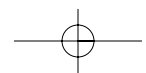
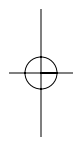
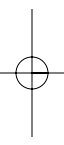
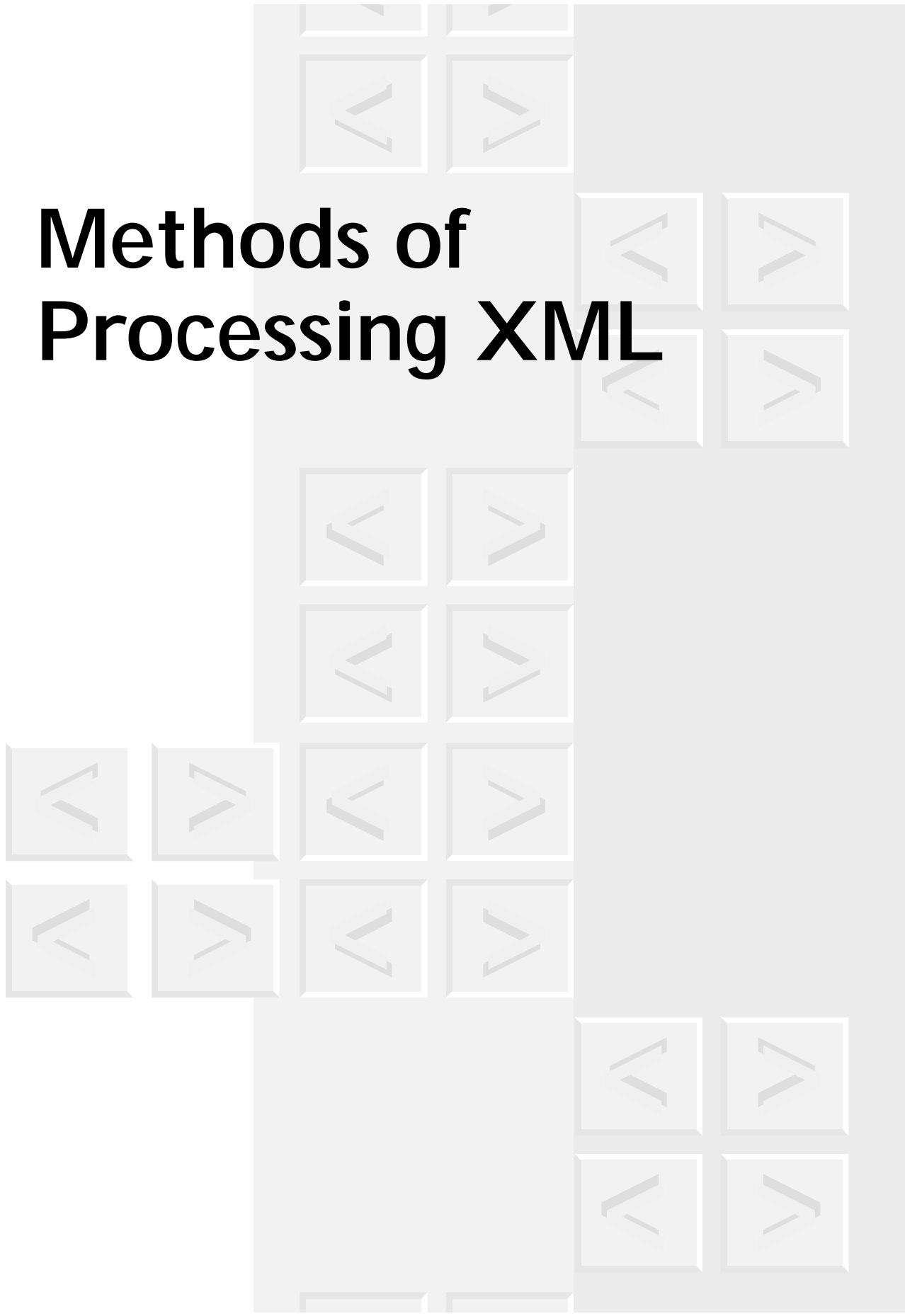


# Methods of Processing XML



# Chapter 3

## 3.1 | Introduction

In this chapter, we will examine the methods of processing XML documents. This foundation will be used throughout the book to assist us in developing higher-level XML processing functions for Enterprise Application Integration (EAI). It is also the most logical place to start a discussion on XML. Without understanding the following concepts, it will be difficult to exercise XML inside more complex business processes.

This chapter starts with an overview of the requirements for parsing XML documents. From there, we will explore some of the programmatic ways that parsed XML data can be accessed. At the end of this chapter, you will have an excellent understanding of how to build higher-level processes based on data formatted in XML.

## 3.2 | Parsing XML

A *parser* is executable processing logic that ensures that a particular document conforms to a specific grammar. Consequently, an XML parser is responsible for ensuring that XML documents conform to the XML grammar. This section examines the details regarding how to parse an XML document.

XML is a specification for a grammar that defines a set of tokens and the sequential ordering of these tokens that allow it to represent data hierarchically. The results of this specification are the following:

1. Documents have a set of rules to be followed for their creation.
2. The grammar defines how to build a process to verify that a document has followed the rules of the grammar, and thus is valid.

This is true of any computer language, whether it is a markup language or a programming language. This is accomplished by means of “production rules.” The rules define the “nonterminal symbols” composed of both tokens and/or nonterminal symbols.

There are many ways to define a grammar, but the most popular one in use today is the Extended Backus-Naur Form (EBNF). The grammar defines the set of acceptable token sequences, which in turn defines the syntactic correctness of a statement in a language.

Example 3.1 below is a production rule for a `First` symbol. It states that the `First` is a nonterminal of a single `Name` symbol.

**Example 3.1: Sample EBNF production rule.**

---

```
First ::= Name
```

---

As the grammar has no production for `Name`, it is a token, or terminal symbol. Every machine-language definition has at least one root production, called a “starting nonterminal,” that breaks down

into a set of productions for statements that can be formed by that language. When reconstructing tokens from a character stream, a parser needs to assume that the starting nonterminal will be satisfied, and attempts to verify that the resulting tokens do indeed match that assumption.

Interestingly, an EBNF grammar is very similar in design to an XML Document Type Definition (DTD) content model; this is by design. In essence, XML with DTDs provides a way to build a generic parser for tag-based notations. This is why XML is called a meta-language. It is a language for creating other languages, just as EBNF is a language for creating other languages.

EBNF nonterminal symbols keep expanding into other nonterminals until a terminal token is eventually reached.

**Example 3.2: An EBNF grammar.**

---

```
Person ::= First Middle Last
First  ::= Name
Middle ::= Name
Last   ::= Name
```

---

In Example 3.2, the starting nonterminal `Person` is comprised of three other nonterminals: `First`, `Middle`, and `Last`. Each of these nonterminals is comprised of a single token called `Name`, which is eventually is composed of a sequence of characters.

Notice also that grammars are designed using a component-like philosophy. The declarations define a set of types. `First`, `Middle`, and `Last` symbols could easily have been declared as nonterminal tokens, but this would make the grammar difficult to change if necessary in the future. Using this current definition, a change to the `Name` token will be propagated all the way back to `Person`.

Once you understand how grammars work, you can read and understand the definitions within the XML 1.0 grammar. Grammars also serve another useful purpose: They are directly convertible to state diagrams, which are then used to code the parsing logic. That is, to simplify the development of the parser code, it is best if the developer first builds a state diagram from the EBNF in the XML specification.

These state diagrams then become the logical statements that eventually identify a particular symbol.

Parsing is traditionally a two-phase process. First, the lexical analyzer parses the document and turns raw character streams into a series of tokens. These tokens are then processed by the syntactic checker and compared against the original EBNF rules to ensure that the document is syntactically correct. Sometimes, with simple and lightweight languages, such as XML, it is possible to compress these two steps into a single function.

Let's explore how the parser will ensure that an XML document is well-formed. Again, a well-formed XML document is one that follows the rules of the 1.0 grammar, but may or may not follow the rules of a DTD. For the rest of this discussion, we will refer to the following XML document (Example 3.3):

---

**Example 3.3: XML Document for Parsing.**

```
<?xml version="1.0"?>
<ROOT>
  <ELEMENT_1>Some Text</ELEMENT_1>
  <ELEMENT_2 attribute="1">
    <ELEMENT_3>Some More Text</ELEMENT_3>
  </ELEMENT_2>
  <?PI_1?>
  <ELEMENT_4/>
  <!-- Comment -->
</ROOT>
```

---

**Example 3.4: The first rule of the XML Grammar.**

---

```
[1] Document ::= prolog element Misc*
```

---

The rule in Example 3.4 tells the parser that it needs to break the entire document up into two or three major symbols (there may be zero `Misc` symbols). The goal of parsing is to ensure that the XML document meets the criteria for being a `Document`. To do this the parser will start with the first symbol and attempt to locate that symbol within the document. If the first symbol is optional (indicated by the \*), then it will look for the first and second symbols, whichever

comes first, and so on until it can make a suitable match, or it fails because no matches occur.

In Example 3.3, the prolog consists of the XML declaration processing instruction. If we had defined a DTD, that too would be considered part of the prolog. Thus, our parser starts reading in characters. It finds the less than (<), question mark (?), x, m, and l, which tell it that this is the XML processing instruction. The parser will continue scanning characters until the greater than (>) character is found. Along the way, the processor will find pseudo-attributes pertinent to parsing, such as the XML version and language encoding. The parser is responsible for extracting these pseudo-attributes and using them to prepare the parser for forthcoming text.

When the parser comes across the > character in the XML declaration processing instruction, it knows it has a complete `XMLDecl` (Declaration 23 as illustrated in Example 3.5). The next step is a bit more tricky for the parser as the next symbol could be a `Misc`, a `doctypeDecl`, or an element.

**Example 3.5: Prolog rule.**

---

```
[22] prolog ::= XMLDecl? Misc* (doctypeDecl Misc*)?
[23] XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDDecl? S? '?>'
[24] VersionInfo ::= S 'version' Eq (' VersionNum ' |
    " VersionNum ")
[25] Eq ::= S? '=' S?
[26] VersionNum ::= ([a-zA-Z0-9_.:] | '-')+
[27] Misc ::= Comment | PI | S
```

---

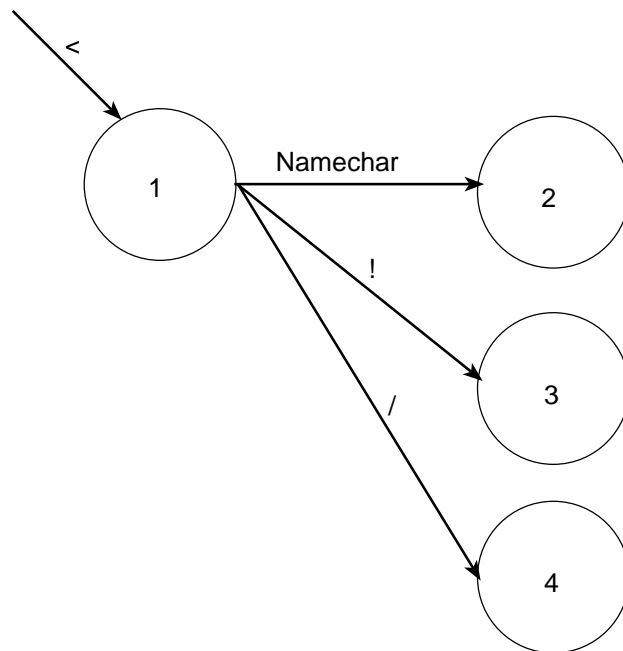
As the parser examines the stream of characters in the XML document from start to finish, it continually manages a stack of symbols. When a token is found, the stack is evaluated to see if the current set of symbols matches a valid XML rule. If it does, those symbols are removed from the stack and replaced with the name of the matched symbol. Eventually, the stack should look just like the rule in Example 3.4.

We mentioned earlier in this chapter that besides helping to identify if a document is syntactically correct, grammars can also assist us in developing the actual code necessary to build the lexical analyzer

(the piece that turns character streams into tokens). To move from EBNF to building the lexical analyzer, we must first build a set of state diagrams. State diagrams visually identify the states that the parser can be in at any one time and the legal states that can be achieved from the current state.

Figure 3-1 below illustrates the state changes that can occur once we've detected the start of an element by the < character. It shows us that the next character must be a legal element type name start character (Namechar), an exclamation mark (!), or a forward slash(/). In turn, states 2, 3, and 4 will be expanded to illustrate the valid states that they support.

As long as the character stream continues along a path of legal states, the lexical analyzer will keep spitting out tokens. However, if a character should force the parser to attempt to create an illegal state,



**Figure 3-1** State diagram.

then the parser will immediately know that the XML document is not well-formed. This should result in a fatal parsing error.

Converting the XML 1.0 grammar to a state diagram is a tedious chore. This is because states operate differently from a grammar. For example, the XML 1.0 grammar breaks down the `prolog` rule into its component symbols and defines rules for them within close proximity (see Example 3.5). However, the developer of a lexical analyzer would rather see all the rules with exclamation points (!) following a less than sign (<) grouped together. That is, grammar represents the syntactic correctness, but the lexical analyzer is interested in lexical—sequences of lexicons—correctness.

### 3.2.1 *Parsing XML in Java*

This section will explore building a syntactical parser in Java. Its job will be to ensure an XML document follows the 1.0 recommendation. Once a token is found, there are a number of different things that can be done with it. Some of these will be explored later in this chapter when we cover the Simple API for XML and the Document Object Model (DOM).

The first step in building a syntax checker for XML is to define a way to ensure that the characters are within the scope of legal characters as defined by the XML 1.0 grammar. There are many algorithms that one could choose to accomplish this. For this book, we will use a Boolean array, where the value of a character acts as an index into the array; legal characters are set to true.



**Note** *The code for parsing shown in this book, though illustrative of the logic and algorithms derived from a state diagram, is not optimized for business solutions. For professional software development we recommend downloading one of the freely available Java-based XML parsers on the Internet; one is supplied with this book.*

```
class CharacterSet {  
    boolean [] setarray = new boolean[0x10000];  
}
```

- 
- *Creates an array of 10000 booleans. Each location represents one Unicode character value.*
- 

```
CharacterSet() {
```

- 
- *CharacterSet() constructor initializes the array.*
- 

```
    for (int i=0; i<0x10000; i++)
        setarray[i]=false;
}
CharacterSet(CharacterSet original) {
```

- 
- *This version of the constructor uses a previously defined CharacterSet object. This is because some character sets share some properties.*
- 

```
    for (int i=0; i<0x10000; i++)
        setarray[i] = original.setarray[i];
}
void SetRange(int start, int stop) {
```

- 
- *The SetRange function sets a sequential range of character values to true.*
- 

```
    for (int i=start; i<=stop; i++)
        setarray[i] = true;
}
void SetRange(CharacterSet base) {
```

- 
- *Updates a range using an alternate CharacterSet object. If the value of the supplied set is not true, it uses the current CharacterSet value.*
- 

```
    for (int i=0; i<0x10000; i++)
        setarray[i] = base.setarray[i] ? true : setarray[i];
}
void SetChar(int location) {
```

- 
- *SetChar sets the value of one character location to true.*
-

```

    setarray[location] = true;
}

boolean inSet(char chkChar) {

```

---

■ *inSet tests whether a character is valid by testing to see if it is in the character set.*

---

```

    return setarray[chkChar];
}
}
class CharacterSets {

```

---

■ *The CharacterSets class builds specific character sets based upon the XML 1.0 specification.*

---

```

CharacterSet setLegalChars = CreateLegalCharSet();
CharacterSet setLetter = CreateLetterSet();
CharacterSet setNameStartChar = CreateNameStartCharSet();
CharacterSet setDigit = CreateDigitSet();
CharacterSet setNameChars = CreateNameCharSet();

private CharacterSet CreateLegalCharSet() {
    CharacterSet cs = new CharacterSet();

    cs.SetChar(0x0009);
    cs.SetChar(0x000A);
    cs.SetChar(0x000D);
    cs.SetRange(0x0020, 0xD7FF);
    cs.SetRange(0xE000, 0xFFFFD);

    return cs;
}

private CharacterSet CreateLetterSet() {
    CharacterSet cs = new CharacterSet();
    . . .
    return cs;
}

private CharacterSet CreateNameStartCharSet() {
    CharacterSet cs = new CharacterSet(setLetter);

```

```

        cs.SetChar('_');
        cs.SetChar(':');
        return cs;
    }

    private CharacterSet CreateDigitSet() {
        CharacterSet cs = new CharacterSet();
        . . .
        return cs;
    }

    private CharacterSet CreateNameCharSet() {
        CharacterSet cs = new CharacterSet (setNameStartChar);
        cs.SetRange(setDigit);
        cs.SetChar('-');
        cs.SetChar('.');
        return cs;
    }
}

```

The lexical analyzer requires five separate character set arrays: `LegalChar set`, `Letter set`, `NameStartChar set`, `Digit set`, and the `NameChar set`, which are defined above to illustrate their configuration. The sets that are missing detail are extremely large and use the character set definitions that can be found at the end of the XML 1.0 specification.

In addition to developing a class to handle validating characters, we have also created a class to represent the XML document being parsed and to keep track of the context. The `Element` class implements `java.util.Vector`, which essentially provides us with dynamic array functionality and allows us to add child elements more easily.

```

import java.util.Vector;
import java.util.Enumeration;
import java.util.Hashtable;

class Element extends Vector {

    private Element parent=null;

```

---

■ *parent Represents the current parent element.*

---

```
private String name = null;
```

---

■ *name represents the element type name of the XML element.*

---

```
boolean collecting = false;
```

---

■ *collecting tells us we're collecting characters that belong to the element's content. When we hit a new element start-tag or this element's end-tag, we will convert the collected characters into a string and add it to the current element's content buffer.*

---

```
private StringBuffer text = new StringBuffer(10);
```

---

■ *text is the buffer for storing the characters until they can be converted into a string.*

---

```
private Hashtable attrs = new Hashtable();
```

---

■ *attrs holds the attributes as key/value pairs in a hashtable.*

---

```
Element() {
```

---

■ *The Element constructor calls the constructor for Vector.*

---

```
    super();  
}
```

```
void setName(String s) {
```

---

■ *Allows the parser to name the element type of the Element object once it is created.*

---

```
    name = s;  
}
```

```
String getName() {
```

- 
- *Allows the parser to retrieve the element type name of an `Element` object.*
- 

```

    return name;
}

void addElement(Element n) {

```

- 
- *Adds a child element to the current element. This builds the graph of objects that represents the XML document hierarchy.*
- 

```

    CheckCollecting();
    super.addElement(n);
}

void setParent(Element p) {

```

- 
- *Sets the parent attribute within the `Element` object. Doing so creates a bi-directional parse tree.*
- 

```

    parent = p;
}
Element getParent() {

```

- 
- *`getParent` provides the parser with a method of moving up and down the tree.*
- 

```

    return parent;
}
void addPCDATA(char p) throws Exception {

```

- 
- *While adding characters, we set `collecting` to true and append to the internal text buffer. If `collecting` is false, we first create a new text buffer.*
- 

```

if (!collecting) {
    text = new StringBuffer(10);
    collecting = true;
}
text.append(p);

```

```

}

void setComplete() {

```

---

■ *This method is called by the parser on the current `Element` when its end-tag is found. It is unnecessary to call this method for an empty `Element` since it will have no PCDATA.*

---

```

    CheckCollecting();
}

void addAttribute(Object key, Object value) {

```

---

■ *`addAttribute` stores a new attribute in the table. If one of that name exists already, it will change the value to that of the new one.*

---

```

    attrs.put(key, value);
}

Object getAttribute(Object key) {

```

---

■ *Gets an attribute value from the set based upon the supplied key.*

---

```

    return (attrs.get(value));
}

void CheckCollecting() {

```

---

■ *This function will take all data text collected so far and convert it into a `String` object to add to the vector.*

---

```

    if (collecting) {
        collecting=false;
        String value = new String(text);
        super.addElement(value);
    }
}
}

```

The next stage of parsing is to read characters from a stream and determine if they are legal XML characters. If they are then they imply some construct that must be determined based upon where they appear. For example, a less than symbol (<) signals the parser that some XML-specific construct is coming. Likewise, a greater than symbol (>) will signal that that construct has terminated.

The following Java source code simplifies parsing by combining syntactic identification with the tokenizing process. This requires that we keep some additional state around so we can identify the context of any character. Of note, we use the design of the application itself to keep state. For example, when a < is found we delegate to the `HandleTag` function. The `HandleTag` function has the job of either finding a valid XML tag or throwing an exception. Therefore, the fact that `HandleTag` function was called represents a state within the parser.

```
import java.io.*;
import java.net.*;
import java.util.Enumeration;
import com.ncfocus.xmlparse.CharacterSets;
import com.ncfocus.xmlparse.Element;
```

```
class XMLParse {

    static char EOF = (char) -1;

    CharacterSets sets = null;
    InputStream stmXML = null;
    Element current = null;
```

---

■ *The parser must always point to the current `Element` object being evaluated.*

---

```
    boolean emptyElement = false;
```

---

■ `emptyElement` *allows the parser to know that it does not need to wait for an end tag.*

---

```
    public static void main (String args[]) {
```

---

■ *main is the root function that will create a parser object and determine if the document is well-formed.*

---

```
XMLParse xp = new XMLParse();
try {
    URL ptrXMLFile = new URL(args[0]);
    xp.stmXML = ptrXMLFile.openStream();
    xp.Parse();
} catch (Exception e) {
    System.err.println("ptrXMLFile:
    "+e.toString());
}
}

void Parse() throws Exception {
```

---

■ *The Parse function will handle top-level identification of the XML grammar and call out to special handling functions based upon the E-BNF grammar rules in the XML specification.*

---

```
sets = new CharacterSets();

try {
    boolean doneParsing=false;
```

---

■ *doneParsing identifies when the end of file has been reached.*

---

```
boolean whitespaceon=false;
```

---

■ *Whitespaeon tells the parser that sequential whitespace characters have been found within parsed character data.*

---

```
while (!doneParsing){
    char parseChar = (char)stmXML.read();
```

---

■ *Reads a character from the XML document.*

---

```
switch (parseChar) {

    case (char)-1:
```

---

■ *If it is the end of file character, the parser must stop.*

---

```
        doneParsing=true;
        break;
    case '<':
```

---

■ *If it is a <, then it is a tag and will be delegated to the `HandleTag` function. If a tag is not found, then this case catches the exception and terminates the parsing.*

---

```
        try {
            HandleTag();
        } catch (Exception e) {
            System.err.println(e.toString());
            doneParsing=true;
        }
        break;

        case ' ':
    case '\t':
    case 0:
```

---

■ *Identifies legal whitespace characters. The block of them will be turned into a single whitespace.*

---

```
        whitespaceon=true;
        break;

    case 0x0A:
```

---

■ *Line feeds are ignored.*

---

```
        break;

    case 0x0D:
```

- 
- *Carriage returns must be propagated into the Element's content.*
- 

```
        if (current != null)
            current.addPCDATA((char)0x0D);
        break;

    case '&':
```

- 
- *Ampersands signal the start of an entity reference.*
- 

```
        HandleEntityReference();
        break;

    default:
```

- 
- *Otherwise, adds the character to the current Element's content.*
- 

```
        if (whitespaceon) {
            whitespaceon=false;
            if (current != null)
                current.addPCDATA(' ');
        }
```

- 
- *Adds characters to the current Element's data text buffer.*
- 

```
        }
        if (current == null)
```

- 
- *This check identifies that there is a single root element, which is a required part of the XML specification.*
- 

```
        throw new
            Exception("XMLParse:" +
                "Expecting Tag");

        if (sets.setLegalChars.inSet
            (parseChar) && current != null)
```

---

■ *Checks to see that the character is a legal XML character.*

---

```

        current.addPCDATA(parseChar);
    else throw new
        Exception("XMLParse: Invalid" +
            "Character");
    }
} catch (Exception e) {
    System.err.println("XMLParse: "+e.toString());
}
}

void HandleTag() throws Exception {

```

---

■ *HandleTag looks at the first character after a < and decides what the markup represents.*

---

```

try {
    char parseChar = (char)stmXML.read();

    switch (parseChar) {
    case '!':

```

---

■ *An exclamation point indicates that it is a comment, not a tag.*

---

```

        HandleExclaim();
        break;

    case '?':

```

---

■ *A question mark indicates a processing instruction.*

---

```

        HandlePI();
        break;

    case '/':

```

---

■ *A forward slash indicates that it is an end-tag.*

---

```

    HandleEndTag();
    break;

default:

```

---

■ *Otherwise, it should be the Element type name, as long as it follows the rules for naming.*

---

```

    if (sets.setNameStartChar.inSet(parseChar))
        HandleStartTag(parseChar);
    else {
        throw new Exception("HandleTag:" +
            "Malformed Tag");
    }
    break;
}
System.out.println();
} catch (Exception e) {
    throw new Exception(e.toString());
}
}

void HandleStartTag(char first) throws Exception {

```

---

■ *HandleStartTag determines if the preceding characters indeed represent a start-tag. Otherwise, it is invalid.*

---

```

char parseChar;
StringBuffer text = new StringBuffer(64);

```

---

■ *This buffer will be used to store the Element type name.*

---

```

text.append(first);

```

---

■ *Because HandleTag pulled the first character out, we needed to add it to the text buffer here to ensure the name was completely recorded.*

---

```

while ((parseChar = (char)stmXML.read()) != '>') {

```

---

■ *We will not stop building the name unless the character is illegal or the > character is found.*

---

```

if (parseChar == EOF)
    throw new Exception("HandleStartTag: Malformed" +
        "Tag");

if (parseChar == '/') {

```

---

■ *A / indicates that it is an empty element. We simply need to retain that state, but not act on it here.*

---

```

    emptyElement=true;
    continue;
}

if (sets.setNameChars.inSet(parseChar))

```

---

■ *Here we check to see that the current character falls within the legal limits of an element type name.*

---

```

    text.append(parseChar);
else if (parseChar == ' ') {

```

---

■ *If a space is found after the name, then we might have attributes. These attributes will be collected by the `HandleAttributes` function.*

---

```

    HandleAttributes();
    break;
}
else
    throw new Exception("HandleStartTag: Invalid" +
        "Tag Name" +
        "Character '"+parseChar+"'");
}
Element em = new Element();

```

---

■ *When we find a legal start-tag, we need to create a new `Element` object to represent it in memory.*

---

```
em.setName (new String(text));
em.setParent(current);

if (current != null)
    current.addElement(em);
```

---

■ *We need to add the `Element` we just created into the hierarchical memory representation of the XML document as long as there is already a root `Element`.*

---

```
if (!emptyElement)
    current = em;
```

---

■ *If the `Element` is not empty, we need to set it as the current `Element` so all future `PCDATA` is directed to its vector until we find the end-tag for this `Element`.*

---

```
    else {
        emptyElement=false;
    }
}

...
}
```

We'll leave it as an exercise for the reader to expand this parser further, but the important concepts are contained within the above code. These are:

1. XML parsers must make sure that characters fall within the acceptable character sets for a particular token designation.
2. XML parsers must first identify a particular token by examining the sequence of characters that represents it.

3. XML parsers must check that a sequence of tokens and nonterminal symbols matches a valid nonterminal symbol within the XML grammar.
4. XML parsers must maintain context within a document to make sure there are matching start- and end-tags for each element and that they do not overlap.

In the next sections, we will further explore how a parser can be extended to offer application-level functionality.

### 3.3 | The Simple API for XML

The Simple API for XML (SAX) is an interface for event-based XML parsing, developed collaboratively by the members of the XML-DEV mailing list, [xml-dev@ic.ac.uk](mailto:xml-dev@ic.ac.uk). Widespread adoption of SAX by many parser developers has made it a de facto industry standard.

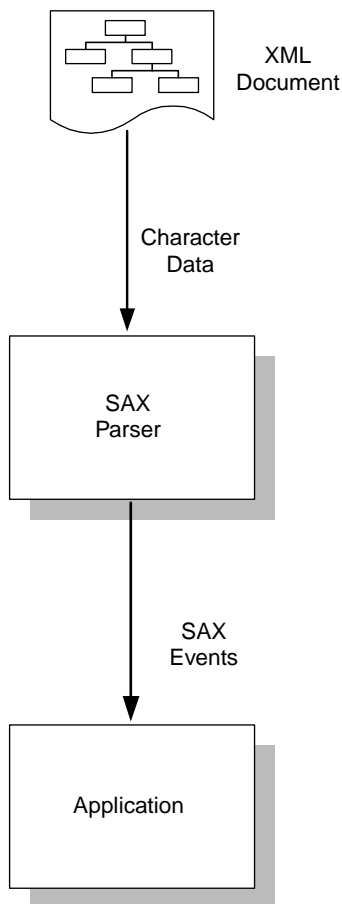
SAX arose out of the need to reuse XML parsers as software components, but to extend them for application-specific use. For example, if you wanted to develop an application that processed `INVOICE` documents represented in XML, then you would need to use an XML parser to read and extract the data. All applications that process XML must encounter this step. However, not all applications process the `INVOICE` document identically. SAX allows an application to receive the extracted data directly from the parser, which it can then use accordingly.

SAX is an event-based parser, which means that it notifies the applications every time it identifies a “parse event” within the XML document. This includes element starts and ends, data content, attributes, processing instructions, whitespace, etc. The next section explains how event parsing works in detail.

### 3.3.1 *Event Parsing*

One output of an XML parser could be a document model. This is an in-memory representation of an XML document as a tree of connected nodes. It is constructed from the parse event information. We will explore this type of output further when we look at the W3C DOM.

A parser does not always construct a document model; instead, it can limit itself to reporting a series of parse events while it processes a document (see Figure 3-2). This places a greater processing bur-



**Figure 3-2** Event Parsing.

den on the application, which must now create the document model from parse events. There is also no assurance that a document is valid or even well-formed until the parser finishes processing the document. But, there are a number of advantages to this approach:

- 1. Small memory footprint.** A document model is often not an efficient means of representing a document. And, when constructed by the parser, the entire model must fit in memory, limiting the size of a document that can be processed. By using an event interface, the application may be able to process very large documents.
- 2. Single mapping to internal structures.** The generic document model constructed by a parser also may not be usable directly by an application, with the application constructing a second model at additional costs in time and memory. This application-specific model can often be constructed directly from parser events, eliminating the cost of constructing two models.
- 3. Non-standard validations.** An event interface also gives an application the means of participating in the validation of a document—application-specific validation that would be difficult to define in a general schema, though this could be done after the document had been parsed. The real advantage here is in being able to clearly identify the place in the text where an error occurred, using information that is not available once the parse is complete.

The use of a standard interface for parse events makes it easy to change an application to use a different parser. One such interface that has become widely supported is SAX.

### 3.3.2 *SAX Interfaces*

SAX defines interfaces for each of the following four different types of parse events, as well as a top-level interface for the parser itself. Table 3-1 explains these interfaces.

The `Parser` and `DocumentHandler` interfaces are central to understanding SAX-based applications. They are also a good place to begin, as they are used to connect the components of an application with the parser itself.

**Table 3-1** SAX interfaces.

<i>Parser Interface</i>	<i>Description</i>
DocumentHandler Interface	Most SAX applications implement this interface. Once the application registers its document handler, the parser uses this interface to report basic document-related events like the start and end of elements and character data.
EntityResolver Interface	This interface is implemented by an application to override a parser's default method for retrieving entities. It is especially useful for applications that build XML documents from databases and for applications that use URI types other than URLs.
DTDHandler Interface	This interface is implemented by an application to receive information about notations and unparsed entities.
ErrorHandler Interface	By default, a parse error results in a <code>SAXParseException</code> being thrown on the parse method. This exception carries details about which document contained the parse error and where in that document the error occurred. When this default behavior is insufficient, the application can register its own error handler with the parser. Note that a custom error handler is needed if warnings and recoverable errors are to be reported.

There are seven methods included in the `Parser` interface. They are outlined in Table 3–2.

**Table 3–2** Parser interface.

<i>Function Name</i>	<i>Description</i>
<code>void setLocale( java.util.Locale locale)</code>	This method allows an application to request a locale for errors and warnings. Parsers are not required to support this capability and may simply throw a <code>SAXException</code> when the method is called.
<code>void setEntityResolver ( EntityResolver resolver)</code>	An application can use this method to register its own implementation of <code>EntityResolver</code> , which is then used by the parser to retrieve any documents or other entities needed by the parser.
<code>void setDTDHandler ( DTDHandler handler)</code>	An application can use this method to register its own implementation of <code>DTDHandler</code> . The registered <code>DTDHandler</code> object then receives information about notations and unparsed entities.
<code>void setDocumentHandler ( DocumentHandler handler)</code>	An application uses this method to register its own implementation of <code>DocumentHandler</code> , to which the parser then passes all document-related events.
<code>void setErrorHandler ( ErrorHandler handler)</code>	By default, errors and warnings are ignored by a parser, with only fatal errors throwing a <code>SAXException</code> . By registering its own <code>ErrorHandler</code> , an application has the option of reporting errors and warnings, as well as being able to abort a parse when an error or warning occurs.
<code>void parse( String systemID)</code>	This method is used by the application to start the parse of a document. The <code>systemID</code> is typically the fully resolved URL of the document.
<code>void parse( InputSource source)</code>	This is an alternative method for starting the parse of a document. The <code>source</code> object may specify a system ID, an <code>InputStream</code> or <code>Reader</code> object. Character encoding and a public ID can be specified in the <code>source</code> object as well.

All document-related events reported by the parser are passed to the application document handler using the `DocumentHandler` interface. There are eight methods included in this interface for passing these events. They are discussed in Table 3-3.

**Table 3-3** DocumentHandler interface.

<i>Function Name</i>	<i>Description</i>
<code>void setDocumentLocator ( Locator locator)</code>	The parser uses this method to share its <code>Locator</code> object with the application. The <code>Locator</code> object identifies the document being parsed and the current position in a document being processed. Other methods in the <code>DocumentHandler</code> interface can all throw <code>SAXParseException</code> s when the document does not conform to the application's requirements. The <code>Locator</code> object can be used to construct this exception, which will then identify where in the document the error occurred.
<code>void startDocument()</code>	This method signals the start of a new document and can be used to reinitialize the document handler, an important consideration when more than one document is being processed.
<code>void endDocument()</code>	This method signals the end of a document and should be used to clear references to any objects no longer in use.
<code>void startElement( String name, Attribute List atts)</code>	This method signals the discovery of a start-tag, and consequently, the start of an XML element. Elements can be nested, with each start element event being paired with an end element event.
<code>void endElement( (String name</code>	This method signals the discovery of an XML end-tag.
<code>void characters( char ch[], int start, int length)</code>	The data text held by an element is passed to the document handler via one or more calls to this method.

*(continued)*

**Table 3-3** DocumentHandler interface. (*continued*)

<i>Function Name</i>	<i>Description</i>
<code>void ignorable   whitespace(     char ch[],     int start,     int length)</code>	Validating parsers will use this method instead of the characters method for whitespace. This event can be ignored.
<code>void processing   Instruction(     String target,     String data)</code>	This method is used to pass a processing instruction to the document handler.

Using the SAX interface is described in the following steps. Let's say we have an application that implements the `DocumentHandler` interface. To parse a document, the application must do the following:

1. Create a `Parser` object. There are various ways to do this, but the most direct approach is simply to create the object using the Java operator `new`. The catch to this direct approach is that we need to know the class name of the parser.

Let's say that we are using the IBM parser. The class in IBM that implements the SAX parser interface is `SAXDriver`, so we can create a parser object with the following line:

```
org.xml.sax.Parser parser = new  
com.ibm.xml.parser.SAXDriver();
```

2. Register the application's `DocumentHandler` object by calling the `setDocumentHandler` method on the parser object.

For example, say we have an object, referenced by the variable `myDocumentHandler`, which is an instance of

a class that implements the SAX `DocumentHandler` interface. The following line will register the object with the parser so that the object will receive all the document events during a parse:

```
parser.setDocumentHandler(myDocumentHandler);
```

We'll examine the `DocumentHandler` interface in more detail shortly.

3. Parse the document by calling the `parse` method on the `parser` object.

For example, say we have a file, "a.xml", in the directory where the program is running (the current working directory). The URL for this file is "file:a.xml". We can now parse this document with the following line of code:

```
parser.parse("file:a.xml");
```

Once a parse is begun, a series of callbacks are made to the application's `DocumentHandler` object. Example 3.6 shows a trace of the events for a very simple document:

**Example 3.6: Sample XML for SAX test.**

---

```
<helloWorld><greeting>Hi!</greeting></helloWorld>
```

---

The expected results of parsing Example 3.6 are indicated below:

1. Event `setDocumentLocator`—Passes the parser's locator object to the application.
2. Event `startDocument`—The document is begun.
3. Event `startElement`—The start-tag for `helloWorld` is processed.
4. Event `startElement`—The start-tag for `greeting` is processed.

5. Event characters—The characters "Hi!" are processed.
6. Event endElement—The end-tag for greeting is processed.
7. Event endElement—The end-tag for helloWorld is processed.
8. Event endDocument—The document is finished.

Taking this one step further, we will now explore the program that reports these various SAX events that occur when parsing a document. Next, we'll examine the output that this program produces when parsing the above `helloWorld` document. First, here's the Java source code:

```
import com.ibm.xml.parser.*;
import org.xml.sax.*;

public class SAXTest implements DocumentHandler {

    public static void main(String args[])
    {
        try
        {
            org.xml.sax.Parser parser = new
com.ibm.xml.parsers.SAXParser();
```

---

■ *Create a new instance of a parser.*

---

```
DocumentHandler myDocumentHandler = new
    SAXTest("IBM");
```

---

■ *Creates an instance of the `DocumentHandler` object that will respond to parser events.*

---

```
parser.setDocumentHandler(myDocumentHandler);
```

---

■ *Register the DocumentHandler object with the parser to facilitate callbacks.*

---

```
parser.parse("file:a.xml");
```

---

■ *Tells the parser to start parsing the file.*

---

```
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

```
protected String prefix;
```

---

■ *prefix will hold the name of the parser we're using for illustrative purposes.*

---

```
public SAXTest(String prefix){  
    this.prefix=prefix;  
}  
  
public void setDocumentLocator(Locator locator) {  
    System.out.println(prefix+": setDocumentLocator");  
}  
  
public void startDocument() throws SAXException {  
    System.out.println(prefix+": startDocument");  
}  
  
public void endDocument() throws SAXException {  
    System.out.println(prefix+": endDocument");  
}  
  
public void startElement(String name, AttributeList atts)  
    throws SAXException {  
    String attsOut=new String();  
    for (int attIndex=0; attIndex<atts.getLength();  
        attIndex++) {  
        attsOut=attsOut.concat("[ " + atts.getName(attIndex) +  
            "=" + atts.getValue(attIndex) + " ]");  
    }  
}
```

---

■ *Here we build a list of all the attributes on the start element using the associated `AttributeList`.*

---

```
    }
    System.out.println(prefix+": startElement
        "+name+attsOut);
}

public void endElement(String name) throws SAXException {
    System.out.println(prefix+": endElement "+name);
}

public void characters(char ch[], int start, int length)
    throws SAXException {
    String cs = new String(ch, start, length);
    System.out.println(prefix+": characters["+cs+"]");
}

public void ignorableWhitespace(char ch[], int start, int
    length) throws SAXException {
    String iws = new String(ch, start, length);
    System.out.println(prefix+": ignorableWhitespace
        ["+iws+"]");
}

public void processingInstruction(String name, String
    remainder) throws SAXException {
    System.out.println(prefix+":
        processingInstruction["+name+"["+remainder+"]");
}
```

---

■ *The above methods implement the `DocumentHandler` interface. All methods must be implemented.*

---

```
}
```

Here are the results of running the above code using the XML4J Java parser version 2.0.6. Notice that it matches our previously expected results.

```
IBM: setDocumentLocator
IBM: startDocument
IBM: startElement helloWorld
IBM: startElement greeting
IBM: characters[Hi!]
IBM: endElement greeting
IBM: endElement helloWorld
IBM: endDocument
```

Using the SAX interfaces, an application can now participate in the validation of a document. It can also create an output stream based on those events or create some application-specific document model, which it can use once the parse of the document is complete. Moreover, by using the SAX interfaces, an application can work closely with a parser without being tied to any particular parser. Should the requirements of an application change, it is still quite easy to replace the parser with a more appropriate selection.

As you can see with this example, when we are done parsing the file, our application terminates because application processing is concurrent with parsing. This programming model is in contrast to the W3C Document Object Model that we are going to look at next. In that programming model, we parse first and process second.

## 3.4 | W3C Document Object Model

Like SAX, the Document Object Model (DOM) is another way that applications can access the data found inside an XML document. However, instead of the application getting the data directly with event handlers, DOM creates an in-memory tree of objects that the applications can traverse to extract the information at will.

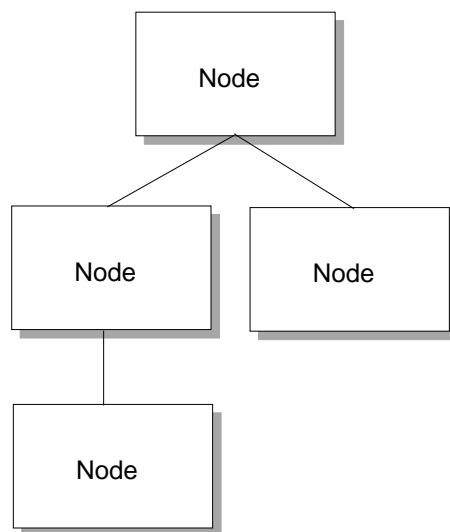
As you may remember from our parsing example at the beginning of this chapter, as part of parsing an XML document, we can create a hierarchical representation in memory of that document consisting of Java objects. The W3C Document Object Model, or DOM, uses this

hierarchical representation to allow developers to further manipulate an XML document.

The DOM has three core functions to XML: 1) it represents the document in memory; 2) it acts as the factory for creating new XML elements; and 3) it offers the necessary functions for manipulating the in-memory representation, also known as the parse tree. With these functions, developers can identify the elements of an XML document, but in contrast to SAX, which we discussed earlier, they will not be able to do this until the parser has completely finished parsing the document.

The benefit gained from using this programming model is that the developer will be able to perform multiple transformations on the XML document in memory, as well as iterate over the elements within the tree and pull information out.

The W3C DOM is designed for use by many programming languages and facilities. Some of these do not have the rich data type support that Java and C++ have. Instead, these environments can simply view the object model as one big array that can contain other arrays. For these impaired facilities, the DOM has a basic “flat” definition (see Figure 3-3).



**Figure 3-3** DOM flat model.

Notice how these implementations of the DOM look at the XML world as if it were a nail and the DOM a hammer. That is, to the flat view, the DOM is just a set of node structures that contain other node structures. This makes it far more difficult to compare and identify these elements. Additionally, these node structures must be associated with all the functions that might ever be performed on an XML element as illustrated by the `Node` interface depicted in Table 3-4 below.

---

**Table 3-4** Node interface.

---

```
Public interface Document extends Node {  
    public DocumentType      getDoctype();  
    public DOMImplementation getImplementation();  
    public Element           getDocumentElement();  
    public Element           createElement(String tagName)  
        throws DOMException;  
    public DocumentFragment createDocumentFragment();  
    public Text              createTextNode(String data);  
    public Comment           createComment(String data);  
    public CDATASection      createCDATASection(String data)  
        throws DOMException;  
    public ProcessingInstruction createProcessingInstruction  
        (String target, String data) throws DOMException;  
    public Attr              createAttribute(String name)  
        throws DOMException;  
    public EntityReference   createEntityReference(String name)  
        throws DOMException;  
    public NodeList          getElementsByTagName(String tagname);  
}
```

---

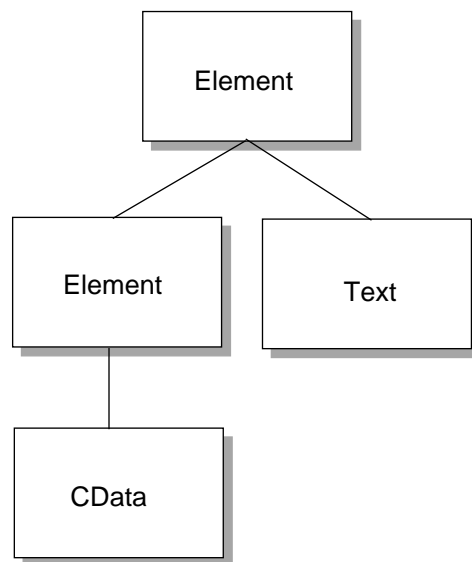
Most developers building EAI solutions, however, use the typed model, also known as the object-oriented model. The typed model provides a separate class definition for each object represented in the tree.

The best way to approach learning the DOM is to start with a visualization of a DOM object tree (see Figure 3-4).

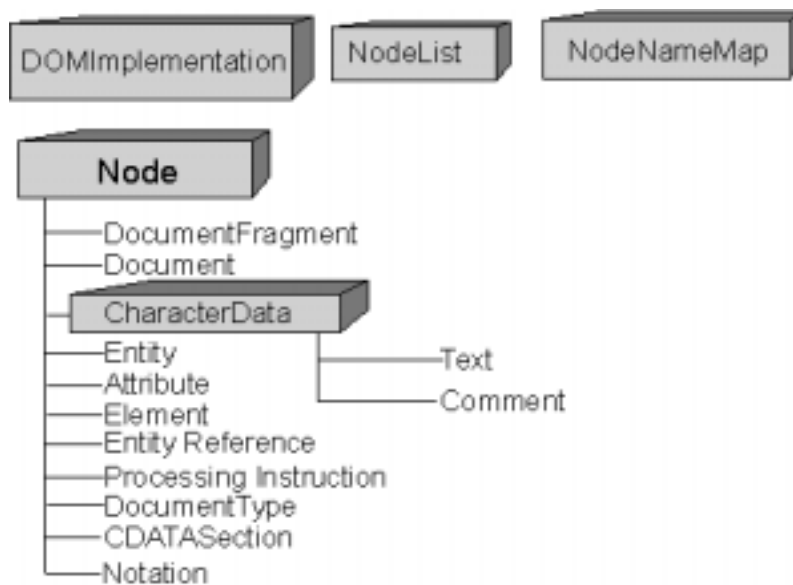
Let's explore some of the more important objects in the typed view as depicted in Figure 3-5. In particular, we will be looking at the `Node`, `Document`, `Element` and `Attr` object classes.

### 3.4.1 *The Node Object*

The `Node` object is the base class that is used to define all other DOM objects. All possible manipulations on the tree can be performed on the `Node` object, and subsequently on all inherited object types.



**Figure 3-4** DOM typed object model.

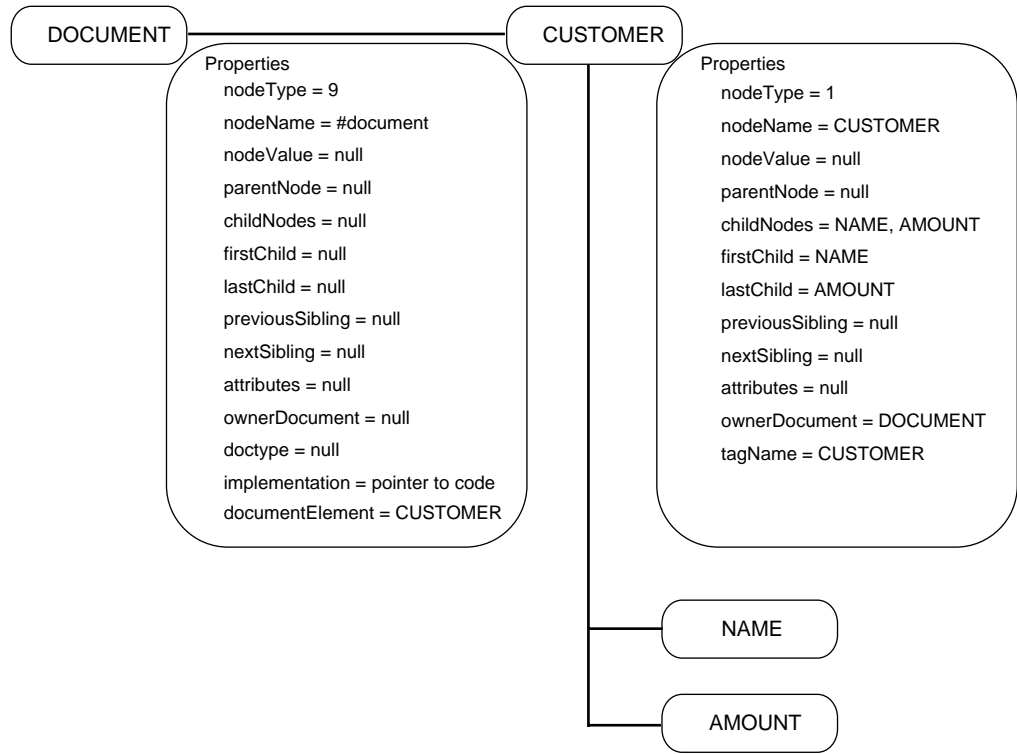


**Figure 3-5** DOM interfaces.

Table 3-4 depicts the Java Language Binding for the `Node` object. The binding represents the mapping from the abstract definition in the DOM specification to a specific programming language; in this case, Java. The W3C also offers bindings for other programming languages and object models, such as COM and CORBA.

Figure 3-6, which is an illustration of an XML document represented as a DOM, is helpful in understanding the relationship between the DOM and an XML document.

Table 3-5 and Table 3-6 provide a more detailed look at the `Node` interface, with Table 3-6 outlining the return values that could be obtained from the `getNodeName` method. It is important to understand the implications of viewing each object in the DOM as a node, and Table 3-6 outlines this best. The `getNodeName` method is overloaded to provide different return values based on the XML object the `Node` represents.



**Figure 3-6** DOM memory representation.

**Table 3-5** The DOM node interface explained.

<i>Function Name</i>	<i>Description</i>
<pre> public static final short ELEMENT_NODE = 1; public static final short ATTRIBUTE_NODE     = 2; public static final short TEXT_NODE = 3; public static final short CDATA_SECTION_     NODE = 4; public static final short ENTITY_     REFERENCE_NODE = 5; public static final short ENTITY_NODE = 6;                 </pre>	<p>These definitions are used by the flat view to determine what XML object the Node represents. This value can be found and compared against the <code>getNodeTypes</code> method.</p>

*(continued)*

**Table 3-5** The DOM node interface explained. *(continued)*

<i>Function Name</i>	<i>Description</i>
<pre>public static final short PROCESSING_   INSTRUCTION_NODE = 7; public static final short COMMENT_NODE = 8; public static final short DOCUMENT_NODE   = 9; public static final short DOCUMENT_TYPE_   NODE = 10; public static final short DOCUMENT_   FRAGMENT_NODE = 11; public static final short NOTATION_NODE   = 12;</pre>	
<pre>public String getNodeValue() throws   DOMException;</pre>	Per Table 3-6
<pre>public NamedNodeMap getAttributes();</pre>	Per Table 3-6
<pre>public void setNodeValue(String node   Value) throws DOMException;</pre>	Attempts to set the value of the Node based upon acceptable values in Table 3-6. Illegal values force the DOMException to be thrown.
<pre>public short getNodeType();</pre>	Returns one of the static node types defined above.
<pre>public Node getParentNode();</pre>	If the current node is not the root element, this method will return the element's parent node.
<pre>public NodeList getChildNodes();</pre>	Returns a NodeList object that contains all the children of the current element.
<pre>public Node getFirstChild(); public Node getLastChild(); public Node getPreviousSibling(); public Node getNextSibling();</pre>	These methods allow the developer to traverse the tree quickly by moving between the child elements and to the element's siblings.

*(continued)*

**Table 3–5** The DOM node interface explained. (*continued*)

<i>Function Name</i>	<i>Description</i>
<code>public Document getOwnerDocument();</code>	Returns the Document object that is associated with the current Node. This document can be used to create additional nodes.
<code>public Node insertBefore(Node newChild, Node refChild) throws DOMException;</code> <code>public Node replaceChild(Node newChild, Node oldChild) throws DOMException;</code> <code>public Node removeChild(Node oldChild) throws DOMException;</code> <code>public Node appendChild(Node newChild) throws DOMException;</code>	These methods provide the developer with a way to transform the tree by adding, changing, and removing nodes. Removal of a node will automatically remove all of its children as well.
<code>public boolean hasChildNodes();</code>	This method allows developers to see if a particular node has children.
<code>public Node cloneNode(boolean deep);</code>	This method creates a separate and distinct copy of the current Node, including all of its children if the parameter <code>deep</code> is true. The DOM does not allow developers to move Node objects between Document objects. The only way to accomplish this is to clone the Node object and insert it into the another Document object.

### ***Sidebar: Is DOM Too Heavyweight For EAI?***

*JP Morgenthal*

*I want this book to be more than a slavish recitation of what you might be able to learn from the W3C specifications. I also want you to think about the applicability of those specifications to your needs. I'll keep these critical insights inside these well-defined sections to differentiate the opinions from the factual content. It is my hope that both will be useful to you.*

**Table 3-6** Return Values for `getNodeName`.

<i>NodeType</i>	<i>nodeName</i>	<i>nodeValue</i>	<i>attributes</i>
Element	TagName	Null	NamedNodeMap
Attr	Name of attribute	Value of attribute	Null
Text	#text	Content of the text node	Null
CDATASection	#cdata-section	Content of the CDATA Section	Null
EntityReference	Name of entity referenced	Null	Null
Entity	Entity name	Null	Null
ProcessingInstruction	Target	Entire content excluding the target	Null
Comment	#comment	Content of the comment	Null
Document	#document	Null	Null
DocumentType	Document type name	Null	Null
DocumentFragment	#document-fragment	Null	Null
Notation	Notation name	Null	Null

*I believe there are a number of problems with the DOM's design that make it unnecessarily heavy for all applications, and some that make it overkill for specific applications of XML, such as messaging and other rudimentary data exchange operations. The following are the reasons behind these conclusions:*

- 1. While both the flat and typed views of an XML document are necessary for the reasons described earlier, the decision to make the `Node` the base class for other DOM objects in the typed view results in a heavyweight object. Because of this decision, every `Comment` object could be treated the same as an `Element` object. That is,*

(continued)

### ***Sidebar: Is DOM Too Heavyweight For EAI?*** (continued)

you could call the function `hasChildNodes` on either one. The problem with this in the typed model is that you do not expect this behavior to be on the `Comment` object, but only on the `Element` object. The result of this is undefined behaviors from legal method (functions in an object) calls. Again, this is understandable in the flat view, where it is up to the developer to first check for a `Node`'s type before deciding what functions can be executed on a particular `Node`, but should not be possible in the typed view.

2. The `Element` object has a function called `normalize` that combines adjacent `Text` objects in the hierarchy. That is, if there are two child objects that are both `Text` objects and follow each other in the tree, then they can be made into one single `Text` object. Moreover, the `Text` object has a `splitText` function that takes a single `Text` object and makes two adjacent `Text` objects out of it by splitting the `Text` object's value at a certain offset within the text string. Clearly, these functions are needed in document-centric processing, but they only further weigh down the DOM in areas where lightweight data processing mechanisms are needed.
3. XML is tedious to build with the DOM. It takes three distinct operations to add a new DOM object to the tree. First, you have to ask the `Document` object to create a new instance of the object type. Second, you have to prepare that object by setting its value, attributes, and children. Finally, you must add the object to the tree using an insert operation. These steps are far too arduous when attempting to build a dynamic XML document in a high-volume environment. It's easier to just hard code the XML creation directly into the program.
4. The DOM provides a simple lookup function to retrieve all the elements that share a particular element type name. However, it retrieves them as a list of objects that have lost their context. For example, requesting the `CUSTOMER` elements from an invoice will give both the `CUSTOMER` element within the `BILLING` element and the `CUSTOMER` element within the `SHIPPING` element. As the application iterates over the returned list of nodes, it is its responsibility to get the parent of each `CUSTOMER` element and look to see what the parent's context is. Clearly, this is functionality that could be added to the DOM and is being examined by the XML-Query Working Group, but as provided, the lookup facilities within the DOM are not overly helpful.

For some applications, these points may seem trivial, but when building XML into a high-volume, server-based data processing environment, these points may have significant implications. I would welcome a W3C activity focused on providing a standard XML programming model optimized for the needs of high performance applications.

### 3.4.2 *The Document Object*

The `Document` object has two roles within the DOM. First, the `Document` object is the object factory for creating other DOM objects. Second, the `Document` object is the container for the document's hierarchical representation and therefore holds the root element. Table 3–7 illustrates and describes the functionality of the `Document` object class.

**Table 3–7** The Document interface.

<i>Function Name</i>	<i>Description</i>
<code>Public DocumentType getDoctype();</code>	Returns the <code>DocumentType</code> object, which contains Document Type Definition (DTD) information, such as entity and notation declarations.
<code>Public DOMImplementation getImplementation();</code>	This method allows developers to check if the current version of the parser will support the features they need, such as XML 1.0 and HTML 4.0.
<code>Public Element getDocumentElement();</code>	Returns the root element of the document. For XML, this is the root element and for HTML, it is “HTML”.
<code>public Element createElement(String tagName) throws DOMException;</code> <code>public DocumentFragment createDocumentFragment();</code> <code>public Text createTextNode(String data);</code> <code>public Comment createComment(String data);</code> <code>public CDATASection createCDATASection(String data) throws DOMException;</code>	These methods create new DOM objects that are not associated with the current <code>Document</code> object.

(continued)

**Table 3–7** The Document interface. (*continued*)

<i>Function Name</i>	<i>Description</i>
<pre>public ProcessingInstruction   createProcessingInstruction(String     target, String data) throws     DOMException; public Attr createAttribute(String   name) throws DOMException; public EntityReference   createEntityReference(String name)   throws DOMException;</pre>	
<pre>public NodeList getElementsByTagName   (String tagname);</pre>	<p>Executes a preorder traversal of the parse tree and returns all elements that match the <code>tagname</code> parameter. A preorder traversal is defined as a visiting of the root node and then the child nodes. A supplied value of <code>'*'</code> will return all nodes.</p>

### 3.4.3 *The Element Object*

The `Element` object contains a deep representation of an XML element. That is, it contains the nodes corresponding to all the text between a start-tag and an end-tag. If that text should contain additional elements, or if the element should contain attributes, those would be captured as well. Table 3–8 illustrates and describes the functionality of the `Element` object class.

### 3.4.4 *The Attr Object*

The `Attr` object represents a single attribute on an element. `Attr` objects are only operated on within the context of an element. Table 3–9 illustrates and describes the functionality of the `Attr` object class.

**Table 3-8** The Element interface.

<i>Function Name</i>	<i>Description</i>
<code>Public String getTagName();</code>	Returns the element type name of the current Element.
<code>Public String getAttribute(String name);</code>	Returns the value of the attribute named by the parameter, or null if it does not exist.
<code>Public void setAttribute(String name, String value) throws DOMException;</code>	Adds the attribute named by the <code>name</code> parameter and sets it to the value defined by the <code>value</code> parameter. If the attribute already exists, then it simply changes the value.
<code>Public void removeAttribute(String name) throws DOMException;</code>	Deletes the attribute from the current element. In contrast to <code>removeChild</code> , this is a permanent deletion.
<code>Public Attr getAttributeNode(String name);</code>	Returns the attribute object that has the name defined by the <code>name</code> parameter.
<code>Public Attr setAttributeNode(Attr newAttr) throws DOMException;</code>	Adds the attribute defined by the attribute object parameter to the current element. If the attribute already exists, the value is copied from the parameter and set on the existing attribute object of the same name.
<code>Public Attr removeAttributeNode(Attr oldAttr) throws DOMException;</code>	Permanently deletes the attribute that is defined by the attribute object parameter <code>oldAttr</code> .
<code>Public NodeList getElementsByTagName(String name);</code>	Executes a pre-order traversal of the current element's descendants and returns all elements that match the <code>tagname</code> parameter. A pre-order traversal is defined as a visiting of the root node and then the child nodes. A supplied value of <code>'*'</code> will return all nodes.
<code>Public void normalize();</code>	Combines adjacent text nodes into a single text node.

**Table 3–9** The Attr interface.

<i>Function Name</i>	<i>Description</i>
<code>Public String getName();</code>	Retrieves the name of the current <code>Attr</code> object.
<code>Public Boolean getSpecified();</code>	If this attribute was given a specific value in the original XML document, this field will be set to true.
<code>public String getValue();</code>	Retrieves the value of the current attribute as a character string.
<code>public void setValue(String value);</code>	Sets the value of the current attribute object. Values are stored as unpaved <code>Text</code> nodes that are children of the attributes.

### 3.4.5 *An Example Of Using DOM from Java*

The following example uses the IBM XML4J parser to read a document from an input stream and create a `Document` object. It takes as input an XML document that represents a list of pages and displays it as a Web page.

Here is a sample of the XML document that it would process:

```
<?xml version="1.0"?>
<PageDB>
<Page ID="pagenotfound"
  title="Sorry, Page Could Not Be Located"
  bgcolor="#ffffff"
  bgimage="BLANK"
  URL="nepage.xml"/>

<Page ID="root"
  URL="pages/root.xml"
  title="NC.Focus Root Page"
  noshow="pages/signup.xml">
  <Description>Contains the default descriptions for all
  other pages</Description>
</Page>
```

```
<Page ID="home"
  bgimage="ncftile.gif"
  public="true"
  title="NC.Focus Home Page">
  <Description>Home Page</Description>
</Page>

<Page ID="sitemaptop"
  public="true"
  URL="sitemaptop.xml"
  title="sitemapheader">
  <Description>Site Map Header</Description>
</Page>

<Page ID="sitemapbottom"
  public="true"
  URL="sitemapbottom.xml"
  title="sitemapbottom">
  <Description>Site Map Footer</Description>
</Page>
</PageDB>
```

The following code will generate a table of links automatically from the XML document below. Included in this code is the ability to filter what is displayed by using the attributes.

```
import com.ibm.xml.parser.*;
import org.w3c.dom.*;

public class sitemap {

  public static void main (String args[]) {
    String stype = null;

    try {
      if (args.length < 1) {
        System.out.println("Usage: sitemap URL [filter]");
        return;
      }
      System.out.println("<HTML><HEAD><TITLE>NC.Focus Site" +
        "Map</TITLE></HEAD>");

      NonValidatingDOMParser parser = new
        NonValidatingDOMParser();
```

- 
- *We create an instance of the IBM XML4J Java parser.*
- 

```
parser.parse(args[0]);
```

- 
- *We tell the parser to parse the XML file indicated by the URL passed as the first argument on the command line.*
- 

```
Document doc = parser.getDocument();
```

- 
- *If the document was successfully parsed, we ask the parser to return to us the DOM Document object.*
- 

```
NodeList pages = doc.getElementsByTagName("Page");
```

- 
- *pages obtains a list of pages by asking the Document object for all elements of type "Page".*
- 

```
System.out.println("<TABLE WIDTH=\"602\" +
  \"BORDER=\"1\"><TR><TD>");
System.out.println("<TABLE WIDTH=\"600\">\r\n");
System.out.println("<TR><TD WIDTH=\"200\"><B>\" +
  \"<CENTER><FONT\" +
  \"FACE=\"Arial\">Page Title</FONT></CENTER></B>\" +
  \"</TD>");
System.out.println("<TD WIDTH=\"300\"><B><CENTER><FONT\" +
  \"FACE=\"Arial\">Description</FONT></CENTER></B>\" +
  \"</TD>");
int ctr = 0;
for (int i=0; i<pages.getLength(); i++) {
```

- 
- *We then iterate over the list of pages, creating one row per Page node.*
- 

```
String colorStr = null;
```

- 
- *colorStr defines the color of the row background.*
-

```
if ((ctr%2) == 0)
    colorStr = " BGCOLOR=\"#00FFFF\" ";
else
    colorStr = " BGCOLOR=\"#FFFF80\" ";
Node x = pages.item(i);
```

---

■ *Individual nodes within the `NodeList` are accessed using the `item` method. Here we set the variable `node` to be the current `Node` object we are processing.*

---

```
NamedNodeMap nnp = node.getAttributes();
```

---

■ *We ask the element we are currently processing for its list of attributes.*

---

```
if (args.length < 2)
    stype = "public";
else
    stype = (String) args[1];
```

---

■ *`stype` is the attribute for which we will filter. If it exists on a page, then the page will be included. If no `stype` was supplied, we will default to the value "public".*

---

```
Node isPublic = nnp.getNamedItem(stype);
```

---

■ *Once we have `stype` set, we ask the `NamedNodeMap`, which contains our attributes, if the attribute exists. If it does, it is returned as a `Node` object. We could decide to manipulate this object as a `Node` as we have done here, or cast the object to an `Attr` object.*

---

```
if (isPublic == null)
    continue;
```

---

■ *If the attribute we are filtering for does not exist, then we move to the next node.*

---

```
ctr++;
Node title = nnp.getNamedItem("title");
```

- 
- *Retrieves the title of the page from the list of attributes.*
- 

```

if (title != null) {
    System.out.println("<TR><TD"+colorStr+">");
    Node url = nnp.getNamedItem("ID");
    System.out.print("<A HREF=\"");
    System.out.print(System.getProperty("pageurl")+"?\" +
        cmd=page&page="+url.getNodeValue());
    System.out.print("\>");
    System.out.println(title.getNodeValue()+"</A>\" +
        </TD>");

```

- 
- *Prints the title of the page into the table. The string we wish to print can be retrieved from a Node object using the `getNodeValue` method.*
- 

```

NodeList children = node.getChildNodes();

```

- 
- *If there's a description, it is stored as a child element of Page, as can be seen in the page called "root" in the XML document above.*
- 

```

System.out.println("<TD"+colorStr+">");
if (children != null) {

```

- 
- *The string we wish to display from the description is actually stored as a `Text` object that is a child to the `Description` element. This statement extracts the `Text` node.*
- 

```

Node desc = children.item(1).getFirstChild();
System.out.println("<CENTER>"+
    desc.getNodeValue() + "</CENTER>");

```

- 
- *In this statement, we simply print the value of the `Text` object, which is the description string.*
- 

```

    }
}
System.out.println("</TABLE></TD></TR></TABLE>");
System.out.println("</BODY></HTML>");

```

```
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

In this example, we use the capabilities of the DOM to create an in-memory representation of a page database. We then use the DOM to extract all the elements that have the name “Page” and we access the attributes on each of those elements to see if they pertain to our filter. It’s also an example of how we might use the DOM to perform a transformation of the data stored in our XML document for another purpose—in this case, to create a Web page. In future chapters, we will use the DOM to provide a model that we can serialize for data exchange.

### 3.5 | Summary

This chapter introduced you to the key methods of dealing with XML documents programmatically. All processing of XML documents starts with a parser, which ensures that the document is well-formed. Some XML parsers are also validating, which means that they can confirm that the document follows the rules specified in the Document Type Definition (DTD).

One method of acting upon the content of an XML document is to integrate document handlers with the parser. This method produces lightweight XML document processing solutions, such as when using the Simple API for XML (SAX). SAX is an event-driven processing model that will call methods inside the developer’s application to process specific parts of an XML document that have been found.

Another method of processing XML documents is to manipulate them as a tree of objects called the Document Object Model (DOM).

The DOM represents a specific set of object classes that represents the XML document in memory as a set of related nodes. The DOM requires more memory and time for processing, but allows the developer to operate directly on the document as a “live” object.

### 3.6 | Looking Ahead

This part outlined the fundamentals of EAI, XML and Java. We also discussed some of the ways XML and Java are providing answers to complex EAI issues today. But, most importantly, we covered the basic methods for processing XML documents—SAX and DOM. The next part delves into using XML in tandem with Enterprise Java APIs for the purpose of building EAI solutions.

