SAX & DOM

CPS 116
Introduction to Database Systems

Announcements (October 30)

- Homework #3 due in one week
- Project milestone #2 due in 1½ week

SAX & DOM

- Both are API's for XML processing
- SAX (Simple API for XML)
  - Started out as a Java API, but now exists for other languages too
- DOM (Document Object Model)
  - Language-neutral API with implementations in Java, C++, etc.
- ☞ JAXP (Java API for XML Processing)
  - Bundled with standard JDK
  - Includes SAX, DOM parsers and XSLT transformers

4

SAX processing model

- Serial access
  - XML document is processed as a stream
  - Only one look at the data
  - Cannot go back to an early portion of the document
- Event-driven
  - A parser generates events as it goes through the document (e.g., start of the document, end of an element, etc.)
  - Application defines event handlers that get invoked when events are generated

5

SAX events

Most frequently used events:

- `startDocument`
- `endDocument`
- `startElement`
- `endElement`
- `characters`
  - Whenever the parser has processed a chunk of character data (without generating other kinds of events)
  - Warning: The parser may generate multiple `characters` events for one piece of text

Whitespace may come up as `characters` or `ignorableWhitespace`, depending on whether a DTD is present

6

A simple SAX example

- Print out text contents of `title` elements

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

public class SaxExample extends DefaultHandler {
  public static void main(String[] argv) throws Exception {
    String fileName = argv[0];
    // Create a SAX parser:
    XMLReader xr = XMLReaderFactory.createXMLReader();
    // Parse the document with this event handler:
    xr.setContentHandler(new SaxExample());
    xr.parse(new InputSource(new FileReader(fileName)));
    return;
  }
  … …
```

A simple SAX example (cont'd)

7

Only relevant when namespace is involved

Assuming no namespace processing, `qname` is tag name

```
private StringBuffer titleStringBuffer = null;
public void startElement(String uri, String localName,
                         String qName,
                         Attributes attributes) {
  if (qName.equals("title"))
    titleStringBuffer = new StringBuffer();
}
public void endElement(String uri, String localName,
                       String qName) {
  if (qName.equals("title")) {
    System.out.println(titleStringBuffer.toString());
    titleStringBuffer = null;
  }
}
public void characters(char[] ch, int start, int length) {
  if (titleStringBuffer != null)
    titleStringBuffer.append(ch, start, length);
}
```

Warning: This code does not handle data with `//title[//title]` pattern

A common mistake

8

What is wrong with the following?

```
private String titleString = null;
public void endElement(String uri, String localName,
                       String qName) {
  // Print the last chunk of characters seen before </title>:
  if (qName.equals("title"))
    System.out.println(titleString);
}
public void characters(char[] ch, int start, int length) {
  titleString = new String(ch, start, length);
}
```

A more complex SAX example

9

- Print out the text contents of top-level section titles in books, i.e., `//book/section/title`
  - Old code would print out all titles, e.g., `//book/title`, `//book//section/title`
  - For simplicity, assume that if we have the pattern `//book/section/title//book/section/title`, we print the higher-level `title` element
- Idea: maintain as state the path from the root

```
private ArrayList path = new ArrayList();
private int pathLengthWhenOutputIsActivated;
```

A more complex SAX example (cont'd)

10

```
public void startElement(String uri, String localName,
                         String qName,
                         Attributes attributes) {
  path.add(qName); // Maintain the path.
  if (path.size() >= 3 &&
      ((String)(path.get(path.size()-1))).equals("title") &&
      ((String)(path.get(path.size()-2))).equals("section") &&
      ((String)(path.get(path.size()-3))).equals("book")) {
    // path matches //book/section/title:
    if (titleStringBuffer == null) {
      pathLengthWhenOutputIsActivated = path.size();
      titleStringBuffer = new StringBuffer();
    }
  }
}
```

A more complex SAX example (cont'd)

11

```
public void endElement(String uri, String localName,
                       String qName) {
  if (titleStringBuffer != null &&
      path.size() == pathLengthWhenOutputIsActivated) {
    // Closing the element that activated output buffering:
    System.out.println(titleStringBuffer.toString());
    titleStringBuffer = null;
  }
  path.remove(path.size()-1); // Maintain the path.
}
public void characters(char[] ch, int start, int length) {
  if (titleStringBuffer != null)
    titleStringBuffer.append(ch, start, length);
}
```

This check prevents premature output in case that `title` has subelements

Would it work if we change this check to `qName.equals("title")`?

DOM processing model

12

- XML is parsed by a parser and converted into an in-memory DOM tree
- DOM API allows an application to
  - Construct a DOM tree from an XML document
  - Traverse and read a DOM tree
  - Construct a new, empty DOM tree from scratch
  - Modify an existing DOM tree
  - Copy subtrees from one DOM tree to another

  etc.

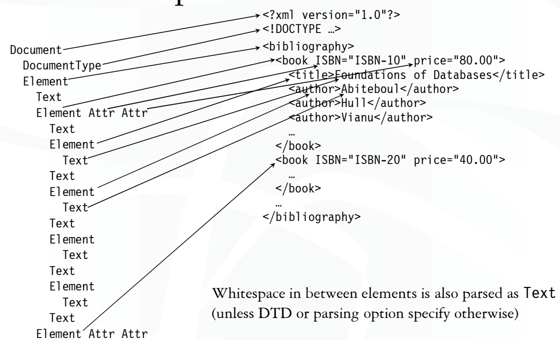
DOM Node's

13

- A DOM tree is made up of Node's
- Most frequently used types of Node's:
  - Document: root of the DOM tree
    - Not the sames as the root element of XML
  - DocumentType: corresponds to the DOCTYPE declaration in an XML document
  - Element: corresponds to an XML element
  - Attr: corresponds to an attribute of an XML element
  - Text: corresponds to chunk of text

DOM example

14



Whitespace in between elements is also parsed as Text (unless DTD or parsing option specify otherwise)

Node interface

15

n.getNodeType() returns the type of Node n

n.getChildNodes() returns a NodeList containing Node n's children

- For example, subelements are children of an Element; DocumentType is a child of the Document

d.getDocumentElement() returns the root Element of Document d

e.getNodeName() returns the tag name of Element e

e.getAttributes() returns a NamedNodeMap (hash table) containing the attributes of Element e

- Attributes are not considered children!

a.getNodeName() returns the name of Attr a

a.getNodeValue() returns the value of Attr a

t.getNodeValue() returns the content of Text t

For convenience: n.getParentNode(), n.getPreviousSibling(), n.getNextSibling(), n.getOwnerDocument(), etc.

Constructing DOM from XML


```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class DomExample {
 public static void main(String[] argv) throws Exception {
  // Parse input XML into a DOM Document:
  DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
  DocumentBuilder builder=factory.newDocumentBuilder();
  Document document=builder.parse(new File(argv[0]));
  // Use the default (identity) Transformer to print the DOM Document:
  TransformerFactory tFactory=TransformerFactory.newInstance();
  Transformer transformer=tFactory.newTransformer();
  transformer.transform(new DOMSource(document),
                        new StreamResult(System.out));
 }
}
```

In general, you can use an XSLT `Transformer` instead

Traversing DOM

- Compute the string value of an XML node

```
public static String convertNodeToString(Node n) {
  // String value of a Text Node is just its content:
  if (n.getNodeType() == Node.TEXT_NODE)
    return n.getNodeValue();
  // String value of a Node of another type is the concatenation
  // of its children's string values:
  String text = "";
  NodeList children = n.getChildNodes();
  for (int i=0; i<children.getLength(); i++) {
    Node child = children.item(i);
    text = text + convertNodeToString(child);
  }
  return text;
}
```

Traversing DOM (cont'd)

- Print out text contents of `title` elements

```
public static void outputTitle(Node n) {
  if (n.getNodeType() == Node.ELEMENT_NODE &&
      n.getNodeName().equals("title")) {
    // This is a title Element; output it:
    System.out.println(convertNodeToString(n));
  } else {
    // Recurse down the tree and look for titles to output:
    NodeList children = n.getChildNodes();
    for (int i=0; i<children.getLength(); i++) {
      Node child = children.item(i);
      outputTitle(child);
    }
  }
}
```

- How would you print out just `//book/section/title`?
  - Use `getParentNode()` to check for `section` parent and `book` grandparent

Constructing DOM from scratch

19

- Construct a DOM `Document` showing all titles as follows:

```
<result>
  <title text="title1"/>
  <title text="title2"/>…
</result>
```

```
public static Document newDocWithTitles(Document inputDoc)
throws Exception {
  // Create a new Document:
  DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
  DocumentBuilder builder=factory.newDocumentBuilder();
  Document newDoc=builder.newDocument();
  // Create the root Element:
  Element newElement=newDoc.createElement("result");
  newDoc.appendChild(newElement);
  // Add titles:
  addTitlesToNewDoc(newDoc, inputDoc);
  return newDoc;
}
```

Constructing DOM from scratch (cont'd)

20

```
public static void addTitlesToNewDoc(Document newDoc, Node n)
throws Exception {
 if (n.getNodeType() == Node.ELEMENT_NODE &&
     n.getNodeName().equals("title")) {
    // This is a title Element; add it:
    Element newElement = newDoc.createElement("title");
    newElement.setAttribute("text", convertNodeToString(n));
    newDoc.getDocumentElement().appendChild(newElement);
  } else {
    // Recurse down the tree and look for titles to add:
    NodeList children = n.getChildNodes();
    for (int i=0; i<children.getLength(); i++) {
      Node child = children.item(i);
      addTitlesToNewDoc(newDoc, child);
    }
  }
}
```

You can only create an `Element` within a `Document`

Copying subtrees in DOM

21

- Construct a DOM `Document` showing all title elements from the input XML

```
public static Document newDocWithTitles2(Document inputDoc)
throws Exception {
  … …
  // Add titles:
  addTitlesToNewDoc2(newDoc, inputDoc);
  … …
}
public static void addTitlesToNewDoc2(Document newDoc, Node n)
throws Exception {
 if (n.getNodeType() == Node.ELEMENT_NODE &&
     n.getNodeName().equals("title")) {
    Node newNode = newDoc.importNode(n, true);
    newDoc.getDocumentElement().appendChild(newNode);
  } else {
    … …
  }
}
```

A `Document` can import (copy) a `Node` from another element
The second argument specifies whether to copy recursively or not

22

Summary: SAX versus DOM

- SAX
  - Because of one-pass processing, a SAX parser is fast, consumes very little memory
  - Applications are responsible for keeping necessary state in memory, and are therefore more difficult to code
- DOM
  - Because the input XML needs to be converted to an in-memory DOM-tree representation, a DOM parser consumes more memory
    - Lazy materialization of DOM tree helps alleviate this problem
  - Applications are easier to develop because of the powerful DOM interface
- Which one scales better for huge XML input?