

Using XML in Wireless Applications

CHAPTER

10

IN THIS CHAPTER

- Overview 336
- XML and Parsing XML Documents 337
- XML Parsers for Wireless Applications 340
- SAX 1.0 Java API For J2ME MIDP 342
- TinyXML Parser for J2ME MIDP 344
- NanoXML Parser for J2ME MIDP 358
- Ælfred Parser for J2ME MIDP 368

Overview

In recent years, the Extensible Markup Language (XML) has been adopted by more and more businesses as an industry standard for data exchange and data sharing. XML provides a system-independent standard format for specifying the information exchanged over networks and between applications.

The concept of XML is fairly simple, but the effectiveness it brings to the distributed computing world is tremendous. It revolutionizes the ways in which companies conduct business online, from Internet content delivery (wired or wireless) to electronic commerce to enterprise computing.

From a developer's perspective, Java makes your application portable among different platforms, and XML makes your data portable among different applications. These languages make our lives easier.

Wireless devices give developers a viable mobile platform to develop applications for consumers and businesses. Most of these wireless applications are not standalone. They need to exchange and share data over the wireless network with other applications including corporate databases, message-oriented middlewares, or other back-end business applications. Using XML in your wireless applications may dramatically reduce development costs and efforts while improving interoperability of your data and the flexibility of your programs.

NOTE

Recently, a special-interest group was formed among major players in the wireless device and PDA space. Its members include Nokia, Palm, Motorola, IBM, and others. The purpose of this group is to come up a vendor-neutral standard based on XML for synchronizing user data such as address books and appointments between wireless devices from different vendors and across different Internet data repository servers. The standard is SyncML (<http://www.syncml.org>). In Chapter 12, "Data Synchronization for Wireless Applications," we will show you an example of how to perform data synchronization between a MIDlet application and an Internet data repository using SyncML.

Transmitting, storing, and parsing XML data may not be an issue for traditional applications running on desktops and servers, but they can become an issue for J2ME developers who are dealing with the CPU, memory, and network bandwidth constraints of wireless devices. So, you should use XML only when it makes sense.

XML parsing creates extra CPU load and memory/storage overhead. Choosing an XML parser for your wireless application is a balance of functionality, performance, and storage overhead. A good XML parser for J2ME should be small yet robust.

Before we get into the details of individual XML parsers, the following section takes a look at XML parsing in general and how it has been used in traditional applications.

XML and Parsing XML Documents

An XML document is a *tagged* data file. The tags in an XML document define the structures and boundaries of the embedded data elements. The syntax of the tags is very similar to that of HTML. Parsing XML simply means retrieving data from an XML document based on its meaning and structure.

Listing 10.1 is a sample XML document that contains a mail message. We will use this sample XML document as the data source for all the sample programs in this chapter. This file is located at <http://www.webyu.com/book/mail.xml>.

LISTING 10.1 mail.xml

```
<?xml version="1.0"?>
<!DOCTYPE mail SYSTEM "http://www.webyu.com/book/mail.dtd" [
  <!ENTITY from "yfeng@webyu.com">
  <!ENTITY to "somebody@somewhere.com">
  <!ENTITY cc "jzhu@webyu.com">
]>
<mail>
  <From> &from; </From>
  <To> &to; </To>
  <Cc> &cc; </Cc>
  <Date>Fri, 12 Jan 2001 10:21:56 -0600</Date>
  <Subject>XML Parsers for J2ME MIDP</Subject>
  <Body language="english">
    Ælfred, NanoXML and TinyXML are the three small-foot-print
    XML parsers that are suitable for J2ME MIDP applications.
    The sizes of these parsers range from 10K to 30K, which fit
    with memory budget of MIDP devices.
  <Signature>
    -----
    Yu Feng
    &from;
    http://www.webyu.com
  </Signature>
</Body>
</mail>
```

In general, there are four main components associated with an XML document: elements, attributes, entities, and DTDs.

An *element* is something that describes a piece of data. An element is comprised of markup tags and the element's content. The following is an element in Listing 10.1:

```
<Subject>XML Parsers for J2ME MIDP</Subject>
```

It contains a start tag `<Subject>`, the content `XML Parsers for J2ME MIDP`, and an end tag `</Subject>`.

An *attribute* is used in an element to provide additional information about the element. It usually resides inside the start tag of an element. In the following example, `language` is an attribute of the element `Body` that describes the language used in the message body.

```
<Body language="english">
```

An *entity* is a virtual storage of a piece of data (either text data or binary data) that you can reference in an XML document. Entities can be further categorized into internal entities and external entities. An internal entity is defined inside an XML document and doesn't reference any outside content. For example, `from` is an internal entity defined in Listing 10.1:

```
<!ENTITY from "yfeng@webyu.com">
```

The entity `from` is later on referenced in the XML document as `&from;`. When the XML document is parsed, the parser simply replaces the entity with its actual value `yfeng@webyu.com`.

An external entity refers to content outside an XML document. Its content is usually a filename or a URL preceded with `SYSTEM` or `PUBLIC` identifier. The following is an example of an external entity `iconimage` that references to a local file called `icon.png`:

```
<!ENTITY iconimage SYSTEM "icon.png" NDATA png>
```

A *Document Type Definition (DTD)* is an optional portion of XML that defines the allowable structure for a particular XML document. Think of DTD as the roadmap and rulebook of the XML document. Listing 10.2 shows the DTD definition for the `mail.xml` shown in Listing 10.1.

LISTING 10.2 mail.dtd

```
<!ELEMENT mail (From,  
                To,  
                Cc,  
                Date,  
                Subject,  
                Body)>  
<!ELEMENT From (#PCDATA)>  
<!ELEMENT To (#PCDATA)>
```

LISTING 10.2 Continued

```
<!ELEMENT Cc          (#PCDATA)>
<!ELEMENT Date        (#PCDATA)>
<!ELEMENT Subject     (#PCDATA)>
<!ELEMENT Signature   (#PCDATA)>
<!ELEMENT Body        (#PCDATA|Signature)+>
```

This DTD basically says that the element `mail` contains six sub-elements: `From`, `To`, `Cc`, `Date`, `Subject`, and `Body`. The term `#PCDATA` refers to the “Parsed Character Data,” which indicates that an element can contain only text. The last line of the DTD definition indicates that the element `Body` could contain mixed contents that include text, sub-element `Signature`, or both.

Event-based XML Parser Versus Tree-based XML Parser

Two types of interfaces are available for parsing XML documents: the event-based interface and the tree-based interface.

An event-based XML parser reports parsing events directly to the application through callback methods. It provides a serial-access mechanism for accessing XML documents. Applications that use a parser’s event-based interface need to implement the interface’s event handlers to receive parsing events.

The Simple API for XML (SAX) is an industry standard event-based interface for XML parsing. The SAX 1.0 Java API defines several callback methods in one of its interface classes. The applications need to implement these callback methods to receive parsing events from the parser. For example, the `startElement()` is one of the callback methods. When a SAX parser reaches the start tag of an element, the application that implements the parser’s `startElement()` method will receive the event, and also receive the tag name through one of the method’s parameters.

A tree-based XML parser reads an entire XML document into an internal tree structure in memory. Each node of the tree represents a piece of data from the original document. It allows an application to navigate and manipulate the parsed data quickly and easily.

The Document Object Model (DOM) is an industry standard tree-based interface for XML parsing. A DOM parser can be very memory- and CPU-intensive because it keeps the whole data structure in memory. A DOM parser may become a performance issue for your wireless applications, especially when the XML document to be parsed is large and complex.

In general, SAX parsers are faster and consume less CPU and memory than DOM parsers. But the SAX parsers allow only serial access to the XML data. DOM parsers’ tree-structured data

is easier to access and manipulate. SAX parsers are often used by Java servlets or network-oriented programs to transmit and receive XML documents in a fast and efficient fashion. DOM parsers are often used for manipulating XML documents.

For traditional Java applications, several Java-based XML parsers are available from different software vendors, such as Sun, IBM, and Microsoft. For example, Sun's Java API for XML Processing (JAXP) package defines both SAX and DOM APIs. These XML parsers provide a rich set of features for dealing with XML data within enterprise applications. But these parsers are too big for J2ME MIDP applications. The total size of JAXP is about 140KB. It doesn't fit on the J2ME MIDP devices that only have a storage budget of a few hundred kilobytes.

However, several small-footprint Java-based XML parsers are good candidates for wireless applications. Their code is small, their performance is robust, and they're very functional.

The following sections take a look at three small-footprint XML parsers: the TinyXML parser, the NanoXML parser, and the Ælfred parser. Using these XML parsers generally creates a storage overhead of 10KB to 30KB. Each parser has its own pros and cons. You will see how to evaluate them based on three criteria: functionality, performance, and code size.

XML Parsers for Wireless Applications

The TinyXML, NanoXML, and Ælfred parsers are all Java based. They were originally designed for use with embedded Java applications or Java applets. With some modifications, they should be able to fit in the J2ME MIDP's resource-constrained environment.

Because all three XML parsers were originally written in J2SE, they must be ported to J2ME MIDP before they can be used with MIDlet applications. Some of the J2ME porting work has already been done by other developers; we collected those porting efforts, made some additional modifications, and repackaged them in a way that can be more easily presented to you.

One of the goals of this book is to give you, as a J2ME MIDP developer, a jump-start on your development efforts. Toward this end, several sample programs that use these parsers are also listed and explained in this chapter.

There are pros and cons associated with using each of these parsers. You must evaluate them individually and use whichever is best suited for your applications. They are all open-source packages, so you can customize them to fit your own development needs as long as the license agreement is met.

Choosing the right XML parser for your application should be based on its functionality, code size, and performance. The following sections compare the three parsers in those areas, and also look at their license differences.

Functionality

All three parsers are non-validating parsers. TinyXML and Ælfred support XML DTD definitions, but NanoXML doesn't. TinyXML and Ælfred support both internal and external entities, but NanoXML doesn't.

All three parsers come with event-based interfaces. NanoXML has a SAX 1.0-compliant event-based interface. TinyXML's event-based interface is proprietary and doesn't support SAX 1.0. Ælfred provides both a proprietary event-based interface and a SAX 1.0-compliant interface.

None of the parsers support the standard DOM interface. However, NanoXML and TinyXML both provide a simple tree-based interface. Ælfred is solely event-based; it doesn't have a tree-based interface.

Code Size

In J2ME MIDP environment, developers must constantly be aware of the CPU and memory usage of their applications. Using XML parsers along with your applications creates a storage overhead of 10KB to 30KB, as shown in Table 10.1. (`sax10_midp.jar` contains the J2ME version of SAX1.0 Java API.)

TABLE 10.1 Comparison of the Parsers' Code Sizes

<i>Package(s)</i>	<i>File(s) to be Included</i>	<i>Size (JAR Compressed)</i>
TinyXML's proprietary event-based interface	<code>tinyxml_event.jar</code>	10KB
TinyXML's proprietary tree-based interface	<code>tinyxml_tree.jar</code>	14KB
NanoXML's proprietary tree-based interface	<code>nanoxml_tree.jar</code>	9KB
NanoXML's SAX interface	<code>nanoxml_sax.jar</code> + <code>sax10_midp.jar</code>	21KB
Ælfred's proprietary event-based interface	<code>aelfred_event.jar</code>	18KB
Ælfred's SAX interface	<code>aelfred_sax.jar</code> + <code>sax10_midp.jar</code>	30KB

The code sizes shown in Table 10.1 are the sizes of compressed JAR files. A Java obfuscator can be used to further reduce the size of these packages.

Using a standard SAX 1.0 interface with these parsers adds about 10K storage overhead on top of existing code (that's the size of SAX 1.0 Java interface); however, using a standard interface offers certain advantages. Your programs will be less reliant on an individual parser, and will therefore be more portable. You'll see an actual example to illustrate this point later in the chapter.

Performance

In general, event-driven interfaces offer better performance than tree-based interfaces because they consume less CPU and memory. However, tree-based interfaces are easier to use because the parsed data is already in a tree-structure.

If you are interested in benchmarking the actual performance of these parsers, you can run the sample code with each parser to parse the same set of XML data and monitor their CPU consumption and memory usage.

Licenses

All three parsers are free, which means they are open-source licensed. Developers have full access to the source code and can modify it for their own use. But make sure that your packaging complies with the individual license agreements.

Generally speaking, Ælfred and NanoXML are under a looser license for commercial and non-commercial use. TinyXML is under a GPL license, which a little more restrictive than the other two.

SAX 1.0 Java API For J2ME MIDP

Author: David Megginson

Original Web URL: <http://www.megginson.com/SAX/SAX1/index.html>

Last package release date: May 11, 1998

Last release version: 1.0

License agreement: Free for both commercial and non-commercial use.

SAX is a standard interface for event-based XML parsing, initiated and maintained by David Megginson (who is also the author of the Ælfred parser). The SAX API specification was developed collaboratively by the members of the XML-DEV mailing list. The SAX API provides a standard way of dealing with XML documents and has been adopted by most XML parser implementations.

The SAX 1.0 API basically defines a number of standard interfaces that every SAX-compliant parser must support. For example, all SAX parsers must support eight callback methods defined in SAX's `DocumentHandler` interface. These callback methods are `startDocument()`,

`endDocument()`, `startElement()`, `endElement()`, `characters()`, `ignorableWhitespace()`, `processingInstruction()`, and `setDocumentLocator()`. The applications that implement these methods will receive parsing events and parsed data from the parser while the XML document is being processed.

The benefit of using standards is that it makes your code portable, the same thing is true with using the SAX API. The following example shows how to plug a different SAX parser into your program without modifying source code. Your code is no longer parser dependent.

The first sample program uses NanoXML's SAX interface:

```
try {
    Parser parser = ParserFactory.makeParser(
        "nanoxml.sax.SAXParser");
    DemoHandler myHandler = new DemoHandler();
    parser.setDocumentHandler(myHandler);
    parser.parse(urlstring);
    resultString = myHandler.getResultString();
} catch (Exception e) {
    System.out.println(e);
}
```

The second sample program uses Ælfred's SAX interface:

```
try {
    Parser parser = ParserFactory.makeParser(
        "com.microstar.xml.SAXDriver");
    DemoHandler myHandler = new DemoHandler();
    parser.setDocumentHandler(myHandler);
    parser.parse(urlstring);
    resultString = myHandler.getResultString();
} catch (Exception e) {
    System.out.println("startApp: " + e);
}
```

The only difference between the two sample programs is the parameter string inside the `ParserFactory.makeParser()` method. The parameter string indicates which parser to use. To make the code more portable, the value of the parameter string can be placed in a resource file or set as a system property, and be read into the application at runtime.

Since both the NanoXML parser and the Ælfred parser provide optional SAX adapters to support SAX 1.0, it is worthwhile to spend some effort porting the original SAX 1.0 Java API to J2ME MIDP.

Porting the original SAX 1.0 Java API (written in J2SE) to J2ME is very straightforward. `java.util.locale` is the only class that needs to be ported to J2ME. For simplicity, an empty

class `java.util.locale` is created for the package. Because there is nothing in the simulated `java.util.locale` class, the original use of `locale` is no longer supported in our J2ME version of the SAX 1.0 Java API. To use the SAX API in your program, make sure `sax10_midp.jar` is included in the Java class path.

The drawback to using the SAX interface in your MIDP application is that you have to include `sax10_midp.jar`, the SAX 1.0 Java API package, which is about 10KB in size. You can find more information about the SAX 1.0 API at <http://www.megginson.com/SAX/SAX1/index.html>.

TinyXML Parser for J2ME MIDP

Author: Tom Gibara

Original Web URL: <http://gibaradunn.srac.org/tiny/index.shtml>

Last package release date: January 2, 2000

Last release version: 0.7

License agreement: GPL <http://www.fsf.org/copyleft/gpl.html>

Christian Sauer has ported a J2ME version of TinyXML (you can find more information at <http://www.kvmworld.com/Articles/TinyXML.shtml>). The TinyXML package used in this book is a combination of Christian's contribution and some modifications we made to preserve the structures and features of the original TinyXML parser.

TinyXML is a non-validating parser. It supports both UTF-8 and UTF-16 encoding in XML documents. It also supports DTD and internal/external entities. It consists of two sets of interfaces: a non-SAX event-based interface and a simple tree-based interface. For convenience, the class files for the event-based interface are all included in `tinymce_event.jar`. The class files for the tree-based interface are all included in `tinymce_tree.jar`.

`tinymce_event.jar` contains the class files of the following Java files:

- `gd/xml/CharacterUtility.java`: Christian Sauer wrote the two static methods `isLetter()` and `isLetterOrDigit()` to replace the equivalent methods in the `Character` class that are not available in J2ME.
- `gd/xml/XMLReader.java`: Defines input/output utility methods for XML parsing.
- `gd/xml/ParseException.java`: Defines the exception for errors generated by the parser.
- `gd/xml/XMLResponder.java`: Defines the interface of all event callback methods.
- `gd/xml/XMLParser.java`: Defines XML parser logic.

`tinymce.jar` contains all the class files in `tinymce_event.jar` plus four additional class files of the following Java files:

- `gd/xml/tiny/ParsedXML.java`: Defines the interface of a tree node for storing an element of the parsed XML data.
- `gd/xml/tiny/ParseNode.java`: Implements the `ParsedXML` interface and defines a tree node to store an element of the parsed XML data.
- `gd/xml/tiny/TinyParser.java`: Defines a wrapper class around `XMLParser` to support the tree-based interface.
- `gd/xml/tiny/TinyResponder.java`: Implements the `XMLResponder`'s callback methods to construct the tree structure out of the parsed XML data.

An Example of Using TinyXML's Event-based Interface

To use TinyXML's event-based interface, you need to include the package `tinymce_event.jar` in your Java class path. You also must implement all the callback methods defined in the `XMLResponder` interface to receive parsing events from the parser.

The following sample application illustrates how to use TinyXML's event-based interface. Our example contains two Java files: `DemoEventResponder.java` (which implements the `XMLResponder` interface) and `tinyEventDemo.java` (a J2ME MIDlet). The XML data source is shown in Listing 10.1 and Listing 10.2.

LISTING 10.3 `DemoEventResponder.java`

```
import javax.microedition.io.*;
import java.util.*;
import java.io.*;
// include TinyXML's event-based interface
import gd.xml.*;

public class DemoEventResponder implements XMLResponder {

    private InputStream is;
    private InputConnection ic;
    private InputStream is_sysid;
    private InputConnection ic_sysid;
    private String prefix;
    private StringBuffer resultStringBuffer;

    public DemoEventResponder(String _url) throws ParseException {
        prefix = "> ";
        resultStringBuffer = new StringBuffer();
    }
}
```

LISTING 10.3 Continued

```
        try {
            ic = (InputConnection) Connector.open(_url);
            is = ic.openInputStream();
        } catch (IOException ioex) {
            throw new ParseException(
                "Failed to open http connection: " + ioex);
        }
    }

    public String getResultString() {
        return resultStringBuffer.toString();
    }

    public void closeConnection() {
        try { if (ic != null) ic.close(); }
        catch (Exception ignored) {}

        try { if (is != null) is.close(); }
        catch (Exception ignored) {}

        try { if (ic_sysid != null) ic_sysid.close(); }
        catch (Exception ignored) {}

        try { if (is_sysid != null) is_sysid.close(); }
        catch (Exception ignored) {}
    }

    public void recordNotationDeclaration(
        String name,String pubID,String sysID)
        throws ParseException {
        System.out.print(prefix+"!NOTATION: "+name);
        resultStringBuffer.append(prefix+"!NOTATION: "+name);
        if (pubID!=null)
        {
            System.out.print("  pubID = "+pubID);
            resultStringBuffer.append("  pubID = "+pubID);
        }
        if (sysID!=null)
        {
            System.out.print("  sysID = "+sysID);
            resultStringBuffer.append("  sysID = "+sysID);
        }
        System.out.println("");
    }
}
```

LISTING 10.3 Continued

```
        resultStringBuffer.append("\n");
    }

    public void recordEntityDeclaration(String name, String value,
        String pubID, String sysID, String notation)
        throws ParseException {
        System.out.print(prefix+"!ENTITY: "+name);
        resultStringBuffer.append(prefix+"!ENTITY: "+name);
        if (value!=null)
        {
            System.out.print("  value = "+value);
            resultStringBuffer.append("  value = "+value);
        }
        if (pubID!=null)
        {
            System.out.print("  pubID = "+pubID);
            resultStringBuffer.append("  pubID = "+pubID);
        }
        if (sysID!=null)
        {
            System.out.print("  sysID = "+sysID);
            resultStringBuffer.append("  sysID = "+sysID);
        }
        if (notation!=null)
        {
            System.out.print("  notation = "+notation);
            resultStringBuffer.append("  notation = "+notation);
        }
        System.out.println("");
        resultStringBuffer.append("\n");
    }

    public void recordElementDeclaration(String name, String content)
        throws ParseException {
        System.out.print(prefix+"!ELEMENT: "+name);
        resultStringBuffer.append(prefix+"!ELEMENT: "+name);
        System.out.println("  content = "+content);
        resultStringBuffer.append("  content = "+content + "\n");
    }

    public void recordAttlistDeclaration(String element, String attr,
        boolean notation, String type, String defmod, String def)
        throws ParseException {
        System.out.print(prefix+"!ATTLIST: "+element);
        resultStringBuffer.append(prefix+"!ATTLIST: "+element);
```

LISTING 10.3 Continued

```
        System.out.print(" attr = "+attr);
        resultStringBuffer.append(" attr = "+attr);
        System.out.print(
            " type = " + ((notation) ? "NOTATIONS " : "") + type);
        resultStringBuffer.append(
            " type = " + ((notation) ? "NOTATIONS " : "") + type);
        System.out.print(" def. modifier = "+defmod);
        resultStringBuffer.append(" def. modifier = "+defmod);
        System.out.println( (def==null) ? " " : " def = "+notation);
        resultStringBuffer.append(
            (def==null) ? "\n" : " def = "+notation + "\n");
    }

    public void recordDoctypeDeclaration(
        String name,String pubID,String sysID)
        throws ParseException {
        System.out.print(prefix+"!DOCTYPE: "+name);
        resultStringBuffer.append(prefix+"!DOCTYPE: "+name);
        if (pubID!=null)
        {
            System.out.print(" pubID = "+pubID);
            resultStringBuffer.append(" pubID = "+pubID);
        }
        if (sysID!=null)
        {
            System.out.print(" sysID = "+sysID);
            resultStringBuffer.append(" sysID = "+sysID);
        }
        System.out.println("");
        resultStringBuffer.append("\n");
        prefix = "";
    }

    public void recordDocStart() {
        System.out.println("Parsing began");
        resultStringBuffer.append("Parsing began\n");
    }

    public void recordDocEnd() {
        System.out.println("");
        resultStringBuffer.append("\n");
        System.out.println("Parsing finished without error");
    }
}
```

LISTING 10.3 Continued

```
        resultStringBuffer.append("Parsing finished without error\n");
    }

    public void recordElementStart(String name, Hashtable attr)
    throws ParseException {
        System.out.println(prefix+"ELEMENT: "+name);
        resultStringBuffer.append(prefix+"ELEMENT: "+name +"\n");
        if (attr!=null) {
            Enumeration e = attr.keys();
            System.out.print(prefix);
            resultStringBuffer.append(prefix);
            String conj = "ATTR: ";
            while (e.hasMoreElements()) {
                Object k = e.nextElement();
                System.out.print(conj+k+" = "+attr.get(k));
                resultStringBuffer.append(conj+k+" = "+attr.get(k));
                conj = ", ";
            }
            System.out.println("");
            resultStringBuffer.append("\n");
        }
        prefix = prefix+" ";
    }

    public void recordElementEnd(String name) throws ParseException {
        prefix = prefix.substring(2);
    }

    public void recordPI(String name, String pValue) {
        System.out.println(prefix+"*"+name+" PI: "+pValue);
        resultStringBuffer.append(prefix+"*"+name+" PI: "+pValue + "\n");
    }

    public void recordCharData(String charData) {
        System.out.println(prefix+charData);
        resultStringBuffer.append(prefix+charData + "\n");
    }

    public void recordComment(String comment) {
        System.out.println(prefix+"*Comment: "+comment);
        resultStringBuffer.append(prefix+"*Comment: "+comment + "\n");
    }
}
```

LISTING 10.3 Continued

```
public InputStream getDocumentStream() throws ParseException {
    return is;
}

public InputStream resolveExternalEntity(
    String name,String pubID,String sysID)
    throws ParseException {
    if (sysID!=null) {
        try {
            ic_sysid = (InputConnection) Connector.open(sysID);
            is_sysid = ic_sysid.openInputStream();
            return is_sysid;
        }
        catch (IOException e) {
            throw new ParseException("Failed to open http connection: "
                + sysID);
        }
    }
    else return null;
}

public InputStream resolveDTDEntity(
    String name, String pubID, String sysID)
    throws ParseException {
    return resolveExternalEntity(name, pubID, sysID);
}
}
```

`DemoEventResponder.java` implements all 15 callback methods defined in TinyXML's event-based interface `XMLResponder`. These callback methods basically are implemented with a number of print statements that print out the contents received from the parser while the XML document is being processed. The callback methods defined in `XMLResponder` are similar to the ones defined in the SAX API. For example, `recordElementStart()`, `recordElementEnd()`, and `recordCharData()` are very similar to the SAX API's `startElement()`, `endElement()`, and `characters()` methods.

The program in Listing 10.4 is a J2ME MIDlet application that reads XML data from `http://www.webyu.com/book/mail.xml` by instantiating a `DemoEventResponder` object `myResponder = new DemoEventResponder(urlstring);`, parses the data using the TinyXML parser by calling `xp.parseXML(myResponder);`, and uses the `DemoEventResponder` callbacks to print the parsing results.

LISTING 10.4 tinyEventDemo.java

```
import java.io.*;
import java.util.*;
import java.lang.String;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import gd.xml.*;

public class tinyEventDemo extends MIDlet implements CommandListener {

    private String url;
    private DemoEventResponder myResponder;

    // GUI component for user to enter url for the xml document
    private Display myDisplay = null;
    private Form mainScreen;
    private TextField requestField;

    // GUI component for displaying xml data content
    private Form resultScreen;
    private StringItem resultField;

    // the "send" button used on mainScreen
    Command sendCommand = new Command("SEND", Command.OK, 1);
    // the "back" button used on resultScreen
    Command backCommand = new Command("BACK", Command.OK, 1);

    public tinyEventDemo() {
        // default url
        url = "http://www.webyu.com/book/mail.xml";

        // initializing the GUI components for entering url
        // for the xml document
        myDisplay = Display.getDisplay(this);
        mainScreen = new Form("Type in a URL:");
        requestField =
            new TextField(null, url,
                100, TextField.URL);
        mainScreen.append(requestField);
        mainScreen.addCommand(sendCommand);
        mainScreen.setCommandListener(this);
    }
}
```

LISTING 10.4 Continued

```
public void startApp() throws MIDletStateChangeException {
    myDisplay.setCurrent(mainScreen);
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
    myResponder.closeConnection();
}

public void commandAction(Command c, Displayable s) {

    // when user clicks on "send" button on mainScreen
    if (c == sendCommand) {

        // retrieving the web url that user entered
        String urlString = requestField.getString();

        String resultstring = "";

        try {
            myResponder = new DemoEventResponder(urlstring);
            XMLParser xp = new XMLParser();
            xp.parseXML(myResponder);
            resultstring = myResponder.getResultString();
        } catch (ParseException e) {
            System.out.println(e);
        }

        // displaying the page content retrieved from web server
        resultScreen = new Form("XML Result:");
        resultField =
            new StringItem(null, resultstring);
        resultScreen.append(resultField);
        resultScreen.addCommand(backCommand);
        resultScreen.setCommandListener(this);
        myDisplay.setCurrent(resultScreen);

    } else if (c == backCommand) {

        // do it all over again
        requestField.setString(url);
    }
}
```

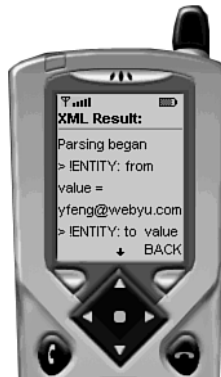
LISTING 10.4 Continued

```
        myDisplay.setCurrent(mainScreen);  
    }  
}  
}
```

Figures 10.1 and 10.2 show the `tinyEventDemo` program.

**FIGURE 10.1**

Entering the Web URL for the XML data to be parsed.

**FIGURE 10.2**

Display for the parsed XML data elements.

PART II

The actual command-line output from this program is as follows:

```
Parsing began
> !ENTITY: from value = yfeng@webyu.com
> !ENTITY: to value = somebody@somewhere.com
> !ENTITY: cc value = jzhu@webyu.com
> !ELEMENT: mail content = (From,
    To,
    Cc,
    Date,
    Subject,
    Body)
> !ELEMENT: From content = (#PCDATA)
> !ELEMENT: To content = (#PCDATA)
> !ELEMENT: Cc content = (#PCDATA)
> !ELEMENT: Date content = (#PCDATA)
> !ELEMENT: Subject content = (#PCDATA)
> !ELEMENT: Signature content = (#PCDATA)
> !ELEMENT: Body content = (#PCDATA|Signature)+
> !DOCTYPE: mail sysID = http://www.webyu.com/book/mail.dtd
ELEMENT: mail
ELEMENT: From
    yfeng@webyu.com
ELEMENT: To
    somebody@somewhere.com
ELEMENT: Cc
    jzhu@webyu.com
ELEMENT: Date
    Fri, 12 Jan 2001 10:21:56 -0600
ELEMENT: Subject
    XML Parsers for J2ME MIDP
ELEMENT: Body
ATTR: language = english
    Ælfred, NanoXML and TinyXML are the three small-foot-print
    XML parsers that are suitable for J2ME MIDP applications.
    The sizes of these parsers range from 10K to 30K, which fit
    with memory budget of MIDP devices.

ELEMENT: Signature
-----
    Yu Feng
    yfeng@webyu.com
    http://www.webyu.com
Parsing finished without error
```

By looking at the output, you can verify two things quickly:

- TinyXML supports DTD and mixed content. The DTD definition `http://www.webyu.com/book/mail.dtd` is parsed correctly, and the sub-element `<Signature>` inside `<Body>` is parsed out correctly based on the DTD definition.
- TinyXML supports entities. The entities `from`, `to`, and `cc` are parsed correctly and used to substitute `&from;`, `&to;`, and `&cc;` with their actual contents.

An Example of Using TinyXML's Tree-based Interface

To use TinyXML's tree-based interface, first you must make sure the package `tinyxml_tree.jar` is included in your Java class path. Unlike using the event-based interface, there is no need to implement the callback methods in `XMLResponder` in order to use the tree-based interface. The callback methods are already implemented in TinyXML's `TinyResponder` class for constructing the tree structure out of parsed XML data.

The sample application in Listing 10.5 illustrates how to use TinyXML's tree-based interface. The same data source `mail.xml` from Listing 10.1 is used in this example. The parser takes `url` as input and returns a tree structure `root: root = TinyParser.parseXML(url);`. Because the parsed result is stored in a tree structure, a recursive method `displayNode()` is used to display the content stored in the tree structure.

LISTING 10.5 `tinyTreeDemo.java`

```
import java.io.*;
import java.util.*;
import java.lang.String;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import gd.xml.*;
import gd.xml.tiny.*;

public class tinyTreeDemo extends MIDlet
    implements CommandListener {

    private String url;
    private ParsedXML root;

    // GUI component for user to enter url for the xml document
    private Display myDisplay = null;
    private Form mainScreen;
    private TextField requestField;
```

LISTING 10.5 Continued

```
// the "send" button used on mainScreen
Command sendCommand = new Command("SEND", Command.OK, 1);

public tinyTreeDemo() {
    url = "http://www.webyu.com/book/mail.xml";

    // initializing the GUI components for entering url
    // for the xml document
    myDisplay = Display.getDisplay(this);
    mainScreen = new Form("Type in a URL:");
    requestField =
        new TextField(null, url,
            100, TextField.URL);
    mainScreen.append(requestField);
    mainScreen.addCommand(sendCommand);
    mainScreen.setCommandListener(this);
}

public void startApp() throws MIDletStateChangeException {
    myDisplay.setCurrent(mainScreen);
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {

    // when user clicks on "send" button on mainScreen
    if (c == sendCommand) {

        // retrieving the web url that user entered
        String urlString = requestField.getString();
        try {
            root = TinyParser.parseXML(url);
            displayNode(root);
        } catch (ParseException e) {
            System.err.println("startApp: " + e);
        }
    }
}

private void displayNode(ParsedXML px) {
```

LISTING 10.5 Continued

```
//choose name
String nodeName = px.getTypeName();
if (px.getName()!=null)
    nodeName += " <" + px.getName() + ">";
String nodeContent = px.getContent();
if (nodeContent==null) nodeContent = "";
System.out.print(nodeName + ":");
System.out.println(nodeContent);

//add subtrees
Enumeration e;

//add attributes
e = px.attributes();
if (e.hasMoreElements()) {
    System.out.print("attribute:");
    while (e.hasMoreElements()) {
        String attrName = (String)e.nextElement();
        System.out.println(
            attrName+ ":" +px.getAttribute(attrName) );
    }
}

e = px.elements();
if (e.hasMoreElements()) {
    while (e.hasMoreElements())
        displayNode((ParsedXML)e.nextElement());
}
}
```

The actual command-line output from this program basically generates the same output as tinyTreeDemo:

```
root:
tag <mail>:
tag <From>:
text:yfeng@webyu.com
tag <To>:
text:somebody@somewhere.com
tag <Cc>:
text:jzhu@webyu.com
tag <Date>:
```

```
text:Fri, 12 Jan 2001 10:21:56 -0600
tag <Subject>:
text:XML Parsers for J2ME MIDP
tag <Body>:
attribute:language:english
text:Ælfred, NanoXML and TinyXML are the three small-foot-print
      XML parsers that are suitable for J2ME MIDP applications.
      The sizes of these parsers range from 10K to 30K, which fit
      with memory budget of MIDP devices.

tag <Signature>:
text:-----
      Yu Feng
      yfeng@webyu.com
      http://www.webyu.com
```

You can find more information about the TinyXML parser at <http://gibaradunn.srac.org/tiny/index.shtml>.

NanoXML Parser for J2ME MIDP

Author: Marc De Scheemaeker
Original Web URL: <http://nanoxml.sourceforge.net/index.html>
Last package release date: November 29, 2000
Last release version: 1.6.7
License agreement: zlib/libpng,
<http://www.opensource.org/licenses/zlib-license.html>

Eric Giguere has ported NanoXML to J2ME (you can find more information at http://www.ericgiguere.com/microjava/cldc_xml.html). The NanoXML package used in this book is a combination of Eric Giguere's contribution and some modifications we made to the parser's SAX adapter.

NanoXML is a non-validating XML parser. It contains two sets of interfaces: a simple tree-based interface and a SAX 1.0-compliant event-based interface.

NanoXML doesn't support DTD and entities, and it doesn't support mixed content either. You can see these limitations from the output of the sample programs in this section.

For convenience, the class files for the NanoXML simple tree-based interface are included in `nanoxml_tree.jar`. The class files for the SAX interface are included in `nanoxml_sax.jar`.

nanoxml_tree.jar contains the class files of the following Java files:

- nanoxml/XMLElement.java: Implements the XML parser functionality. Ported to J2ME by Eric Giguere.
- nanoxml/XMLParseException.java: Defines the exception for errors generated by the parser.

nanoxml_sax.jar contains all the class files in nanoxml_tree.jar and the class files of the following Java files to support SAX 1.0 API:

- nanoxml/sax/SAXLocator.java: Implements the Locator interface in SAX 1.0 Java API.
- nanoxml/sax/SAXParser.java: The SAX adapter of NanoXML parser.

An Example of Using NanoXML's Tree-based Interface

To use NanoXML's tree-based interface, nanoxml_tree.jar must be included in your Java class path.

The sample application in Listing 10.6 (nanoTreeDemo.java) illustrates how to use NanoXML's tree-based interface. The XML data source is shown in Listing 10.1 and Listing 10.2. The program reads the XML document for the URL and passes the whole content of the document as a big string to the parser: `foo.parseString(xml.toString(), 0)`; `foo` itself is a tree-structure that contains all the parsed data. A recursive method `displayTree()` is used to display the content stored in the tree structure.

LISTING 10.6 nanoTreeDemo.java

```
import java.io.*;
import java.util.*;
import java.lang.String;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import nanoxml.*;

public class nanoTreeDemo extends MIDlet implements CommandListener {

    private String url;

    // for output display
    private StringBuffer resultStringBuffer;

    // GUI component for user to enter url for the xml document
    private Display myDisplay = null;
```

LISTING 10.6 Continued

```
private Form mainScreen;
private TextField requestField;

// GUI component for displaying xml data content
private Form resultScreen;
private StringItem resultField;

// the "send" button used on mainScreen
Command sendCommand = new Command("SEND", Command.OK, 1);
// the "back" button used on resultScreen
Command backCommand = new Command("BACK", Command.OK, 1);

public nanoTreeDemo() {
    // default url
    url = "http://www.webyu.com/book/mail.xml";

    resultStringBuffer = new StringBuffer();

    // initializing the GUI components for entering url
    // for the xml document
    myDisplay = Display.getDisplay(this);
    mainScreen = new Form("Type in a URL:");
    requestField =
    new TextField(null, url,
    100, TextField.URL);
    mainScreen.append(requestField);
    mainScreen.addCommand(sendCommand);
    mainScreen.setCommandListener(this);
}

public void startApp() throws MIDletStateChangeException {
    myDisplay.setCurrent(mainScreen);
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {
```

LISTING 10.6 Continued

```
// when user clicks on "send" button on mainScreen
if (c == sendCommand) {

    // retrieving the web url that user entered
    String urlString = requestField.getString();

    InputConnection ic = null;
    InputStream is = null;

    StringBuffer xml = new StringBuffer();
    try {
        ic = (InputConnection) Connector.open(url);
        is = ic.openInputStream();
        int ch;
        while ( (ch = is.read()) != -1) {
            xml.append((char) ch);
        }
    } catch (IOException e) {
    } finally {
        try { if (ic!=null) ic.close(); }
        catch (Exception e) {}
        try { if (is!=null) is.close(); }
        catch (Exception e) {}
    }

    try {

        XMLElement foo = new XMLElement();
        foo.parseString(xml.toString(), 0);
        displayTree(foo);

        System.out.println("");
        System.out.println(
            "-----original XML-----");
        System.out.println(foo);
        System.out.println(
            "-----original XML-----");

    } catch (Exception e) {
        System.err.println(e);
    }

    // displaying the page content retrieved from web server
    resultScreen = new Form("XML Result:");
    resultField =
```

LISTING 10.6 Continued

```
        new StringItem(null, resultStringBuffer.toString());
        resultScreen.append(resultField);
        resultScreen.addCommand(backCommand);
        resultScreen.setCommandListener(this);
        myDisplay.setCurrent(resultScreen);

    } else if (c == backCommand) {

        // do it all over again
        requestField.setString(url);
        myDisplay.setCurrent(mainScreen);
    }
}

public void displayTree(XMLElement node) {
    if ( node.getTagName() != null )
    {
        System.out.println("<" + node.getTagName() + ">");
        resultStringBuffer.append(
            "<" + node.getTagName() + ">\n");
    }

    if ( node.getContents() != null )
    {
        System.out.println("    contents: " + node.getContents());
        resultStringBuffer.append(
            "    contents: " + node.getContents() + "\n");
    }

    Enumeration enum = node.enumerateChildren();
    while (enum.hasMoreElements()) {
        XMLElement bar = (XMLElement)(enum.nextElement());
        displayTree(bar);
    }
}
}
```

The actual command-line output from nanoTreeDemo is as follows:

```
<mail>
<From>
    contents:    &from;
<To>
    contents:    &to;
```

```
<Cc>
  contents:      &cc;
<Date>
  contents: Fri, 12 Jan 2001 10:21:56 -0600
<Subject>
  contents: XML Parsers for J2ME MIDP
<Body>
  contents:
    Ælfred, NanoXML and TinyXML are the three small-foot-print
    XML parsers that are suitable for J2ME MIDP applications.
    The sizes of these parsers range from 10K to 30K, which fit
    with memory budget of MIDP devices.
    <Signature>
    -----
    Yu Feng
    &from;
    http://www.webyu.com
    </Signature>
```

Because NanoXML doesn't support DTD and entities, the DTD definition is ignored and as a result the sub-element `<Signature>` is treated as part of the `<Body>` contents. The entities `from`, `to` and `cc` are not substituted correctly either in the result.

An Example of Using NanoXML's SAX Interface

The following sample application illustrates how to use NanoXML's SAX interface. Because NanoXML supports SAX 1.0 Java API, both `nanoxml_sax.jar` and `sax10_midp.jar` must be included in your Java class path.

To use the SAX 1.0 API, you need to implement all the callback methods defined in the `DocumentHandler` interface. You can also extend the convenience class `HandlerBase` when you need to only implement part of the interface. The `HandlerBase` class implements the default behavior of the `DocumentHandler` interface. Both `DocumentHandler` and `HandlerBase` can be found in the SAX 1.0 Java API.

The example in Listings 10.7 and 10.8 consists of two files: `DemoHandler.java` and `nanoSaxDemo.java`. `DemoHandler` extends SAX API's convenience class `HandlerBase` and overrides its `getResultString()`, `startDocument()`, `endDocument()`, `resolveEntity()`, `startElement()`, `endElement()`, `characters()`, and `processingInstruction()` methods. `nanoSaxDemo` reads the XML data from a Web server and calls NanoXML parser to parse the data.

LISTING 10.7 DemoHandler.java

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.Reader;

public class DemoHandler extends HandlerBase {

    private StringBuffer resultStringBuffer;

    public DemoHandler() {
        resultStringBuffer = new StringBuffer();
    }

    public String getResultString() {
        return resultStringBuffer.toString();
    }

    public void startDocument ()
    throws SAXException
    {
        System.out.println("Start parsing document >>>>");
    }

    public void endDocument ()
    throws SAXException
    {
        System.out.println("Finish parsing document <<<<");
    }

    public InputSource resolveEntity (
        String publicId, String systemId)
    throws SAXException
    {
        return new InputSource(systemId);
    }

    public void startElement (
        String elname, AttributeList attributes)
    throws SAXException
    {
        System.out.println("<" + elname + ">");
        AttributeListImpl myatts =
            new AttributeListImpl(attributes);
        if ( myatts.getLength() > 0 ) {
            System.out.print("  Attribute:\t");
            resultStringBuffer.append("  Attribute:");
        }
    }
}
```

LISTING 10.7 Continued

```
        for (int i = 0; i < myatts.getLength(); i++)
        {
            System.out.print(myatts.getName(i) + ":");
            resultStringBuffer.append(
                myatts.getName(i) + ":");
            System.out.println(myatts.getValue(i));
            resultStringBuffer.append(
                myatts.getValue(i) + "\n");
        }
    }

    public void endElement (String elname)
    throws SAXException
    {
        System.out.println(" END <" + elname + ">");
        resultStringBuffer.append(" END <" + elname + ">\n");
    }

    public void characters (char ch[], int start, int length)
    throws SAXException
    {
        String contents = new String(ch, start, length);
        System.out.println(" Contents:\t" + contents);
        resultStringBuffer.append(
            " contents:" + contents + "\n");
    }

    public void processingInstruction (String target, String data)
    throws SAXException
    {
        System.out.println("PI:");
        System.out.println("target:" + target);
        System.out.println("data:" + data);
    }
}
```

LISTING 10.8 nanoSaxDemo.java

```
import java.io.*;
import java.util.*;
import java.lang.String;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
```

LISTING 10.8 Continued

```
import javax.microedition.midlet.*;
import javax.xml.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class nanoSaxDemo extends MIDlet
    implements CommandListener {

    private String url;

    // GUI component for user to enter url for the xml document
    private Display myDisplay = null;
    private Form mainScreen;
    private TextField requestField;

    // GUI component for displaying xml data content
    private Form resultScreen;
    private StringItem resultField;

    // the "send" button used on mainScreen
    Command sendCommand = new Command("SEND", Command.OK, 1);
    // the "back" button used on resultScreen
    Command backCommand = new Command("BACK", Command.OK, 1);

    public nanoSaxDemo() {
        // default url
        url = "http://www.webyu.com/book/book.xml";

        // initializing the GUI components for entering url
        // for the xml document
        myDisplay = Display.getDisplay(this);
        mainScreen = new Form("Type in a URL:");
        requestField =
            new TextField(null, url,
                100, TextField.URL);
        mainScreen.append(requestField);
        mainScreen.addCommand(sendCommand);
        mainScreen.setCommandListener(this);
    }

    public void startApp() throws MIDletStateChangeException {

        myDisplay.setCurrent(mainScreen);
    }
}
```

LISTING 10.8 Continued

```
public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {

    // when user clicks on "send" button on mainScreen
    if (c == sendCommand) {

        // retrieving the web url that user entered
        String urlString = requestField.getString();

        String resultString = "";

        try {

            Parser parser =
                ParserFactory.makeParser("nanoxml.sax.SAXParser");
            DemoHandler myHandler = new DemoHandler();
            parser.setDocumentHandler(myHandler);
            parser.parse(urlString);
            resultString = myHandler.getResultString();

        } catch (Exception e) {
            System.err.println(e);
        }

        // displaying the page content retrieved from web server
        resultScreen = new Form("XML Result:");
        resultField =
            new StringItem(null, resultString);
        resultScreen.append(resultField);
        resultScreen.addCommand(backCommand);
        resultScreen.setCommandListener(this);
        myDisplay.setCurrent(resultScreen);

    } else if (c == backCommand) {
        // do it all over again
        requestField.setString(url);
        myDisplay.setCurrent(mainScreen);
    }
}
}
```

The actual command-line output from nanoSaxDemo is as follows:

```
Start parsing document >>>>
<mail>
<From>
  Contents:      &from;
END <From>
<To>
  Contents:      &to;
END <To>
<Cc>
  Contents:      &cc;
END <Cc>
<Date>
  Contents:      Fri, 12 Jan 2001 10:21:56 -0600
END <Date>
<Subject>
  Contents:      XML Parsers for J2ME MIDP
END <Subject>
<Body>
  Attribute:     LANGUAGE:english
  Contents:
    Ælfred, NanoXML and TinyXML are the three small-foot-print
    XML parsers that are suitable for J2ME MIDP applications.
    The sizes of these parsers range from 10K to 30K, which fit
    with memory budget of MIDP devices.
  <Signature>
  -----
  Yu Feng
  &from;
  http://www.webyu.com
  </Signature>

END <Body>
END <mail>
Finish parsing document <<<<
```

You can find more information about the NanoXML parser at
<http://nanoxml.sourceforge.net/index.html>.

Ælfred Parser for J2ME MIDP

Author: David Megginson

Original Web URL: <http://www.microstar.com/aelfred.html>

Last package release date: July 2, 1998

Last release version: 1.2
License agreement: Free for both commercial and non-commercial use.
Please see the original package for the actual license agreement.

The *Ælfred* parser was written by David Megginson, who also maintains the SAX APIs. The parser is named for the Saxon king *Ælfred*. Here is an interesting quote from the author:

“*Ælfred* the Great was king of Wessex, and at least nominally of all England, at the time of his death in 899AD. *Ælfred* introduced a wide-spread literacy program in the hope that his people would learn to read English, at least, if Latin was too difficult for them. This *Ælfred* hopes to bring another sort of literacy to Java, using XML, at least, if full SGML is too difficult. The initial “AE” (“*Æ*” in ISO-8859-1) is also a reminder that XML is not limited to ASCII.”

Jun Fujisawa has ported *Ælfred* to Palm KVM (you can find more information at <http://fujisawa.org/palm/>). However, the source code of this port is not available to the public. The *Ælfred* parser used in this book is version modified by the authors. The two main modifications to the original source code simulate the unsupported J2SE classes and change J2SE networking to J2ME networking.

Ælfred is a fast, non-validating XML parser. It contains two sets of event-based interfaces: a proprietary event-based interface and a SAX 1.0-compliant interface. It supports DTD and entities and has the capability of handling international encoding.

For convenience, the class files for the proprietary event-based interface are included in `aelfred_event.jar`. The class files for the SAX interface are included in `aelfred_sax.jar`.

`aelfred_event.jar` contains the class files of the following Java programs:

- `com/microstar/xml/HandlerBase.java`: A convenience class that implements the `XmlHandler` interface. If users don't want to implement all the methods from `XmlHandler`, they can simply extend from this base class and only implement a subset of callback methods.
- `com/microstar/xml/XmlHandler.java`: An interface that defines the event-driven callback methods that must be implemented by applications that use the *Ælfred* parser. This interface is very similar to the `DocumentHandler` interface defined in the SAX 1.0 Java API.
- `com/microstar/xml/XmlException.java`: Defines the exception for errors generated by the parser.
- `com/microstar/xml/XmlParser.java`: Defines the XML parser that parses XML documents and generates parsing events through callbacks.

`aelfred_sax.jar` contains all the class files in `aelfred_event.jar` and the class file of the SAX adapter for the *Ælfred* parser, `com/microstar/xml/SAXDriver.java`. `SAXDriver.java` implements the SAX 1.0 Java interfaces.

An Example of Using Ælfred's Proprietary Event-based Interface

To use Ælfred's proprietary event-based interface, `aelfred_event.jar` must be included in your Java class path.

The sample programs in Listings 10.9 and 10.10 show how to use Ælfred's proprietary event-based interface, which can be roughly described as a simplified SAX interface. The example consists of two files: `DemoEventHandler.java` and `aelfredEventDemo.java`.

`DemoEventHandler.java` implements all the callback methods defined in Ælfred's `XMLHandler` interface. These callback methods are very similar to SAX's callback methods, a lot of them even share the same method names. `aelfredEventDemo.java` is a MIDlet application that calls the Ælfred parser. The XML data source is shown in Listing 10.1 and Listing 10.2.

LISTING 10.9 `DemoEventHandler.java`

```
import java.io.InputStream;
import java.io.Reader;
import javax.microedition.io.*;

import com.microstar.xml.XmlHandler;
import com.microstar.xml.XmlParser;

public class DemoEventHandler implements XmlHandler {

    public XmlParser parser;
    private StringBuffer resultStringBuffer;

    public String getResultString() {
        return resultStringBuffer.toString();
    }

    public DemoEventHandler() {
        resultStringBuffer = new StringBuffer();
    }

    public Object resolveEntity (
        String publicId, String systemId)
    {
        System.out.println("Resolving entity: pubid="
            + publicId + ", sysid=" + systemId);
        resultStringBuffer.append("Resolving entity: pubid="
            + publicId + ", sysid=" + systemId + "\n");
    }
}
```

LISTING 10.9 Continued

```
        return null;
    }

    public void startExternalEntity (String systemId)
    {
        System.out.println(
            "Starting external entity: " + systemId);
        resultStringBuffer.append(
            "Starting external entity: " + systemId + "\n");
    }

    public void endExternalEntity (String systemId)
    {
        System.out.println(
            "Ending external entity: " + systemId);
        resultStringBuffer.append(
            "Ending external entity: " + systemId + "\n");
    }

    public void startDocument ()
    {
        System.out.println("Start parsing document >>>>");
        resultStringBuffer.append("Start parsing document >>>>\n");
    }

    public void endDocument ()
    {
        System.out.println("Finish parsing document <<<<");
        resultStringBuffer.append("Finish parsing document <<<<\n");
    }

    public void doctypeDecl (String name,
        String pubid, String sysid)
    {
        System.out.println("Doctype declaration: " + name
            + ", pubid=" + pubid + ", sysid=" + sysid);
        resultStringBuffer.append("Doctype declaration: "
            + name + ", pubid=" + pubid + ", sysid=" + sysid + "\n");
    }

    public void attribute (String name, String value,
        boolean isSpecified)
    {
        System.out.print(" Attribute:\t" + name + ":" + value);
        resultStringBuffer.append(
```

LISTING 10.9 Continued

```
        " Attribute:\t" + name + ":" + value);
    }

    public void startElement (String name)
    {
        System.out.println(" <" + name + ">\n");
        resultStringBuffer.append(" <" + name + ">\n");
    }

    public void endElement (String name)
    {
        System.out.println(" END <" + name + ">");
        resultStringBuffer.append(" END <" + name + ">\n");
    }

    public void charData (char ch[], int start, int length)
    {
        String contents = new String(ch, start, length);
        System.out.println(" Contents:\t" + contents);
        resultStringBuffer.append(" contents:" + contents + "\n");
    }

    public void ignorableWhitespace (char ch[],
    int start, int length)
    {
    }

    public void processingInstruction (String target,
    String data)
    {
    }

    public void error (String message,
    String url, int line, int column)
    {
        System.out.println("FATAL ERROR: " + message);
        System.out.println(" at " + url.toString() + ": line "
        + line + " column " + column);
        throw new Error(message);
    }

    void doParse (String url)
    throws java.lang.Exception
    {
```

LISTING 10.9 Continued

```
        parser = new XmlParser();
        parser.setHandler(this);
        parser.parse(url, null, (String)null);
    }

    String escape (char ch[], int length)
    {
        StringBuffer out = new StringBuffer();
        for (int i = 0; i < length; i++) {
            switch (ch[i]) {
                case '\\':
                    out.append("\\\\");
                    break;
                case '\n':
                    out.append("\\n");
                    break;
                case '\t':
                    out.append("\\t");
                    break;
                case '\r':
                    out.append("\\r");
                    break;
                case '\f':
                    out.append("\\f");
                    break;
                default:
                    out.append(ch[i]);
                    break;
            }
        }
        return out.toString();
    }
}
```

LISTING 10.10 aelfredEventDemo.java

```
import java.io.*;
import java.lang.String;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
```

LISTING 10.10 Continued

```
import javax.microedition.midlet.*;
import com.microstar.xml.*;

public class aelfredEventDemo extends MIDlet implements CommandListener {

    private String url;

    // GUI component for user to enter url for the xml document
    private Display myDisplay = null;
    private Form mainScreen;
    private TextField requestField;

    // GUI component for displaying xml data content
    private Form resultScreen;
    private StringItem resultField;

    // the "send" button used on mainScreen
    Command sendCommand = new Command("SEND", Command.OK, 1);
    // the "back" button used on resultScreen
    Command backCommand = new Command("BACK", Command.OK, 1);

    public aelfredEventDemo() {
        url = "http://www.webyu.com/book/mail.xml";

        // initializing the GUI components for entering url for the xml document
        myDisplay = Display.getDisplay(this);
        mainScreen = new Form("Type in a URL:");
        requestField =
            new TextField(null, url,
                100, TextField.URL);
        mainScreen.append(requestField);
        mainScreen.addCommand(sendCommand);
        mainScreen.setCommandListener(this);
    }

    public void startApp() throws MIDletStateChangeException {

        myDisplay.setCurrent(mainScreen);
    }

    public void pauseApp() {
    }
}
```

LISTING 10.10 Continued

```
public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {

    // when user clicks on "send" button on mainScreen
    if (c == sendCommand) {

        // retrieving the web url that user entered
        String urlString = requestField.getString();

        String resultString = "";

        try {
            XmlParser parser = new XmlParser();
            DemoEventHandler myHandler = new DemoEventHandler();
            parser.setHandler(myHandler);
            parser.parse(urlString, (String) null, (String) null);
            resultString = myHandler.getResultString();
        } catch (Exception e) {
            System.err.println("startApp: " + e);
        }

        // displaying the page content retrieved from web server
        resultScreen = new Form("XML Result:");
        resultField =
            new StringItem(null, resultString);
        resultScreen.append(resultField);
        resultScreen.addCommand(backCommand);
        resultScreen.setCommandListener(this);
        myDisplay.setCurrent(resultScreen);

    } else if (c == backCommand) {
        // do it all over again
        requestField.setString(url);
        myDisplay.setCurrent(mainScreen);
    }
}
}
```

The output of `aelfredEventDemo` is as follows:

```
Start parsing document >>>>
Resolving entity: pubid=null, sysid=http://www.webyu.com/book/mail.xml
Starting external entity: http://www.webyu.com/book/mail.xml
```

```
Resolving entity: pubid=null, sysid=http://www.webyu.com/book/mail.dtd
Starting external entity: http://www.webyu.com/book/mail.dtd
Ending external entity: http://www.webyu.com/book/mail.dtd
Doctype declaration: mail, pubid=null,
sysid=http://www.webyu.com/book/mail.dtd
<mail>
  <From>
    Contents:          yfeng@webyu.com
  END <From>
  <To>
    Contents:          somebody@somewhere.com
  END <To>
  <Cc>
    Contents:          jzhu@webyu.com
  END <Cc>
  <Date>
    Contents:          Fri, 12 Jan 2001 10:21:56 -0600
  END <Date>
  <Subject>
    Contents:          XML Parsers for J2ME MIDP
  END <Subject>
  Attribute:          language:english <Body>
  Contents:
    Ælfred, NanoXML and TinyXML are the three small-foot-print
    XML parsers that are suitable for J2ME MIDP applications.
    The sizes of these parsers range from 10K to 30K, which fit
    with memory budget of MIDP devices.
  <Signature>
    Contents:
      -----
      Yu Feng
      yfeng@webyu.com
      http://www.webyu.com
    END <Signature>
  Contents:
  END <Body>
END <mail>
Ending external entity: http://www.webyu.com/book/mail.xml
Finish parsing document <<<<
```

Similar to the output generated by TinyXML, the DTD and entities are parsed and used correctly in this sample.

An Example of Using Ælfred's SAX Interface

To use Ælfred's SAX interface, both `ælfred_sax.jar` and `sax10_midp.jar` must be included in your Java class path.

The sample program contains two Java files: `DemoHandler.java` and `aelfredSaxDemo.java`. `DemoHandler.java` is the same as Listing 10.7, and `aelfredSaxDemo.java` is almost identical to Listing 10.8 (`nanoSaxDemo.java`). The only difference between the two programs is that `nanoSaxDemo.java` (Listing 10.8) uses

```
Parser parser = ParserFactory.makeParser("nanoxml.sax.SAXParser");
```

and `aelfredSaxDemo.java` uses

```
Parser parser = ParserFactory.makeParser("com.microstar.xml.SAXDriver");
```

You should have figured out by now why this happens. It shows the benefit of using the standard SAX interface. The SAX API acts as a dashboard for XML parsers: The parsers from different vendors can be easily swapped in and out from an application without extra coding.

Here is the output from the `aelfredSaxDemo`:

```
Start parsing document >>>>
```

```
<mail>
<From>
  Contents:      yfeng@webyu.com
END <From>
<To>
  Contents:      somebody@somewhere.com
END <To>
<Cc>
  Contents:      jzhu@webyu.com
END <Cc>
<Date>
  Contents:      Fri, 12 Jan 2001 10:21:56 -0600
END <Date>
<Subject>
  Contents:      XML Parsers for J2ME MIDP
END <Subject>
<Body>
  Attribute:    language:english
  Contents:
    Ælfred, NanoXML and TinyXML are the three small-foot-print
    XML parsers that are suitable for J2ME MIDP applications.
    The sizes of these parsers range from 10K to 30K, which fit
    with memory budget of MIDP devices.

<Signature>
  Contents:
  -----
  Yu Feng
  yfeng@webyu.com
  http://www.webyu.com
```

PART II

```
END <Signature>
  Contents:

END <Body>
END <mail>
endDoc
Finish parsing document <<<<
```

You can find more information about the Ælfred parser at <http://www.microstar.com/aelfred.html>.

NOTE

One thing we didn't show in this chapter is the actual CPU and memory consumption of the three parsers we've discussed. You should be able to run the sample programs against your own XML data to get a performance benchmark and decide for yourself which parser is more suitable for your application. Roughly speaking, the NanoXML and TinyXML parsers consume half amount of the memory that Ælfred parser consumes, and the TinyXML parser runs a little faster than the NanoXML and Ælfred parsers against our sample XML documents.

Summary

This chapter discussed three small-footprint XML parsers. TinyXML is a fairly small and fast parser with an event-based interface at 10KB and a tree-based interface at 14KB. Because it is licensed under GPL, certain restrictions apply for both commercial and non-commercial uses. You should consult with the license agreement before packaging and distributing it with your applications.

NanoXML has a simple tree-based interface that takes up only 9KB. This interface is simple and easy to use, but it may not be suitable for parsing large XML files in the memory-constrained environment. Some of the features, such as DTD support and entities, are not supported by the NanoXML parser.

The event-based interfaces make Ælfred a fast XML parser. It supports a more complete set of XML features than the other two parsers, but this parser is the biggest in size. The code size of the proprietary event-based interface is about 18KB, and the size of the SAX compliant interface is about 30KB including the SAX 1.0 Java API.

In addition to the three XML parsers shown in this chapter, several other XML parsers (such as the kXML parser) might be good candidates for J2ME MIDP applications. You can find more information about kXML at <http://www.kxml.de>.