
SAX Tutorial 1

Author Name : Nazmul Idris
Created On : May 23, 1999
Modified On : June 4, 1999 8:03 pm

Table of contents

Overview

What is SAX

Three steps to SAX

Step1: Creating a custom object model

Step2: Creating a SAX parser

Step3: Creating a DocumentHandler

Presentation layers for the object model

Presentation Layer 1: Swing

Presentation Layer 2: Servlet

Source code

Overview

XML 1.0 allows you to encode your information in textual form and it allows you to create tags which allow you to structure the information stored in XML documents. This information must at some point be read by some program to do something useful, like viewing, modifying or printing it. In order for your programs to access this information you can use the SAX (Simple API for XML) or the DOM (Document Object Model) APIs. Both these APIs must be implemented by the XML parser of your choice (which also must be written in the programming language of your choice).

For Java, these parsers include the Sun TR2 XML Parser, Datachannel XJ2, IBM XML Parser for Java, and OpenXML (among many others). All of these parsers implement the SAX API (and also the DOM API). There are fewer differences in the implementation of SAX compared to the implementation of DOM 1.0 (simply because SAX is so much smaller and simpler than DOM).

So your Java programs must use an XML Parser written in Java that implements the SAX API in order for you to use SAX. In this tutorial, I illustrate the use of the Sun TR2 Parser, and I will show you how you can modify the code very simply in order to use the XML parser of your choice.

What is SAX

SAX stands for the Simple API for XML. Unlike DOM (Document Object Model) which creates a tree based representation for the information in your XML documents, SAX does not have a default object model. This means that when you create a SAX parser and read in a document (unlike DOM) you will not be given a nice default object model. A SAX parser is only required to read in your XML document and fire events based on the things it encounters in your XML document. Events are fired when the following things happen:

- open element tags are encountered in your document
- close element tags are encountered in your document
- #PCDATA and CDATA sections are encountered in your document
- processing instructions, comments, entity declarations, are encountered in your document.

We will start by looking at the open and close element tag events and the #PCDATA and CDATA events. One thing to remember about SAX is that the sequence of these events is very important, because the sequence in which events are fired determines how you will have to interpret each event.

Three steps to SAX

Since SAX does not come with a default object model representation for the data in your XML document, the first thing you have to when using SAX is create your own custom Java object model for your data. This could be something as simple as creating an AddressBook class if your XML document is an address book. After your custom object model has been created to “hold” your data (inside your Java program), the next step is creating a SAX document handler to create instances of your custom object models from the information stored in the XML document. This “document handler” is a listener for the various events that are fired by the SAX parser based on the contents of your XML document. This is very similar to the AWT 1.1 Event Delegation Model, where UI components generate events based on user input and event listeners perform some useful function when these events are fired. Most of the work in using SAX is in creating this document handler. Once you have created the custom object model and the SAX document handler you can use the SAX parser to create instances of your custom object model based on the data stored in your XML documents.

This process is illustrated using an example in the following paragraphs. I will show you how to perform these 3 steps for an AddressBook example. The example problem is that I have an XML document which contains my address book and I would like to view this address book using a Swing program and a Servlet. Also, I would like to use a SAX parser to do this instead of using a DOM parser. The first thing to do is create an object model and deal with the SAX parser issues before even thinking about the presentation layers (Swing and Servlet) for my object model (AddressBook). Here is what my address book XML document looks like:

```
<?xml version = "1.0"?>
<addressbook>
  <person>
    <lastname>Idris</lastname>
    <firstname>Nazmul</firstname>
    <company>The Bean Factory, LLC.</company>
    <email>xml@beanfactory.com</email>
  </person>
</addressbook>
```

The three steps to using SAX in your programs are:

1. Creating a custom object model (like Person and AddressBook classes)

2. Creating a SAX parser
3. Creating a DocumentHandler (to turn your XML document into instances of your custom object model).

Step1: Creating a custom object model

I have created a simple Java object model to represent the information in my address book XML document. I created 2 classes, an AddressBook class and a Person class. My object model is a simple mapping from the elements into classes. The following is a description of these classes:

- The AddressBook class is a container of Person objects. The AddressBook class is a simple adapter over the java.util.List interface. The AddressBook class has methods to allow you to add Person objects, get Person objects, and find out how many Person objects are in the AddressBook. The addressbook element maps to the AddressBook class.
- The Person class simply holds 4 String objects, the last name, first name, email and company name. This information is embedded within the <person> tag. So the person element maps into the Person class. The firstname, lastname, company and email elements map into String class.

Here is a listing of the Person class:

```
public class Person{
// Data Members
String fname, lname, company, email;

// accessor methods
public String getCompany(){return company;}
public String getEmail(){return email;}
public String getFirstName(){return fname;}
public String getLastName(){return lname;}

// mutator methods
public void setLastName( String s ){lname = s;}
public void setFirstName( String s ){fname = s;}
public void setCompany( String s ){company = s;}
public void setEmail( String s ){email = s;}

// toXML() method
public String toXML(){
    StringBuffer sb = new StringBuffer();
    sb.append( "<PERSON>\n" );
    sb.append( "\t<LASTNAME>"+lname+"</LASTNAME>\n" );
    sb.append( "\t<FIRSTNAME>"+fname+"</FIRSTNAME>\n" );
}
```

```
sb.append( "\t<COMPANY>"+company+"</COMPANY>\n" );
sb.append( "\t<EMAIL>"+email+"</EMAIL>\n" );
sb.append( "</PERSON>\n" );
return sb.toString();
}}
```

Please note the toXML() method. This method returns a String that contains the XML representation of a Person object. This kind of method is not only very useful for debugging, but it can also be used to save Person objects to an XML file (or other kind of XML persistence/storage engine). The AddressBook class also has a toXML() method, and that method uses the Person class's toXML() method too.

Here is a listing of the AddressBook class:

```
public class AddressBook{
// Data Members
List persons = new java.util.ArrayList();

// mutator method
public void addPerson( Person p ){persons.add( p );}

// accessor methods
public int getSize(){ return persons.size();}
public Person getPerson( int i ){
    return (Person)persons.get( i );}

// toXML method
public String toXML(){
    StringBuffer sb = new StringBuffer();
    sb.append( "<?xml version=\"1.0\"?>\n" );
    sb.append( "<ADDRESSBOOK>\n\n" );
    for(int i=0; i<persons.size(); i++) {
        sb.append( getPerson(i).toXML() );
        sb.append( "\n" );
    }
    sb.append( "</ADDRESSBOOK>" );
    return sb.toString();
}}
```

As you can see these are very simple classes. The interesting part (in this case) is Step3.

Step2: Creating a SAX parser

Creating a SAX parser is quite easy and you have to create an XML document handler class for the parser (so that something useful gets done as the parser parses the XML document).

Here is code to create a SAX parser:

```
import java.net.*;
import java.io.*;
import org.xml.sax.*;
...
try{
    //create an InputSource from the XML document source
    InputStreamReader isr = new InputStreamReader(
        new URL("http://host/AddressBook.xml").openStream();
        //new FileReader( new File( "AddressBook.xml" ) )
    );
    InputSource is = new InputSource( isr );

    //create an documenthandler to create obj model
    DocumentHandler handler = //new YourHandler();

    //create a SAX parser using SAX interfaces and classes
    String parserClassName = "com.sun.xml.parser.Parser";
    org.xml.sax.Parser.parser = org.xml.sax.helpers.ParserFactory.
        makeParser( parserClassName );

    //create document handler to do something useful
    //with the XML document being parsed by the parser.
    parser.setDocumentHandler( handler );
    parser.parse( is );
}
catch(Throwable t){
    System.out.println( t );
    t.printStackTrace();
}
```

The code example above uses the Sun TR2 parser. The classes used from TR2 include the `com.sun.xml.parser.Parser` which is used to create a non-validating SAX parser.

You can use any parser that supports SAX. You have to change 1 thing:

- Replace the value of the `parserClassName` string with the class name of the SAX Parser class of your choice.

You might be wondering what an `InputSource` class does. An `InputSource` is analogous to an `InputStream`. An `InputSource` is an encapsulation over a byte stream or character stream and it also includes a system and public identifier (which amount to a URI). In fact, in order to create in `InputSource`, you have to pass the constructor an `InputStream` object reference.

Step3: Creating a DocumentHandler

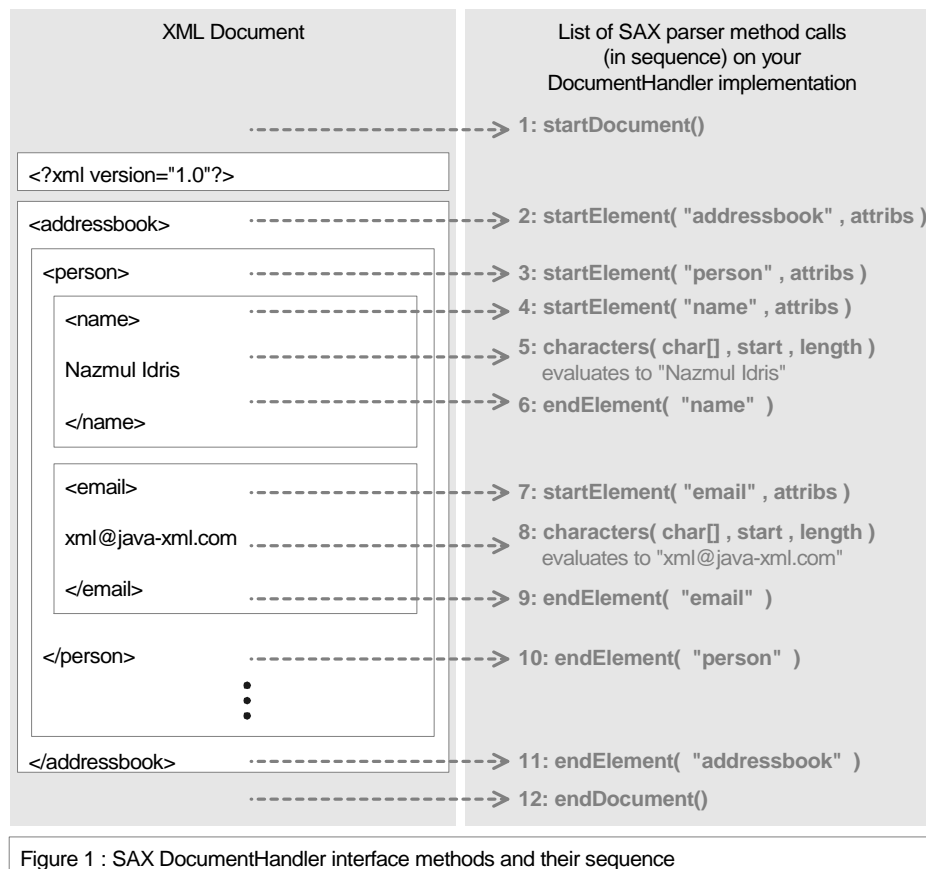
The SAX parser that was created in Step2 reads an XML document and fires events as it encounters open tags, close tags, CDATA and #PCDATA sections, etc. These events are fired as the SAX parser reads the XML document from top to bottom, a tag at a time. In order for the SAX parser to notify some object that these events are occurring, an interface called `DocumentHandler` is used (its in the `org.xml.sax` package). There are 3 other interfaces that exist called `EntityResolver`, `DTDHandler` and `ErrorHandler`. These 4 interfaces together include all the methods that correspond to all possible events that the SAX parser can fire (as its reading an XML document). The most frequently used interface is the `DocumentHandler` interface. So you have to provide an implementation of at least the `DocumentHandler` interface to the SAX parser, which then will invoke the right methods in the right sequence on your `DocumentHandler` implementation class. As the SAX parser reads an XML document, events are fired, which are then translated into method calls on all the “registered document event listeners” (which is your `DocumentHandler` implementation class). So as these events are fired, as the XML document is read, method calls are made on your `Document Handler` implementation class. This class must do something useful with these method calls and the sequence of the calls.

Figure 1 shows the sequence of method calls that the SAX parser makes on your `DocumentHandler` interface implementation class. The sequence of method calls are numbered from 1 to 12. You can see from this picture how the SAX parser turns the XML document into a bunch of events, which are in turn translated into a bunch of method calls in your `DocumentHandler` interface implementation class.

The `DocumentHandler` interface contains methods in it which deal primarily with element open and close tags, attributes, #PCDATA and CDATA sections. Once you create your `DocumentHandler` implementation class you must tell the SAX parser to use it. The following code snippet does this trick.

```
org.xml.sax.Parser.parser = //create a SAX parser
DocumentHandler handler = //instantiate your implementation
parser.setDocumentHandler( handler );
```

There are 3 more interfaces in addition to the `DocumentHandler` interface: `EntityResolver`, `DTDHandler`, `ErrorHandler`. The following sections describe each of these interfaces in addition to the `HandlerBase` class. Then I describe the actual implementation of the `DocumentHandler` interface (that creates the `AddressBook` object).



ErrorHandler interface

The ErrorHandler interface methods deal with custom error handling behaviors that you want to implement in your handler, in response to erroneous conditions in your source XML document. The SAX parser is not required to do any kind of error handling, it is the programmer's responsibility to put any error handling and exception-throwing code in the implementation class of this interface. You don't have to implement this interface. If you extend the HandlerBase class, a default implementation is already provided which throws a SAXException when errors are encountered in the XML document being parsed. The methods in the ErrorHandler interface are: error(SAXParseException e), fatalError(SAXParseException e) and warning(SAXParseException e) methods.

If you choose to perform custom error handling (in the implementation class of your ErrorHandler interface) you have to tell the SAX parser to use your class as an error handler. The following code snippet does this:

```
org.xml.sax.Parser.parser = //create a SAX parser
ErrorHandler handler = //instantiate your implementation
parser.setErrorHandler( handler );
```

DTDHandler interface

The DTDHandler interface methods deal with entity and notation declaration sections in the DTDs of the XML documents that you reading. The SAX parser is not required to do anything with these notation and entity declarations, if you wish to do something with these things then you have to write a class that implements the DTDHandler interface and tell the SAX parser to use it. The DTDHandler interface has the following methods: notationDecl(String name , String publicId, String systemId), and unparsedEntityDecl((String name, String publicId, String systemId, String notationName).

If you choose to do something useful with entity and notation declarations, you have to tell the SAX parser to use your class as a DTD handler. The following code snippet does this:

```
org.xml.sax.Parser.parser = //create a SAX parser
DTDHandler handler = //instantiate your DTDHandler implementation
parser.setDTDHandler( handler );
```

EntityResolver interface

The EntityResolver interface deals with allowing you to create customized InputSource objects for external entities. These external entities could be DTDs that are located by using a URI to a remote resource or any other resource that is external to your local system. These external entities or resources are located using URIs and by implementing the EntityResolver interface you can define custom code for creating an InputSource given an external entity. The EntityResolver interface only has one method: InputSource resolveEntity (String publicId, String systemId).

If you choose to provide customized behaviors for creating InputSources from external entities, you have to tell the SAX parser to use your class that implements the EntityResolver interface. The following code snippet does this:

```
org.xml.sax.Parser.parser = //create a SAX parser
EntityResolver handler = //instantiate your implementation
parser.setDTDHandler( handler );
```

HandlerBase class

Now, instead of implementing each method in the 4 interfaces (DocumentHandler, EntityResolver, DTDHandler and ErrorHandler) you can make your SAX handler class extend the org.xml.sax.HandlerBase class. The HandlerBase class provides empty implementations for each of the 4 SAX handler interfaces (DocumentHandler, EntityResolver, DTDHandler and ErrorHandler). Extending the HandlerBase saves a lot of time because you can only override the implementations for the methods that you are interested in using. I will demonstrate how to create an AddressBook from an XML document by extending the HandlerBase class.

Building the actual object model (using DocumentHandler)

In converting the address book XML document (described in the “Three steps to SAX” section), I use a DocumentHandler to create Person objects that I insert in one AddressBook object (the code

for these classes is given in the “Step1: Creating a custom object model” section). Instead of creating a `DocumentHandler` class by implementing the `DocumentHandler` interface directly, I choose to extend the `HandlerBase` class (as it saves a lot of time). I am only going to use three methods that are available in the `DocumentHandler` interface: `startElement(...)`, `endElement(...)` and `characters(...)`.

Now, an address book contains person elements, which in turn contain name and email elements. This is how I have to think in order to write a `DocumentHandler`:

- each person element has to be converted into a `Person` object and inserted into one `AddressBook` object.
- the `lastname`, `firstname`, `company` and `email` elements have to be converted to `String` objects and put inside a `Person` object.

So as you can see, my `DocumentHandler` implementation must create one `AddressBook` object and many `Person` objects (using the information in the XML document). The SAX parser reports this information to my `DocumentHandler` by making a sequence of method calls as shown in Figure 1. I must use this sequence of method calls and remember “where” in the XML document I am in order to create `Person` objects (and add them to the `AddressBook` object).

In order to remember “where” in the XML document I currently am in, I use a `String` to remember the name of the last tag that was encountered. Every time an `startElement(“person”, ..)` method is called on my handler (by the SAX parser), I create a new `Person` object (and save a reference to it). Thereafter, when I get `endElement(“person”, ..)` method called on my handler, I add the current `Person` object to my `AddressBook` object.

In between the `startElement(“person”, ...)` and `endElement(“person”, ...)` method calls I get 4 more `startElement(...)` and `endElement(...)` method calls for the “`lastname`”, “`firstname`”, “`email`” and “`company`” tags. I have to save the values that are contained inside each of these 4 elements in 4 `Strings`, and add these values to the current `Person` object that I just created.

In a nutshell, this is how you have to interpret the sequence of method calls in order to create your object model. You can think of this process as a document -> event -> method -> object model mapping. You have to provide the document and the object model and the method handler. SAX only takes care of the event generation and method invocation (on your handler implementations).

This all sounds a lot more complicated than it really is. Here a partial listing of the code for my `DocumentHandler` implementation (and `HandlerBase` subclass) that I call `SaxAddressBookHandler.java`:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.ParserFactory;
import com.sun.xml.parser.Resolver;

public class SaxAddressBookHandler extends HandlerBase{
    // data members
    private AddressBook ab = new AddressBook();
    private Person p = null;           //temp Person ref
```

```
private String currentElement = null; //current element name

// AddressBook accessor method
public AddressBook getAddressBook(){    return ab; }

// HandlerBase method overrides. This is SAX.
/*
This method is called when the SAX parser encounters an open element
tag. Must remember which element tag was just opened (so that the
characters(..) method can do something useful with the data that is
read by the parser.
*/
public void startElement( String name , AttributeList atts ){
    if( name.equalsIgnoreCase("LASTNAME") ) {
        currentElement = "LASTNAME";
    }
    else if( name.equalsIgnoreCase("FIRSTNAME") ) {
        currentElement = "FIRSTNAME";
    }
    else if( name.equalsIgnoreCase("COMPANY") ) {
        currentElement = "COMPANY";
    }
    else if( name.equalsIgnoreCase("EMAIL") ) {
        currentElement = "EMAIL";
    }
    else if( name.equalsIgnoreCase("PERSON") ) {
        p = new Person();
    }
}

/*
This method is called when the SAX parser encounters a close element
tag. If the person tag is closed, then the person objec must be
added to the AddressBook (ab).
*/
public void endElement( String name ){
    if( name.equalsIgnoreCase("PERSON") ) {
        ab.addPerson( p );
    }
}
```

```
        p = null;
    }
}

/*
This method is called when the SAX parser encounters #PCDATA or CDATA.
It is important to remember which element tag was just opened so that
this data can be put in the right object.
I had to trim() the textual data and make sure that empty data is
just ignored.
Also the start index and length integer must be used to retrieve only
a portion of the data stored in the char[].
*/
public void characters( char ch[], int start, int length ){
    //dont try to read ch[] as it will go on forever, must use the
    //range provided by the SAX parser.
    String value = new String( ch , start , length );
    if(!value.trim().equals("")) {
        if( currentElement.equalsIgnoreCase("FIRSTNAME") ) {
            p.setFirstName( value );
        }
        else if( currentElement.equalsIgnoreCase("LASTNAME") ) {
            p.setLastName( value );
        }
        else if( currentElement.equalsIgnoreCase("COMPANY") ) {
            p.setCompany( value );
        }
        else if( currentElement.equalsIgnoreCase("EMAIL") ) {
            p.setEmail( value );
        }
    }
}
```

This class is what makes it all happen. In your projects, you will have to do something very similar. This is a simple example by its very design to show you the basic idea. In your real world projects, the code in your DocumentHandler implementation might be very complex in order to deal with a complex XML document.

Now that you have seen the object model creation (using SAX from XML documents) part, the next step involves displaying this information to your users using a presentation layer (in our case, Swing and Servlet).

Presentation layers for the object model

Once the SAX document handler and Java object models have been created, the presentation layer becomes important. Here is where the user will “see” the object model (that came from the XML document via the SAX document handler you write). The presentation layer is the View (in an MVC context). The view should be independent of the model, so we can create any number of presentation layers (or views). Two are shown here, one is Swing based for local display, another is for web based display using HTML and Servlets. Both of the presentation layers shown below are read-only or view-only layers, they don’t allow editing of the address book XML document. In future SAX tutorials, I will show you how to create editable presentation layers.

Presentation Layer 1: Swing

In order to display the object model (AddressBook) using Swing, I choose to use a JTable to display it in. This simply involves the creation of a TableModel implementation class (that sits on top of the AddressBook class). I call this TableModel implementation class AddressBookTableModelAdapter and it is a TableModel (interface) adapter for the AddressBook. This adapter class simply implements the TableModel, but the methods for the TableModel implementation are actually implemented by making use of methods available to the AddressBook class.

Here is a listing of the AddressBookTableModelAdapter class:

```
import java.awt.*;                //AWT classes
import java.awt.event.*;          //AWT event classes
import java.util.*;               //Vectors, etc
import java.io.*;                 //Serializable, etc
import java.net.*;                //Network classes
import javax.swing.*;             //Swing classes
import javax.swing.event.*;        //Swing events
import javax.swing.table.*;        //JTable models
import javax.swing.tree.*;        //JTree models
import javax.swing.border.*;      //JComponent Borders

public class AddressBookTableModelAdapter
implements TableModel {
// Data Members
protected java.util.List listeners = new ArrayList();
protected static final Class[] colClasses = {
    String.class ,
    String.class ,
    String.class ,
    String.class
}
```

```
};

protected static final String[] colNames = {
    "LASTNAME" ,
    "FIRSTNAME" ,
    "COMPANY" ,
    "EMAIL"
};

protected static final int LASTNAME_COL = 0 ,
    FIRSTNAME_COL = 1 ,
    COMPANY_COL = 2 ,
    EMAIL_COL = 3;

protected AddressBook addressBook;

// Constructor
public AddressBookTableModelAdapter( AddressBook a ){
    addressBook = a;
}

// TableModel impl
public int getRowCount(){
    return addressBook.getSize();
}

public int getColumnCount(){
    return colClasses.length;
}

public String getColumnName(int c){
    return colNames[ c ];
}

public Class getColumnClass(int c){
    return colClasses[ c ];
}

public boolean isCellEditable(int r, int c){
    return false;
}
```

```
public Object getValueAt(int r, int c){
    //row corresponds to person
    Person p = addressBook.getPerson( r );

    //column corresponds to person field
    if( c == LASTNAME_COL ) {
        return p.getLastName();
    }
    else if( c == FIRSTNAME_COL ) {
        return p.getFirstName();
    }
    else if( c == COMPANY_COL ) {
        return p.getCompany();
    }
    else if( c == EMAIL_COL ) {
        return p.getEmail();
    }
    else return "No value for this col";
}

public void setValueAt(Object v, int r, int c){
    if( r == addressBook.getSize() ) return;
}

public void addTableModelListener(TableModelListener l){
    if(!listeners.contains(l)) {
        listeners.add( l );
    }
}

public void removeTableModelListener(TableModelListener l){
    if(listeners.contains(l)) {
        listeners.remove(l);
    }
}

// Event Utility Methods
```

```

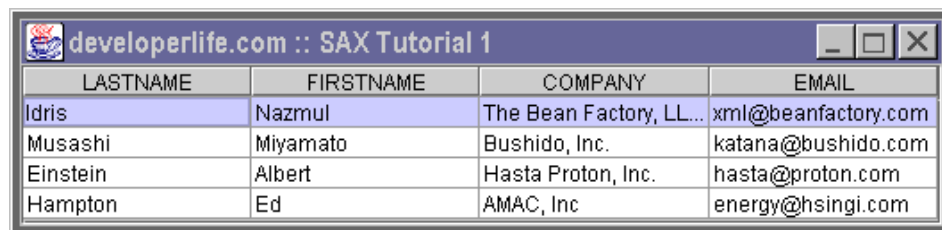
public void fireTableChanged(){
    TableModelEvent e = new TableModelEvent( this );
    ArrayList copy = new ArrayList( listeners );
    for(int i=0; i<copy.size(); i++) {
        ((TableModelListener)copy.get(i)).tableChanged( e );
    }
}
}
} //end class AddressBookTableModelAdapter

```

The AddressBookFrame class (that is provided with the source code for this tutorial) can be run in order to see this Swing user interface in action. The AddressBookFrame does the following things:

- uses the SaxAddressBookConverter to create an AddressBook (using SAX); the actual AddressBook.xml document that is used is located at <http://beanfactory.com/xml/Address-Book.xml>
- takes this AddressBook object and creates an AddressBookTableModelAdapter class with it
- creates a JTable with this AddressBookTableModelAdapter as its TableModel
- displays the JTable to a JFrame (after putting it in a JScrollPane).

Figure 2 is a screenshot of what this Swing program looks like.



LASTNAME	FIRSTNAME	COMPANY	EMAIL
Idris	Nazmul	The Bean Factory, LL...	xml@beanfactory.com
Musashi	Miyamoto	Bushido, Inc.	katana@bushido.com
Einstein	Albert	Hasta Proton, Inc.	hasta@proton.com
Hampton	Ed	AMAC, Inc	energy@hsingi.com

Figure 2 : Screenshot of Swing presentation layer of the AddressBook

Presentation Layer 2: Servlet

In order to display the AddressBook to HTML (by using a Servlet), it is possible to generate the HTML by using the AddressBook object itself. Instead of doing this, I choose to use the AddressBookTableModelAdapter that has already been written. By reusing the adapter class, I can save myself a lot of coding because the adapter (which implements the TableModel) can easily be converted into HTML with very little work.

```

import java.util.*; //Vectors, etc
import java.io.*; //Serializable, etc
import java.net.*; //Network classes
import javax.servlet.*; //Servlet classes
import javax.servlet.http.*; //Servlet classes

```



```
public class AddressBookServlet extends HttpServlet {
// doGet() method
protected void doGet( HttpServletRequest req,
                      HttpServletResponse res)
throws ServletException, IOException{
    //create an AddressBook obj using a SaxAddressBookConverter
    AddressBookTableModelAdapter model =
        new AddressBookTableModelAdapter(
            new SaxAddressBookConverter().getAddressBook() );

    //output the HTML to the res.outputstream
    res.setContentType("text/html");

    PrintWriter out = new PrintWriter(res.getOutputStream());
    out.print( "<html>" );
    out.print( "<title>" );
    out.print( "SAX Tutorial Part 1" );
    out.print( "</title>" );
    out.print( "<center>" );
    out.print( "<head><pre>" );
    out.print( "http://beanfactory.com/xml/AddressBook.xml" );
    out.print( "</pre></head><hr>" );

    //format the table
    out.print( "<table BORDER=0 CELLSPACING=2 " );
    out.print( "CELLPADDING=10 BGCOLOR=\"#CFCFFF\" >" );
    out.print( "<tr>");

    //display table column
    for(int i=0; i < model.getColumnCount(); i++){
        out.print( "<td><b><center>" +
                    model.getColumnName( i ) +
                    "</center></b></td>" );
    }

    out.print( "</tr>" );

    //need to iterate the doc to get the fields in it
```

```
for(int r=0; r < model.getRowCount(); r++) {
    out.print( "<tr>" );

    for(int c=0; c < model.getColumnCount(); c++) {
        out.print( "<td>" );
        out.print( model.getValueAt( r , c ) );
        out.print( "</td>" );
    }//end for c=0...
    out.print( "</tr>" );

} //end for r=0...

out.print( "</table>" );
out.print( "<hr>Copyright The Bean Factory, LLC." );
out.print( " 1998-1999. All Rights Reserved." );
out.print( "</center>" );
out.println("</body>");
out.println("</html>");
out.flush();
}
} //end class AddressBookServlet
```

The AddressBookServlet does the following things:

- uses the SaxAddressBookConverter to create an AddressBook (using SAX); the actual AddressBook.xml document that is used is located at <http://beanfactory.com/xml/Address-Book.xml>
- takes this AddressBook object and creates an AddressBookTableModelAdapter class with it
- iterates through the AddressBookTableModelAdapter and creates an HTML table by running through each row of the TableModel.

Figure 3 is a screen shot of what this servlet looks like.

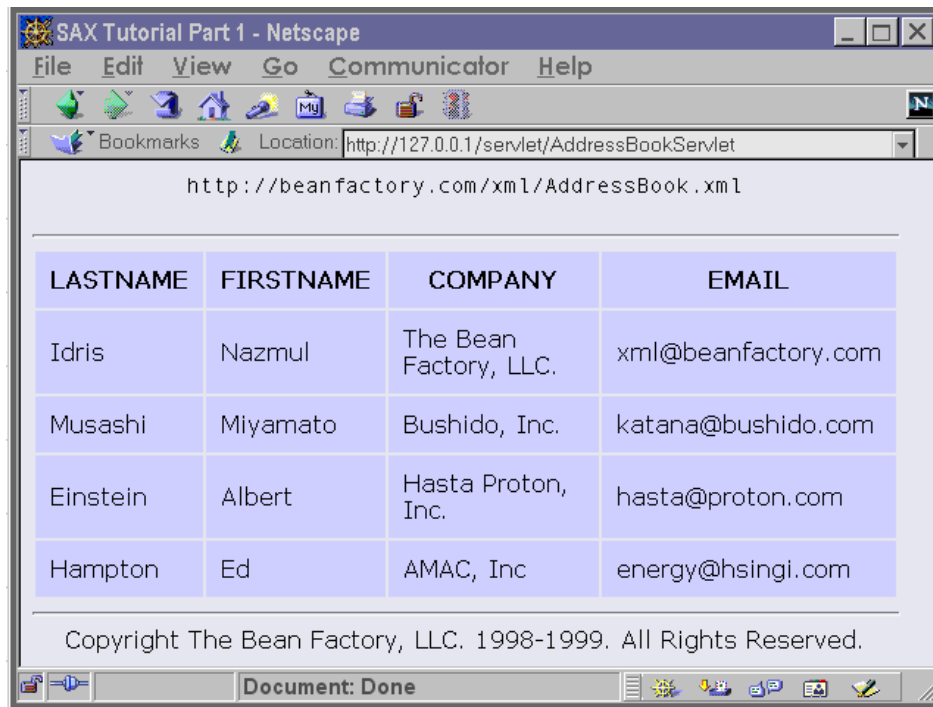


Figure 3 : Screenshot of Servlet presentation layer of the AddressBook

Source code

All the source code on the site is provided in one zip file. The com.developerlife.saxtutorial1 package contains the classes used in this tutorial. A description of these classes are provided in Table 1.

Table 1: Source code classes and descriptions

Source file	Description
Package com.developerlife.saxtutorial1	Contains all the classes for this tutorial
AddressBook.java	Object model for information store in the address book XML document
AddressBookFrame.java	This class displays the Swing based user interface for the address book
AddressBookServlet	This class is the Servlet that displays the address book as an HTML document

Table 1: Source code classes and descriptions

Source file	Description
AddressBookTableModelAdapter	This class is an adapter that fits the TableModel interface on top of the AddressBook class
Person	This class contains information stored in a person element of the address book XML document
SaxAddressBookConverter	This class creates a SAX parser and reads an XML address book document, and uses the SaxAddressBookHandler to create an AddressBook object
SaxAddressBookHandler	This class implements most of the DocumentHandler SAX interface and is responsible for converting the address book XML document into an AddressBook object

Running the programs

In order to run the Swing or Servlet program you must download the zip file that contains all the sources and unzip it to some folder on your harddrive (and make sure to recreate all the folders contained in the zip file).

Swing

If you are trying to run the Swing program, then make sure you have JDK1.2 VM on your machine. Using the Java2 VM, run the following from the command prompt:

```
java com.developerlife.saxtutorial1.AddressBookFrame
```

Please make sure that you are in the folder in which you unzipped the source archive. If you are behind a firewall, then this program will not work because it tries to load an XML document from <http://beanfactory.com/xml/AddressBook.xml>. In order to make this program work behind a firewall, you must tell the Java2 VM that you are behind a firewall by doing the following:

```
java -DproxySet=true -DproxyHost=PROXYSERVER -DproxyPort=PORT  
com.developerlife.saxtutorial1.AddressBookFrame
```

where PROXYSERVER is the hostname or IP address of your proxy server and PORT is the port number of the proxy server.

Alternatively, you can run the following code before you use URL.openStream() method:

```
System.getProperties().put("proxySet", "true");  
System.getProperties().put("proxyHost", "PROXYSERVER");
```

```
System.getProperties().put("proxyPort", "PORT");
```

Using either of the 2 ways shown above, your Java VM will try to use the proxy server that is installed in your network.

Servlet

If you are trying to run the Servlet program, make sure you have a Servlet 2.0 compliant Servlet engine and webserver installed on your machine. Then you have to make sure to copy the `com.developerlife.saxtutorial1` package to your servlet engines servlet class folder. Then invoke the servlet from your web browser by using the following URL:

```
http://HOST:PORT/servlet/  
com.developerlife.saxtutorial1.AddressBookServlet
```

where HOST is the hostname or IP address of the machine running the servlet engine and PORT is the port number of the servlet engine. Now some servlet engines might require you to put in a different URI than what I have shown above. Regardless of what your servlet engine expects you to do, you must specify the Servlet class that must be run by using `com.developerlife.saxtutorial1.AddressBookServlet`.

If you are behind a firewall you should put the following lines of code in your Servlet:

```
System.getProperties().put("proxySet", "true");  
System.getProperties().put("proxyHost", "PROXYSERVER");  
System.getProperties().put("proxyPort", "PORT");
```

where the PROXYSERVER is the hostname or IP address of the proxy server and PORT is the port number of the proxy server.

I hope you enjoyed this tutorial, I will have more complicated SAX tutorials online at a later time.