Simple API for XML

SAX

This API was developed originally as a set of Java interfaces and classes, although working versions exist in several other programming languages.

The development went through several stages, and that fact accounts for the two stages used when creating a parser.

```
SAXParserFactory factory =
                    SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
XMLReader xmlReader = saxParser.getXMLReader();
```

It is the XMLReader (actually an interface) object that has the *parse* method.

The *parse* methods take an InputSource as it parameter or a String representing a URI.

Event-based Parsing

Unlike a DOM parser, a SAX parser creates no parse tree.

A SAX parser can be viewed as a scanner that reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document.

These tokens are processed in the same order that they appear in the document.

A SAX parser interacts with an application program by reporting to the application the nature of the tokens that the parser has encountered as they occur.

The application program provides an "event" handler that must be registered with the parser.

As the pieces of the XML document are identified, callback methods in the handler are invoked with the relevant information.

ContentHandler Interface

This interface specifies the callback methods that the SAX parser uses to notify an application program of the components of the XML document that it has seen.

The ContentHandler interface has eleven methods.

void startDocument()

Called at the beginning of a document.

void endDocument()

Called at the end of a document.

void startElement(String uri, String localName,
String qName, Attributes atts)

Called at the beginning of an element.

void endElement(String uri, String localName,
String qName)

Called at the end of an element.

void characters(**char** [] ch, **int** start, **int** length)

Called when character data is encountered.

void ignorableWhitespace(**char** [] ch, **int** start, **int** length)

Called when a DTD is present and ignorable whitespace is encountered.

void processingInstruction(String target, String data)

Called when a processing instruction is recognized.

void setDocumentLocator(Locator locator)

Provides a Locator object that can be used to identify positions in the document.

void skippedEntity(String name)

Called when an unresolved entity is encountered.

void startPrefixMapping(String prefix, String uri)

Called when a new namespace mapping is defined.

void endPrefixMapping(String prefix)

Called when a namespace definition ends its scope.

Attributes Interface

This interface specifies methods for processing the attributes connected to an element.

```
int getLength()                    // number of attributes

String getQName(int index)

String getValue(int index)

String getValue(String qname)
```

Example: SAX Parser Reports

In this program we provide a ContentHandler that reports the elements, text content, and processing instructions found in an XML document.

File: ReportHandler.java

```
import org.xml.sax.ContentHandler;
import org.xml.sax.Attributes;
import org.xml.sax.Locator;

class ReportHandler implements ContentHandler
{
   public void startDocument()
   {
      System.out.println("Starting the document.");
   }

   public void endDocument()
   {
      System.out.println("Ending the document.");
   }

   public void startElement(String uri, String localName,
                            String qualName, Attributes attribs)
   {
      System.out.println("Start tag: " + qualName);
      for (int k=0; k<attribs.getLength(); k++)
      {
         System.out.print("Attribute: ");
         System.out.print(attribs.getQName(k));
         System.out.print(" = ");
         System.out.println(attribs.getValue(k));
      }
   }
```

```
public void endElement(String uri, String localName,
                                              String qualName)
{
   System.out.println("End tag: " + qualName);
}

public void characters(char [] chars, int start, int length)
{
   System.out.print("Characters: ");
   showString(chars, start, length);
}

public void ignorableWhitespace(char [] chars,
                                          int start, int length)
{
   System.out.print("Ignorable whitespace: ");
   showString(chars, start, length);
}

public void processingInstruction(String target,
                                          String data)
{
   System.out.println("Processing instruction");
   System.out.println("\tTarget: " + target);
   System.out.println("\tData  : " + data);
}

public void setDocumentLocator(Locator loc)
{}

public void skippedEntity(String name)
{}

public void startPrefixMapping(String prefix, String uri)
{}

public void endPrefixMapping(String prefix)
{}
```

```
    private void showString(char [] chars, int start, int leng)
    {
        String string = new String(chars, start, leng);
        string = string.replaceAll(" ", "+");
        string = string.replaceAll("\n", "[nl]");
        string = string.replaceAll("\t", "[tab]");
        System.out.println(string);
    }
}
```

In the class with the main method that calls the SAX parser, we create a ReportHandler object and register it with the XMLReader object.

Additionally, we register an instance of the MyErrorHandler class that can be found in the DOM chapter. This object ensures that parser errors are handled gracefully.

File: SaxParse.java

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import java.io.*;

public class SaxParse
{
    static public void main(String [] args)
    {
        if (args.length != 1)
        {
            System.out.println("Usage: java SaxParse xmlFile");
            return;
        }
        SAXParserFactory factory =
                            SAXParserFactory.newInstance();
```

```
        try
        {
           SAXParser saxParser = factory.newSAXParser();
           XMLReader xmlReader =
                                   saxParser.getXMLReader();
           xmlReader.setContentHandler(new ReportHandler())
           xmlReader.setErrorHandler(new MyErrorHandler());
           xmlReader.parse(args[0]);
        }
        catch (ParserConfigurationException e)
        { System.out.println("Parser configuration error"); }
        catch (SAXException e)
        { System.out.println("Parsing error."); }
        catch (IOException e)
        { System.out.println("IO error."); }
     }
  }
```

Executing SaxParse

To test the SAX parser we use XML documents similar to the ones used for the DOM parser.

In the first example, the XML document has no DOCTYPE (no DTD), so the entity references have been removed along with the CDATA section.

File: rt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- rt.xml -->
<?SaxParse usage="java SaxParser root.xml"?>
<root>
  <child position="first">
     <name>Eileen Dover</name>
  </child>
```

```
    <child position="second">
      <name>Amanda Reckonwith</name>
    </child>
  </root>
```

% **java SaxParse rt.xml**

```
Starting the document.
Processing instruction
    Target: SaxParse
    Data  : usage="java SaxParser root.xml"
Start tag: root
Characters: [nl]+++
Start tag: child
Attribute: position = first
Characters: [nl]+++++++
Start tag: name
Characters: Eileen+Dover
End tag: name
Characters: [nl]+++
End tag: child
Characters: [nl]+++
Start tag: child
Attribute: position = second
Characters: [nl]+++++++
Start tag: name
Characters: Amanda+Reckonwith
End tag: name
Characters: [nl]+++
End tag: child
Characters: [nl]
End tag: root
Ending the document.
```

Note that all white space in the XML document is handled by the *characters* method. Without a DTD, the parser has no way to interpret certain characters as ignorable white space.

In the second example we have an XML document with an external DTD specification that defines two entities.

File: root.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "root.dtd">
<!-- root.xml -->
<?SaxParse usage="java SaxParser root.xml"?>
<root>
  <child position="first">
     <name>Eileen &last1;</name>
  </child>
  <child position="second">
     <name><![CDATA[<<<Amanda>>>]]> &last2;</name>
  </child>
</root>
```

File: root.dtd

```
<!ELEMENT root (child*)>
<!ELEMENT child (name)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST child position NMTOKEN #REQUIRED>
<!ENTITY last1 "Dover">
<!ENTITY last2 "Reckonwith">
```

% **java SaxParse root.xml**

```
Starting the document.
Processing instruction
    Target: SaxParse
    Data  : usage="java SaxParser root.xml"
Start tag: root
Ignorable whitespace: [nl]+++
Start tag: child
Attribute: position = first
Ignorable whitespace: [nl]+++++++
Start tag: name
```

```
Characters: Eileen+
Characters: Dover
End tag: name
Ignorable whitespace: [nl]+++
End tag: child
Ignorable whitespace: [nl]+++
Start tag: child
Attribute: position = second
Ignorable whitespace: [nl]+++++++
Start tag: name
Characters: <<<Amanda>>>
Characters: +
Characters: Reckonwith
End tag: name
Ignorable whitespace: [nl]+++
End tag: child
Ignorable whitespace: [nl]
End tag: root
Ending the document.
```

Observations

- Because of the DTD, the parser can recognize and report when characters are ignorable white space.
- Both entity references have been resolved by the parser.
- The CDATA section has been integrated into the XML document.
- The *characters* method has processed some of the text content in pieces.
- A SAX parser ignores comments completely.

DefaultHandler

As a convenience the *org.xml.sax.helpers* package has a class DefaultHandler that implements the ContentHandler interface with empty methods.

We can alter the ReportHandler class to start with

```
import org.xml.sax.helpers.DefaultHandler;
class ReportHandler extends DefaultHandler
{
```

Then we can omit the methods *setDocumentLocator*, *skippedEntity*, *startPrefixMapping*, and *endPrefixMapping* that had empty bodies in ReportHandler.java.

The DefaultHandler class also implements the ErrorHandler interface, the DTDHandler interface, and the EntityResolver interface.

When to Use SAX

The SAX parser works very differently from the DOM parser.

Situations where SAX works well:

- You can process the XML document in a linear fashion from the top down.
- The document is not deeply nested.
- You are processing a very large XML document whose DOM tree would consume too much memory. Typical DOM implementations use ten bytes of memory to represent one byte of XML
- The problem to be solved involves only part of the XML document.
- Data is available as soon as it is seen by the parser, so SAX works well for an XML document that arrives over a stream.

Disadvantages of SAX

- We have no random access to an XML document since it is processed in a forward-only manner.
- If you need to keep track of data the parser has seen or change the order of items, you must write the code and store the data on your own.

Problem Solving with SAX

To illustrate text processing using SAX we want to take the *phoneA.xml* document and extract certain information from it to produce a text file showing the names, phone numbers, and cities in the XML document.

As a reminder, here is part of the document.

```
<?xml version="1.0">
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
   <entry>
     <name>
       <first>Rusty</first>
       <last>Nail</last>
     </name>
     <phone>335-0055</phone>
     <city>Iowa City</city>
   </entry>
   <entry>
     <name gender="female">
       <first>Pearl</first>
       <middle>E.</middle>
       <last>Gates</last>
     </name>
     <phone>335-4582</phone>
     <city>North Liberty</city>
   </entry>
```

```
        :
</phoneNumbers>
```

In the first part of the solution, the external DTD specification is missing.

Here is the text file we want to create.

Name	Phone	City
Rusty Nail	335-0055	Iowa City
Mr. Justin Case	354-9876	Coralville
Ms. Pearl E. Gates	335-4582	North Liberty
Ms. Helen Back	337-5967	

SAX Driver

This program follows the format that we used before except that we use an InputSource as the parameter to the *parse* method.

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import java.io.*;

public class SaxPhone
{
   static public void main(String [] arg)
   {
      String inName = null, outName = null;
      if (arg.length == 2)
      {
         inName = arg[0];   outName = arg[1];
      }
```

```
        else
        {  System.out.println(
                        "Usage: SaxPhone <infile> <outfile>");
           return;
        }
        try
        {  SAXParserFactory spf =
                           SAXParserFactory.newInstance();
           SAXParser parser = spf.newSAXParser();
           XMLReader reader = parser.getXMLReader();
           reader.setErrorHandler(new MyErrorHandler());
           PrintWriter out =
                        new PrintWriter(new FileWriter(outName));
           reader.setContentHandler(
                                       new PhoneHandler(out));
           reader.parse(new InputSource(
                                       new FileReader(inName)));
           out.close();
        }
        catch (ParserConfigurationException e)
        { System.out.println("Parser configuration error"); }
        catch (SAXException e)
        { System.out.println("Parsing error."); }
        catch (IOException e)
        { System.out.println("IO error."); }
     }
  }
```

The actual text processing will be found in the ContentHandler class.

Remember we get only one pass through the XML document.

To extract the appropriate text content, we need to know where the parser is as it reads the document.

The idea is to set a boolean flag when we enter a particular element and turn it off when we leave the element.

If we encounter textual information, using the *characters* method, when a certain element flag is set, we retrieve that information and store it.

To make the organization of data simpler, we will use variations of the Entry and Name classes from the DOM chapter.

Entry Class

Since we will be setting properties (instance variables) in this class at different points in our handler, we will need mutator methods for that purpose.

```
class Entry
{
    private Name name;
    private String gender, phone, city;

    Entry() { }     // other constructors have been removed

    Name getName() {  return name;  }

    void setName(Name n) {  name = n;  }

    String getGender() {  return gender;  }

    void setGender(String g) {  gender = g;  }

    String getPhone() {  return phone;  }

    void setPhone(String p) {  phone = p;  }

    String getCity()
    {   if (city==null) return "";
        else return city;
    }
```

```
    void setCity(String c) {  city = c;  }

    public String toString()        // this method has changed
    {
       String gen = "";
       if (gender != null && gender.length() > 0)
          gen = "gender = " + gender + "\n";
       if (city != null && city.length()>0) city = city + "\n";
       else city= "";
       return name + "\n" + gen + phone + "\n" + city;
    }
}
```

Name Class

The original constructors in both classes have been removed since we no longer need them.

The no-parameter constructors are written explicitly, although this step is redundant now.

```
class Name
{
   private String first, middle, last;

   Name() { }    // other constructors have been removed

   String getFirst() {  return first;  }
   void setFirst(String f) {  first = f;  }

   String getMiddle() {  return middle;  }
   void setMiddle(String m) {  middle = m;  }

   String getLast() {  return last;  }
   void setLast(String lt) {  last = lt;  }
```

```
    public String toString()
    {
        if (middle == null || middle.equals(""))
            return first + " " + last;
        else
            return first + " " + middle + " " + last;
    }
}
```

Phone Handler

The ContentHander has boolean instance variables for each of the elements that we are interested in.

It also has instance variables for the data that we will collect, namely an Entry object, a Name object, and a List object to hold the Entry objects.

Finally we need a PrintWriter object for writing to the output file.

Observe that the program uses the generic list structures provided in Java 1.5, along with the new version of the **for** command.

Formatting the output into columns for the resulting file is performed by a method called *format*.

```
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
import java.util.*;
import java.io.*;

public class PhoneHandler extends DefaultHandler
{
    private boolean inFirst = false, inMiddle = false,
                    inLast = false,  inPhone = false,  inCity = false;
    private Entry entry;
    private Name name;
```

```
private List<Entry> entryList;
private PrintWriter pw;

PhoneHandler(PrintWriter p)
{
   pw = p;
}

public void startDocument()
{
   entryList = new ArrayList<Entry>();
   format(" Name", " Phone", " City");
   format(" ----", " -----", " ------");
}

public void endDocument()     // entry list is complete
{
   for (Entry e : entryList)         // the new for command
   {
      String title = "";
      if ("male".equals(e.getGender())) title = "Mr. ";
      if ("female".equals(e.getGender())) title = "Ms. ";
      format(title+e.getName(), e.getPhone(), e.getCity());
   }     // note order of arguments to equals
}        // e.getGender() may be null

public void startElement(String namespaceURI,
         String localName, String qName, Attributes atts)
{
   if (qName.equals("entry"))
      entry = new Entry();
   else if (qName.equals("name"))
   {
      name = new Name();
      entry.setGender(atts.getValue("gender"));
   }                                    // gender may be null
```

```
    else if (qName.equals("first"))  inFirst = true;
    else if (qName.equals("middle"))  inMiddle = true;
    else if (qName.equals("last"))  inLast = true;
    else if (qName.equals("phone"))  inPhone = true;
    else if (qName.equals("city"))  inCity = true;
  }

  public void endElement(String namespaceURI,
                         String localName, String qName)
  {
    if (qName.equals("entry"))
      entryList.add(entry);
    else if (qName.equals("name"))
      entry.setName(name);
    else if (qName.equals("first"))  inFirst = false;
    else if (qName.equals("middle"))  inMiddle = false;
    else if (qName.equals("last"))  inLast = false;
    else if (qName.equals("phone"))  inPhone = false;
    else if (qName.equals("city"))  inCity = false;
  }

  public void characters(char [] ch, int start, int length)
  {
    String str = new String(ch, start, length);
    if (inFirst)  name.setFirst(str);
    else if (inMiddle)  name.setMiddle(str);
    else if (inLast)  name.setLast(str);
    else if (inPhone)  entry.setPhone(str);
    else if (inCity)  entry.setCity(str);
  }
```

```
    private void format(String c1, String c2, String c3)
    {
       String line = c1+ sp.substring(0, 25-c1.length());
       line = line + c2 + sp.substring(0, 15-c2.length());
       pw.println(line + c3);
    }

    private static String sp = "                    ";
}
```

Execution

```
% java SaxPhone phoneA.xml phone.out
```

The text file *phone.out* has the formatted list of names, phone numbers, and cities that we desire.

To make sense of PhoneHandler, trace the code for each of the elements that will be encountered as the XML document is parsed.

Trace the elements *entry*, *name*, *phone*, *city*, *first*, *middle*, and *last*.

A Problem

We have already noticed that the *characters* method may not collect all of the text data in an element's content in a single invocation.

To see what this possibility can do to the PhoneHandler class, consider this XML document with a CDATA section and an entity reference.

File: pnums.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM "pnums.dtd">
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
   <entry>
     <name>
       <first>Rusty</first>
       <last>Nail</last>
     </name>
     <phone>335-0055</phone>
     <city>Iowa &city;</city>
   </entry>
   <entry>
     <name gender="male">
       <first>Justin</first>
       <last>Case</last>
     </name>
     <phone>354-9876</phone>
     <city>Coralville</city>
   </entry>
   <entry>
     <name gender="female">
       <first>Pearl</first>
       <middle>E.</middle>
       <last>Gates</last>
     </name>
     <phone>335-4582</phone>
     <city>North <![CDATA[<<Liberty>>]]></city>
   </entry>
   <entry>
     <name gender="female">
       <first>Helen</first>
       <last>Back</last>
     </name>
     <phone>337-5967</phone>
   </entry>
  </entries>
</phoneNumbers>
```

File: pnums.dtd

```
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
<!ELEMENT entry (name, phone, city?)>
<!ELEMENT name (first, middle?, last)>
<!ATTLIST name gender (female | male) #IMPLIED>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ENTITY city "City">
```

Execution of SaxPhone

% **java SaxPhone pnums.xml pnums.out**

File: pnums.out

Name	Phone	City
Rusty Nail	335-0055	City
Mr. Justin Case	354-9876	Coralville
Ms. Pearl E. Gates	335-4582	<<Liberty>>
Ms. Helen Back	337-5967	

A Problem

When the CDATA section and the entity reference are encountered by the SAX parser, it makes two separate calls of the method *characters* with the separate pieces of data, "Iowa " and "City" in the first case and "North " and "<<Liberty>>" in the second.

```
public void characters(char [] ch, int start, int length)
{
   String str = new String(ch, start, length);
   if (inFirst)  name.setFirst(str);
   else if (inMiddle)  name.setMiddle(str);
   else if (inLast)  name.setLast(str);
   else if (inPhone)  entry.setPhone(str);
   else if (inCity)  entry.setCity(str);
}
```

Only the second strings "City" and "<<Liberty>>" are finally stored in the *city* field of the Entry object.

Solution

First we add an instance variable to the class to collect the text content produced by several call to *characters* within the same element.

```
private String content;
```

We need to change three methods in the new class PHandler: *startElement*, *endElement*, and *characters*.

When an element is first encountered, initialize the *content* variable to an empty string.

```
public void startElement(String namespaceURI,
      String localName, String qName, Attributes atts)
{
   content = "";
      :
      :     // rest of the method is the same
}
```

When we are done processing an element, the variable *content* should contain all of the text in the element's content. Set the corresponding field at this time.

```
public void endElement(String namespaceURI,
                       String localName, String qName)
{
   if (qName.equals("entry"))
      entryList.add(entry);
   else if (qName.equals("name"))
      entry.setName(name);
   else if (qName.equals("first"))
   {
      name.setFirst(content); inFirst = false;
   }
   else if (qName.equals("middle"))
   {
      name.setMiddle(content); inMiddle = false;
   }
   else if (qName.equals("last"))
   {
      name.setLast(content); inLast = false;
   }
   else if (qName.equals("phone"))
   {
      entry.setPhone(content); inPhone = false;
   }
   else if (qName.equals("city"))
   {
      entry.setCity(content); inCity = false;
   }
}
```

Finally, in the characters method we collect all of the strings that make up the content of a particular element.

```
public void characters(char [] ch, int start, int length)
{
   content = content + new String(ch, start, length);
}
```

Execution

% **java SPhone pnums.xml pnums.out**

File: pnums.out

Name	Phone	City
Rusty Nail	335-0055	Iowa City
Mr. Justin Case	354-9876	Coralville
Ms. Pearl E. Gates	335-4582	North <<Liberty>>
Ms. Helen Back	337-5967	

Another Problem

Many times the text content of an element is provided on more than one line in an XML document.

For example, alter the two *city* elements in *phoneA.xml* that contain city names made from two words.

```
   :
<city>Iowa
        City</city>
   :
<city>North
        Liberty</city>
   :
```

The revised version of PhoneHandler.java collects both words in the city name, but it also retains the whitespace between the two words.

% **java SPhone ph.xml ph.out**

File: ph.out

Name	Phone	City
Rusty Nail	335-0055	Iowa City
Mr. Justin Case	354-9876	Coralville
Ms. Pearl E. Gates	335-4582	North Liberty
Ms. Helen Back	337-5967	

Solution

We can solve this problem by using the *trim* method for String objects in the *characters* method to remove whitespace that occurs between the pieces of text that are collected during the multiple calls of the method.

We need to enter a space since the two lines of text need some kind of separater.

```
public void characters(char [] ch, int start, int length)
{
   String s = new String(ch, start, length).trim() + " ";
   content = content + s;
}
```

This solutions is somewhat *ad hoc* in the sense that it may not fit all situations.

Mixed Content

Consider the difficulty of processing mixed content in this manner.

One global instance variable, *content*, will no longer work with mixed content.

<tag1>Some text<tag2>Inner text</tag2>Some more</tag1>

Need two boolean variables.

inTag1 && !inTag2 ⇒ "Some text" and "Some more"

inTag1&& inTag2 ⇒ "Inner text"

String Concatenation

If the *characters* method is called many times to concatenate the text that makes up the content of elements, the String operation may be too inefficient.

Alternative: Use a StringBuffer with the *append* method.

Make the following changes:

1. Instance variable

```
private StringBuffer content;
```

2. Inside *startElement*

```
content = new StringBuffer();
```

3. Inside *endElement*

```
name.setFirst(content.toString());
name.setMiddle(content.toString());
name.setLast(content.toString());
entry.setPhone(content.toString());
entry.setCity(content.toString());
```

4. Body of *characers*

```
content.append(ch, start, length);
```

Alternate Formatting

Java 1.5 includes a method *printf* with formatting strings similar to those in C.

```
private void format(String c1, String c2, String c3)
{
   pw.printf("%-25s%-15s%-20s\n", c1, c2, c3);
}
```

Formatting Conventions

%-25s	Print first string left-justified in a field of width 25
%-15s	Print second string left-justified in a field of width 15
%-20s	Print third string left-justified in a field of width 20

Namespaces and SAX

The following XML document is intended as a vehicle for investigating the handling of namespaces with SAX.

File: ns.xml

```
<?xml version="1.0"?>
<!DOCTYPE ph:phoneNumbers SYSTEM "ns.dtd">
<ph:phoneNumbers
                xmlns:ph="http://slonnegr.cs.uiowa.edu/phone">
  <title>Phone Numbers</title>
  <ph:entries>
    <en:entry xmlns:en="http://slonnegr.cs.uiowa.edu/entry">
      <ph:name>Rusty Nail</ph:name>
       <en:phone>335-0055</en:phone>
      <city>Iowa City</city>
    </en:entry>
  </ph:entries>
</ph:phoneNumbers>
```

Identifying Namespaces

This first handler recognizes the scope of namespace declarations and shows the names passed to the *startElement* method.

File: NHandler.java

```
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
import org.xml.sax.Locator;

public class NHandler extends DefaultHandler
{
   private Locator locator;

   public void startElement(String uri, String localName,
                            String qualName, Attributes atts)
   {
      System.out.println("Line: " + locator.getLineNumber());
      System.out.println("Qualified name: " + qualName);
      System.out.println("Namespace uri:  " + uri);
      System.out.println("Local name:     " + localName);
      System.out.println();

      for (int k=0; k<atts.getLength(); k++)
      {
         System.out.print("Attribute: " + atts.getQName(k));
         System.out.print(atts.getQName(k));
         System.out.print(" = " + atts.getValue(k));
      }
   }

   public void setDocumentLocator(Locator loc)
   {
      System.out.println("In setDocumentLocator");
      locator = loc;
   }

   public void startPrefixMapping(String prefix, String uri)
   {
      System.out.println("Start prefix: " + prefix + "=" + uri);
      System.out.println();
   }
```

```
    public void endPrefixMapping(String prefix)
    {
        System.out.println("End prefix: " + prefix);
        System.out.println();
    }
}
```

Execution

% **java NParse ns.xml**

```
In setDocumentLocator
Start prefix: ph=http://slonnegr.cs.uiowa.edu/phone

Line: 3
Qualified name: ph:phoneNumbers
Namespace uri: http://slonnegr.cs.uiowa.edu/phone
Local name: phoneNumbers

Line: 4
Qualified name: title
Namespace uri:
Local name: title

Line: 5
Qualified name: ph:entries
Namespace uri: http://slonnegr.cs.uiowa.edu/phone
Local name: entries

Start prefix: en=http://slonnegr.cs.uiowa.edu/entry

Line: 6
Qualified name: en:entry
Namespace uri: http://slonnegr.cs.uiowa.edu/entry
Local name: entry

Line: 7
Qualified name: ph:name
Namespace uri: http://slonnegr.cs.uiowa.edu/phone
Local name: name
```

Line: 8
Qualified name: en:phone
Namespace uri: http://slonnegr.cs.uiowa.edu/entry
Local name: phone

Line: 9
Qualified name: city
Namespace uri:
Local name: city

End prefix: en

End prefix: ph

Observations

- The *startPrefixMapping* method is called before the *startElement* method for the element containing the namespace attribute.
- For each element not in a namespace, the *uri* value is an empty string.
- When a *uri* value (a namespace) is absent, the local name may be undefined, so use the qualified name.
- This handler uses the Locator object to print line numbers for each element found.
- For reasons unknown to me, the attributes in the elements *phoneNumbers* and *entry* are not recognized by SAX even though they have been declared in a DTD.
- Local names are recognized in this example because the SAX factory has been made aware of namespaces.

```
factory.setNamespaceAware(true);
```

Processing Namespaces

By testing the *uri* value in *startElement*, we can determine whether the element is in a namespace.

Here is a short handler that illustrates how elements in namespaces can be recognized.

File: THandler.java

```
import org.xml.sax.helpers.DefaultHandler;

public class THandler extends DefaultHandler
{
   public void startElement(String uri, String locName,
                            String qName, Attributes attribs)
   {
      if ("".equals(uri))
         System.out.println("Element: " + qName);
      else
         System.out.println("Element: {" + uri + "} "
                                                + locName);
   }
}
```

Execution

% **java TParse ns.xml**

```
Element: {http://slonnegr.cs.uiowa.edu/phone} phoneNumbers
Element: title
Element: {http://slonnegr.cs.uiowa.edu/phone} entries
Element: {http://slonnegr.cs.uiowa.edu/entry} entry
Element: {http://slonnegr.cs.uiowa.edu/phone} name
Element: {http://slonnegr.cs.uiowa.edu/entry} phone
Element: city
```

Using a Stack to Process Elements

The nested structure of elements in an XML document embodies the basic form of a stack: the current end tag must match the start tag processed most recently but not yet matched.

As before we handle an element of complex type as an object that will contain the information inside of the element, and an element of simple type will be represented as a field of some primitive type or String (textual data).

Basic Strategy

In *startElement*

- When an element of complex type appears, push an object of its corresponding class onto the stack.
- When an element of simple type appears, push a StringBuffer to hold its textual content onto the stack and indicate that the *characters* method can start collecting character data.

In *endElement*

- When a tag for a complex element is encountered, pop the object and pass it on to its containing element, which is now on the top of the stack.
- When a tag for a simple element is encountered, pop the StringBuffer object, use that value to set the field in the object on the top of the stack, and tell the *characters* method to stop collecting text.

In *characters*

- Append the current array of characters onto the StringBuffer object that resides on the top of the stack.

The last object popped from the stack should contain a structure that contains all of the information gleaned from the XML document.

Stacks in Java

Java has a class in the package *java.util* whose objects implement a stack of objects.

Since Java 1.5 introduced generics, the stack can have a type parameter E representing the class whose objects will populate the stack.

To create a stack object we can write:

```
java.util.Stack<E> stack = new java.util.Stack<E>();
```

where E represents the type (class or interface) of the items on the stack.

We use the following instance methods from Stack:

E push(E item)

Pushes an item onto the top of this stack and returns that item.

E pop()

Removes the object at the top of this stack and returns that object as the value of this function.

E peek()

Looks at the object at the top of this stack without removing it from the stack.

XML Document to Process

The XML document on the next page describes a collection of books and magazines in a very small library.

The catalog consists of a list of books and magazines.

A magazine contains a list of articles.

File: library.xml

```
<?xml version="1.0"?>
<catalog library="XML Library">
  <book>
    <author>Luke Upp</author>
    <title>Oliver Twist Learns XML</title>
  </book>
  <book>
    <author>Cliff Hanger</author>
    <title>Romeo and Java</title>
  </book>
  <magazine>
    <name>XML Today</name>
    <article page="5">
      <headline>XML Can Be Your Friend</headline>
    </article>
    <article page="29">
      <headline>The XML Diet</headline>
    </article>
    <article page="59">
      <headline>SAX: The Inside Story</headline>
    </article>
  </magazine>
  <book>
    <author>Woody Glenn</author>
    <title>Tale of Two DTDs</title>
  </book>
  <magazine>
    <name>Readers XML Digest</name>
    <article page="17">
      <headline>Humor in XML</headline>
    </article>
    <article page="47">
      <headline>XML Condensed</headline>
    </article>
  </magazine>
  <book>
    <author>Lance Boyle</author>
    <title>War and Peace and XML</title>
  </book>
</catalog>
```

Java Classes for the Complex Elements

Some of the classes rely on the no-parameter constuctor that is supplied by the compiler automatically.

All collections have their components typed using the generics mechanism in Java 1.5.

Several of these classes require an import statement for the package *java.util*, which has been omitted to save space.

```
public class Catalog
{
   private List<Book> books;
   private List<Magazine> magazines;

   Catalog()
   {
      books = new ArrayList<Book>();
      magazines = new ArrayList<Magazine>();
   }

   void addBook(Book b)
   {  books.add(b);  }

   void addMagazine(Magazine m)
   {  magazines.add(m);  }

   public String toString()
   {
      StringBuffer buf = new StringBuffer(">>> Books <<<\n");
      for (int k=0; k<books.size(); k++)
         buf.append(books.get(k)).append('\n');
      buf.append(">>> Magazines <<<\n");
      for (int k=0; k<magazines.size(); k++)
         buf.append(magazines.get(k)).append('\n');
      return buf.toString();
   }
}
```

```
public class Book
{
   private String author;
   private String title;

   void setAuthor(String a)
   {  author = a;  }

   void setTitle(String t)
   {  title  = t;  }

   public String toString()
   {
      return "Book: \n   Author='" + author +
                     "' \n   Title='" + title + "'";
   }          // note the apostrophes surrounding
}             // the author and the title


public class Magazine
{
   private String name;
   private List<Article> articles;

   Magazine()
   {
      articles = new ArrayList<Article>();
   }

   void setName(String n)
   {  name = n;  }

   void addArticle(Article a)
   {  articles.add(a);  }
```

```
    public String toString()
    {
        StringBuffer buf = new StringBuffer("Magazine:");
        buf.append("\n   Name='" + name + "'");
        for (int k=0; k<articles.size(); k++)
            buf.append(articles.get(k));
        return buf.toString();
    }
}


public class Article
{
    private String headline, page;

    void setHeadline(String h)
    {  headline = h;  }

    void setPage(String p)
    {  page = p;  }

    public String toString()
    {
        return "\n   Article: Headline='" + headline +
                        "' on page='" + page + "'";
    }
}
```

Java code for the handler class appears on the next page.

It has three instance variables:

catalog	refers to the Catalog object that contains the information gleaned from the XML document.
stack	the Stack object used to process the elements of the document.
isReadyForText	signals the *characters* method to collect text.

File: SaxCatalog.java

```
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import java.util.*;

public class SaxCatalog extends DefaultHandler
{
   private Catalog catalog;

   private Stack<Object> stack = new Stack<Object>();

   private boolean isReadyForText = false;


   Catalog getCatalog()
   { return catalog; }


   public void startElement(String uri, String locName,
                            String qName, Attributes atts)
   {
      // If next element is complex, push a new instance
      // on the stack. If element has properties, set them
      // in the new instance.
      if (qName.equals("catalog"))
         stack.push(new Catalog());
      else if (qName.equals("book"))
         stack.push(new Book());
      else if (qName.equals("magazine"))
         stack.push(new Magazine());
```

```
    else if (qName.equals("article"))
    {
       stack.push(new Article());
       String page = atts.getValue("page");
       if (page==null) page = "unknown";
       ((Article)stack.peek()).setPage(page);
    }
    // If next element is simple, push an empty
    // StringBuffer. This makes the stack ready
    // to accept character text.
    else if (qName.equals("title") ||
             qName.equals("author") ||
             qName.equals("name") ||
             qName.equals("headline"))
    {
       stack.push(new StringBuffer());
       isReadyForText = true;
    }
}

public void endElement(String uri, String locName,
                                          String qName)
{
    // Recognized text is always content of an element.
    // When the element closes, no more text should
    // be expected.
    isReadyForText = false;

    // Pop stack and add to 'parent' element, which is
    // the next item on the stack.
    // Pop stack first, then peek at top element.
    Object obj = stack.pop();
```

```
        if (qName.equals("catalog"))
           catalog = (Catalog)obj;
        else if (qName.equals("book"))
            ((Catalog)stack.peek()).addBook((Book)obj);
        else if (qName.equals("magazine"))
            ((Catalog)stack.peek()).
                              addMagazine((Magazine)obj);
        else if (qName.equals("article"))
            ((Magazine)stack.peek()).addArticle((Article)obj);
        // For simple elements, pop StringBuffer and convert
        // to String.
        else if (qName.equals("title"))
            ((Book)stack.peek()).setTitle(obj.toString());
        else if (qName.equals("author"))
            ((Book)stack.peek()).setAuthor(obj.toString());
        else if (qName.equals("name"))
            ((Magazine)stack.peek()).setName(obj.toString());
        else if (qName.equals("headline"))
            ((Article)stack.peek()).setHeadline(obj.toString());
        // If none of the above, it is an unexpected element:
        // necessary to push popped element back.
        else stack.push(obj);
    }

    public void characters(char [] data, int start, int leng)
    {
        // If stack is ready, collect data for element.
        if (isReadyForText)
            ((StringBuffer)stack.peek()).
                                    append(data, start, leng);
    }
}
```

Execution

% **java SaxDriver library.xml**

```
>>> Books <<<
Book:
  Author='Luke Upp'
  Title='Oliver Twist Learns XML'
Book:
  Author='Cliff Hanger'
  Title='Romeo and Java'
Book:
  Author='Woody Glenn'
  Title='Tale of Two DTDs'
Book:
  Author='Lance Boyle'
  Title='War and Peace and XML'
>>> Magazines <<<
Magazine:
  Name='XML Today'
  Article: Headline='XML Can Be Your Friend' on page='5'
  Article: Headline='The XML Diet' on page='29'
  Article: Headline='SAX: The Inside Story' on page='59'
Magazine:
  Name='Readers XML Digest'
  Article: Headline='Humor in XML' on page='17'
  Article: Headline='XML Condensed' on page='47'
```

The next page has a trace of the stack as the XML document is parsed.

Each line on the right shows the classes of the objects on the stack at that point in the execution.

Stack Trace

```
<?xml version="1.0"?>                                  Catalog
<catalog library="XML Library">                        Catalog  Book
  <book>                                               Catalog  Book  StringBuffer
    <author> Luke Upp</author>                         Catalog  Book
                                                       Catalog  Book  StringBuffer
    <title>Oliver Twist Learns XML</title>             Catalog  Book
                                                       Catalog
  </book>                                              Catalog  Book
  <book>                                               Catalog  Book  StringBuffer
    <author>Cliff Hanger</author>                      Catalog  Book
                                                       Catalog  Book  StringBuffer
    <title>Romeo and Java</title>                      Catalog  Book
                                                       Catalog
  </book>                                              Catalog  Magazine
  <magazine>                                           Catalog  Magazine  StringBuffer
    <name>XML Today</name>                             Catalog  Magazine
                                                       Catalog  Magazine  Article
    <article page="5">                                 Catalog  Magazine  Article  StringBuffer
      <headline>XML Can Be Your Friend                 Catalog  Magazine  Article
      </headline>                                      Catalog  Magazine
    </article>                                         Catalog  Magazine  Article
    <article page="29">                                Catalog  Magazine  Article  StringBuffer
      <headline>The XML Diet</headline>                Catalog  Magazine  Article
                                                       Catalog  Magazine
    </article>                                         Catalog  Magazine  Article
    <article page="59">                                Catalog  Magazine  Article  StringBuffer
      <headline>SAX: The Inside Story                  Catalog  Magazine  Article
      </headline>                                      Catalog  Magazine
    </article>                                         Catalog
  </magazine>                                          Catalog  Book
  <book>                                               Catalog  Book  StringBuffer
    <author>Woody Glenn</author>                       Catalog  Book
                                                       Catalog  Book  StringBuffer
    <title>Tale of Two DTDs</title>                    Catalog  Book
                                                       Catalog
  </book>                                              Catalog  Magazine
  <magazine>                                           Catalog  Magazine  StringBuffer
    <name>Readers XML Digest</name>                    Catalog  Magazine
                                                       Catalog  Magazine  Article
    <article page="17">                                Catalog  Magazine  Article  StringBuffer
      <headline>Humor in XML</headline>                Catalog  Magazine  Article
                                                       Catalog  Magazine
    </article>                                         Catalog  Magazine  Article
    <article page="47">                                Catalog  Magazine  Article  StringBuffer
      <headline>XML Condensed</headline>               Catalog  Magazine  Article
                                                       Catalog  Magazine
    </article>                                         Catalog
  </magazine>
</catalog>
```