Introduction to XML Programming

Chris Panayiotou

General Purpose XML Programming

- Needed for:
  - domain-specific applications
  - implementing new generic tools
- Important components:
  - parsing XML documents into XML trees
  - navigating through XML trees
  - manipulating XML trees
  - serializing XML trees as XML documents
- There are many APIs (standards) for manipulating XML
  - Examples include SAX, DOM

What are XML APIs for?

- You want to read/write data from/to XML files, and you don't want to write an XML parser.
- Applications:
  - processing an XML-tagged corpus
  - saving configs, prefs, parameters, etc. as XML files
  - sharing results with outside users in portable format
    - example: typed dependency relations
  - alternative to serialization for persistent stores
    - doesn't break with changes to class definition
    - human-readable

Overview of JAXP

- JAXP = Java API for XML Processing
  - Provides a common interface for creating and using the standard SAX, DOM, and XSLT APIs in Java.
  - All JAXP packages are included standard in JDK 1.4+. The key packages are:

javax.xml.parsers	The main JAXP APIs, which provide a common interface for various SAX and DOM parsers.
org.w3c.dom	Defines the Document class (a DOM), as well as classes for all of the components of a DOM.
org.xml.sax	Defines the basic SAX APIs.
javax.xml.transform	Defines the XSLT APIs that let you transform XML into other forms. (Not covered today.)

JAXP XML Parsers

- javax.xml.parsers defines abstract classes DocumentBuilder (for DOM) and SAXParser (for SAX).
  - It also defines factory classes DocumentBuilderFactory and SAXParserFactory. By default, these give you the "reference implementation" of DocumentBuilder and SAXParser, but they are intended to be vendor-neutral factory classes, so that you could swap in a different implementation if you preferred.
- The JDK includes three XML parser implementations from Apache:
  - Crimson: The original. Small and fast. Based on code donated to Apache by Sun. Standard implementation for J2SE 1.4.
  - Xerces: More features. Supports XML Schema. Based on code donated to Apache by IBM.
  - Xerces 2: The future. Standard implementation for J2SE 5.0.

SAX vs. DOM

- Java-specific
- interprets XML as a stream of events
- you supply event-handling callbacks
- SAX parser invokes your event-handlers as it parses
- doesn't build data model in memory
- serial access
- very fast, lightweight
- good choice when
  - no data model is needed, or
  - natural structure for data model is list, matrix, etc.

- W3C standard for representing structured documents
- platform and language neutral (not Java-specific!)
- interprets XML as a tree of nodes
- builds data model in memory
- enables random access to data
- therefore good for interactive apps
- more CPU- and memory-intensive
- good choice when data model has natural tree structure

SAX = Simple API for XML

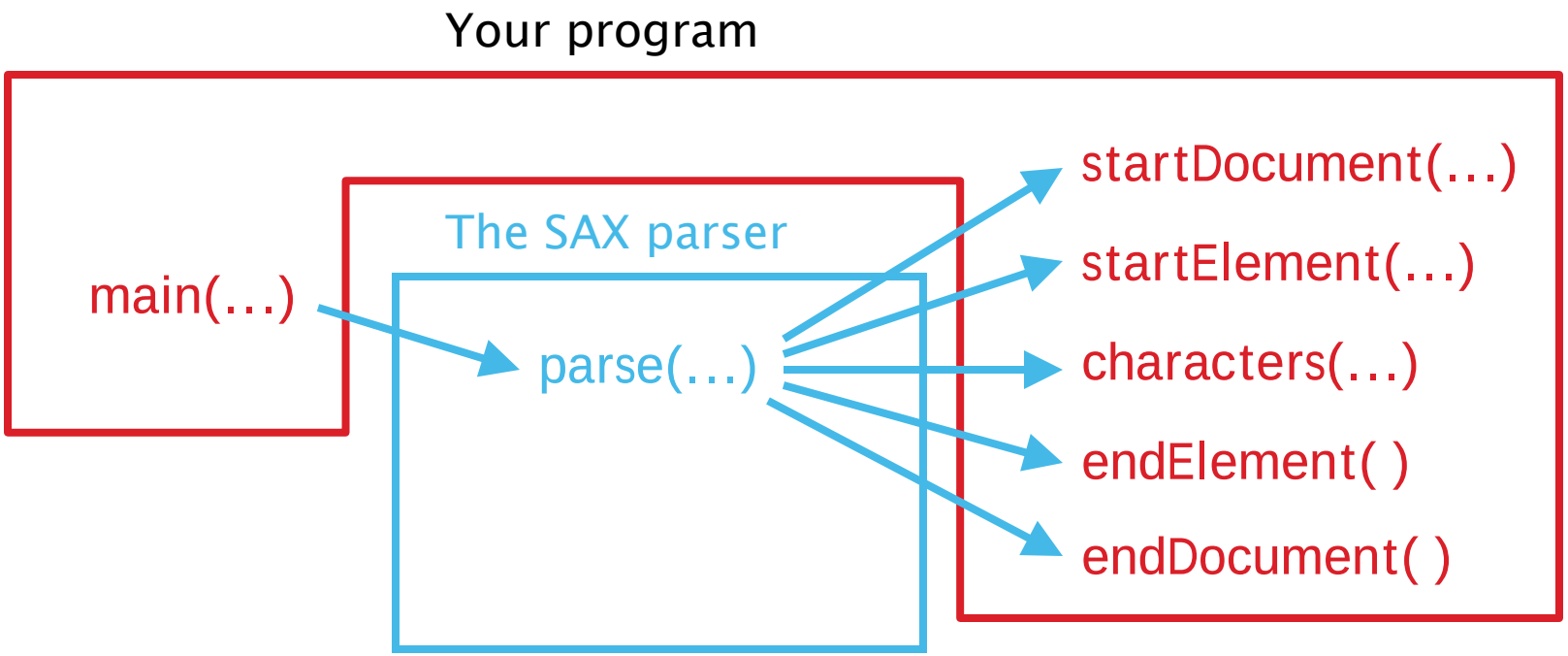
DOM = Document Object Model

SAX vs. DOM

- DOM reads the entire XML document into memory and stores it as a tree data structure
- SAX reads the XML document and calls one of your methods for each element or block of text that it encounters
- Consequences:
  - DOM provides “random access” into the XML document
  - SAX provides only sequential access to the XML document
  - DOM is slow and requires huge amounts of memory, so it cannot be used for large XML documents
  - SAX is fast and requires very little memory, so it can be used for huge documents (or large numbers of documents)
    - This makes SAX much more popular for web sites
  - Some DOM implementations have methods for changing the XML document in memory; SAX implementations do not

Callbacks

- SAX works through callbacks: you call the parser, it calls methods that you supply



Using SAX

- Here's the standard recipe for starting with SAX:

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

// get a SAXParser object
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();

// invoke parser using your custom content handler
saxParser.parse(inputStream, myContentHandler);
saxParser.parse(file, myContentHandler);
saxParser.parse(url, myContentHandler);
```

- (This reflects SAX 1, which you can still use, but SAX 2 prefers a new incantation…)

Using SAX 2

- In SAX 2, the following usage is preferred:

```
// tell SAX which XML parser you want (here, it's Crimson)
System.setProperty("org.xml.sax.driver",
    "org.apache.crimson.parser.XMLReaderImpl");

// get an XMLReader object
XMLReader reader = XMLReaderFactory.createXMLReader();

// tell the XMLReader to use your custom content handler
reader.setContentHandler(myContentHandler);

// Have the XMLReader parse input from Reader myReader:
reader.parse(new InputSource(myReader));
```

- myContentHandler is class that you should write

Defining a ContentHandler

- Easiest route: define a new class which extends org.xml.sax.helpers.DefaultHandler.
- Override event-handling methods from DefaultHandler:

```
startDocument()     // receive notice of start of document
endDocument()       // receive notice of end of document
startElement()      // receive notice of start of each element
endElement()        // receive notice of end of each element

characters()        // receive a chunk of character data
error()             // receive notice of recoverable parser error
                    // ...plus more...
```

startElement()and endElement()

- The SAXParser invokes your callbacks to notify you of events:

```
startElement(String namespaceURI,        // for use w/ namespaces
             String localName,           // for use w/ namespaces
             String qName,               // "qualified" name -- use this one!
             Attributes atts)

endElement(String namespaceURI,
           String localName,
           String qName)
```

- For simple usage, ignore namespaceURI and localName, and just use qName (the “qualified” name).
- XML namespaces are described in an appendix, below.
- startElement() and endElement() events *always* come in pairs:
  - “<foo/>” will generate calls:

```
startElement("", "", "foo", null)
endElement("", "", "foo“)
```

SAX Attributes

- Every call to startElement() includes an Attributes object which represents all the XML attributes for that element.
- Methods in the Attributes interface:

```
getLength()                    // return number of attributes
getIndex(String qName)         // look up attribute's index by qName
getValue(String qName)         // look up attribute's value by qName
getValue(int index)            // look up attribute's value by index
                               // ... and others …
```

SAX characters()

- The characters() event handler receives notification of character data (i.e. content that is not part of an XML element):

```
public void characters(char[] ch,     // buffer containing chars
                       int start,     // start position in buffer
                       int length)    // num of chars to read
```

- May be called multiple times within each block of character data—for example, once per line.
- So, you may want to use calls to characters() to accumulate characters in a StringBuffer, and stop accumulating at the next call to startElement().

Simple SAX program

- The following program is adapted from CodeNotes® for XML by Gregory Brill, pages 158-159
- The program consists of two classes:
  - Sample -- This class contains the main method; it
    - Gets a factory to make parsers
    - Gets a parser from the factory
    - Creates a Handler object to handle callbacks from the parser
    - Tells the parser which handler to send its callbacks to
    - Reads and parses the input XML file
  - Handler -- This class contains handlers for three kinds of callbacks:
    - startElement callbacks, generated when a start tag is seen
    - endElement callbacks, generated when an end tag is seen
    - characters callbacks, generated for the contents of an element

The Sample class

```
import javax.xml.parsers.*;   // for both SAX and DOM
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class Sample {
    public static void main(String args[]) {
        // Create a parser factory
        SAXParserFactory factory = SAXParserFactory.newInstance();
        // Tell factory that the parser must understand namespaces
        factory.setNamespaceAware(true);
        // Make the parser
         SAXParser saxParser = factory.newSAXParser();
         XMLReader parser = saxParser.getXMLReader();
        // Create a handler and tell the parser to use it
         parser.setContentHandler(new Handler());
         // Finally, read and parse the document
         parser.parse("hello.xml");
    }
```

The Handler class

```
public class Handler extends DefaultHandler {
// DefaultHandler is an adapter class that defines these methods and others as do-nothing
// methods, to be overridden as desired. We will define three very similar methods to
// handle (1) start tags, (2) contents, and (3) end tags--our methods will just print a line

    // SAX calls this method when it encounters a start tag
    public void startElement(String namespaceURI,  String localName, String qualifiedName,
            Attributes attr) throws SAXException {
        System.out.println("startElement: " + qualifiedName);
    }
   // SAX calls this method to pass in character data
    public void characters(char ch[], int start, int length)  throws SAXException {
        System.out.println("characters: \"" + new String(ch, start, length) + "\"");
    }
    // SAX call this method when it encounters an end tag
    public void endElement(String nsURI, String lName, String qName)  throws SAXException {
        System.out.println("Element: /" + qName);
    }
}
```

Results

- If the file hello.xml contains:

```
<?xml version="1.0"?>
<display>Hello World!</display>
```

- Then the output from running java Sample will be:

```
startElement: display
characters: "Hello World!"
Element: /display
```

More results

```
<?xml version="1.0"?>
<display>
   <i>Hello</i> World!
</display>
```

- Notice that the root element, <display>, now contains a nested element <i> and some whitespace (including newlines)
- The result will be as shown at the right:

```
startElement: display
characters: ""          // empty String
characters: "
"                       // new line
characters: "      "    // spaces
startElement: i
characters: "Hello"
endElement: /i
characters:  "World!"
characters: "
“                       // new line
endElement: /display
```

Hello.xml

Result

Example Notes: Parser factories

- A factory is an alternative to constructors
- To create a SAX parser factory, call this method:
  SAXParserFactory.newInstance()
  - This returns an object of type SAXParserFactory
  - It may throw a FactoryConfigurationError
- You can then say what kind of parser you want:
  - public void setNamespaceAware(boolean awareness)
    - Used if you are using namespaces
    - The default is false
  - public void setValidating(boolean validating)
    - Used if you want to validate against a DTD
    - The default is false
    - Validation will give an error if you don't have a DTD

Example Notes: Getting a parser

- Once you have a SAXParserFactory set up you can create a parser with:

```
SAXParser saxParser = factory.newSAXParser();
XMLReader parser = saxParser.getXMLReader();
```

- Older implementation may use Parser instead of XMLReader
  - Parser is SAX1, not SAX2, and is now deprecated
  - SAX2 supports namespaces and some new parser properties
- SAXParser is not thread-safe
  - To use it in multiple threads, create a separate SAXParser for each thread

Example Notes: Declaring which handler to use

- Since the SAX parser will be calling our methods, we need to supply these methods
- In the example these are in a separate class, *Handler*
- We need to tell the parser where to find the methods:

```
parser.setContentHandler(new Handler());
```

- Finally, we call the parser and tell it what file to parse:

```
parser.parse("hello.xml");
```

- Everything else will be done in the handler methods

SAX handlers

- A callback handler for SAX must implement these four interfaces:
  - interface ContentHandler
    - This is the most important interface--it handles basic parsing callbacks, such as element starts and ends
  - interface DTDHandler
    - Handles *only* notation and unparsed entity declarations
  - interface EntityResolver
    - Does customized handling for external entities
  - interface ErrorHandler
    - Must be implemented or parsing errors will be *ignored!*
- You could implement all these interfaces yourself, but that's a lot of work--it's easier to use an adapter class

Class DefaultHandler

- As already mentioned the easiest way to create a SAX handler is to extend the class DefaultHandler
- DefaultHandler is in package org.xml.sax.helpers
- DefaultHandler implements ContentHandler, DTDHandler, EntityResolver, and ErrorHandler
- DefaultHandler is an *adapter class*--it provides *empty* methods for every method declared in each of the four interfaces
- To use this class, *extend* it and *override* the methods that are important to your application
  - We already covered the most basic methods
  - You can find more methods in the methods in the ContentHandler and ErrorHandler interfaces

Whitespace

- Whitespace is a major nuisance
  - Whitespace is characters; characters are PCDATA
  - If you are validating, the parser will ignore whitespace where PCDATA is not allowed by the DTD
  - If you are not validating, the parser cannot ignore whitespace
  - If you ignore whitespace, you lose your indentation
- To ignore whitespace
  - When validating happens automatically
  - When not validating use the String function trim() to remove whitespace and then check the result to see if it is the empty string

Handling ignorable whitespace

- A *nonvalidating* parser cannot ignore whitespace, because it cannot distinguish it from real data
- A *validating* parser can, and *does*, ignore whitespace where character data is not allowed
  - For processing XML, this is usually what you want
  - However, if you are manipulating and *writing out* XML, discarding whitespace ruins your indentation
  - To capture ignorable whitespace, you can override this method (defined in DefaultHandler):

```
public void ignorableWhitespace(char[] ch, int start, int length)
  throws SAXException
```

  - Parameters are the same as those for characters

Error Handling with SAX

- SAX error handling is unusual
- Most errors are *ignored* unless you register an *error handler* (org.xml.sax.ErrorHandler)
  - Ignored errors can cause bizarre behavior
  - Failing to provide an error handler is unwise
- The ErrorHandler interface has the following methods:
  - public void fatalError (SAXParseException exception)
    throws SAXException // XML not well structured
  - public void error (SAXParseException exception)
    throws SAXException // XML validation error
  - public void warning (SAXParseException exception)
    throws SAXException // minor problem

Error Handling with SAX

- If you are extending DefaultHandler, it implements ErrorHandler and registers itself
  - DefaultHandler's version of fatalError() throws a SAXException, but...
  - its error() and warning() methods do nothing!
- You should override these methods
- Note that the only kind of exception your override methods can throw is a SAXException
  - When you override a method, you cannot *add* exception types
  - If you need to throw another kind of exception, say an IOException, you can *encapsulate* it in a SAXException:

```
catch (IOException ioException) {
    throw new SAXException("I/O error: ", ioException)
}
```

Error Handling with SAX

- If you are *not* extending DefaultHandler:
  - Create a new class (e.g. MyErrorHandler) that implements ErrorHandler
  - Create a new object of this class
  - Tell your XMLReader object about it by sending calling the method *setErrorHandler(ErrorHandler handler)*
- Example:

```
XMLReader parser = saxParser.getXMLReader();
parser.setErrorHandler(new MyErrorHandler());
```

DOM: What is it?

- An object-based, language-neutral API for XML and HTML documents
  - allows programs and scripts to build documents, navigate their structure, add, modify or delete elements and content
  - Provides a foundation for developing querying, filtering, transformation, rendering etc. applications on top of DOM implementations
- Based on OO concepts:
  - methods – to access or change object's state)
  - interfaces – declaration of a set of methods
  - objects – encapsulation of data and methods
- Roughly similar to the XSLT/XPath data model
  - Tree-like structure implied by the abstract relationships defined by the programming interfaces
- Essentially it allows treating XML documents as trees comprised of nodes

A simple DOM program

- This program is adapted from CodeNotes® for XML by Gregory Brill, page 128

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class SimpleDom {
    public static void main(String args[]) {
        try {
            // Create a DOM parser
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            // Load an XML file for parsing
            Document document = builder.parse("hello.xml");
            // Find the content of the root element and prints it
            Element root = document.getDocumentElement();
            Node textNode = root.getFirstChild();
            System.out.println(textNode.getNodeValue());
        } catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
}
```

Reading in the tree

- The parse method reads in the entire XML document and represents it as a tree in memory
  - For a large document, parsing could take a while
  - If you want to interact with your program while it is parsing, you need to parse in a separate thread
    - Once parsing starts, you cannot interrupt or stop it
    - Do not try to access the parse tree until parsing is done
- An XML parse tree may require up to ten times as much memory as the original XML document
  - If you have a lot of tree manipulation to do, DOM is much more convenient than SAX
  - If you don't have a lot of tree manipulation to do, consider using SAX instead

Structure of the DOM tree

- The DOM tree is composed of Node objects
- Node is an interface
  - Some of the more important subinterfaces are Element, Attr, and Text
    - An Element node may have children
    - Attr and Text nodes are leaves
  - Additional types are Document, ProcessingInstruction, Comment, Entity, CDATASection and several others
- Hence, the DOM tree is composed entirely of Node objects, but the *Node objects* can be downcast into more specific types as needed

Methods of Node objects

- The results returned by getNodeName(), getNodeValue(), getNodeType() and getAttributes() depend on the subtype of the node, as follows
  - Tip: You can use switch to easily tell what kind of a node you are dealing
  - switch(node.getNodeType()) {
    case Node.ELEMENT_NODE:
    Element element = (Element)node; ...; break;
    case Node.TEXT_NODE:
    Text text = (Text)node; … break;
    case Node.ATTRIBUTE_NODE:
    Attr attr = (Attr)node; … break;
    default: …
    }

	Element	Text	Attr
getNodeName()	*tag name*	"#text"	*name of attribute*
getNodeValue()	null	*text contents*	*value of attribute*
getNodeType()	ELEMENT_NODE	TEXT_NODE	ATTRIBUTE_NODE
getAttributes()	NamedNodeMap	null	null

Methods of Node objects

- Tree-walking operations that return a Node:
  - getParentNode()
  - getFirstChild()
  - getNextSibling()
  - getPreviousSibling()
  - getLastChild()

- Tests that return a boolean:
  - hasAttributes()
  - hasChildNodes()

Methods of Element objects

- String getTagName()
  - Returns the name of the tag
- boolean hasAttribute(String name)
  - Returns true if this Element has the named attribute
- String getAttribute(String name)
  - Returns the (String) value of the named attribute
- boolean hasAttributes()
  - Returns true if this Element has any attributes
  - This method is actually inherited from Node
    - Returns false if it is applied to a Node that isn't an Element
- NamedNodeMap getAttributes()
  - Returns a NamedNodeMap of all the Element's attributes
  - This method is actually inherited from Node
    - Returns null if it is applied to a Node that isn't an Element

NamedNodeMap

- The node.getAttributes() method returns a NamedNodeMap
  - Because NamedNodeMaps are used for other kinds of nodes (elsewhere in Java), the contents are treated as general Nodes, not specifically as Attrs
- Some methods of NamedNodeMap are:
  - getNamedItem(String name) returns (as a Node) the attribute with the given name
  - getLength() returns (as an int) the number of Nodes in this NamedNodeMap
  - item(int index) returns (as a Node) the n^{th} item
    - This operation lets you conveniently step through all the nodes in the NamedNodeMap
    - Java does not guarantee the order in which nodes are returned

Methods of Text objects

- Text is a subinterface of CharacterData and inherits the following methods (among others):
  - public String getData() throws DOMException
    - Returns the text contents of this Text node
  - public int getLength()
    - Returns the number of Unicode characters in the text
  - public String substringData(int offset, int count) throws DOMException
    - Returns a substring of the text contents

Methods of Attr objects

- String getName()
  - Returns the name of this attribute.
- Element getOwnerElement()
  - Returns the Element node this attribute is attached to, or null if this attribute is not in use
- boolean getSpecified()
  - Returns true if this attribute was explicitly given a value in the original document
- String getValue()
  - Returns the value of the attribute as a String

Preorder traversal

- The DOM is stored in memory as a tree
- An easy way to traverse a tree is in preorder
  - That is we first visit the root and then traverse each subtree, in order

```
static void simplePreorderPrint(String indent, Node node) {
        printNode(indent, node);
        if(node.hasChildNodes()) {
            Node child = node.getFirstChild();
            while (child != null) {
                simplePreorderPrint(indent + "  ", child);
                child = child.getNextSibling();
            }
        }
    }
static void printNode(String indent, Node node) {
        System.out.print(indent);
        System.out.print(node.getNodeType() + " ");
        System.out.print(node.getNodeName() + " ");
        System.out.print(node.getNodeValue() + " ");
        System.out.println(node.getAttributes());
    }
```

Trying out the program

```
<?xml version="1.0"?>
<novel>
  <chapter num="1">The Beginning</chapter>
  <chapter num="2">The Middle</chapter>
  <chapter num="3">The End</chapter>
</novel>
```

- Things to think about:
  - What are the numbers?
  - Are the nulls in the right places?
  - Is the indentation as expected?
  - How could this program be improved?

```
1 novel null
  3 #text
   null
  1 chapter null  num="1"
    3 #text The Beginning null
  3 #text
   null
  1 chapter null  num="2"
    3 #text The Middle null
  3 #text
   null
  1 chapter null  num="3"
    3 #text The End null
  3 #text
 null
```

Input

Output

Additional DOM methods

- There are some methods that allow you to modify the DOM tree, for example:
  - setNodeValue(String nodeValue)
  - insertBefore(Node newChild, Node refChild)
- Java provides a large number of these operations
- These operations are not part of the W3C specifications
- There is no standardized way to write out a DOM as an XML document
  - It isn't that hard to write out the XML
  - The previous program is a good start on outputting XML