XML Content Handlers

This article is one in a series describing an approach to building rich clients that permit lightweight AJAX-style interactions, but use Java technology instead of JavaScript, and XML instead of HTML. This particular article describes how to build and use XML content handlers in such applications, and assumes that you have already read **Asynchronous Web Clients**, the first article in this series, available at http://www.openmarkup.net/docs/tutorials/AsyncClients.pdf.

Overview

In an asynchronous web client, raw content is loaded and processed in the background using a different thread from the one that initiated the load. While raw content is processed in the background thread, processed chunks of it are asynchronously consumed by the client in the event handling thread. The basic design involves three cooperating objects, each responsible for a specific part of the interaction. First, a **content loader** is responsible for fetching content from a web service or data source after passing request parameters, if any, to the server. Second, a **content handler** is responsible for interpreting, filtering, and manipulating the fetched content to realize domain-specific objects from the content. Finally, a **content consumer** is responsible for assimilating the realized objects into the presentation.

Content loaders, content handlers, and content consumers implement the `ContentLoader`, `ContentHandler`, and `ContentConsumer` interfaces, respectively. When the received content consists of XML data, a `ContentHandler` has a wide variety of choices in how it handles that content. An obvious choice is to simply treat XML content as plain text, and extract meaningful chunks of *information* from it using **regular expressions**. More interesting content handlers typically use a standard **SAX** or **DOM parser** to extract information from the content. Additionally, a content handler may preprocess the XML content using **XSLT** to transform the data to another format, or postprocess it with an **object realizer** to realize domain-specific objects from it. This article shows how to build content handlers that exercise some of these options.

We begin by briefly touching upon basic DOM programming, and then show how to accomplish more with less code using an object realizer and a reusable tag library.

DOM Programming

A popular way to handle XML content is to first construct a **document object model** from the content using a DOM parser, and then use the DOM programming interfaces to access and manipulate the document object model. In this article, we will look at simple examples of DOM programming using Document Object Model Level 3 interfaces.

DOM Level 3 introduces several new features, such as a standard way to create a DOM `Document`, and the ability to bind application-specific *non*-XML data to a DOM `Document`. Whenever we need a DOM parser, we use the bootstrapping feature of DOM Level 3 Core to first obtain a reference to the **DOM Implementation Registry**. The registry provides an implementation-independent starting point for applications. From the registry, we obtain a suitable **DOM implementation class** based on the DOM features we want to use. Finally, we use the DOM implementation class to create parsers, as well as objects that provide input parameters to parsers. Since this requires repetitive boilerplate code, we define the `DOMBootstrapper` class shown in Listing 1 to reduce the tedium.

Listing 1: DOMBootstrapper class

```
public class DOMBootstrapper {
    public static DOMImplementation getImplementation(String features) {
        return registry == null ? null : registry.getDOMImplementation(features);
    }

    private static DOMImplementationRegistry    registry;
    static {
        try {
            System.setProperty(DOMImplementationRegistry.PROPERTY,
            "com.sun.org.apache.xerces.internal.dom.DOMImplementationSourceImpl");
            registry = DOMImplementationRegistry.newInstance();
        }
        catch (Exception ex) {
            System.out.println("Could not create DOMImplementationRegistry: " + ex);
        }
    }
}
```

The `static` initializer block initializes the `registry` member with a new instance of `DOMImplementationRegistry`, and the `getImplementation` method uses that registry to look up and return a `DOMImplementation` instance that supports the requested features.

Listing 2 shows how to use `DOMBootstrapper` to obtain a DOM implementation that supports the DOM Level 3 **Load and Save** feature (indicated by `"LS"`). The parser we create in this example operates synchronously with respect to the current thread.

Listing 2: Using the DOMBootstrapper class

```
DOMImplementationLS impl =
        (DOMImplementationLS) DOMBootstrapper.getImplementation("LS");  // Load&Save

if (impl != null) {
    LSParser parser = impl.createLSParser(MODE_SYNCHRONOUS, null);
    Document document = parser.parseURI("foo.xml");
}
```

Listing 3 shows how to create an `LSInput` object using a DOM implementation obtained from `DOMBootstrapper`, and use it to pass input parameters to the parser.

Listing 3: Using the DOMBootstrapper class

```
DOMImplementationLS impl =
        (DOMImplementationLS) DOMBootstrapper.getImplementation("LS");  // Load&Save

if (impl != null) {
    LSInput   input = impl.createLSInput();
    input.setCharacterStream(reader);    // reader is an instance of java.io.Reader
    input.setSystemId(systemID);         // optional, but recommended
    LSParser  parser = impl.createLSParser(MODE_SYNCHRONOUS, null);
    Document  document = parser.parse(input);
}
```

When passing an instance of a `Reader` to the parser through the `setCharacterStream` method of `LSInput`, it is not *necessary* to set the input object's system ID (URI reference indicating where input data is to be fetched from). This is because the parser will only attempt to fetch the content identified by the URI reference if there is no other input available in the input source. However, setting the system ID is useful, even when it is not required, as it may help the parser in resolving relative URI references that might be embedded in the content being parsed.

A parser created with the `MODE_SYNCHRONOUS` flag runs synchronously with respect to the thread in which it was created. This means that the parser's `parse` method does not return until the entire input has been parsed, and a new `Document` has been constructed. Consequently, if the parser is run in the Event Dispatch Thread, the user interface may become unresponsive until the DOM has been built. However, when a synchronous parser is created within a `ContentHandler` that is running in a separate worker thread, the user interface remains responsive, even when the `ContentLoader` is triggered in the Event Dispatch Thread.

Example: Parsing an RSS Feed

We now look at an example of basic DOM programming in a `ContentHandler`. Listing 4 shows a class called `ChannelParser` that implements the `ContentHandler` interface, and uses DOM Level 3 calls to create a DOM parser for parsing an **RSS feed**.

Listing 4: ChannelParser class

```
public class ChannelParser implements ContentHandler<Element, Object> {
    public Element getObject(String systemID, MediaType contentType, int length,
        Map<String,String> params, InputStream stream, ProgressTracker<Object> o) {
        try {
            if (contentType.isXML()) {
                BufferedReader   reader = new BufferedReader(
                                                new InputStreamReader(stream));
                LSInput       input = domImpl.createLSInput();
                input.setCharacterStream(reader);
                input.setSystemId(systemID);
                LSParser      parser = domImpl.createLSParser(MODE_SYNCHRONOUS, null);
                Document      document = parser.parse(input);
                reader.close();

                NodeList  nodes = document.getElementsByTagName("channel");
                return nodes == null ? null : (Element) nodes.item(0);
            }
        }
        catch (Exception ex) {
            System.out.println(ex);
        }
        return null;
    }
    private final static DOMImplementationLS    domImpl =
        (DOMImplementationLS) DOMBootstrapper.getImplementation("LS");
}
```

The `getObject` method of this class parses an RSS document supplied to it through the input stream, and returns a reference to the `channel` DOM element as its result.

Listing 5 shows how to write a ContentConsumer that extracts information about syndicated items from the channel Element, and uses that to populate an instance of JList. The SyndicatedItem class used in this example is one of several defined in a sample class library built specifically to illustrate many of the concepts in this article.

Listing 5: Extracting information from a DOM Element

```
public class ChannelDOMConsumer extends AbstractContentConsumer<Element, Object> {
    public ChannelDOMConsumer(JList listBox) {
        this.listBox = listBox;
    }

    @Override public void assimilate(boolean cancelled, Element channelElem) {
        if (!cancelled && channelElem != null) {
            DefaultListModel listModel = new DefaultListModel();
            NodeList itemElements = channelElem.getElementsByTagName("item");
            int max = itemElements.getLength();
            for (int i = 0; i < max; i++) {
                Element   itemElement = (Element) itemElements.item(i);
                SyndicatedItem   item = getSyndicatedItem(itemElement); // see below
                listModel.addElement(item);   // add domain object: SyndicatedItem
            }
            listBox.setModel(listModel);
        }
    }

    private SyndicatedItem getSyndicatedItem(Element itemElement) {
        SyndicatedItem item = new SyndicatedItem(); // from Syndication library
        try {  // extract domain-specific information from DOM Element
            item.setTitle(itemElement.getElementsByTagName(
                "title").item(0).getTextContent());
            item.setDescription(itemElement.getElementsByTagName(
                "description").item(0).getTextContent());
            item.setLink(new URL(itemElement.getElementsByTagName(
                "link").item(0).getTextContent()));
            // ... and so on
        }
        catch (Exception ex) {
            System.out.println("Could not populate syndicated item: " + ex);
        }
        return item;  // SyndicatedItem realized from DOM Element
    }

    private final JList listBox;
}
```

This type of code is easy to write, and there are plenty of good articles and books that do a better job describing DOM programming than what this short article can possibly do.

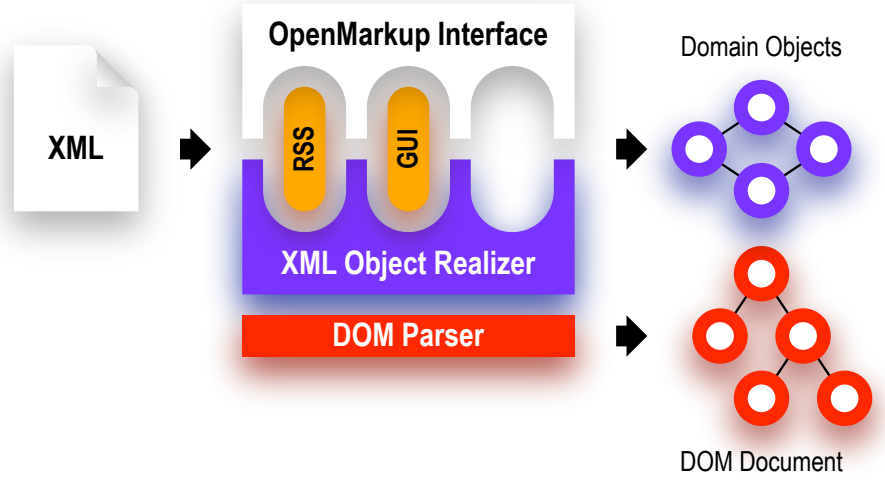
This type of code can also be tedious to write, especially when it has to be done over and over again. For XML documents with stable, well-defined formats, a better alternative is to use prefabricated components that can automatically realize domain-specific objects from XML descriptions of those objects without requiring low level DOM programming.

Object Realization

Object realization is a process by which objects are created, configured, and processed according to machine-readable descriptions of those objects. It includes post-instantiation tasks, such as configuring objects with additional attributes or properties, connecting them with other objects to create complex object compositions, or otherwise manipulating them according to control information embedded in the object descriptions.

An **XML object realizer** works in concert with a DOM parser and one or more pluggable **tag libraries** to realize domain-specific components from their XML descriptions. As shown in Figure 1, XML object realization results in the production of two object structures — a DOM document consisting of a *tree* of DOM elements, and an object *graph* comprised of domain-specific components that are realized from the DOM document.

Figure 1: Object realizer with plug-in tag libraries



An object realizer first routes the input XML document to a DOM parser for creating a document object model in memory. This document object model is an instance of a custom **document implementation class**, called `DOMDocument`, that the DOM parser is configured with. In this document object model, each element in the document is represented by an instance of a custom **element implementation class** that is provided by a specialized tag library registered with the object realizer.

The element implementation classes in a tag library know how to extract domain-specific information from the document information set and use that to realize **domain objects** that are free of any vestiges of the document object model. A call to realize the domain object corresponding to an element in the document starts the construction of the object graph, as related objects are realized and assimilated according to rules defined by the tag library.

To summarize, the document object model is created by a DOM parser using a custom document implementation class that the parser is configured with, while the object graph is created by custom element implementation classes provided by various tag libraries. An object realizer allows applications to use domain objects without getting tangled in XML code that is not germane to the application, effectively shifting the burden of XML programming from the application developer to the tag library developer.

The remainder of this article describes how to *use* an object realizer and tag libraries. For information on how to *define* a tag library, see **Object Realization Tag Libraries**.

Interfaces

An instance of an element implementation class provides the run-time representation of an element, as well as a way to obtain the domain object realized from the element. To support both, OpenMarkup defines an interface called `OMElement` that inherits DOM operations from the `Element` interface, while adding new operations for object realization. To ensure that all elements in a document object model support these operations, OpenMarkup requires a document implementation class to implement the `OMDocument` interface, which redeclares the element creation methods of `Document` to return instances of the **covariant type**, `OMElement`, instead of the original return type, `Element`.

The object realizer component itself implements the `ObjectRealizer` interface, which declares operations for obtaining an `OMDocument`, an `OMElement`, or a domain object from an XML input source. For details on these OpenMarkup interfaces, refer to the API documentation at http://www.openmarkup.net/docs/api/om/.

Realizing domain objects from XML

We now look at an example of a `ContentHandler` that uses an object realizer to process an RSS feed using a sample **RSS tag library** to realize objects specific to the *content syndication* domain. The content handler class, called `ChannelBuilder`, is shown in Listing 6. It is worth contrasting this `ChannelBuilder` class with the `ChannelParser` class shown earlier in Listing 4. Unlike the `ChannelParser` class, whose `getObject` method returns an ordinary DOM `Element`, the `ChannelBuilder` class returns a domain-specific `Channel` object that is realized from the input XML by an object realizer configured with the RSS tag library.

Listing 6: ChannelBuilder class

```
public class ChannelBuilder implements ContentHandler<Channel, Object> {
   public Channel getObject(String systemID, MediaType contentType, int length,
      Map<String,String> params, InputStream stream, ProgressTracker<Object> o) {
      try {
         if (contentType.isXML()) {
            BufferedReader   reader = new BufferedReader(
                                          new InputStreamReader(stream));
            OMDocument document = or.getDocument(reader, systemID);
            reader.close();
            ChannelElement   channelElem = (ChannelElement) document.evaluate(
                                          null, "/rss/channel", NODE);
            return channelElem.getObject();   // realize domain object
         }
      }
      catch (Exception ex) {
         System.out.println(ex);
      }
      return null;
   }

   private final static ObjectRealizer or =
                           new ObjectRealizerImpl(RSSVocabulary.get());
}
```

The last two lines of the `ChannelBuilder` class show how to create an object realizer and configure it with the RSS tag library. The tag library, identified by the `RSSVocabulary` class in this example, is actually a collection of classes that implement the RSS object realization code. The `ChannelBuilder` class stores a reference to the object realizer in a private final static member.

The `getObject` method of `ChannelBuilder` calls the `getDocument` method of the object realizer to parse the input data and prime the resulting document object model for subsequent object realization chores. It also closes the reader after the document has been parsed.

The `getObject` method of `ChannelBuilder` then tries to obtain a `ChannelElement` object from the document object model by asking the document to `evaluate` an XPath expression referring to the RSS `channel` element. The first argument to the `evaluate` method is `null`, indicating that the XPath expression does not depend on any **namespace context**. This is because RSS element types do not belong to any namespace. The second argument is the actual XPath expression identifying the `channel` element under the `rss` root element, while the third argument indicates that we are interested in a single DOM `Node`. The `ChannelBuilder` class *knows* that the `evaluate` method will return an instance of `ChannelElement` for this expression because the `RSSVocabulary` class states that the element implementation class for an RSS `channel` element type is `ChannelElement`.

The `ChannelElement` class is one of several in the RSS tag library. It inherits the implementation of the `Element` interface along with basic object realization features from its ancestor classes. The most interesting feature of this class is that we can realize a domain-specific `Channel` object from a `ChannelElement` object simply by calling *its* `getObject` method. That is exactly what we do in the following return statement. Note that the `Channel` class encapsulates domain knowledge about syndicated channels, but knows nothing about XML in general, and RSS in particular.

Example: Showing syndicated items in a Swing JList

To consume the output of a `ChannelBuilder`, we now develop a `ContentConsumer` that fills an instance of `JList` with channel items contained in the realized `Channel` object. Listing 7 shows the code for this class called `ChannelListUpdater`. Compared to the code in Listing 5, the code in Listing 7 is remarkably clear, as all of the domain-specific object realization code is neatly tucked away inside the *reusable* RSS tag library.

Listing 7: ChannelListUpdater

```
public class ChannelListUpdater extends AbstractContentConsumer<Channel, Object> {
    public ChannelListUpdater(JList listBox) {
        this.listBox = listBox;
    }
    @Override public void assimilate(boolean cancelled, Channel channel) {
        if (!cancelled && channel != null) {
            List<SyndicatedItem> items = channel.getItems();
            // Use DefaultComboBoxModel as ListModel for "convenience"
            // as it offers a constructor that accepts an array...
            listBox.setModel(new DefaultComboBoxModel(items.toArray()));
        }
    }
    private final JList listBox;
}
```

Listing 8 shows a simple application that uses a `ContentLoader` to load an RSS feed from a **java.net forum**. Notice that there are no visible hints of Java Generics in the main application code.

Listing 8: Using ChannelBuilder and ChannelListUpdater

```
public class SyndicatedItemsList {
    public static void main(String[] args) {
        JList  listBox = new JList();
        ContentLoader itemsLoader = ContentLoaderFactory.createContentLoader(
            new ChannelBuilder(), new ChannelListUpdater(listBox), null,
            "http://forums.java.net/jive/rss/rssmessages.jsp?categoryID=29");
        itemsLoader.load();
        // Build rest of the UI while content is loaded in a background thread
        JFrame frame = new JFrame();
        JScrollPane  scrollPane = new JScrollPane(listBox);
        frame.add(scrollPane);
        frame.setSize(600, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

The application first sets up a `ContentLoader` and triggers its `load` method in the main application thread. The content loader loads the specified RSS feed in a separate *background* thread, and delivers the feed to an instance of `ChannelBuilder` in the same background thread. The application continues to build the rest of the user interface in the main application thread.

Meanwhile, `ChannelBuilder` uses an object realizer configured with the RSS tag library to realize a `Channel` object from the loaded content, and returns the realized object to the content loader in the same background thread in which it received the raw RSS feed. Upon receiving the realized `Channel` object from the `ChannelBuilder` instance, the content loader delivers the realized object to the `ChannelListUpdater` instance in the Event Dispatch Thread, where `ChannelListUpdater` populates the `JList` instance with `SyndicatedItem` objects obtained from the realized `Channel` object.

Open Formats

An XML schema is **open** if instance documents written to that schema may contain elements and attributes *not* defined in the schema while still remaining valid against the schema. The RSS format, for example, is "open" by this definition, because it allows RSS documents to contain arbitrary elements as long as they are defined within *a* namespace — *any* namespace, for that matter.

The RSS feed from a **java.net forum** shown in Listing 9 contains several elements whose type is not defined by the RSS specification. For example, all elements with the `jf` prefix belong to a third-party *proprietary* namespace identified by the namespace URI "`http://www.jivesoftware.com/xmlns/jiveforum/rss`".

Obviously, a *domain-specific* tag library (such as an RSS tag library) cannot be expected to have built-in support for arbitrary *application-specific* extensions (such as those for Jive forum). Consequently, applications that use predefined tag libraries for open document formats must be prepared to handle content that might not be understood by the tag libraries.

Listing 9: RSS feed with application-specific extensions

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0"
   xmlns:dc="http://purl.org/dc/elements/1.1"
   xmlns:jf="http://www.jivesoftware.com/xmlns/jiveforums/rss">

   <channel>
      <title>java.net Forums</title>
      <link>http://forums.java.net/jive</link>
      <description>List of forum topics</description>
      <language>en</language>

      <item>
         <title>OpenMarkup 2.0</title>
         <link>http://forums.java.net/jive/thread.jspa?messageID=99703</link>
         <description><![CDATA[Item description...]]></description>
         <dc:creator>rameshgupta</dc:creator>
         <jf:creationDate>Apr 1, 2006 12:00:00 AM</jf:creationDate>
         <jf:modificationDate>Apr 1, 2006 12:00:00 AM</jf:modificationDate>
         <jf:author>rameshgupta</jf:author>
         <jf:replyCount>2</jf:replyCount>
      </item>
      <!-- More item elements... -->
   </channel>
</rss>
```

Handling application-specific extensions

There are several ways to handle application-specific extensions in XML documents. If the extensions are well-defined, stable, and frequently used, it is best to extend a tag library by adding support for the extension elements and attributes. With this approach, a tag library handles custom elements and attributes just like built-in elements and attributes.

Another way to handle application-specific extensions in XML documents is to combine tag library usage with standard DOM programming. Under this approach, a content consumer has direct access to the document object model created by an object realizer. As a result, it can freely make DOM calls, while still retaining the ability to pluck domain objects from elements in the DOM tree using the `getObject` method of `OMElement`.

To show how this might work, we now look at an example in which we propagate an instance of `OMElement` from a content handler to a content consumer. We continue to exploit an RSS feed as our archetypal example of an open document format to demonstrate handling of application-specific extensions in XML documents.

Listing 10 shows a content handler class called `ChannelElementBuilder` that returns an instance of a `ChannelElement` from its `getObject` method. The only difference between this class and the `ChannelBuilder` class shown in Listing 6 is that its `getObject` method returns the `ChannelElement` node in the DOM tree rather than the `Channel` object realized from that element node. The `ChannelElementBuilder` class *knows* that the `evaluate` method of `OMDocument` will return an instance of `ChannelElement` because the `RSSVocabulary` class states that the element implementation class for an RSS `channel` element type is `ChannelElement`.

Listing 10: RSS feed with application-specific extensions

```
public class ChannelElementBuilder
    implements ContentHandler<ChannelElement, Object> {
    public ChannelElement getObject(String systemID, MediaType type, int length,
        Map<String,String> params, InputStream stream, ProgressTracker<Object> o) {
        try {
            if (type.isXML()) {
                BufferedReader   reader = new BufferedReader(
                                                    new InputStreamReader(stream));
                OMDocument document = or.getDocument(reader, systemID);
                reader.close();
                return (ChannelElement) document.evaluate(
                                                    null, "/rss/channel", NODE);
            }
        }
        catch (Exception ex) {
            System.out.println(ex);
        }
        return null;
    }

    private final static ObjectRealizer or =
                            new ObjectRealizerImpl(RSSVocabulary.get());
}
```

Since a `ChannelElement` is fundamentally a DOM `Element`, it is amenable to standard DOM programming. In addition. a `ChannelElement` may also realize a `Channel` object on demand using information in the DOM.

Combining Object Realization and DOM Programming

We now look at an example in which we combine object realization and DOM programming techniques. Listing 11 shows a content consumer that knows how to consume a `ChannelElement` object and extract domain objects, such as a list of `SyndicatedItem` objects from that element. In addition, it knows how to extract application-specific information, such as the *reply count* for a message from proprietary elements embedded inside the RSS feed.

The content consumer class, called `ChannelTableUpdater`, uses information contained in an RSS feed to update an instance of `JTable` with a basic `TableModel` containing two columns — one showing the message title and the other showing the message reply count. For clarity, we have deliberately kept the table model in this example very simple. More sophisticated implementations can bind a model directly to the underlying DOM tree and object graph.

The `assimilate` method of `ChannelTableUpdater` first obtains a list of all `item` elements in the feed by evaluating an XPath expression in the context of the `channel` element. Since RSS element types do not belong to any namespace, we pass `null` for the namespace context in the first argument to the `evaluate` method. The second argument is an XPath expression that identifies all `item` elements under the `channel` element. Finally, the third argument indicates that we are expecting a `NodeList` as the result.

Listing 11: Extracting domain-specific and application-specific information

```
public class ChannelTableUpdater
    extends AbstractContentConsumer<ChannelElement, Object> {
    public ChannelTableUpdater(JTable table) {
        this.table = table;
    }

    @Override public void assimilate(boolean cancelled, ChannelElement chnlElem) {
        if (!cancelled && chnlElem != null) {
            try {
                NodeList itemElements = (NodeList) chnlElem.evaluate(
                    null, "item", NODESET);    // evaluate “item” under chnlElem
                int rowCount = itemElements.getLength();    // number of item elements
                Object[][]   tableData = new Object[rowCount][2];    // two columns
                for (int i = 0; i < rowCount; i++) {
                    ItemElement itemElement = (ItemElement) itemElements.item(i);
                    SyndicatedItem item = itemElement.getObject();    // domain object
                    String replyCount = (String) itemElement.evaluate(
                        nsContext, "ns:replyCount/text()", STRING);
                    tableData[i][0] = item.getTitle();                // first column
                    tableData[i][1] = Integer.parseInt(replyCount);   // second column
                }
                table.setModel(new DefaultTableModel(
                    tableData, new String[]{"Item","Replies"})); // data, column names
            }
            catch (Exception ex) {
                System.out.println(getClass() + " error: " + ex);
            }
        }
    }

    private final static NamespaceContext nsContext = new NamespaceContextSupport(
        NULL_NS_URI, //Default ns URI (use NULL_NS_URI because RSS is not in any ns)
        // map additional namespace prefixes to namespace URIs, if any (in pairs):
        "ns", "http://www.jivesoftware.com/xmlns/jiveforums/rss");

    private final JTable table;
}
```

Once we obtain the list of `item` element nodes, we allocate a two dimensional array to hold references to objects in our table model. The `ChannelTableUpdater` class *knows* that the `evaluate` method of `OMElement` will return a `NodeList` of `ItemElement` nodes in this case because the `RSSVocabulary` class states that the element implementation class for an RSS `item` element type is `ItemElement`.

Note that `ChannelElement` and `ItemElement` return different types of objects from their `getObject` method. `ChannelElement` implements `OMElement<Channel>` to return a `Channel` object, but `ItemElement` implements `OMElement<SyndicatedItem>` to return a `SyndicatedItem` object. Recall that `Channel` and `SyndicatedItem` are specific to the content syndication domain, but have no intrinsic knowledge of XML in general, and RSS in particular.

While looping over the list of `ItemElement` nodes, we also want to extract application-specific information, such as the value of the `replyCount` element defined in the Jive forum namespace `http://www.jivesoftware.com/xmlns/jiveforums/rss`. We accomplish that by evaluating another XPath expression, `"ns:replyCount/text()"` — this time within the context of each `ItemElement` instance.

Since `replyCount` belongs to a specific namespace, we must specify a `NamespaceContext` in the first argument to the `evaluate` method. We set up a singleton `NamespaceContext` object, associating the arbitrarily chosen prefix `ns` with the Jive forum namespace. Once that is done, we can refer to the Jive forum namespace simply by using the prefix `ns` in the XPath expression that we pass as the second argument to the `evaluate` method. The final argument to the `evaluate` method indicates that the returned object is a `String`. The next two lines populate a row in the two-dimensional array that we use in constructing our table model. After we exit the loop, we create an instance of `DefaultTableModel` using the array we just populated. Finally, we update the table to use the new table model.

Listing 12 shows how to use `ChannelElementBuilder` and `ChannelTableUpdater` with a `ContentLoader`. Once again, notice that there are no visible hints of Java Generics in the main application code.

Listing 12: Building a table of syndicated items from an RSS feed with application-specific elements

```
public class SyndicatedItemsTable {
    public static void main(String[] args) {
        JTable table = new JTable();
        ContentLoader items = ContentLoaderFactory.createContentLoader(
            new ChannelElementBuilder(), new ChannelTableUpdater(table), null,
            "http://forums.java.net/jive/rss/rssmessages.jsp?categoryID=29");
        items.load();
        JFrame frame = new JFrame();
        JScrollPane scrollPane = new JScrollPane(table);
        frame.add(scrollPane);
        frame.setSize(600, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Conclusion

The field of XML processing is much more expansive than what this short article delves in. This article shows only a couple of ways in which content handlers and content consumers can process XML content.

Standard DOM programming is easy and straightforward, but it can also be tedious. An object realizer makes it easy to extract information from XML documents in a form that is *directly* usable by an application. In some respects, this approach is comparable to object-relational mapping in which relational data is mapped to domain objects. In this case, however, the mapping is between DOM objects (which carry information in XML documents) and domain objects (which might not know anything about XML but can encapsulate domain-specific information as well as behavior). Reusable domain-specific object realization libraries help streamline application code by allowing you to focus on application logic rather than the mechanics of converting XML documents to domain objects.