/2000/09/xmldsig#">
```

Listing 3-16

A portion of an XML Signature document

```
<Reference xmlns="http://www.w3.org/2000/09/xmldsig#"
           URI="http://www.rsasecurity.com" >
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
</Reference>
```

Listing 3-17

The use of *general entities* in an XML document

```
<?xml version="1.0"?>
<!DOCTYPE Signature
    [
      <!ENTITY  dsig  "http://www.w3.org/2000/09/xmldsig#">
      <!ENTITY  alg   "http://www.w3.org/2000/09/xmldsig#sha1">
      <!ENTITY  val   "http://www.rsasecurity.com">
    ]
>
<Reference xmlns="&dsig;" URI="&val;" >
  <DigestMethod Algorithm="&alg;"/>
  <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
</Reference>
```

The value on the left (`dsig`) is replaced with the string value on the right in the actual markup. To specify the *general entity* within the markup, the `&` and `;` syntax is used to delimit the general entity name. These are marked with bold in Listing 3-17. The use of general entities within the markup can make the XML document much easier to read, especially when there is a large amount of text that must be repeated throughout a document. Some of the XML Security standards make use of general entities to provide for more readable examples.

All three of the general entities shown in Listing 3-17 are inside the *document prolog.* The document prolog consists of some mandatory pieces, which at the very least must include the *XML declaration,* and the *document type declaration.* The XML declaration is the string value `<?xml version="1.0"?>`. If a document prolog is included in an XML document, it *must* begin with this declaration. The XML declaration does little more than communicate the version number along with some other semantics such as the *document encoding* and whether the XML document refers to external files. The XML declaration is shown in Listing 3-17 on the first line.

Immediately following the XML declaration is the document type declaration. The document type declaration is designed to provide a grammar for the given XML document. It begins with the string `<!DOCTYPE`, is followed with a string (`Signature`), and must end with a closing `>`. Inside the document type declaration is where *markup declarations* such as our *general entities* belong. (The general entities are actually inside the *internal subset*, but we'll ignore this detail for now.) The reader should observe the syntax of both the XML declaration and the document type declaration. These declarations do not represent well-formed XML; that is, the syntax is special and doesn't use the element and attribute syntax shared

by the main markup. We will return to this point later and see what is being done to alleviate this funny syntax.

At this point the reader should have a basic grasp of the syntax of the document prolog. Aside from the *general entity* (replacement text) declarations, the document prolog usually fulfills a more dignified role as the chief mechanism for assigning a *formal grammar* for the current XML document via something called a *document type definition*. This term is deceptively similar to document type declaration, but refers to something a bit different. The *document type definition* (DTD) is the *collection* of internal and external resources (internal to the current XML document) that collectively provide a *formal grammar* for the XML document. This topic is discussed in more detail in the following section.

Document Type Definition (DTD)

This section discusses the document type definition (DTD), which is the set of rules and constraints for providing a *formal grammar* for an XML document. First we will look at a simple case of a DTD with a fictional markup language. Following this we will examine parts of a real DTD and see if we can make our way through it.

The DTD

In an earlier section the reader was challenged to consider Listing 3-1 and answer the question: "What language is that?" The answer was: "It is a *fictional* markup language that uses the syntax of XML."

This fictional language has a fairly well defined syntax described in the well-formed constraints for XML. It is easy for us to construct legal documents because we have no rules other than those imposed by the XML Recommendation. For example, consider Listing 3-18 that uses the same elements as Listing 3-1.

Listing 3-18

A fictional language that uses XML with no additional constraints

```
<Good_Beer>
  <Food> Samuel Adams </Food>
  <FrenchFries> Guinness </FrenchFries>
  <Beers>
    <Bad_Beer> Budweiser </Bad_Beer>
    <Bad_Beer> Fosters </Bad_Beer>
  </Beers>
</Good_Beer>
```

The reader should be staring blankly at Listing 3-18 and probably wondering why the elements don't make any sense. This is intentional—the syntax of XML is extensible. In fact, with only the well-formed constraints, one can argue that the syntax is *too* extensible. Why? Right now we can create semantically meaningless element combinations that amount to gibberish. We have created a fictional markup language, but at this point we have no way to constrain it in any meaningful way. We can't yet tell *valid* XML documents from *invalid* XML documents for our particular language. We don't have a measure for validity yet so we don't know if Listing 3-1 or Listing 3-18 is legal or illegal even though they are both *well-formed*.

The term *valid* has a special meaning in the context of XML. An XML document is said to be valid if it has been compared against a formal grammar and has not violated any parts of this grammar. This grammar is the document type definition and is usually a file somewhere that contains the rules for a particular markup language. Once the document type definition has been created, we can associate it with a given XML document via the document type declaration. To provide some motivation for this, let's create a simple grammar for Listing 3-1 that allows us to discern *valid* and *invalid* instances of our fictional markup language. Let's call this particular markup language the *Food* language, where our three food groups consist of good beers, bad beers, and french fries—a true diet of champions. Listing 3-19 shows an example of a simple document type definition that lives in a file separate from the main markup file.

The set of constraints shown in Listing 3-19 is roughly equivalent to the following English description: A document that uses `<Food>` as its root element must contain exactly two elements, `<FrenchFries>` and `<Beers>`. The `<FrenchFries>` element must contain some character data and the `<Beers>` element must contain at least one `<Good_Beer>` element or `<Bad_Beer>` element. Both a `<Good_Beer>` and `<Bad_Beer>` element must contain character data.

The syntax used for the document type definition can be quite intuitive, even though it looks weird. Without explicitly defining each line, the

Listing 3-19

Some constraints for the *Food* markup language

```
<!ELEMENT Food (FrenchFries, Beers)>
<!ELEMENT FrenchFries (#PCDATA)>
<!ELEMENT Beers (Good_Beer+ | Bad_Beer+)>
<!ELEMENT Good_Beer (#PCDATA)>
<!ELEMENT Bad_Beer (#PCDATA)>
```

reader should be able to follow the given English description and deduce the meaning of most of the notation used. The fancy name for each line in Listing 3-19 is *element declaration.* Each of these element declarations defines constraints for the element name. Let's look at the first element declaration:

```
<!ELEMENT Food (FrenchFries, Beers)>
```

Each element declaration begins with the string <!ELEMENT followed by the element name, and then the *content-model* and finally the closing character >. In the previous small example the element name is Food and the content-model we are declaring for this element is a sequence of child elements in a specific order. The order is denoted by the order of the element names within the parentheses. The first child element must be FrenchFries and the second (and final) child element must be Beers. The next element declaration is even simpler:

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

The previous example simply says that the FrenchFries element must contain only character data—it cannot contain any elements. The odd-looking keyword #PCDATA stands for *parsed character data*. The third element declaration gets a bit more complex, but it is still readable:

```
<!ELEMENT Beers (Good_Beer+ | Bad_Beer+)>
```

The content model here says that the Beers element must contain a *choice* (denoted by the | character) of *one or more* (denoted by the + symbol) Good_Beer or Bad_Beer elements. Cardinality operators such as +, ?, and * are used throughout document type definitions and mean *one or more, zero or one,* or *zero or more,* respectively. We will see these cardinality operators again in Chapter 4 when we look at the structure of the XML Signature. The last two element declarations are simply repeats of the second declaration and merely constrain the Good_Beer and Bad_Beer elements to only contain character data.

The reader should have a basic understanding of how to put together a simple grammar for a custom markup language. The reader is challenged to reconsider Listing 3-1. Is this XML document valid? Does it conform to the grammar set forth in Listing 3-19? The correct answer is no. The reason is because the <Beers> element contains more than one child element—it contains two <Good_Beer> elements and two <Bad_Beer> elements. Our document type definition constrains the <Beers> element

to a *choice* of either one, but not both. This is the constraint defined by the third element declaration.

Now that we have a simple document type definition, we need to associate this with an instance of our *Food* markup language. This is done so a processing application can find the formal grammar and perform the check in a seamless way. Luckily, the syntax to accomplish this association is not difficult. A complete, valid, *Food* XML document that points to an external document type definition is shown in Listing 3-20.

Listing 3-20 uses what is called a *system identifier* (denoted by the `SYSTEM` keyword) to inform the processing application of the document type definition. In this case, the DTD file lives somewhere on a server called food.com. While a remote URI is shown in Listing 3-20, any valid URI can be used for the system identifier value. A filename by itself usually signifies that the DTD file is in the same local directory as the XML document.

A Real DTD

This section looks at pieces of the DTD for an XML Signature (the entire DTD is boring). The reader doesn't have to know much about an XML Signature yet—the details will be covered in Chapter 4. This section is intended to give the reader some practice reading through a real DTD instead of a fake markup language about beer and french fries.

The first piece we are going to look at is the element declaration for the parent element, which is the `<Signature>` element, as follows:

```
<!ELEMENT Signature (SignedInfo, SignatureValue, KeyInfo?, Object*)>
```

Luckily, this element declaration is simple and is similar to a declaration made in the DTD for the *Food* markup language. The declaration says that a `<Signature>` element must contain a `<SignedInfo>` element, `<SignatureValue>` element, zero or one `<KeyInfo>` element, and zero or more `<Object>` elements. The beauty of this declaration is that it is

Listing 3-20

A valid instance of the *Food* markup language

```
<?xml version="1.0"?>
<!DOCTYPE Food SYSTEM "http://food.com/food.dtd">
<Food>
  <FrenchFries> Curly Fries </FrenchFries>
  <Beers>
    <Good_Beer> Samuel Adams </Good_Beer>
  </Beers>
</Food>
```

fairly easy to read and we don't yet have to know a thing about an XML Signature to understand the constraints. Here is another piece, which puts a constraint on the contents of the `<SignatureValue>` element such that it only contains character data and no elements:

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

Here is another piece of the XML Signature DTD that gives constraints for the `<SignedInfo>` element:

```
<!ELEMENT SignedInfo (CanonicalizationMethod,SignatureMethod,Reference+)>
```

The constraint simply says that a `<SignedInfo>` element must contain a `<CanonicalizationMethod>` element, a `<SignatureMethod>` element, and one or more `<Reference>` elements.

Learning More about DTDs

These small sections only give the reader the absolute basics with regard to DTDs, and the reader is urged to consult the references section at the end of this book for more information. Some might argue that we made the discussion easier than it really is, and in some cases we have simplified a few things. We omitted *attribute declarations* and *parameter entities*, both of which are found in the actual XML Signature DTD. An attribute declaration is similar to an entity declaration, but it provides constraints for attributes instead of entities and a parameter entity is similar to a general entity, but is intended for use inside a DTD.

XML Schema

Like document type definitions, the XML Schema definition language is used to describe the structure of XML. One view of XML Schema is an overhaul or upgrade of DTDs. Newcomers to XML often get frustrated by the fact that there are two tools for accomplishing roughly the same task. The frustration continues when it is also learned that DTDs are being replaced with corresponding XML Schema definitions.

The DTD faces two major problems:

- DTDs aren't powerful enough to provide sophisticated constraints on an XML document.
- DTDs don't *use* XML.

The first issue sounds plausible, but the second needs a bit of explanation. Reconsider the DTD syntax, specifically any of the previously discussed element declarations. The careful reader should notice that the syntax doesn't correspond to well-formed XML (that is, syntax in which element declarations are made in between an opening `<!` and closing `>` character). There is no end tag for the element declaration. The syntax of DTDs is a bit ironic; it can't be processed in the same way as the document it constrains. There is really no reason for this. The XML syntax is powerful and general enough to model constraints on instance documents that use XML. Switching to an XML-based markup language that constrains XML document instances is inevitable and makes processing the formal grammar easier. In addition, XML Schema is *extensible* in contrast to the DTD language, which is fixed and part of the XML 1.0 Recommendation.

We will not go into any specifics of XML Schema in this book; the topic is simply too large and complex to fit inside a single chapter, let alone a single section. To give the reader a flavor for what XML Schema looks like, we will present part of the Schema definition from the XML Signature Recommendation for a quick tour. The reader should visit the references section for more information on XML Schema. Consider Listing 3-21.

Fortunately, XML Schema is quite intuitive and extensive knowledge of the schema definition language is not required for discerning some basic constraints. XML Schema is much more general in its scope than DTDs and has the features of a programming language.

The two basic *types* in XML Schema are *simple types* and *complex types*. In general, complex types are types that contain other elements while

Listing 3-21

The XML Schema Definition for the `<Signature>` element

```
<element name="Signature">
  <complexType>
    <sequence>
      <element ref="ds:SignedInfo"/>
      <element ref="ds:SignatureValue"/>
      <element ref="ds:KeyInfo" minOccurs="0"/>
      <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
      <attribute name="Id" type="ID" use="optional"/>
  </complexType>
</element>
```

simple types cannot. In Listing 3-21, the element `<Signature>` is defined to be a complex type that contains a sequence of four elements: `<SignedInfo>`, `<SignatureValue>`, `<KeyInfo>`, and `<Object>`. The last two elements, `<KeyInfo>` and `<Object>`, have additional constraints declared using attributes that provide the same functionality as the DTD cardinality operators. Both of these elements are optional and `<KeyInfo>` is constrained to a single instance while `<Object>` is unbounded. Finally, there is also a provision for an attribute value called `Id`, which is also optional.

Processing XML

The reader should now have a fairly good idea of how XML documents are structured and validated, and should understand the difference between a markup language and a meta-language. Given a document that uses XML markup, the reader should be able to tell if it is well formed and should be able to discern some basic validity constraints with the help of a DTD. This section of the primer marks a shift from examining the syntax of XML to understanding how XML is *processed*. XML documents are cool to look at and fun to create, but unless we understand how they are processed and dealt with, the previous section is more a thumb-twiddling exercise rather than something practical.

The next few topics on our plate include understanding the Document Object Model (DOM), which is an API for doing practical things with structured documents, as well as the XPath data model, which is used in the XML Security standards. We will also look at some source code in Java that shows how to use the Apache Xerces toolkit to do a few practical things with XML documents such as parse them and output basic information.

The Document Object Model (DOM)

The DOM is an Application Programming Interface (API) meant for *structured documents*. The reader may be wondering why we are discussing an API at this stage in the book and may also wonder about the relevance of this section altogether. The reason the DOM is important is because it is highly standardized and represents a widely used and accepted program-

ming model for structured documents. This makes the API important because most XML Security implementations (XML Signature or XML Encryption) will have support in some way or another for the DOM; in essence, these implementations are usually written directly on top of the DOM and rely on its functionality and semantics. Because of this, it is an important building block in XML Security.

To support standardization, there are various levels of the DOM. The term *level* is akin to a version number for the DOM, where higher levels represent increased functionality. Our discussion here will include the features and behavior found only in DOM Level 1, and more specifically DOM Level 1 Core, which is the smaller subset of DOM Level 1.

Structured Documents vs. Structured Data

The reader may notice that we have not explicitly mentioned XML documents, but instead began this section using the more general term *structured documents*. The reason for this change in terminology is related to the definition of the *Document Object Model*. The DOM is an API designed for structured documents in general and isn't an API exclusive to XML documents. For example, the DOM can also model HTML documents using the same interfaces.

Our focus with the DOM will be XML documents, and because of this, the more specific term *structured data* is slightly more appropriate. The term *document* can be confusing because a document as such usually implies some sort of presentation coupling such as fonts, colors, graphics, or multimedia. These types of additional presentation semantics are out of scope for XML documents that represent security objects such as an XML Signature or encrypted XML element. We will look at the DOM in terms of structured data, instead of its wider scope of structured documents.

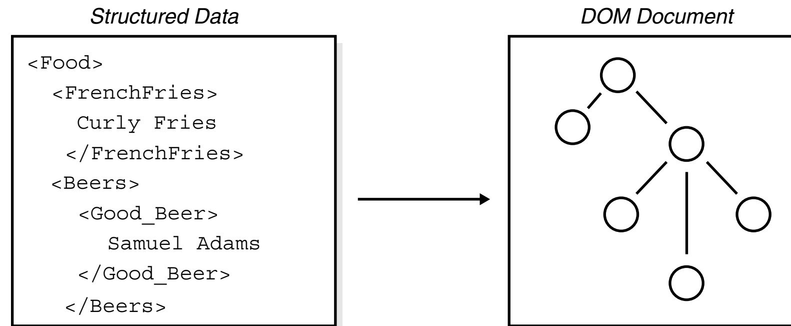
DOM Interfaces

DOM is a collection of interfaces for manipulating structured data in memory using *objects*. We are not throwing out a new term; the term *object* is the same as that found in an object-oriented programming language such as C++ or Java. Simply put, given arbitrary structured data, the DOM specifies interfaces that can be used to access and manipulate this data at a programming language level. A picture of this process is shown in Figure 3-2.

Figure 3-2 shows a *simplified* view of how the DOM looks at structured data. The DOM uses a tree-like structure to model the relationships

Figure 3-2

A simplified view of a DOM document object



between its interfaces, but does not constrain the actual implementation *to* a tree data structure. Put another way, the DOM appears to act like a tree from the outside, but a particular *implementation* of the data structures is not constrained to a tree structure. A tree structure is an obvious model for structured data such as an XML document. The parent element represents the root of the tree and each child element represents child nodes of the root and so on. A better picture of the scope of the DOM is shown in Figure 3-3.

In Figure 3-3 a new box appears with a question mark inside of it. This signifies that the actual implementation of the DOM is out of scope. A direct relationship between the tree-like interface provided by the DOM and the underlying implementation isn't necessary. DOM merely specifies interfaces; the underlying implementation is up to the vendor that takes on the task of creating a usable DOM API. The outside of the DOM—the user-accessible API—is fixed and has a logical structure that matches that of a tree. The inside—the actual implementation of the DOM interfaces—is not constrained and may or may not match the outer tree view. The idea here is that the DOM be a *portable* interface. For example, it may be desirable to add a DOM interface over some other legacy API that deals with structured data. Because of this, the DOM explicitly separates itself from the implementation of its objects.

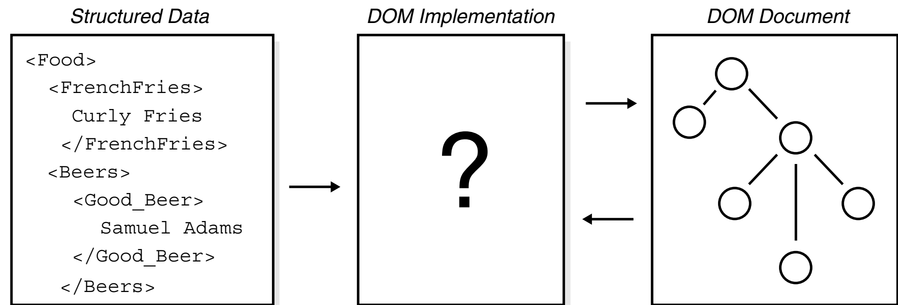
This key idea is important to remember; often the DOM will appear clunky and obtuse for basic tasks. The reason is because it is not designed with any one document format in mind, but instead for *arbitrary structured documents*, which may or may not include XML documents.

Inheritance View vs. Flattened View

We will examine the DOM as it is specified for an object-oriented programming language such as Java. DOM APIs exist for many scripting lan-

Figure 3-3

The scope of the DOM



guages and as such exist in two different views: a *flattened view* and an *inheritance view*. Not all languages support object-oriented features such as inheritance and this creates some redundancy in the API. This redundancy adds to the feature set in terms of extra functions and can cause confusion. The flattened view is out of scope for our discussion and the reader should visit the references section at the end of this book for more information on the DOM. All of the upcoming code examples will be given in Java, and, because of this, our primary view of the DOM will be the *inheritance view.*

Complete understanding of the inheritance view of the DOM comes from understanding just two objects: `Node` and `Document`. It is difficult to say which object is more fundamental; a clear understanding of both is a necessity for doing anything useful with the DOM. The logical structure of the DOM is a tree, and every object in this conceptual tree is some type of `Node` object. Consider Figure 3-4.

The first thing to notice about Figure 3-4 is that every node in the conceptual tree has been replaced with a concrete interface called `org.w3c.dom.Node`. The name of the object comes from the *Java language binding* for the DOM. A given *language binding* for the DOM is defined by the W3C and doesn't refer to a given DOM implementation or API. All compliant DOM implementations for Java *must* use the `org.w3c.dom.*` packages that define the DOM interfaces. Our focus with the DOM will be Java and it is appropriate the use the fully qualified interface names at this point in time.

The second thing to notice about Figure 3-4 is that all of the objects that represent the structured document in the figure are identical. All nodes are `org.w3c.dom.Node` objects. This is only true from an object-oriented *subtype* relationship. This is difficult to describe, but easy to show

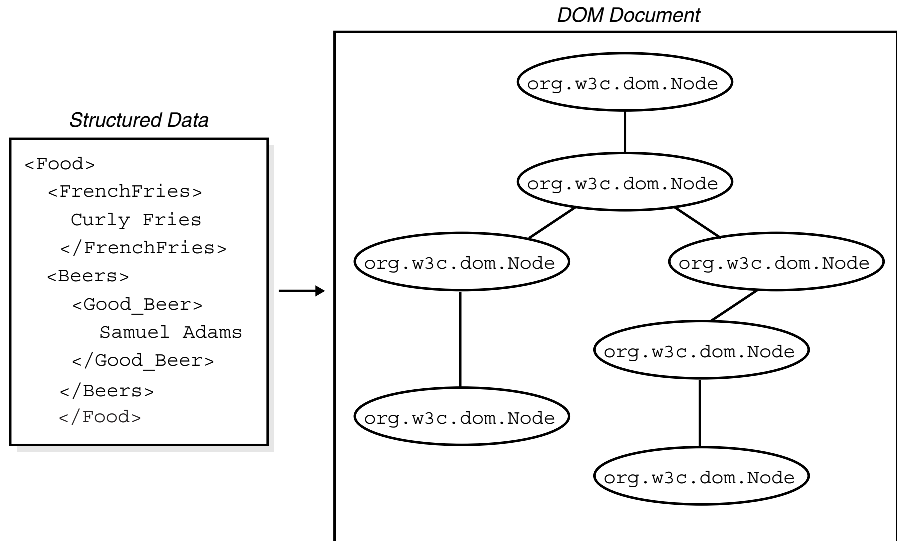


Figure 3-4

Node objects in the DOM tree

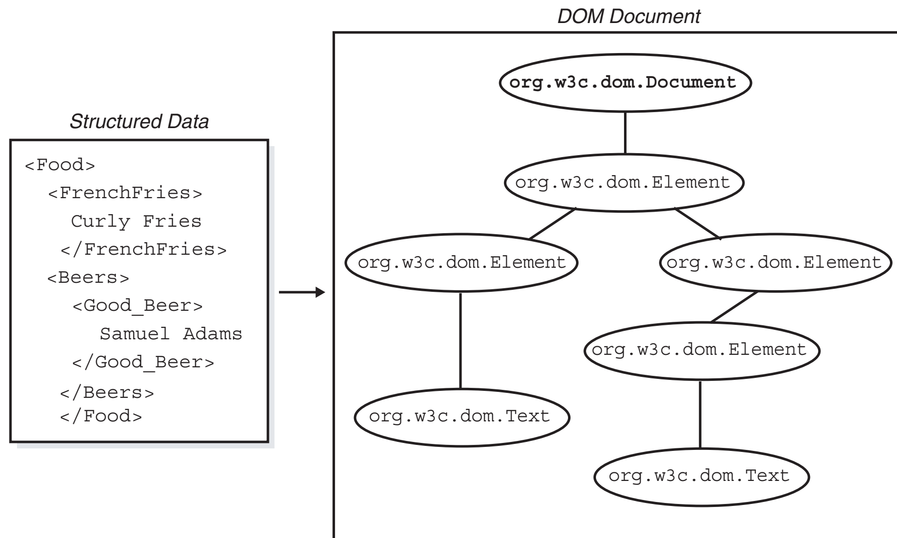


Figure 3-5

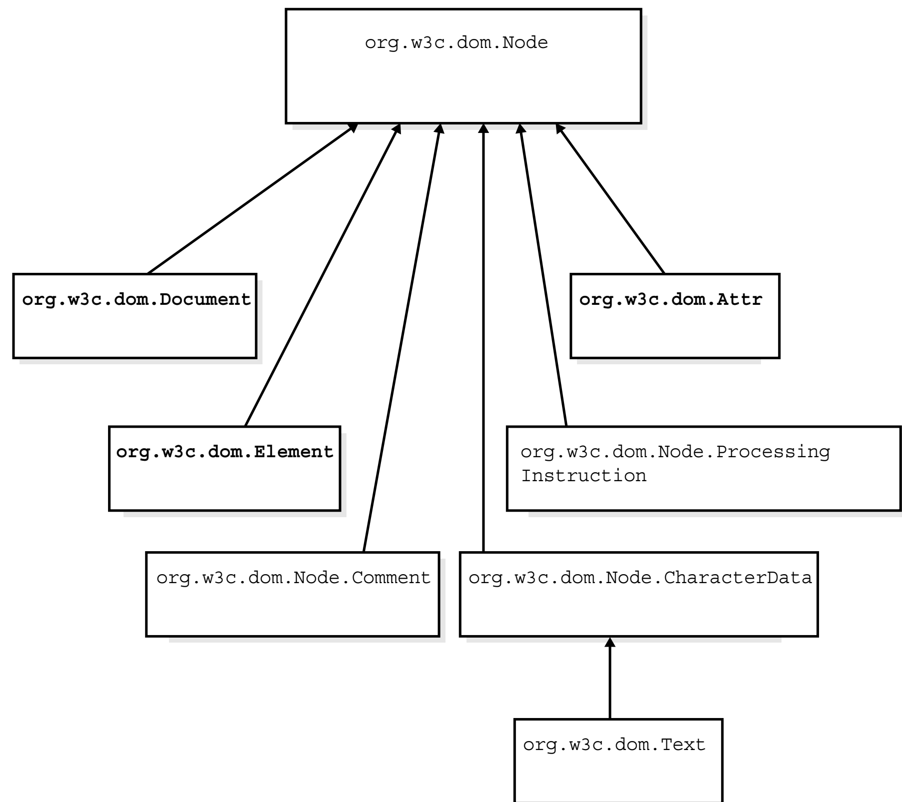
DOM objects

with a picture. Figure 3-5 shows the true objects for the sample XML document, and Figure 3-6 shows the parent-child relationships of the objects.

Figure 3-6 shows the parent child relationships of some common `org.w3c.dom.Node` subtypes. Not all possible subtypes are shown, and the most common ones are marked in bold. The most important thing to

Figure 3-6

DOM class hierarchy



notice about Figure 3-6 is that all of the classes shown are subclasses of `org.w3c.dom.Node` and because of this they properly fulfill the subtype relationship. Each subtype of `org.w3c.dom.Node` is designed to represent something in the structured document. For example, the `org.w3c.dom.Element` type represents *elements* in an XML document and the `org.w3c.dom.Attr` type represents *attributes* in an XML document. The list goes on; there are over 15 different subtypes of `org.w3c.dom.Node`. We will not cover them all here, because only a few have immediate interest to us. The reader should refer to the references section for more complete information on the DOM.

The careful reader should notice that there appears to be a mismatch between Figure 3-5 and the structured data shown. That is, there is an extra `org.w3c.dom.Document` node at the root of the conceptual tree that has no obvious match to anything in the XML document shown in the figure. This extra node is the main entry point into the structured

data and the real root of the XML document is actually the first child in the DOM tree.

This point showcases the importance of the `org.w3c.dom.Document` object, which represents the *entire* structured document. Once an `org.w3c.dom.Document` object is obtained for a given structured document, the user can access various `org.w3c.dom.Node` objects that comprise the document tree. Any use of the DOM to model an existing structured document begins with the creation of a `Document` object. Ironically, the actual bootstrapping of the `Document` object is left out of scope —it is the responsibility of the specific DOM *implementation* to provide the user with the necessary methods to create an instance of `org.w3c.dom.Document`. The next section shows how this bootstrapping process works with the Xerces XML Parser.

Bootstrapping with Xerces

Our first goal in this section is to use the Xerces XML Parser[1] (which has support for the DOM language bindings) to create an `org.w3c.dom.Document` object from some sort of real XML data. Once we create the `org.w3c.dom.Document` object, we can traverse the logical tree structure and get information about the XML document. Consider Listing 3-22.

The first things to note about Listing 3-22 are the import statements at the top. The previously discussed *DOM Java language bindings* and the Xerces DOM parser implementation are both added here. The two sepa-

Listing 3-22

Using Xerces

```
// Import the DOM Java language bindings
import org.w3c.dom.*;
// Import the DOM parser implementation
import org.apache.xerces.parsers.DOMParser;
class CodeListing31 {

  public static void main (String args[]) throws Exception {
    // Make a new DOM Parser
    DOMParser domParser = new DOMParser();
    // Parse an input document
    domParser.parse("food.xml");
    // Get the org.w3c.dom.Document node
    Document documentNode = domParser.getDocument();
    // Get the first child
    Element rootNode = (Element)documentNode.getFirstChild();
    // What is the name?
    System.out.println("Root element: " + "<"+ rootNode.getTagName()+">");
  }
}
```

[1]Xerces can be downloaded for free at http://xml.apache.og.

rate `import` statements showcase the separation of the parser implementation from the DOM interfaces that are specified in the language binding.

Once the proper DOM interfaces and the parser implementation have been imported, the actual constructor is called for the `DOMParser` class. This call and the next two calls are not relying on the DOM API; they are calls that are proprietary to the Xerces processor. In fact, the only calls in the code listing that use the DOM API are the second to last and last function calls.

Once an instance of the `DOMParser` class has been created, a call to the `parse()` function occurs. This is a *blocking call*. In other words, the program is halted while the DOM Parser reads from the specified XML file. This may not seem like a big deal, but when the size of the XML file grows to hundreds or thousands of lines this call has the potential to take a great deal of time.

Once the parsing is complete we are ready to obtain an `org.w3c.dom.Document` object with a call to `getDocument()`. This call returns the `org.w3c.dom.Document` object as specified by the DOM. The reader may think of this call as the transition point from the proprietary Xerces parser to the standard DOM APIs. From here the idea is to use only function calls that are specified by the DOM.

The first call obtains the actual root element in the XML document. In Listing 3-22 we are assuming that the input is the food XML document shown in Figure 3-5; the call to `getFirstChild()` actually obtains the `org.w3c.dom.Element` object that corresponds to the `<Food>` element. The actual string text in between the tag markup (`<` and `>`) in the input document is printed out with the final `getTagName()` call. The result of Listing 3-22 is the stunning output: `<Food>`.

This is hardly useful, but the sample does showcase how the logical structure of the DOM works. When using the DOM it is often desirable to rely on recursive semantics to traverse a structured document. Consider Listing 3-23, which prints out all of the `org.w3c.dom.Element` nodes and `org.w3c.dom.Text` nodes in the `food.xml` document.

All of the gory details are in the `printNode()` function. The idea here is to give this function an `org.w3c.dom.Node` object and it will determine the *type* of node and then make a printing decision. In Listing 3-23 we pass the first child of the `org.w3c.dom.Document` node (the first child is the document element) directly into the `printNode()` function. The first thing done inside `printNode()` is to determine which type of node we have; this function is trivial and only deals with two types of

Listing 3-23

Some recursion with the DOM

```
import org.w3c.dom.*;
// Import the DOM parser implementation
import org.apache.xerces.parsers.DOMParser;
class CodeListing32 {
  public static void main (String args[]) throws Exception {
    // Make a new DOM Parser
    DOMParser domParser = new DOMParser();
    // Parse an input document
    domParser.parse("food.xml");
    // Get the org.w3c.dom.Document node
    Document documentNode = domParser.getDocument();
    // Get the first child
    Element rootNode = (Element)documentNode.getFirstChild();
    // Let's print out all the element names and text nodes
    printNode(rootNode);
  }
  public static void printNode(Node nodeToPrint) {
    int type = nodeToPrint.getNodeType();

    if (type == Node.ELEMENT_NODE) {
      String nodeName = nodeToPrint.getNodeName();
      System.out.println("Element Node Found: <" +nodeName+">");
      NodeList childNodes = nodeToPrint.getChildNodes();
      if (childNodes != null) {
        for (int i=0; i<childNodes.getLength(); i++) {
          printNode(childNodes.item(i));
        }
      }
   }
      if (type == Node.TEXT_NODE) {
      String textValue = nodeToPrint.getNodeValue();
      System.out.println("Text Node Found: " +textValue);
    }
  }
}
```

nodes, so the choice is either `Node.TEXT_NODE` or `Node.ELEMENT_NODE`. These static identifiers are simply integers used by the DOM implementation to distinguish between different node types. If our node is an element, we first determine the name using `getNodeName()` and then print this out.

Once the parent element has been printed, the child nodes are next in line. A simple `for` loop is used to iterate through these and print them out one by one. The object used to hold the list of nodes is the `org.w3c.dom.NodeList` object. This object is a bit unique in its semantics. At first glance this object appears to model a list of nodes that can be accessed like an array. For example, in Listing 3-23 we use a for loop to iterate through this `org.w3c.dom.NodeList` object with an `item()`

function that appears to give us the contents at each position in the node list. Despite the way this object looks, it does *not* have perfect list-like or array-like semantics. This can be confusing for newcomers to the DOM.

The `org.w3c.dom.NodeList` is a *linear view* of the document tree. This means that if you *add* a node to an `org.w3c.dom.NodeList`, you are adding a node to the tree. Similarly, if you remove a node, a node gets removed from the tree. As the tree gets updated, the `NodeList` changes; there is no need to update the `org.w3c.NodeList`—it will change by itself. `NodeLists` are useful when it is desirable to do operations that require sequential access (such as printing out the children of a given node).

The second case is that of a *text node*. Text nodes are pervasive throughout even the simplest XML document (such as ours) because white space is considered significant in XML documents. This means that the `printNode()` function will be called more times than the visible contents of `food.xml`. Further, this means that we lied a bit in Figure 3-5. Figure 3-5 shows the document structure not counting white space in the original document. In other words, the tree shown in Figure 3-5 has white space nodes (which are proper `org.w3c.dom.Node` subclasses) removed for the sake of clarity. For example, the reader can see how the white space is counted by looking at the output of Listing 3-23, shown in Listing 3-24.

There are a total of seven `org.w3c.dom.Text` nodes found in the `food.xml` document, even though only two are apparent (`Curly Fries` and `Samuel Adams`).

Beyond DOM

The previous two sections represent a whirlwind tour of a common paradigm for processing XML data. The DOM and its tree structure model is only one way of processing structured documents. The tree structure

Listing 3-24

The output from Listing 3-23

```
Element Node Found: <Food>
Text Node Found:
Element Node Found: <FrenchFries>
Text Node Found:
    Curly Fries
Text Node Found:
Element Node Found: <Beers>
Text Node Found:
Element Node Found: <Good_Beer>
Text Node Found:
      Samuel Adams
Text Node Found:
Text Node Found:
```

model is easy to understand and straightforward to implement, but it is not always ideal for every practical situation. The previously mentioned *blocking* `parse()` call in Xerces can present a problem for memory constrained environments; every time a document is parsed, a logical structure is built. Applications that only need access to a single node of the tree must incur a large performance penalty in terms of memory.

To avoid this type of performance tradeoff, another processing paradigm is used that sees the structured document as a stream of events. For example, instead of building a logical structure in memory that matches the document, the document is seen as a continuous stream of pieces and components. Each piece of the structured document (this can be an element or attribute) usually represents some sort of *event* and something important is done upon the receipt of the event. XML documents processed as a stream have a performance benefit because no logical in-memory structure is created and the programmer has the flexibility to store the pieces that are needed as they come. The standard API for this type of processing is called the *Simple API for XML Processing* (SAX). We will not discuss the SAX or its use in this book, but it should be considered for applications that don't want to be locked in to the more memory intensive DOM paradigm. The reader should visit the references at the end of this book for more information on SAX.

The XPath Data Model

This next section marks a shift from the practical DOM API to the more conceptual XPath data model. The DOM structure model is intended to be an API for applications that process XML. The XPath data model, while similar to the DOM in its specification, is intended to be a *conceptual structure model* for an XML document.

There are two potentially confusing things about XPath and the XPath data model. First, the XPath data model appears to be very similar to the DOM structure model. Both data models use a logical tree that relies on nodes to represent pieces of the input document (such as elements and attributes). Further, they also have similar constructs for representing a *collection* of nodes. The key idea about the XPath data model is that it is only conceptual and exists as a standard way of referring to an XML document from an intellectual perspective. This means it gets used a lot in the XML standards and drafts, including the XML Security standards. For example, understanding the XPath data model is useful in under-

standing how the XML Signature Recommendation processes data as XML. The DOM model can't be directly used in this case because it is an actual API and specifying XML standards in terms of the DOM would tightly couple a given standard to a mode of implementation, which is out of scope for most XML-related standards.

The second confusing thing about XPath is that it is also a specification of a *path language* for traversing an XML document. This generally adds to the muddle because most people use XPath to write expressions for transforming and selecting pieces of an XML document. This will *not* be our main focus here; the data model that XPath provides is what is most important because it allows us to understand how XML standards view an XML document. Once the data model is understood, the reader is in good shape for understanding how XML documents are transformed.

XPath Nodes

The main construct in the XPath data model is the concept of a node. A node represents an actual piece of an XML document. The difference between a DOM-based `org.w3c.dom.Node` and an XPath node is the scope of what can be represented. An XPath node has a smaller scope in most respects and contributes to a slightly simplified conceptual view of an XML document.

For example, the DOM has an interface called `org.w3c.dom.EntityReferences` that extends `org.w3c.dom.Node` and can be used to model entity references in the input XML document. This enables a user to count the number of entity references in an input document—the point being that entity references show up in the DOM's view of an XML document. As another example, the DOM has an interface called `org.w3c.dom.DocumentType` that also extends `org.w3c.dom.Node`. This interfaces allows for access to information inside the document prolog, specifically the DTD, enabling a user to read parts of the DTD and print them out. Again, the DTD is in DOM's view of the XML document.

The situation is a bit different for the XPath data model; there are only seven conceptual node types. Because there are more than seven possible constructs in an XML document, XPath can't model everything and the view is necessarily simplified. The data model defined by the XPath Recommendation consists of the following seven node types: *root nodes, element nodes, text nodes, attribute nodes, namespace nodes, processing instruction nodes,* and *comment nodes*. Our aim is to eventually describe how a given XML document gets chopped up into these node types. That

is, we are about to describe the process by which the XPath data model is *applied* to a given XML document.

Before we discuss the actual nodes themselves, we need to approach something more fundamental, called *document order,* which is the order in which the XPath data model is applied.

Document Order

The term *document order* refers to the order in which the various node types are created, based on a real, physical XML document. This ordering is actually quite straightforward and simple, and can be best described as a list of rules that provide the "cooked" view of the XML document. Document order is summarized as follows:

Document Order

Nodes are organized in the order in which they appear in the XML representation with these constraints:

1. The root node comes first.
2. Element nodes occur before their children.
3. The attribute and namespace nodes of an element occur before the children of the element.
4. The namespace nodes occur before the attribute nodes.
5. The relative order of namespace nodes and attribute nodes is implementation dependent.

Everything about document order is straightforward except for the third point. First, XPath contrasts the DOM in that it respects namespace nodes, where DOM Level 1 just considers namespaces to be attributes (which they properly are). Further, an element node has an associated set of attribute nodes and namespace nodes that aren't defined to be proper children of the associated element node. Attribute nodes and namespace nodes are considered to be associated with or properties of a given element node. This view makes a lot of sense because visually an element and its attributes are adjacent in the XML document. The last point here is that document order looks at an XML document after general entities have been expanded. This is where some of the simplification occurs. The XPath data model doesn't have an entity node and simply treats replaced

entities as text. Remember, with the XPath data model, whatever was in your original XML document can only be seen as one of seven node types (six really, since the root node is fixed).

Now it is time to examine the seven node types and see how they are created from an example XML document. We will attempt to form the XPath data model view of Listing 3-25, which is an updated version of an earlier food XML document.

In Listing 3-25 we have updated our food XML document with some comments, a default namespace, and some attributes. From here, let's look at the node types in detail and see what we can come up with for an XPath data model.

Root Node

Only one root node represents every XML document in the XPath data model in its entirety (not just the document element). This point is often confusing for readers, many ask: "Why have a root node, when we really want to get to the document element of the XML data?" An easy way to explain this is to realize that other items (such as the comments shown) appear as siblings to the document element. This means that if we are to maintain our logical tree structure, we need a conceptual root node to hold the rest of the nodes that are neither parent nor child to the document element (root element). In Listing 3-25 there is one root node with three children (in document order): a comment node, an element (the document element), and a comment node. There is no xml declaration node, so this piece of information is also lost in the XPath data model view. Our conceptual XPath tree thus far is shown in Figure 3-7.

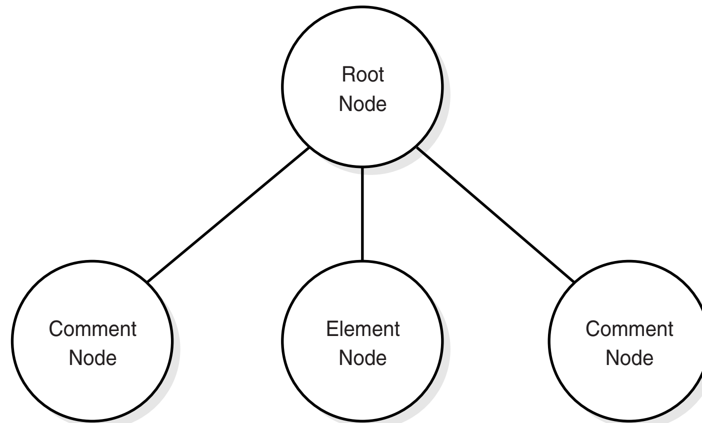
Listing 3-25

Updated "food" XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Here is a healthy meal -->
<Food xmlns="http://food.com">
  <FrenchFries Size=Large" Salted="True">
    Curly Fries
  </FrenchFries>
  <Beers Size="Pint">
    <Good_Beer>
      Samuel Adams
    </Good_Beer>
  </Beers>
</Food>
<!-- don't forget to always drink good beer -->
```

Figure 3-7

The beginnings of the XPath tree for Listing 3-25



Element Node

Element nodes represent actual elements in the physical XML document. There is nothing too tricky here. The only other thing to note is that the possible child nodes of an element node includes element nodes, comment nodes, processing instruction nodes, and text nodes. All of these nodes haven't been discussed yet, but we will get to them.

If we add element nodes to our XPath tree, the result looks something like Figure 3-8. A total of three element nodes are added, one for each element in Listing 3-25 (excluding the root element).

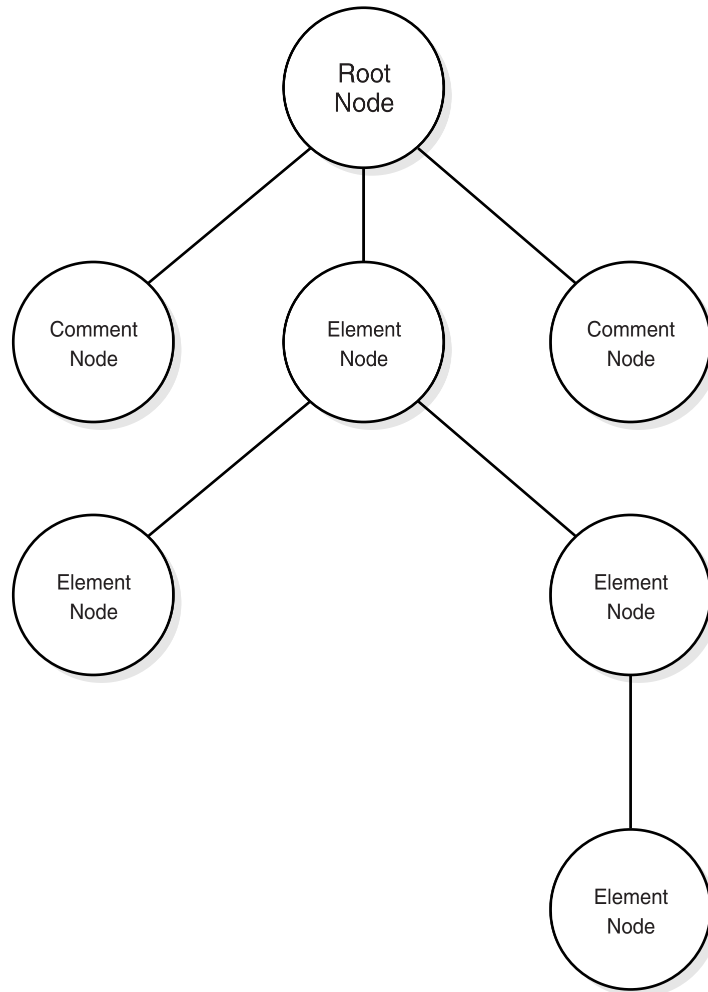
Attribute Node

Attribute nodes are slightly more confusing than element nodes. The relationship between an element and its attributes can be thought of as *association*. An element may have attribute nodes associated with it. The confusing part is that the XPath Recommendation defines the element node-bearing attributes to be the parent node of the attributes, but the attributes are not child nodes of the element node. This sentence can be confusing because the term *parent* when used in discussions of a tree structure usually logically implies the presence of *children*. XPath uses the term *set* to describe the attributes associated with a given element—we will expand upon this set idea and use such a notation for our expanded XPath view of Listing 3-25, which now contains attributes. This is shown in Figure 3-9.

The reader should notice that in Figure 3-9 the relationship of the attribute sets to each element. Visually, it makes sense to call the element

Figure 3-8

Adding element nodes to the XPath tree

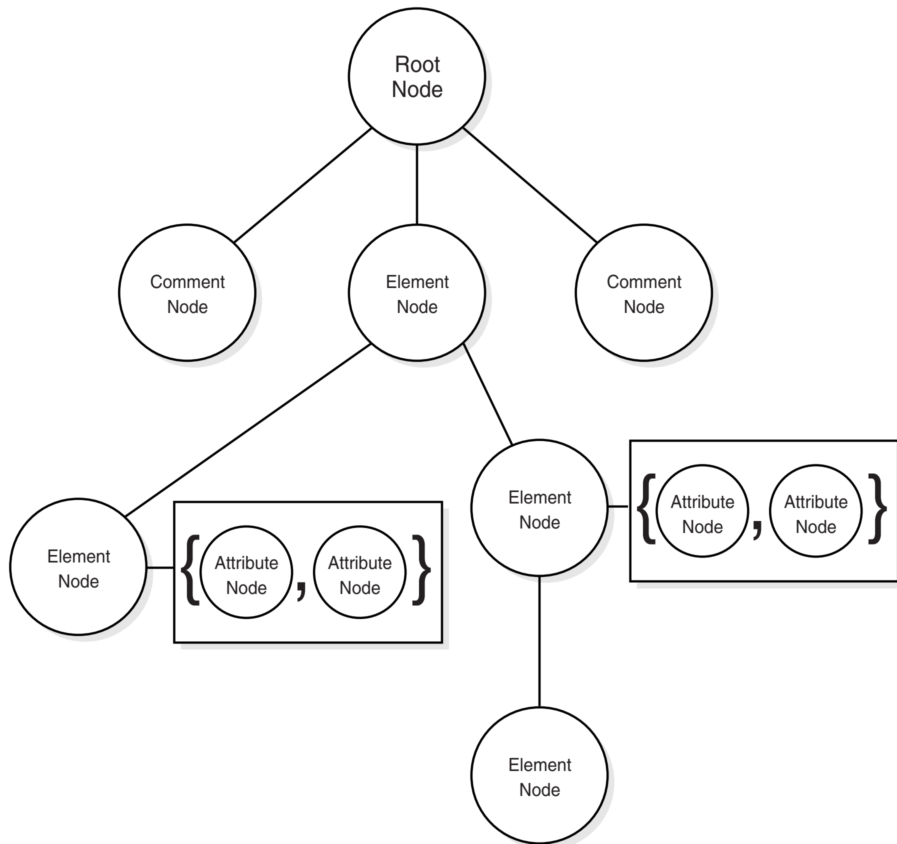


bearing the attributes a parent of the attributes, but it should also be clear that the attributes are not proper children of the element because they are not really intermediate nodes in the logical tree.

Another thing to note is that attribute nodes only appear in the XPath view for attribute nodes explicitly specified in the physical XML document being modeled (or those specified with default values in the DTD). A DTD can specify that attributes can take on default values. This isn't something that was covered earlier, and the reader is urged to visit the reference section at the end of the book for more information about DTDs.

Figure 3-9

Adding attributes to the XPath tree



In short, there can be some ambiguities between XPath views of an XML document because XPath doesn't require that the DTD be read (the XML Parser used may not support it). This means that it is quite possible for two identical XML documents to produce different XPath nodes based on the presence or absence of the DTD.

Namespace Nodes

The treatment of a namespace node is similar to an attribute node because namespaces are spiced up attributes from an XML syntax standpoint. Similarly, a given element node is associated with a set of namespace nodes. There is one namespace node for every namespace that is in scope for the current element. For example, if a namespace node was

declared on an ancestor node and is still in scope (it hasn't been overridden or undeclared), then there is a namespace node for this namespace.

Finally, the additional qualifier here is that the namespace nodes should be first in the set; for the picture we will use the same set for both the attribute nodes and namespace nodes. Listing 3-25 only has one namespace node, the default namespace. This namespace, however, is in scope for the entire food XML document. This means that there is a namespace node for *every* element node in our picture. The updated picture is shown in Figure 3-10.

In Figure 3-10 the picture looks a bit skewed as we try to fit everything together. The reader should notice that we have put the namespace nodes first in the associated set. This has to do with document order and the constraint that says namespace nodes must occur before attribute nodes.

Text Node

Text nodes represent any sort of text in the document. There is an XPath text node for all text in the document except for text inside comments, processing instructions, and attribute values. The XPath view of an XML document simplifies text defined using predefined character entities or residing in CDATA sections. That is, an XPath text node doesn't tell you if the text *came from* a predefined character entity or a CDATA section. This information is essentially lost; XPath models all text the same way. Consider the following short example:

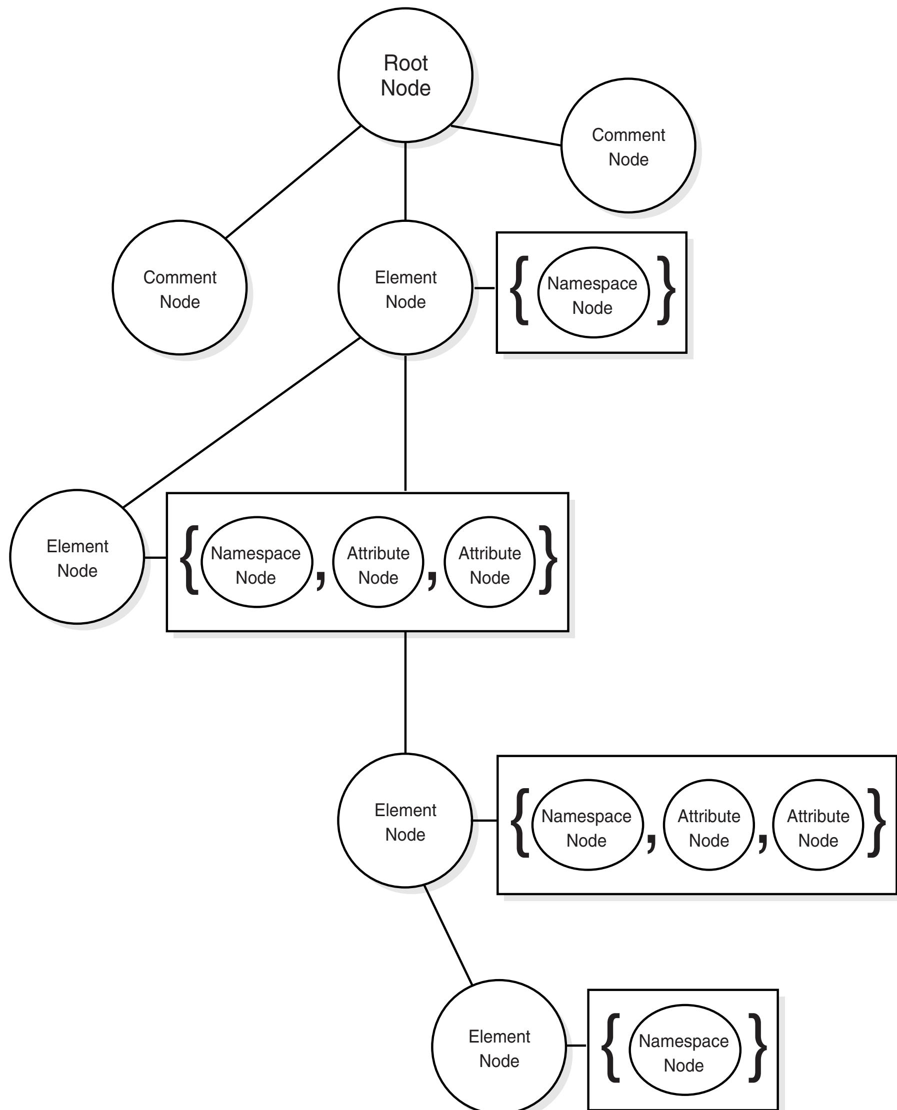
```
<Sample>
  <element1> I love XPath! </element1>
  <element2> <![ CDATA [ I love XPath ]]> </element2>
</Sample>
```

XPath sees both <element1> and <element2> as an element node and a text node; the presence of the CDATA section is lost. From this it follows that with the XPath view of an XML document it can't be determined if a given text node had its origins as a CDATA section or simply normal markup. Figure 3-11 shows the final picture for our XPath view of Listing 3-25.

In Figure 3-11 two text nodes were added. One text node corresponds to the Curly Fries text and one text node corresponds to the Samuel Adams text.

Figure 3-10

Adding namespace nodes to the XPath tree



Processing Instruction Node

This node is rather boring for our purposes. In short, XPath can model any processing instruction in the physical XML document except for processing instructions inside the DTD. There is a single processing instruction node for every actual processing instruction found inside the body of the document.

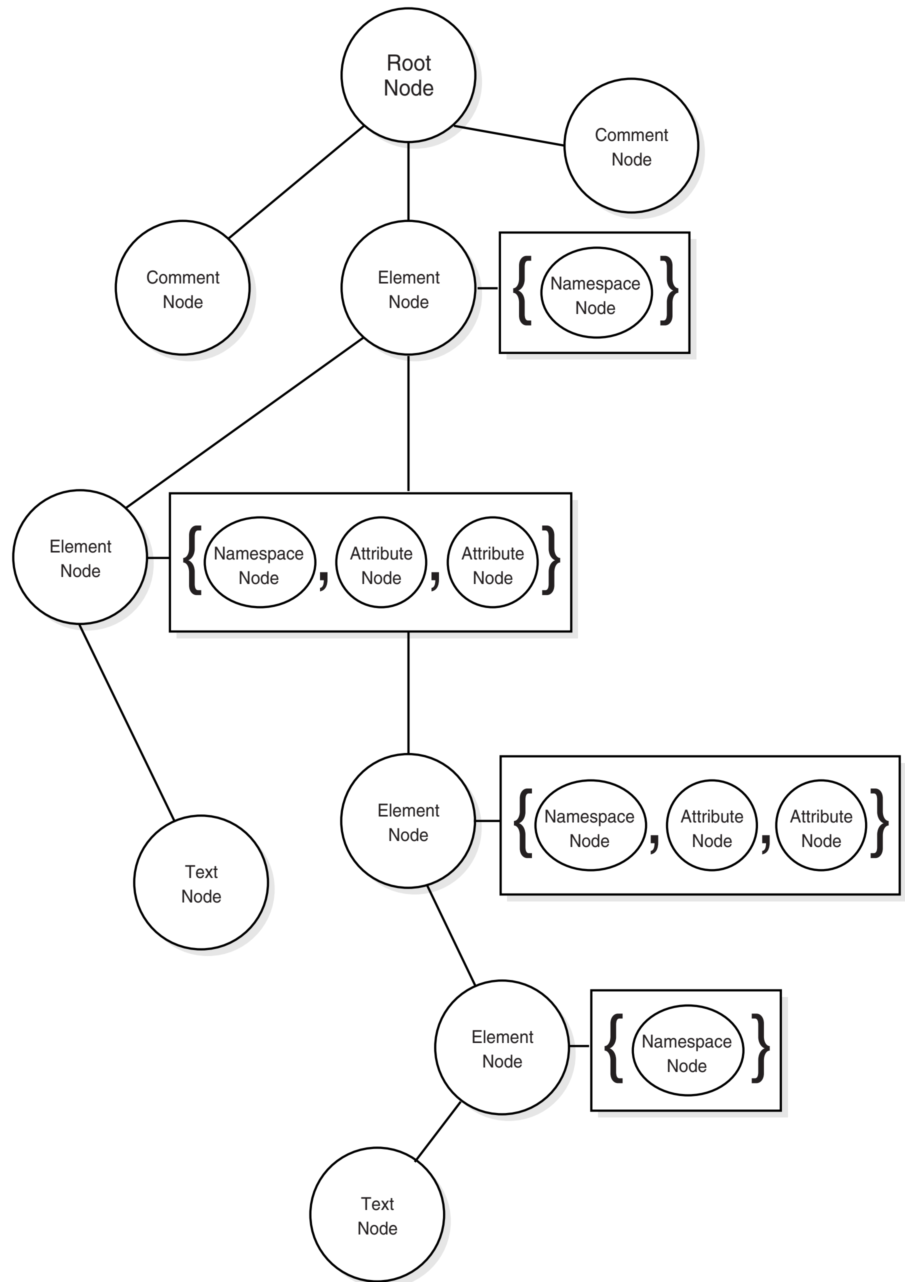


Figure 3-11

The final XPath view for Listing 3-24

Comment Node

We have already seen this node in action. There is one comment node for every actual comment found in the physical XML document. Again, any comment can be modeled except for comments inside the DTD.

XPath Node Set

Understanding the XPath data model is important because of a single term: *node-set*. This term is used throughout the XML Security standards and is an unordered collection of XPath nodes. Now that you know what the possible node types are, you also now know what a node-set is in this context. The node-set is simply an XPath tree that has been serialized from a tree to a flat list. The reader should also understand that the standard `org.w3c.dom.NodeList` object cannot be used to model an XPath node-set without some changes. A node-set is a proper list, while the `org.w3c.dom.NodeList` is a linear view. This is an important distinction to make because, at first glance, the two concepts appear to be interchangeable.

More on XPath

The main focus of the XPath Recommendation is not the previous data model. The data model is actually the last topic in the XPath Recommendation (some would consider this odd, since it is foundational for XPath). XPath is more concerned with the language and function library for traversing through an XML document and selecting document subsets. This discussion, while interesting, falls out of scope for our goals in this book. The reader will see some simple (largely self-explanatory) XPath expressions in Chapters 5 and 6 that perform some basic selection, but an in-depth tutorial on the matter is left to another resource. The reader should visit the references section for more information on XPath.

Chapter Summary

This chapter has been an XML primer on XML divided into two subsections: syntax and processing. The chapter began with discussing the fun-

damentals of XML syntax including a discussion of elements, attributes, and documents. Well-formed XML was then discussed with some introductory material on XML namespaces and document type definitions, including a small amount of information on XML Schema. The processing section focused on two main topics: The Document Object Model and the XPath data model. Both are fundamental for processing XML as a logical tree structure. The distinction was made between the DOM, which is a practical API, and the XPath data model, which is a conceptual tree model used in XML standards. This distinction was noted as important because both models use a node-based tree structure, but have a different purpose and semantics.