Chapter 4

Converting Flat Files to XML

Relatively little of the world's data is currently stored in XML. Much of it is stored in flat files as tab-delimited text, comma-separated values, or some similar format. More is locked up in databases of one kind or another, whether relational, hierarchical, or object based. Even more is hidden inside unstructured documents, including Microsoft Word files, HTML documents, and plain text. XML tools are not suitable for working with any of this.

There are no magic bullets that will convert all of your data to semantically tagged XML. There are a few specialized programs that convert certain formats such as Word documents to particular XML applications such as XHTML. However, the output from even the best of these tools often needs to be cleaned up by hand. How much clean-up work you need to do generally depends on how structured the data format is to start with and how clean the data is. It's relatively easy to encode a relational table from a DB2 database as XML because it already has a lot of structure and a mandatory schema. It's a lot harder in practice to convert tab-delimited text files because those tend to be full of mistakes and dirty data. Records are missing fields. Fields get swapped with each other. A field that is supposed to contain a number between 1 and 12 may contain a list of foodstuffs the data entry clerk was supposed to buy on his way home one day. All of these things can and do happen, and you have to account for them regardless of what you're doing with such data, whether that's converting it to XML or summarizing it for an annual report.

When you're tasked with converting legacy data to XML, you just have to roll up your sleeves and attack the problem. You need to understand the current structure of the data. You need to write a program that reads the input format and writes out XML. You need to debug the inevitable problems that arise when the data in the input isn't exactly you thought it was or what it was supposed to be. By far the hardest part of this problem is parsing the input data, in whatever form it takes. Once you've loaded the data into your program, writing it back out again in XML is a cakewalk.

■ The Budget

As an example of this process, I'm going to use U.S. federal government budget authorization data, which the Office of Management and Budget (OMB) publishes in a variety of equivalent flat formats, even though the data itself is relatively unflat. This is a good example of the sort of legacy data developers often have to deal with. The complete document *[http://w3.access.gpo.gov/usbudget/fFY2002/db.html]* consists of 3,185 line items. Each line item consists of 43 separate fields. In the comma-separated values (CSV) version of the file, a typical line item looks like this:

```
"418","National Endowment for the Humanities","00","National Endow-
ment for the Humanities","0200","National Endowment for the Humani-
ties: grants and administration","59","503","Research and general
education  aids","Discretionary","On-budget",  0,  0,  0,  121275,
145231, 150100, 151299, 130560, 135447, 140118, 139478, 132582,
138890, 140435, 153000, 156910, 170002, 175955, 177413, 177491,
172000, 110000, 110000, 111000, 112000, 115000, 120000, 121000,
124000, 126000, 129000, 132000
```

Each field is separated from the following field by a comma. Strings are enclosed in double quotes, and may contain commas that do not delimit fields. Dollar amounts are expressed as integers divided by 1,000. That is, the last value in the above line is 132 million dollars, not 132 thousand dollars. Table 4.1 identifies the 43 separate fields.

Table 4.1 Public Budget Database Field Descriptions

Field Number	Field Name	Description
1	Agency code	A unique, three-digit, numerical code for the cabinet department or independent agency.
2	Agency name	The name of the cabinet-level department (e.g., Department of Defense) or independent agency (e.g., Peace Corps). Even though only the executive branch has true agencies, for purposes of the budget, offices within the legislative branch are given the agency name "Legislative Branch" and offices within the judicial branch are given the agency name "Judicial Branch." Agency names have a maximum of 89 characters.
3	Bureau code	A two-digit numerical code for the bureau within the cabinet department or independent agency; bureau codes are unique only within an agency.
4	Bureau name	The name of the suboffice within the agency (e.g., Coast Guard or Federal Aviation Administration). Budget amounts for the agency as a whole that are not part of a specific bureau are generally categorized under a fictional bureau with the same name as the agency. Bureau names have a maximum of 89 characters.
5	Account code	A four-digit code (outlays) or six-digit code (offsetting receipts) for the account within the bureau.
6	Account name	The name of the budgeted function within the bureau. Account names have a maximum of 160 characters.
7	Treasury Agency code	A two-digit numerical code for the agency, assigned by the Treasury Department.
8	Subfunction code	A three-digit numerical code for the subfunction within an account.
9	Subfunction title	The name of the subfunction within the account. Subfunction titles have a maximum of 72 characters.

(continued)

Table 4.1 *continued*

Field Number	Field Name	Description
10	BEA category	Budget Enforcement Act category: “Mandatory,” “Discretionary,” or “Net interest.”
11	On- and off-budget indicator	“On-budget” or “Off-budget”; Social Security trust funds and the Postal Service are off-budget; all other accounts are on-budget.
12	1976 value	Actual amounts, in thousands of dollars, for fiscal year (FY) 1976. Budget authority is usually shown as a positive value. Offsetting receipts are usually negative values.
13	TQ value	Actual amount, in thousands of dollars, for the “transitional quarter” in 1976 when the government shifted the start of its fiscal year ahead from July to October.
14–37	1977–2000	Actual amounts, in thousands of dollars, for each fiscal year from 1977 to 2000.
38–end	2001–2006	Estimated amounts, in thousands of dollars, for FY 2001 through FY 2006.

Source: Adapted from U.S. Office of Management and Budget. Budget Analysis Branch. *Public Budget Database User's Guide: Budget of the United States Government, Fiscal Year 2002 [http://a257.gakamaitech.net/7/257/2422/09apr20010800/www.gpo.gov/usbudget/FY2002/pdf/db_guide.pdf]*, April 9, 2001, 8.

The Model

Now that we've inspected the existing format, the next step is to design the Java data structures that will hold this information inside the program after it is parsed. Sometimes you'll want to design custom Java classes that closely represent the data. If I were writing a budget analysis program, I might do that here. But because we're not planning to do anything more complex than write the data back out again as XML, the generic data structures in the Java Collections API will more than suffice.

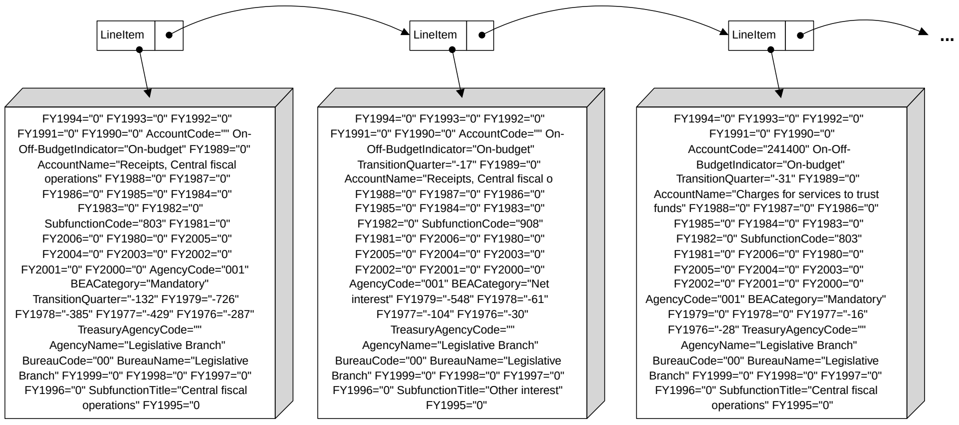
LineItem
LineItem
LineItem
...
FY1994="0" FY1993="0" FY1992="0" FY1991="0" FY1990="0" AccountCode="" On-Off-BudgetIndicator="On-budget" FY1989="0" AccountName="Receipts, Central fiscal operations" FY1988="0" FY1987="0" FY1986="0" FY1985="0" FY1984="0" FY1983="0" FY1982="0" SubfunctionCode="803" FY1981="0" FY2006="0" FY1980="0" FY2005="0" FY2004="0" FY2003="0" FY2002="0" FY2001="0" FY2000="0" AgencyCode="001" BEACategory="Mandatory" TransitionQuarter="-132" FY1979="-726" FY1978="-385" FY1977="-429" FY1976="-287" TreasuryAgencyCode="" AgencyName="Legislative Branch" BureauCode="00" BureauName="Legislative Branch" FY1999="0" FY1998="0" FY1997="0" FY1996="0" SubfunctionTitle="Central fiscal operations" FY1995="0
FY1994="0" FY1993="0" FY1992="0" FY1991="0" FY1990="0" AccountCode="" On-Off-BudgetIndicator="On-budget" TransitionQuarter="-17" FY1989="0" AccountName="Receipts, Central fiscal o FY1988="0" FY1987="0" FY1986="0" FY1985="0" FY1984="0" FY1983="0" FY1982="0" SubfunctionCode="908" FY1981="0" FY2006="0" FY1980="0" FY2005="0" FY2004="0" FY2003="0" FY2002="0" FY2001="0" FY2000="0" AgencyCode="001" BEACategory="Net interest" FY1979="-548" FY1978="-61" FY1977="-104" FY1976="-30" TreasuryAgencyCode="" AgencyName="Legislative Branch" BureauCode="00" BureauName="Legislative Branch" FY1999="0" FY1998="0" FY1997="0" FY1996="0" SubfunctionTitle="Other interest" FY1995="0"
FY1994="0" FY1993="0" FY1992="0" FY1991="0" FY1990="0" AccountCode="241400" On-Off-BudgetIndicator="On-budget" TransitionQuarter="-31" FY1989="0" AccountName="Charges for services to trust funds" FY1988="0" FY1987="0" FY1986="0" FY1985="0" FY1984="0" FY1983="0" FY1982="0" SubfunctionCode="803" FY1981="0" FY2006="0" FY1980="0" FY2005="0" FY2004="0" FY2003="0" FY2002="0" FY2001="0" FY2000="0" AgencyCode="001" BEACategory="Mandatory" FY1979="0" FY1978="0" FY1977="-16" FY1976="-28" TreasuryAgencyCode="" AgencyName="Legislative Branch" BureauCode="00" BureauName="Legislative Branch" FY1999="0" FY1998="0" FY1997="0" FY1996="0" SubfunctionTitle="Central fiscal operations" FY1995="0"

Figure 4.1 The List of Maps Data Structure for the Budget

There are an indefinite number of records from year to year as new budget items are added. Thus the list of records will be kept in a `java.util.ArrayList`. Any other form of `java.util.List` such as a `Vector` or a `LinkedList` would work equally well. After initial construction, I'll only access this object through the methods of the abstract `List` superclass. The program will not depend on any implementation details of the list.

The records themselves can be represented as arrays, vectors, instances of a custom class, hash tables, or maps. If the data is reasonably clean, I find it easier to use a custom class or a map. An array or vector works well when there may be extra data in some lines or perhaps missing information. In my initial experiments, the data proved to be fairly clean, so I chose to use a `Map`. The keys will be reasonable approximations to the field names, so they can be stored in a static array for easy extraction and iteration in a later part of the code. Again, there are no API calls that set this up for you. You have to do it yourself.

When complete, you'll have a list of maps, one map for each record, as diagrammed in Figure 4.1. This is very close to the form of the input data and still requires manipulation before it's in the form for the output data. Some manipulations may be straightforward. For example, it's very easy to extract all of the data for 1982: just iterate through the list and pull out only the fields that are relevant to 1982 from each map. Other manipulations are more complex. For example, if you wanted to convert this into a hierarchical structure in which each bureau was part of its agency, you might need to use a sorted data structure or make multiple passes through the list. You might want to reorganize the data by calendar year instead of fiscal year. Or perhaps beyond merely reorganizing, you might want to perform some calculations on the data, such as summing the total budget for each agency each year. Whatever output you want, it's just a matter of writing the code to generate it. Once the input data has been parsed, it's easy to write it out as XML.

■ Input

By far the hardest part of this or any similar problem is parsing the non-XML input data. Everything else pales by comparison. Unlike parsing XML, you generally cannot rely on a library to do the hard work for you. You have to do it yourself. And also unlike XML, there's little guarantee that the data is well-formed. More likely than not, you will encounter incorrectly formatted data.

In this case, because the records are separated into lines, I'll read each line, one at a time, using the `readLine()` method of `java.io.BufferedReader`. This method works well enough as long as the data is in a file, although it's potentially buggy when the data is served over a network socket.

Each line is dissected into its component fields inside the `splitLine()` method. Each record is stored in its own map. The keys for the map are read from a constant array, because the fields are always in the same position in each record.

Caution

For parsing the data out of each line, a lot of Java developers immediately reach for the `java.util.StringTokenizer` or `java.io.StreamTokenizer` classes. Don't. These classes are very strangely designed and rarely do what developers expect them to do. For example, if `StreamTokenizer` encounters a `\n` inside a string literal, it will convert it to a linefeed. This makes sense when parsing Java source code, but in most other environments `\n` is just another two characters with no special meaning. Java's tokenizer classes are designed for and suited to parsing Java source code. They are not suitable for reading tab- or comma-delimited data. If you want to design your program around a tokenization function, you should write one yourself that behaves appropriately for your data format.

Example 4.1 shows the input code. To use this, open an input stream to the file containing the budget data and pass that stream as an argument to the `parse()` method. You'll get back a `List` containing the parsed data. Each object in this list is a `Map` containing the data for one line item. Both keys and values for this map are strings. Because the keys are constant, they're stored in a final static array named `keys`. At various times I plan to use the keys as XML element names, XML attribute names, or SQL field names. Therefore, it's necessary to begin each of them with letters. Thus the keys for the fiscal year fields will be named FY1976, FY1977, FY1978, and so forth instead of just 1976, 1977, 1978, and so forth. This means we won't trivially be able to store the keys as ints. However, this turns out not to have been the case anyway because one of the year fields turns out to be the transitional quarter in 1976—which does not represent a full year and does not have a numeric name.

Caution

In 1976 the government's fiscal year shifted forward one quarter. As a result, the 1977 fiscal year started in October, a quarter after the 1976 fiscal year ended. There was a transitional quarter from July through September that year; therefore, some of the data actually represents less than a whole year. Here, the special case is very much a result of the data itself. Thus the data can't be fixed but still requires extra code, making the examples less clean than they otherwise would be.

This sort of funky data (a year with only three months in it that can easily be confused with another year) is exactly the sort of thing you have to watch out for when processing legacy data. The real world does not always fit into neatly typed categories. There's almost always some outlier data that just doesn't fit the schema. All too often it's been forced into the existing system by some manager or data entry clerk in ways the original designers never intended. This happens *all the time*. You cannot assume the data actually adheres to its schema, either implicit or explicit.

The code to parse each line of input is hidden inside the private `splitLine()` method. This code is relatively complex. It iterates through the record looking for comma delimiter characters but has to ignore commas that appear inside quoted strings. Furthermore, it must recognize that the end of the string delimits the last token. Even so, this method is not very robust. It will throw an uncaught exception if any quotes are omitted, or if there are too few fields. It will not notice and report the error if a record contains too many fields.

Example 4.1 A Class That Parses Comma-Separated Values into a List of HashMaps

```
import java.io.*;
import java.util.*;


public class BudgetData {

  public static List parse(InputStream src) throws IOException {

    // The document as published by the OMB is encoded in Latin-1
  InputStreamReader isr = new InputStreamReader(src, "8859_1");
  BufferedReader in = new BufferedReader(isr);
  List records = new ArrayList();
  String lineItem;
  while ((lineItem = in.readLine()) != null) {
    records.add(splitLine(lineItem));
  }
  return records;

}

// the field names in order
public final static String[] keys = {
  "AgencyCode",
  "AgencyName",
  "BureauCode",
  "BureauName",
  "AccountCode",
  "AccountName",
  "TreasuryAgencyCode",
  "SubfunctionCode",
  "SubfunctionTitle",
  "BEACategory",
  "On-Off-BudgetIndicator",
  "FY1976", "TransitionQuarter", "FY1977", "FY1978", "FY1979",
  "FY1980", "FY1981", "FY1982", "FY1983", "FY1984", "FY1985",
  "FY1986", "FY1987", "FY1988", "FY1989", "FY1990", "FY1991",
  "FY1992", "FY1993", "FY1994", "FY1995", "FY1996", "FY1997",
  "FY1998", "FY1999", "FY2000", "FY2001", "FY2002", "FY2003",
  "FY2004", "FY2005", "FY2006"
 };

private static Map splitLine(String record) {

  record = record.trim();

  int index = 0;
  Map result = new HashMap();
  for (int i = 0; i < keys.length; i++) {
    //find the next comma
    StringBuffer sb = new StringBuffer();
```

```
        char c;
        boolean inString = false;
        while (true) {
          c = record.charAt(index);
          if (!inString && c == '"') inString = true;
          else if (inString && c == '"') inString = false;
          else if (!inString && c == ',') break;
          else sb.append(c);
          index++;
          if (index == record.length()) break;
        }
        String s = sb.toString().trim();
        result.put(keys[i], s);
        index++;
      }

      return result;

    }

  }
```

Determining the Output Format

Once all the data has been stored in a convenient data structure, you will need to write it out as XML. The first step in this process is deciding what the XML should look like. XML does offer the possibility of a much more hierarchical structure than is available in the flat format. For example, you might sort budgets by department and year. You might use the natural hierarchy of the government in which agencies contain bureaus, bureaus contain accounts, and accounts contain subfunctions. I'll demonstrate several possibilities, beginning with the simplest, a fairly flat XML representation that stays very close to the original flat format.

As published the data is essentially one table. Thus the simplest output format merely duplicates this table structure. This table will be the root element, here called `Budget`. Each record in the table—that is, each line in the text file—will be encoded as a separate `LineItem` element. Each field will be encoded in a child element of the `LineItem` element whose name is the name of that field. Example 4.2 produces this data.

Example 4.2 Naively Reproducing the Original Budget Table Structure in XML

```
import java.io.*;
import java.util.*;


public class FlatXMLBudget {

  public static void convert(List data, OutputStream out)
   throws IOException {

    Writer wout = new OutputStreamWriter(out, "UTF8");
    wout.write("<?xml version=\"1.0\"?>\r\n");
    wout.write("<Budget>\r\n");

    Iterator records = data.iterator();
    while (records.hasNext()) {
      wout.write("  <LineItem>\r\n");
      Map record = (Map) records.next();
      Set fields = record.entrySet();
      Iterator entries = fields.iterator();
      while (entries.hasNext()) {
        Map.Entry entry = (Map.Entry) entries.next();
        String name = (String) entry.getKey();
        String value = (String) entry.getValue();
        // some of the values contain ampersands and less than
        // signs that must be escaped
        value = escapeText(value);

        wout.write("    <" + name + ">");
        wout.write(value);
        wout.write("</" + name + ">\r\n");
      }
      wout.write("  </LineItem>\r\n");
    }
    wout.write("</Budget>\r\n");
    wout.flush();

  }
```

```
public static String escapeText(String s) {

  if (s.indexOf('&') != -1 || s.indexOf('<') != -1
   || s.indexOf('>') != -1) {
    StringBuffer result = new StringBuffer(s.length() + 4);
    for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);
      if (c == '&') result.append("&amp;");
      else if (c == '<') result.append("&lt;");
      else if (c == '>') result.append("&gt;");
      else result.append(c);
    }
    return result.toString();
  }
  else {
    return s;
  }

}

public static void main(String[] args) {

  try {

    if (args.length < 1) {
     System.out.println("Usage: FlatXMLBudget infile outfile");
     return;
    }

    InputStream in = new FileInputStream(args[0]);
    OutputStream out;
    if (args.length < 2) {
      out = System.out;
    }
    else {
      out = new FileOutputStream(args[1]);
    }

    List results = BudgetData.parse(in);
    convert(results, out);
  }
  catch (IOException e) {
        System.err.println(e);
      }

    }

  }
```

The main() method reads the name of an input and output file from the command line. It parses the input file using the previously designed BudgetData.parse() method. This produces a List of line items. This list is passed to the convert() method along with an OutputStream to which the output will be written. I used an OutputStream rather than a Writer here because with a Writer it's not possible to pick your encoding, whereas with an OutputStream you can choose the encoding when you chain the Writer to it.

The convert() method iterates through that list. It extracts each record in turn, outputs a <LineItem> start-tag, then iterates through the Map representing the record, outputting each record in turn. The keys in the Map serve double duty, also becoming the names of the child elements of LineItem. The escapeText() method turns any <, >, or & characters that appear in the value into their respective escape sequences. Finally, the </LineItem> end-tag is output.

To some extent I've exposed more of the details here than is ideal. This group of classes is not as well encapsulated as I normally prefer. For the most part that's because I'm deliberately trying to show you how everything works internally. I'm not viewing this as reusable code. If I did need to make this more reusable, I would probably define a Budget class that contained a list of LineItem objects. Public methods in these classes would hide the detailed storage as a List of Maps.

Here's the prolog, root element start-tag, and first record from the XML output:

```
<?xml version="1.0"?>
<Budget>
  <LineItem>
    <FY1994>0</FY1994>
    <FY1993>0</FY1993>
    <FY1992>0</FY1992>
    <FY1991>0</FY1991>
    <FY1990>0</FY1990>
    <AccountCode></AccountCode>
    <On-Off-BudgetIndicator>On-budget</On-Off-BudgetIndicator>
    <FY1989>0</FY1989>
   <AccountName>Receipts, Central fiscal operations</AccountName>
    <FY1988>0</FY1988>
  <FY1987>0</FY1987>
  <FY1986>0</FY1986>
  <FY1985>0</FY1985>
  <FY1984>0</FY1984>
  <FY1983>0</FY1983>
  <FY1982>0</FY1982>
  <SubfunctionCode>803</SubfunctionCode>
  <FY1981>0</FY1981>
  <FY2006>0</FY2006>
  <FY1980>0</FY1980>
  <FY2005>0</FY2005>
  <FY2004>0</FY2004>
  <FY2003>0</FY2003>
  <FY2002>0</FY2002>
  <FY2001>0</FY2001>
  <FY2000>0</FY2000>
  <AgencyCode>001</AgencyCode>
  <BEACategory>Mandatory</BEACategory>
  <TransitionQuarter>-132</TransitionQuarter>
  <FY1979>-726</FY1979>
  <FY1978>-385</FY1978>
  <FY1977>-429</FY1977>
  <FY1976>-287</FY1976>
  <TreasuryAgencyCode></TreasuryAgencyCode>
  <AgencyName>Legislative Branch</AgencyName>
  <BureauCode>00</BureauCode>
  <BureauName>Legislative Branch</BureauName>
  <FY1999>0</FY1999>
  <FY1998>0</FY1998>
  <FY1997>0</FY1997>
  <FY1996>0</FY1996>
 <SubfunctionTitle>Central fiscal operations</SubfunctionTitle>
  <FY1995>0</FY1995>
</LineItem>
```

As you can see, one thing lost in the transition to XML is the order of the input fields. This is a side effect of storing each record as an unordered `HashMap`. However, order is really not important for this use case, so nothing significant has been lost. Order is rarely a problem for data-oriented XML applications. If order is important in your data, you can maintain it by holding the fields with a list or an array instead of a map. You could even use a SortedMap with a custom implementation of the Comparable interface for the keys you store in the map.

This design is somewhat memory intensive. The entire input data is read in and parsed before the first line is written out. In this case, the size of the input data is reasonably small for modern systems (just over a megabyte), so this isn't really an issue. If the data were much bigger, however, it would make sense to stream it in reasonably sized chunks. Each line item could be output as soon as it was read rather than waiting for the last line item to be read. For example, instead of storing each record in a big list, the BudgetData.parse() method could immediately write the data onto a Writer:

```
public static void parse(InputStream src, OutputStream out)
 throws IOException {

  Writer wout = new OutputStreamWriter(out, "UTF8");
  wout.write("<?xml version=\"1.0\"?>\r\n");
  wout.write("<Budget>\r\n");

  // The document as published by the OMB is encoded in Latin-1
  InputStreamReader isr = new InputStreamReader(src, "8859_1");
  BufferedReader in = new BufferedReader(isr);
  String lineItem;
  while ((lineItem = in.readLine()) != null) {
    Map data = splitLine(lineItem);
    Iterator records = data.iterator();
    wout.write("  <LineItem>\r\n");
    Set fields = records.entrySet();
    Iterator entries = fields.iterator();
    while (entries.hasNext()) {
      Map.Entry entry = (Map.Entry) entries.next();
      String name = (String) entry.getKey();
      String value = (String) entry.getValue();
      // some of the values contain ampersands and less than
      // signs that must be escaped
      value = escapeText(value);

      wout.write("    <" + name + ">");
      wout.write(value);
      wout.write("</" + name + ">\r\n");
    }  // end entries loop
    wout.write("  </LineItem>\r\n");
    }  // end lineitem loop

    wout.write("</Budget>\r\n");
    wout.flush();

  }
```

The disadvantage to this approach is that it only works when the output is fairly close to the input. In this case, each line of text in the input produces a single element in the output. The other extreme would be reordering the line items by year, essentially exchanging the rows and the columns. In this case, each element of output would include some content from every line of text. We'll see some examples of this (and some in-between cases) shortly. In these cases, a streaming architecture would require multiple passes over the input data and be very inefficient. Still, if the data set were large enough, that might be the only workable approach.

Validation

Error checking in this program is minimal. For example, the `convert()` method does not verify that each item in the `List` is in fact a `Map`. If somebody passes in a `List` that contains non-`Map` objects, then an uncaught `ClassCastException` will be thrown. Furthermore, `convert()` does not check that each `Map` has all 43 required entries or that the keys in the `Map` match the known key list. No exception will be thrown in these cases. However, if a record turns out to be short a few fields or to have the wrong fields, then validation of the output should pick it up.

The first test you should make of the output is for simple well-formedness. Whatever tool you normally use to check well-formedness, for example, Xerces-J's `sax.Counter` *[http://xml.apache.org/xerces2-j/samples-sax.html#Counter]*, will suffice. The first time I ran the output from this program through `sax.Counter`, it indeed told me the output was not well-formed. The problem was that some of the names and titles used & characters that had to be escaped, so I added the `escapeText()` method to the `FlatXMLBudget` class, and then reran the program. I checked my output again, and it was still malformed. Although I'd written the `escapeText()` method, I had forgotten to invoke it. I edited `FlatXMLBudget` one more time and reran the program. My output was still malformed. This time I'd forgotten to recompile the program before rerunning it, so I recompiled and ran it one more time. Finally my output proved to be well-formed. This is just normal back-and-forth development. Although the test is specific to XML, there's nothing XML-specific about the process. You keep fixing problems and trying again until you succeed. Many fixes will simply expose new bugs and errors. Eventually you fix the last one, and the program runs correctly.

Once you're confident that your program is outputting well-formed XML, the next step is to validate it. Even if your use case doesn't actually require valid XML, validation is still an invaluable testing and debugging tool. Example 4.3 is a schema for the intended output of Example 4.2. I chose to use a schema rather than a DTD so I could check that each field contained the correct form of data.

Example 4.3 A Schema for the XML Budget Data

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="Budget">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="LineItem" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:all>
              <xsd:element name="AccountCode">
                <xsd:simpleType>
                  <xsd:union
                       memberTypes="FourDigitCode SixDigitCode"/>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="AccountName">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="160"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="AgencyCode"
                           type="ThreeDigitCode"/>
              <xsd:element name="TreasuryAgencyCode"
                           type="TwoDigitCode"/>
              <xsd:element name="AgencyName">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="89"/>
                  </xsd:restriction>
                </xsd:simpleType>
```

```
</xsd:element>
<xsd:element name="BEACategory">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Mandatory"/>
      <xsd:enumeration value="Discretionary"/>
      <xsd:enumeration value="Net interest"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="BureauCode"
             type="TwoDigitCode"/>
<xsd:element name="BureauName">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="89"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="On-Off-BudgetIndicator">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="On-budget"/>
      <xsd:enumeration value="Off-budget"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="SubfunctionCode"
             type="ThreeDigitCode"/>
<xsd:element name="SubfunctionTitle">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="72"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="FY1976" type="xsd:integer"/>
<xsd:element name="TransitionQuarter"
             type="xsd:integer"/>
<xsd:element name="FY1977" type="xsd:integer"/>
<xsd:element name="FY1978" type="xsd:integer"/>
```

```
            <xsd:element name="FY1979" type="xsd:integer"/>
            <xsd:element name="FY1980" type="xsd:integer"/>
            <xsd:element name="FY1981" type="xsd:integer"/>
            <xsd:element name="FY1982" type="xsd:integer"/>
            <xsd:element name="FY1983" type="xsd:integer"/>
            <xsd:element name="FY1984" type="xsd:integer"/>
            <xsd:element name="FY1985" type="xsd:integer"/>
            <xsd:element name="FY1986" type="xsd:integer"/>
            <xsd:element name="FY1987" type="xsd:integer"/>
            <xsd:element name="FY1988" type="xsd:integer"/>
            <xsd:element name="FY1989" type="xsd:integer"/>
            <xsd:element name="FY1990" type="xsd:integer"/>
            <xsd:element name="FY1991" type="xsd:integer"/>
            <xsd:element name="FY1992" type="xsd:integer"/>
            <xsd:element name="FY1993" type="xsd:integer"/>
            <xsd:element name="FY1994" type="xsd:integer"/>
            <xsd:element name="FY1995" type="xsd:integer"/>
            <xsd:element name="FY1996" type="xsd:integer"/>
            <xsd:element name="FY1997" type="xsd:integer"/>
            <xsd:element name="FY1998" type="xsd:integer"/>
            <xsd:element name="FY1999" type="xsd:integer"/>
            <xsd:element name="FY2000" type="xsd:integer"/>
            <xsd:element name="FY2001" type="xsd:integer"/>
            <xsd:element name="FY2002" type="xsd:integer"/>
            <xsd:element name="FY2003" type="xsd:integer"/>
            <xsd:element name="FY2004" type="xsd:integer"/>
            <xsd:element name="FY2005" type="xsd:integer"/>
            <xsd:element name="FY2006" type="xsd:integer"/>
          </xsd:all>
        </xsd:complexType>
       </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="TwoDigitCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9][0-9]"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
  <xsd:simpleType name="ThreeDigitCode">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9][0-9][0-9]"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="FourDigitCode">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9][0-9][0-9][0-9]"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="SixDigitCode">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9][0-9][0-9][0-9][0-9][0-9]"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

As well as indicating bugs in your code that you need to fix, schema checking can uncover dirty input data that isn't what it's supposed to be. This happens *all the time* in real-world data. When I validated the XML output against this schema, a few problems did crop up. A few account codes were blank, and the third BEA category was actually "Net interest," not "Net Interest" as the documentation claimed. However, these problems were quite minor overall. In fact, I was quite surprised that there weren't more problems. The programmers at the OMB are doing a much better than average job.

When faced with dirty data that's static, the normal solution is to edit the input data by hand and rerun the program. However if the data is dynamic (for example, the sales figures for various retail locations submitted to a central server when each franchise closes down for the night), then you'll have to write extra code to massage the bad data into an acceptable format, at least until you can fix the process that's generating the dirty data.

Attributes

Given that the order of child elements really isn't significant within a `LineItem` and that no two child elements are repeated, it seems that some of this data could be stored as attributes. Besides the obvious change in the `convert()` method, this also requires changing the `escapeText()` method to escape any embedded quotes, which are legal in element content but not attribute values. Example 4.4 demonstrates this.

Example 4.4 Converting to XML with Attributes

```
import java.io.*;
import java.util.*;


public class AttributesXMLBudget {

  public static void convert(List data, OutputStream out)
   throws IOException {

    Writer wout = new OutputStreamWriter(out, "UTF8");
    wout.write("<?xml version=\"1.0\"?>\r\n");
    wout.write("<Budget>\r\n");

    Iterator records = data.iterator();
    while (records.hasNext()) {
      wout.write("  <ListItem");
      Map record = (Map) records.next();

      // write the attributes
      writeAttribute(wout, "AgencyCode", record);
      writeAttribute(wout, "AgencyName", record);
      writeAttribute(wout, "BureauCode", record);
      writeAttribute(wout, "BureauName", record);
      writeAttribute(wout, "AccountCode", record);
      writeAttribute(wout, "AccountName", record);
      writeAttribute(wout, "TreasuryAgencyCode", record);
      writeAttribute(wout, "SubfunctionCode", record);
      writeAttribute(wout, "SubfunctionTitle", record);
      writeAttribute(wout, "BEACategory", record);
      writeAttribute(wout, "BudgetIndicator", record);
      wout.write(">\r\n");
      writeAmount(wout, "1976", record);
      wout.write("    <Amount year='TransitionQuarter'>");
      wout.write(
        escapeText((String) record.get("TransitionQuarter"))
      );
      wout.write("</Amount>\r\n");
      for (int year=1977; year <= 2006; year++) {
        writeAmount(wout, String.valueOf(year), record);
      }
      wout.write("  </ListItem>\r\n");
    }
    wout.write("</Budget>\r\n");
    wout.flush();

  }

  // Just a couple of private methods to factor out repeated code
  private static void writeAttribute(Writer out, String name,
   Map record) throws IOException {
    out.write(" " + name + "='"
     + escapeText((String) record.get(name)) + "'");
  }

  private static void writeAmount(Writer out, String year,
   Map record) throws IOException {
    out.write("    <Amount year='" + year + "'>");
    out.write(escapeText((String) record.get("FY" + year)));
    out.write("</Amount>\r\n");
  }

  public static String escapeText(String s) {

    if (s.indexOf('&') != -1 || s.indexOf('<') != -1
     || s.indexOf('>') != -1 || s.indexOf('"') != -1
     || s.indexOf('\'') != -1 ) {
      StringBuffer result = new StringBuffer(s.length() + 6);
      for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '&') result.append("&amp;");
        else if (c == '<') result.append("&lt;");
        else if (c == '"') result.append("&quot;");
        else if (c == '\'') result.append("&apos;");
        else if (c == '>') result.append("&gt;");
        else result.append(c);
      }
      return result.toString();
    }
    else {
      return s;
    }

  }

  public static void main(String[] args) {

    try {

      if (args.length < 1) {
        System.out.println(
         "Usage: AttributesXMLBudget infile outfile"
        );
        return;
      }

      InputStream in = new FileInputStream(args[0]);
      OutputStream out;
      if (args.length < 2) {
        out = System.out;
      }
      else {
        out = new FileOutputStream(args[1]);
      }

      List results = BudgetData.parse(in);
      convert(results, out);
    }
    catch (IOException e) {
      System.err.println(e);
    }

  }

}
```

In this case, because different fields are treated differently, the program can't simply iterate through the map. It must ask for each field by name and put it where it belongs. This has the advantage of making the output a little more readable to humans, as the following fragment of output demonstrates:

```
<?xml version="1.0"?>
<Budget>
  <LineItem AgencyCode='001' AgencyName='Legislative Branch'
            BureauCode='00'
            BureauName='Legislative Branch' AccountCode=''
            AccountName='receipts, Central fiscal operations'
            TreasuryAgencyCode='' SubfunctionCode='803'
            SubfunctionTitle='Central fiscal operations'
            BEACategory='mandatory'
            On-Off-BudgetIndicator='On-budget'>
    <Amount year='1976'>-287</Amount>
    <Amount year='TransitionalQuarter'>-132</Amount>
    <Amount year='1977'>-429</Amount>
    <Amount year='1978'>-385</Amount>
    <Amount year='1979'>-726</Amount>
    <Amount year='1980'>0</Amount>
    <Amount year='1981'>0</Amount>
    <Amount year='1982'>0</Amount>
    <Amount year='1983'>0</Amount>
    <Amount year='1984'>0</Amount>
    <Amount year='1985'>0</Amount>
    <Amount year='1986'>0</Amount>
    <Amount year='1987'>0</Amount>
    <Amount year='1988'>0</Amount>
    <Amount year='1989'>0</Amount>
    <Amount year='1990'>0</Amount>
    <Amount year='1991'>0</Amount>
    <Amount year='1992'>0</Amount>
    <Amount year='1993'>0</Amount>
    <Amount year='1994'>0</Amount>
    <Amount year='1995'>0</Amount>
    <Amount year='1996'>0</Amount>
    <Amount year='1997'>0</Amount>
    <Amount year='1998'>0</Amount>
    <Amount year='1999'>0</Amount>
    <Amount year='2000'>0</Amount>
    <Amount year='2001'>0</Amount>
    <Amount year='2002'>0</Amount>
    <Amount year='2003'>0</Amount>
    <Amount year='2004'>0</Amount>
    <Amount year='2005'>0</Amount>
    <Amount year='2006'>0</Amount>
  </LineItem>
```

The attribute version is a little more compact than the element-only alternative. It also has the advantage of keeping the years in chronological order.

Clearly, there are many other ways you might choose to divide the output between elements and attributes. For example, you might make all the names elements but all the codes attributes. There's certainly more than one way to do it. The modifications needed to produce slightly different forms of this output are straightforward.

Building Hierarchical Structures from Flat Data

Sometimes you want to convert only a subset of the available data. For example, you might want to store each year in a separate XML document. With a few tweaks to the basic code, this is not hard to do. Another possibility is to arrange the data hierarchically, with `Agency` elements containing `Bureau` elements, which contain `Account` elements, which contain `Subfunction` elements. This is a little more complex, but far from impossible. Or perhaps you want to massage the data while converting it, for example by changing the amounts of thousands of dollars to amounts of single dollars. Or you could do all three, in which case the hierarchy would be arranged something like in Example 4.5.

Example 4.5 A Hierarchical Arrangement of the Budget Data

```
<?xml version="1.0"?>
<Budget year="2001">
 <Agency>
  <Name>Farm Credit System Financial Assistance Corporation</Name>
  <Code>354</Code>
   <Bureau>
   <Name>Farm Credit System Financial Assistance Corporation</Name>
      <Code>00</Code>
      <Account>
        <Name>Financial assistance corporation trust fund</Name>
        <Code>8202</Code>
        <BEACategory>Mandatory</BEACategory>
      <Subfunction>
        <Title>Farm income stabilization</Title>
        <Code>351</Code>
        <Amount>-1000</Amount>
      </Subfunction>
    </Account>
  </Bureau>
</Agency>
<Agency>
  <Name>Department of Education</Name>
  <Code>018</Code>
  <Bureau>
    <Name>Office of Elementary and Secondary Education</Name>
    <Code>10</Code>
    <Account>
      <Name>Reading excellence</Name>
      <Code>0011</Code>
      <BEACategory>Discretionary</BEACategory>
      <Subfunction>
        <Title>
          Elementary, secondary, and vocational education
        </Title>
        <Code>501</Code>
        <Amount>286000000</Amount>
      </Subfunction>
    </Account>
    <Account>
      <Name>Indian education</Name>
      <Code>0101</Code>
      <BEACategory>Discretionary</BEACategory>
      <Subfunction>
        <Title>
          Elementary, secondary, and vocational education
        </Title>
        <Code>501</Code>
        <Amount>116000000</Amount>
      </Subfunction>
    </Account>
    <!-- more accounts...-->
  </Bureau>
```

```
      <Bureau>
        <Name>
        Office of Bilingual Education and Minority Languages Affairs
        </Name>
        <Code>15</Code>
        <Account>
          <Name>Bilingual and immigrant education</Name>
          <Code>1300</Code>
          <BEACategory>Discretionary</BEACategory>
          <Subfunction>
            <Title>
              Elementary, secondary, and vocational education
            </Title>
            <Code>501</Code>
            <Amount>460000000</Amount>
          </Subfunction>
        </Account>
      </Bureau>
      <!-- more bureaus...-->
    </Agency>
    <!-- many more agencies... -->
  </Budget>
```

There are two basic approaches to imposing hierarchy on the flat structures we're starting with. One is to build the hierarchy into the Java data structures in memory, enabling these data structures to be output in XML quite simply. The alternative is to leave the data structures flat in memory, but to try to insert the hierarchy as the output code is being written. The latter approach is not too difficult when the data can be sorted easily, but most of the time I find the former approach to be simpler.

Although all the data could still be stored in lists of lists, and lists of maps, and maps of lists, and so forth, I really do find that at this point the data structures are clearer if we name them. And in Java, there isn't any real difference between a named data structure and a class. Thus we can describe the structure as classes and objects. Informally, the structure looks like this:

```
class Budget {

  String year;
  List   agencies;
}

class Agency {

  String code;
  String treasuryCode;
  String name;
  List   bureaus;

}

class Bureau {

  String code;
  String name;
  List   accounts;

}

class Account {

  String code;
  String name;
  String BEACategory;
  List   subfunctions;

}

class Subfunction {

  String code;
  String title;
  long   amount;

}
```

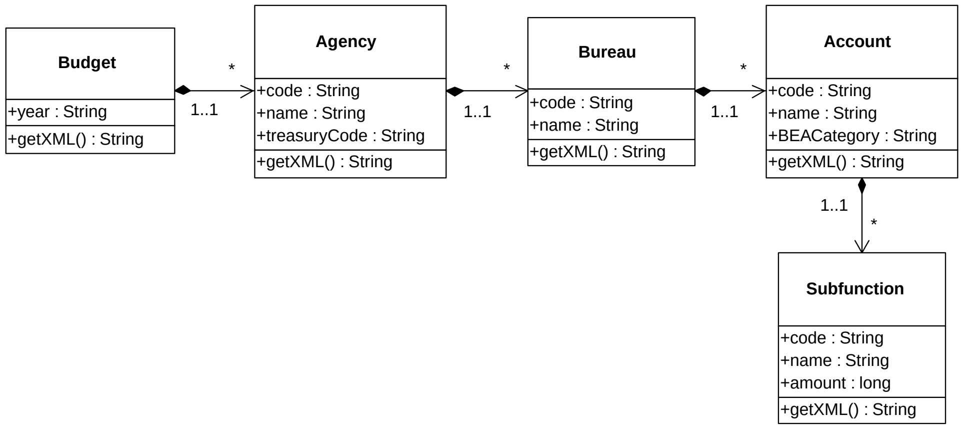

Figure 4.2 A UML Diagram for the Budget Class Hierarchy

Of course, we'll also need a number of methods in these classes. For the most part, we will use factory methods instead of constructors to guarantee that 19 IRS `Bureau` objects aren't created just because the IRS has 19 line items in the budget. (One IRS is bad enough. :-)) Each class will have a `getXML()` method that returns a `String` containing the XML form of the object. And each class will have a method to add a child node of the appropriate type. Figure 4.2 illustrates a UML static structure diagram for the complete group of classes.

Note

I must admit that the lack of generic types (templates to C++ programmers) hurts a little here. For example, I want to say that an `Agency` contains a list of `Bureaus`. However, all I can really say is that an `Agency` contains a list of objects, and this list happens to be named `bureaus`. It's not a huge problem—if it were, I could always define my own custom list classes with the appropriate types—but it does lead to a less natural solution than generic types do. I thought seriously about using generics for this example, but that would limit this code to Java 1.4. Maybe for the second edition.

The `Budget` class, show in Example 4.6, is at the top of the tree. It contains a list of agencies and a year. Line items encoded as maps by `BudgetData` are fed into the tree through the `add()` method. This method breaks up that map into the relevant parts, encodes each as an object, adds each of those objects to the tree in the right place, and stores the agency in the list of agencies. The various `add()` methods in both this and the other classes are responsible for putting each piece in the right place.

Example 4.6 The Budget Class

```
import java.util.*;


public class Budget {

  private List   agencies = new ArrayList();
  private String year;

  public Budget(String year) {
    this.year = year;
  }

  public void add(Agency agency) {
    if (!agencies.contains(agency)) agencies.add(agency);
  }

  public void add(Map lineItem) {

    String agencyName = (String) lineItem.get("AgencyName");
    agencyName = escapeText(agencyName);
    String agencyCode = (String) lineItem.get("AgencyCode");
    String treasuryAgencyCode
     = (String) lineItem.get("TreasuryAgencyCode");
    Agency agency = Agency.getInstance(agencyName, agencyCode,
     treasuryAgencyCode, year);
    this.add(agency);

    String bureauName = (String) lineItem.get("BureauName");
    bureauName = escapeText(bureauName);
    String bureauCode = (String) lineItem.get("BureauCode");
    Bureau bureau = Bureau.getInstance(bureauName, bureauCode,
     agencyCode, year);
    agency.add(bureau);
    // Names and codes of two accounts in different bureaus
    // can be the same
    String accountName = (String) lineItem.get("AccountName");
    accountName = escapeText(accountName);
    String accountCode = (String) lineItem.get("AccountCode");
    String category    = (String) lineItem.get("BEACategory");
    Account account = Account.getInstance(accountName,
     accountCode, category, bureauCode, agencyCode, year);
    bureau.add(account);

    // Names and codes of two subfunctions in different accounts
    // can be the same
    String subfunctionTitle
     = escapeText((String) lineItem.get("SubfunctionTitle"));
    String subfunctionCode
     = (String) lineItem.get("SubfunctionCode");
    String yearKey = year;
    if (!yearKey.equals("TransitionQuarter")) {
      yearKey = "FY" + year;
    }
    long amount
     = 1000L * Long.parseLong((String) lineItem.get(yearKey));
    Subfunction subfunction = new Subfunction(subfunctionTitle,
     subfunctionCode, amount);
    account.add(subfunction);

  }

  public String getXML() {

    StringBuffer result = new StringBuffer("<Budget year='"
     + this.year +"'>\r\n");
    Iterator iterator = agencies.iterator();
    while (iterator.hasNext()) {
      Agency agency = (Agency) iterator.next();
      result.append(agency.getXML());
    }
    result.append("</Budget>\r\n");
    return result.toString();

  }
  public static String escapeText(String s) {

    if (s.indexOf('&') != -1 || s.indexOf('<') != -1
     || s.indexOf('>') != -1 || s.indexOf('"') != -1
     || s.indexOf('\'') != -1 ) {
      StringBuffer result = new StringBuffer(s.length() + 6);
      for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '&') result.append("&amp;");
        else if (c == '<') result.append("&lt;");
        else if (c == '"') result.append("&quot;");
        else if (c == '\'') result.append("&apos;");
        else if (c == '>') result.append("&gt;");
        else result.append(c);
      }
      return result.toString();
    }
    else {
      return s;
    }

  }

}
```

Most of the other classes are structurally the same but with subtle variations. Each has a list of its children, a private constructor, a public factory method, an `add()` method to add a child of the appropriate type, and a `getXML()` method that returns the XML representation of the object as a `String`. I'll begin with the `Agency` class in Example 4.7.

Example 4.7 The Agency Class

```
import java.util.*;


public class Agency {

  private String code;
  private String name;
private String treasuryCode;
private String year;

private List   bureaus = new ArrayList();

private static Map instances = new HashMap();

// A private constructor so instantiators
// have to use the factory method
private Agency(String name, String code, String treasuryCode,
        String year) {

  this.name = name;
  this.code = code;
  this.treasuryCode = treasuryCode;
  this.year = year;

}

public static Agency getInstance(String name, String code,
  String treasuryCode, String year) {

  // Agencies can be uniquely identified by code alone
  String key = code+" "+year;
  Agency agency = (Agency) instances.get(key);
  if (agency == null) {
    agency = new Agency(name, code, treasuryCode, year);
    instances.put(key, agency);
  }

  return agency;

}

public void add(Bureau b) {
  if (!bureaus.contains(b)) {
      bureaus.add(b);
  }
}
```

```
  public String getXML() {

    StringBuffer result = new StringBuffer("  <Agency>\r\n");
    result.append("    <Name>" + name + "</Name>\r\n");
    result.append("    <Code>" + code + "</Code>\r\n");
    result.append("    <TreasuryAgencyCode>" + treasuryCode
     + "</TreasuryAgencyCode>\r\n");
    Iterator iterator = bureaus.iterator();
    while (iterator.hasNext()) {
      Bureau bureau = (Bureau) iterator.next();
      result.append(bureau.getXML());
    }
    result.append("  </Agency>\r\n");
    return result.toString();

  }

}
```

This is not a general-purpose class. The public interface is limited to a factory method, a method to add a `Bureau` to the `Agency`, and a method to get the XML representation of the object as a `String`. There aren't even any getter methods. I've deliberately chosen not to burden this and the other classes with a lot of error checking. I'm assuming that will be done both in the database before the data is input here, and later through validation after the XML document has been produced. Doing it here too seemed a tad superfluous.

Clients retrieve instances of the class using the `getInstance()` factory method. The first time that this class is asked to get a particular agency, it constructs it using the private constructor, stores it in the static `instances` map, and returns a reference to it. Subsequent requests will return the formerly created object from the map. The key for the map is the concatenation of the agency's code and year.

The `add()` method simply stores `Bureau` objects in the `bureaus` list. Since `bureaus` is private and accessed only through this `add()` method, we can be certain that no non-`Bureau` objects will ever get added to this list. We can further guarantee that the list doesn't contain any duplicates.

The `getXML()` method returns an XML representation of this `Agency` object. It loops through all of the bureaus the agency contains, and uses their `getXML()` methods to obtain the XML representations for each of them. The XML is returned as a `String`. An alternative approach that might be more memory efficient and perhaps a little faster, especially in streaming applications, would be to pass the `OutputStream` or `Writer` onto which the XML should be written to the `getXML()`

method. In essence, this would be a `writeXML()` method rather than a `getXML()` method.

The code here is far from the most efficient possible. A lot of optimizations could be done, including many that would build far fewer intermediate string representations of each element. For example, you could pass in the `StringBuffer` to append to to each `getXML()` method. However, because this is intended primarily as a one-off solution, the optimization didn't seem to be worth the investment of time or the added complexity.

The `Bureau` class, shown in Example 4.8, is much the same, except that it contains a list of accounts. Furthermore, both an agency code and a bureau code are required to uniquely identify bureaus. The bureau code alone is not sufficient, so the keys for the `instances` map are formed by concatenating the agency code, the bureau code, and the year.

Example 4.8 The Bureau Class

```
import java.util.*;

public class Bureau {

  // Agency code plus bureau code uniquely identify a bureau.
  // Bureau code alone is definitely not sufficient.
  private String code;
  private String name;
  private String year;
  private String agencyCode;

  private List   accounts = new ArrayList();

  private static Map instances = new HashMap();

  // Use a private constructor so instantiators
  // have to use the factory method
  private Bureau(String name, String code, String agencyCode,
    String year) {

    this.name = name;
    this.code = code;
    this.agencyCode = agencyCode;
    this.year = year;
  }

  public static Bureau getInstance(String name, String code,
   String agencyCode, String year) {

    String key = agencyCode+" "+code+" "+year;
    Bureau bureau = (Bureau) instances.get(key);
    if (bureau == null) {
      bureau = new Bureau(name, code, agencyCode, year);
      instances.put(key, bureau);
    }

    return bureau;

  }

  public void add(Account account) {
    if (!accounts.contains(account)) accounts.add(account);
  }

  public String getXML() {

    StringBuffer result = new StringBuffer("    <Bureau>\r\n");
    result.append("      <Name>" + name + "</Name>\r\n");
    result.append("      <Code>" + code + "</Code>\r\n");
    Iterator iterator = accounts.iterator();
    while (iterator.hasNext()) {
      Account account = (Account) iterator.next();
      result.append(account.getXML());
    }
    result.append("    </Bureau>\r\n");
    return result.toString();

  }

}
```

The `Account` class, shown in Example 4.9, is similar to `Bureau` and `Agency`. Accounts are uniquely identified by an account code, a bureau code, an agency code, and a BEA code. Otherwise, this class is structured the same as the preceding two.

Example 4.9 The Account Class

```
import java.util.*;


public class Account {

  // An account is uniquely identified by account code,
  // bureau code, agency code and BEA category
  private String code;
  private String name;
  private String BEACategory;
  private String bureauCode;
  private String agencyCode;
  private String year;

  private List   subfunctions = new ArrayList();

  private static Map instances = new HashMap();

  // Use a private constructor so clients
  // have to use the factory method
  private Account(String name, String code, String BEACategory,
   String bureauCode, String agencyCode, String year) {

    this.name = name;
    this.code = code;
    this.BEACategory = BEACategory;
    this.bureauCode = bureauCode;
    this.agencyCode = agencyCode;
    this.year = year;

  }

  public static Account getInstance(String name, String code,
   String BEACategory, String bureauCode, String agencyCode,
   String year) {

    String key = code + " " + BEACategory + " " + bureauCode
     + " " + agencyCode + " " + year;
    Account account = (Account) instances.get(key);
    if (account == null) {
      account = new Account(name, code, BEACategory, bureauCode,
       agencyCode, year);
      instances.put(key, account);
    }

    return account;

  }

  public void add(Subfunction sfx) {
    if (!subfunctions.contains(sfx)) subfunctions.add(sfx);
  }

  public String getXML() {

    StringBuffer result = new StringBuffer();
    result.append("      <Account>\r\n");
    result.append("        <Name>" + name + "</Name>\r\n");
    result.append("        <Code>" + code + "</Code>\r\n");
    result.append("        <BEACategory>" + BEACategory
     + "</BEACategory>\r\n");
    Iterator iterator = subfunctions.iterator();
    while (iterator.hasNext()) {
      Subfunction subfunction = (Subfunction) iterator.next();
      result.append(subfunction.getXML());
    }
    result.append("      </Account>\r\n");
    return result.toString();

  }

}
```

Finally, the `Subfunction` class, shown in Example 4.10, contains a title, a code, and an amount of money for a given year. Unlike agencies, bureaus, and accounts, subfunctions are not necessarily unique. For example, the “Department of Defense-Military” subfunction appears in 222 different accounts. Therefore, this class does not use a factory method, just a regular constructor.

Example 4.10 The Subfunction Class

```
import java.util.*;


public class Subfunction {

  private String code;
  private String title;
  private long   amount;

  public Subfunction(String title, String code, long amount) {

    this.title  = title;
    this.code   = code;
    this.amount = amount;

  }

  public String getXML() {

    StringBuffer result
      = new StringBuffer("        <Subfunction>\r\n");
    result.append("          <Name>" + title + "</Name>\r\n");
    result.append("          <Code>" + code + "</Code>\r\n");
    result.append("          <Amount>");
    result.append(amount + "</Amount>\r\n");
    result.append("        </Subfunction>\r\n");
    return result.toString();

  }

}
```

The ultimate plan is to read all the records of the budget data, and then use that information to create objects of these classes. As each object is created, it is added to its parent object's list. When all records have been processed, the `getXML()` method of the `Budget` object is invoked to retrieve the XML representation of the budget. This XML can be written onto an output stream. Example 4.11 accomplishes this. It has a `main()` method to provide a simple user interface. This simply reads the year and the input and output file names from the user, and calls `convert()`. The `convert()` method delegates reading the input to the `BudgetData` class, and the data-structure building and most of the output to the `Budget` class.

Example 4.11 The Driver Class That Builds the Data Structure and Writes It Out Again

```
import java.io.*;
import java.util.*;


public class HierarchicalXMLBudget {

  public static void convert(List budgetData, String year,
   OutputStream out) throws IOException {

    Budget budget = new Budget(year);
    Iterator records = budgetData.iterator();
    while (records.hasNext()) {
      Map lineItem = (Map) records.next();
      budget.add(lineItem);
    }

    Writer wout = new OutputStreamWriter(out, "UTF8");
    wout.write("<?xml version=\"1.0\"?>\r\n");
    wout.write(budget.getXML());
    wout.flush();

  }

  public static void main(String[] args) {

    try {

      if (args.length < 2) {
        System.out.println(
         "Usage: HierarchicalXMLBudget year infile outfile");
        return;
      }
```

```
        // simple error checking on the year value
        try {
          if (!args[0].equals("TransitionQuarter")) {
            Integer.parseInt(args[0]);
          }
        }
        catch (NumberFormatException e) {
          System.out.println(
           "Usage: HierarchicalXMLBudget year infile outfile");
          return;
        }

        InputStream in = new FileInputStream(args[1]);
        OutputStream out;
        if (args.length < 3) {
          out = System.out;
        }
        else {
          out = new FileOutputStream(args[2]);
        }

        List results = BudgetData.parse(in);
        convert(results, args[0], out);
      }
      catch (IOException e) {
        System.err.println(e);
      }

    }

  }
```

Here's the start of a sample run:

```
C:\XMLJAVA>java HierarchicalXMLBudget 2002 budauth.txt
<?xml version="1.0"?>
<Budget year='2002'>
   <Agency>
     <Name>Legislative Branch</Name>
     <Code>001</Code>
     <TreasuryAgencyCode></TreasuryAgencyCode>
     <Bureau>
```

```
            <Name>Legislative Branch</Name>
            <Code>00</Code>
            <Account>
              <Name>Receipts, Central fiscal operations</Name>
              <Code></Code>
              <BEACategory>Mandatory</BEACategory>
              <Subfunction>
                <Name>Central fiscal operations</Name>
                <Code>803</Code>
                <Amount>0</Amount>
              </Subfunction>
            </Account>
            <Account>
              <Name>Receipts, Central fiscal operations</Name>
              <Code></Code>
              <BEACategory>Net interest</BEACategory>
              <Subfunction>
...
```

The technique used here is extremely powerful. The data is read one item at a time. Each item is stored in a data structure, which automatically moves each item into its appropriate slot. The final result is generated by a single method call to the root of the data structure.

Note
A similar approach would be to define classes that represent the XML elements and attributes rather than the input data. Indeed this is the approach taken by the Document Object Model (DOM). I'll explore this alternative in later chapters.

Alternatives to Java

When all you have is a hammer, most problems look a lot like nails. Since you're reading this book, I'm willing to bet that Java is your hammer of choice, and indeed Java is a very powerful hammer. But sometimes you really could use a screwdriver, and this may be one of those times. I must admit that the solution for imposing hierarchy developed in the last section feels more than a little like pounding a screw with a hammer. Maybe it would be better to use the hammer to set the screw, but then use a screwdriver to drive it in. In this section I want to explore a few possible screwdrivers, including XSLT and XQuery. Rather than using such complex Java code, I'll do the following: First I'll use Java to get the data into the same simple XML format as that produced by Example 4.2, which closely matches the flat input data. Then I'll use XSLT to transform this simple intermediate XML format into the less flat final XML format. To refresh your memory, the flat XML data is organized like this:

```
<?xml version="1.0"?>
<Budget>
  <LineItem>
    <FY1994>-1982</FY1994>
    <FY1993>4946</FY1993>
    <FY1992>-3251</FY1992>
    <FY1991>-17373</FY1991>
    <FY1990>-90008</FY1990>
    <AccountCode>265197</AccountCode>
    <On-Off-BudgetIndicator>On-budget</On-Off-BudgetIndicator>
    <TransitionQuarter>0</TransitionQuarter>
    <FY1989>-80069</FY1989>
    <AccountName>Sale of scrap and salvage materials</AccountName>
    <FY1988>-72411</FY1988>
    <FY1987>-60964</FY1987>
    <FY1986>-61462</FY1986>
    <FY1985>-68182</FY1985>
    <FY1984>-79482</FY1984>
    <FY1983>0</FY1983>
    <FY1982>0</FY1982>
    <SubfunctionCode>051</SubfunctionCode>
    <FY1981>0</FY1981>
    <FY2006>-1000</FY2006>
    <FY1980>0</FY1980>
    <FY2005>-1000</FY2005>
    <FY2004>-1000</FY2004>
    <FY2003>-1000</FY2003>
    <FY2002>-1000</FY2002>
    <FY2001>-1000</FY2001>
    <FY2000>-2000</FY2000>
    <AgencyCode>007</AgencyCode>
    <BEACategory>Mandatory</BEACategory>
    <FY1979>0</FY1979>
    <FY1978>0</FY1978>
```

```
        <FY1977>0</FY1977>
        <FY1976>0</FY1976>
        <TreasuryAgencyCode>97</TreasuryAgencyCode>
        <AgencyName>Department of Defense--Military</AgencyName>
        <BureauCode>00</BureauCode>
        <BureauName>Department of Defense--Military</BureauName>
        <FY1999>-1000</FY1999>
        <FY1998>-2000</FY1998>
        <FY1997>-4000</FY1997>
        <FY1996>-1000</FY1996>
        <SubfunctionTitle>Department of Defense-Military
         </SubfunctionTitle>
        <FY1995>-1000</FY1995>
      </LineItem>
      <!-- several thousand more LineItem elements... -->
    </Budget>
```

Imposing Hierarchy with XSLT

The XSLT stylesheet shown in Example 4.12 will convert flat XML budget data of this type into an output document of the same form produced by Example 4.11. Because the input file is so large, you may need to raise the memory allocation for your XSLT processor before running the transform.

Example 4.12 An XSLT Stylesheet That Converts Flat XML Data to Hierarchical XML Data

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- Try to make the output look half decent -->
  <xsl:output indent="yes" encoding="ISO-8859-1"/>

  <!-- Muenchian method -->
  <xsl:key name="agencies" match="LineItem" use="AgencyCode"/>
  <xsl:key name="bureaus"  match="LineItem"
    use="concat(AgencyCode,'+',BureauCode)"/>
  <xsl:key name="accounts" match="LineItem"
    use="concat(AgencyCode,'+',BureauCode,'+',AccountCode)"/>
  <xsl:key name="subfunctions" match="LineItem"
  use="concat(AgencyCode,'+',BureauCode,'+',AccountCode,
  '+',SubfunctionCode)"/>

<xsl:template match="Budget">
  <Budget year='2001'>
    <xsl:for-each select="LineItem[generate-id()
     = generate-id(key('agencies',AgencyCode)[1])]">
      <Agency>
        <Name><xsl:value-of select="AgencyName"/></Name>
        <Code><xsl:value-of select="AgencyCode"/></Code>
        <xsl:for-each
          select="/Budget/LineItem[AgencyCode
          =current()/AgencyCode]
           [generate-id() =
             generate-id(key('bureaus',
                   concat(AgencyCode, '+', BureauCode))[1])]">
          <Bureau>
            <Name><xsl:value-of select="BureauName"/></Name>
            <Code><xsl:value-of select="BureauCode"/></Code>
            <xsl:for-each select="/Budget/LineItem
                [AgencyCode=current()/AgencyCode]
                [BureauCode=current()/BureauCode]
                [generate-id() = generate-id(key('accounts',
                 concat(AgencyCode,'+',BureauCode,'+',
                                          AccountCode))[1])]">
              <Account>
                <Name>
                  <xsl:value-of select="AccountName"/>
                </Name>
                <Code>
                  <xsl:value-of select="AccountCode"/>
                </Code>
                <xsl:for-each select=
                  "/Budget/LineItem
                   [AgencyCode=current()/AgencyCode]
                   [BureauCode=current()/BureauCode]
                   [AccountCode=current()/AccountCode]
                   [generate-id()=generate-id(
                     key('subfunctions' concat(AgencyCode,'+',
                     BureauCode,'+',AccountCode,'+',
                     SubfunctionCode))[1])]">
```

```
                        <Subfunction BEACategory="{BEACategory}"
                         BudgetIndicator="{On-Off-BudgetIndicator}">
                          <Title>
                           <xsl:value-of select="SubfunctionTitle"/>
                          </Title>
                          <Code>
                           <xsl:value-of  select="SubfunctionCode"/>
                          </Code>
                          <Amount>
                            <xsl:value-of select="FY2001"/>
                          </Amount>
                        </Subfunction>
                      </xsl:for-each>
                    </Account>
                  </xsl:for-each>
                </Bureau>
              </xsl:for-each>
            </Agency>
          </xsl:for-each>
        </Budget>
      </xsl:template>

    </xsl:stylesheet>
```

The algorithm for converting flat data to hierarchical data with XSLT is known as the *Muenchian method* after its inventor, Steve Muench of Oracle. The trick of the Muenchian method is to use the `xsl:key` element and the `key()` function to create node sets of all the `LineItem` elements that share the same agency, bureau, account, or subfunction. Inside the template, the `generate-id()` function is used to compare the current node to the *first* node in any given group. Output is generated only if we are indeed processing the first `Agency`, `Bureau`, `Account,` or `Subfunction` element with a specified code. Also note, that the `select` attributes in the `xsl:for-each` elements keep returning to the root rather than processing children and descendants as is customary. This reflects the fact that the hierarchy in the input is not the same as the hierarchy in the output.

One minor advantage of using XSLT instead of Java data structures is that XSLT preserves the order of the input data. You'll notice that the output begins with the Legislative Branch agency, bureau, and Receipts, Central fiscal operations account—the same as the input data does. This was not the case for the output produced by Java.

Note

XSLT 2.0 will make it much easier to write stylesheets that group elements in this fashion. This will likely involve a new `xsl:for-each-group` element that groups elements according to an XPath expression, and a `current-group()` function that selects all members of the current group so that they can be processed together.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Budget year="2001">
   <Agency>
      <Name>Legislative Branch</Name>
      <Code>001</Code>
      <Bureau>
         <Name>Legislative Branch</Name>
         <Code>00</Code>
         <Account>
            <Name>Receipts, Central fiscal operations</Name>
            <Code/>
            <Subfunction BEACategory="Mandatory"
              BudgetIndicator="On-budget">
               <Title>Central fiscal operations</Title>
               <Code>803</Code>
               <Amount>0</Amount>
            </Subfunction>
            <Subfunction BEACategory="Net interest"
              BudgetIndicator="On-budget">
               <Title>Other interest</Title>
               <Code>908</Code>
               <Amount>0</Amount>
            </Subfunction>
         </Account>
         <Account>
            <Name>Charges for services to trust funds</Name>
            ...
```

The XML Query Language

XSLT is Turing complete. Nonetheless, some operations are more than a little cumbersome in XSLT. XSLT's inventors definitely did not envision using the Muenchian method to impose hierarchy. The W3C has begun work on a language more

Caution
This section describes bleeding-edge technology. The broad picture presented here is likely to be correct, but the details are almost certain to change. Furthermore, the exact subset of XQuery implemented by early experimental tools varies significantly from one product to the next.

suitable for querying XML documents, called, simply enough, the *XML Query Language,* or *XQuery* for short. XQuery is to XML documents what SQL is to relational tables. However, XQuery is limited to SELECT. It has no equivalent of INSERT, UPDATE, or DELETE. It is a read-only language.

XQuery queries are not in general well-formed XML. Although there is an XML syntax for XQuery, it is not intended to be used by human beings. Instead humans are supposed to write in a more natural 4GL syntax, which will be compiled into XML documents if necessary. If you think about it, this shouldn't be so surprising: SQL statements aren't tables. Why should XQuery statements be XML documents?

The basic nature of an XQuery query is the FLWR (pronounced "flower") statement. *FLWR* is the acronym for for-let-where-return, the basic form of an XQuery query. In brief, `for` each node in a node set, `let` a variable have a certain value, `where` some condition is true, and `return` an XML fragment based on the values of these variables. Variables are set and XML is returned using XPath 2.0 expressions.

For example, here's an XQuery that generates a list of agency names from the flat XML budget:

```
for $name in document("budauth.xml")/Budget/LineItem/AgencyName
return $name
```

The `for` clause iterates over every node in the node set returned by the XPath 2.0 expression `document("budauth.xml")/Budget/LineItem/AgencyName`. This expression returns a node set containing 3,175 `AgencyName` elements. The XQuery variable `$name` is set to each of these elements in turn. The `return` clause is evaluated for each value of `$name`. In this case, the `return` clause says simply to return the node to which the `$name` variable currently points. In this example, the `$name` variable always points to an `AgencyName` element; therefore, the output would begin like this:

```
<AgencyName>Legislative Branch</AgencyName>
<AgencyName>Legislative Branch</AgencyName>
<AgencyName>Legislative Branch</AgencyName>
```

```
<AgencyName>Legislative Branch</AgencyName>
<AgencyName>Legislative Branch</AgencyName>
...
```

This is not a well-formed XML document because it does not have a root element. However, it is a well-formed XML document *fragment*.

You can use the XPath 2.0 `distinct-values()` function around the XPath expression to select only one of each `AgencyName` element:

```
for $name in distinct-values(document("budauth.xml")/Budget/
    LineItem/AgencyName)
return $name
```

The output would now begin like this, listing each agency name only once:

```
<AgencyName>Legislative Branch</AgencyName>
<AgencyName>Judicial Branch</AgencyName>
<AgencyName>Department of Agriculture</AgencyName>
<AgencyName>Department of Commerce</AgencyName>
<AgencyName>Department of Defense--Military</AgencyName>
...
```

As well as copying existing elements, XQuery can create new elements. You can type the tags precisely where you want them to appear. To include the value of a variable (or other expression) inside the tags, enclose it in curly braces. For example, the following query places `<Name>` and `</Name>` tags around each agency name, rather than `<AgencyName>` and `</AgencyName>`. Notice also that it selects only the text content of each `AgencyName` element, rather than the complete element node:

```
for $name in distinct-values(
    document("budauth.xml")//AgencyName/text())
return <Name>{ $name }</Name>
```

The output now begins like this:

```
<Name>Legislative Branch</Name>
<Name>Judicial Branch</Name>
<Name>Department of Agriculture</Name>
<Name>Department of Commerce</Name>
```

```
<Name>Department of Defense--Military</Name>
...
```

More complex queries typically require multiple variables. These can be set in a `let` clause based on XPath expressions that refer to the variable in the `for` clause. For example, this query selects distinct agency codes but returns agency names:

```
for $code in distinct-values(document("budauth.xml")//AgencyCode)
let $name := $code/../AgencyName
return $name
```

A `where` clause can further restrict the members of the node set for which results are generated. `where` conditions can use boolean connectors such as `and`, `or`, and `not()`. For example, this query finds all the bureaus in the Department of Agriculture:

```
for $bureau in distinct-values(document("budauth.xml")/Budget/
    LineItem/BureauName)
where $bureau/../AgencyName = "Department of Agriculture"
return $bureau
```

XQuery expressions can nest. That is, the `return` statement of the FLWR may contain another FLWR. For example, this statement lists all the bureau names inside their respective agencies:

```
for $ac in distinct-values(document("budauth.xml")//AgencyCode)
return
  <Agency>
    <Name>{ $ac/../AgencyName/text() }</Name>
    {
   for $bc in distinct-values(document("budauth.xml")//BureauCode)
      where $bc/../AgencyCode = $ac
      return
        <Bureau>
          <Name>{ $bc/../BureauName/text() }</Name>
        </Bureau>
    }
  </Agency>
```

The output now begins like this:

```
<Agency>
  <Name>Legislative Branch</Name>
  <Bureau>
    <Name>Legislative Branch</Name>
  </Bureau>
  <Bureau>
    <Name>Senate</Name>
  </Bureau>
  <Bureau>
    <Name>House of Representatives</Name>
  </Bureau>
  <Bureau>
    <Name>Joint Items</Name>
  </Bureau>
...
```

This is all the syntax needed to write a query that will convert flat budget data such as that produced by Example 4.2 into a hierarchical XML document. Example 4.13, which selects the data from 2001, demonstrates such a query.

Example 4.13 An XQuery That Converts Flat Data to Hierarchical Data

```
<Budget year="2001">
  {
  for $ac in distinct-values(document("budauth.xml")//AgencyCode)
  return
    <Agency>
      <Name>{ $ac/../AgencyName/text() }</Name>
      <Code>{ $ac/text() }</Code>
      {
        for $bc
         in distinct-values(document("budauth.xml")//BureauCode)
        where $bc/../AgencyCode = $ac
        return
          <Bureau>
            <Name>{ $bc/../BureauName/text() }</Name>
            <Code>{ $bc/text() }</Code>
            {
            for $acct in distinct-values(
                  document("budauth.xml")//AccountCode)
                 where $acct/../AgencyCode = $ac
                  AND $acct/../BureauCode = $bc
                 return
                   <Account
                     BEACategory="{ $acct/../BEACategory/text() }">
                     <Name>{ $acct/../AccountName/text() }</Name>
                     <Code>{ $acct/text() }</Code>
                     {
                       for $sfx
                         in document("budauth.xml")//SubfunctionCode
                       where $sfx/../AgencyCode = $ac
                         and $sfx/../BureauCode = $bc
                         and $sfx/../AccountCode = $acct
                       return
                         <Subfunction>
                      <Title>{$sfx/../SubfunctionTitle/text()}</Title>
                           <Code>{ $sfx/text() }</Code>
                           <Amount>{ $sfx/../FY2001/text() }</Amount>
                         </Subfunction>
                    }
                 </Account>
                }
              </Bureau>
          }
        </Agency>
      }
    </Budget>
```

There's a lot more to XQuery, but this should give you an idea of what it can do. It's definitely worth a look any time you need to perform database-like operations on XML documents.

Relational Databases

Relational databases are another common source of data for XML documents. Some databases such as FileMaker Pro have built-in support for outputting tables as XML documents. Others do not. However, even if your database can export tables as XML documents, its XML format may not be the XML format you want. Fortunately, as long as there's a JDBC driver for your database of choice, it's not hard to extract the information from it and write that information into an XML document in the desired form.

For this example, I'll use the same budget data previously read out of a CSV file in a single relational table that reflects the original flat structure of the files distributed by the Office of Management and Budget. Doubtless they have the data in their own relational databases, probably divided up into multiple tables; but they don't publish it that way. They do state, "If you plan to use these data in a relational database, you should designate the following fields as 'primary' to uniquely identify each row of data: agency code, bureau code, account code, subfunction code, BEA category, Grant/Nongrant, and On- Off-budget field." The SQL `CREATE TABLE` statement that initializes this table is as follows:

```
CREATE TABLE BudgetAuthorizationTable (
  AgencyCode             CHAR(3),
  AgencyName             VARCHAR(89),
  BureauCode             CHAR(2),
  BureauName             VARCHAR(89),
  AccountCode            VARCHAR(6),
  AccountName            VARCHAR(160),
  TreasuryAgencyCode     CHAR(2),
  SubfunctionCode        CHAR(3),
  SubfunctionTitle       VARCHAR(72),
  BEACategory            VARCHAR(13),
  On-Off-BudgetIndicator VARCHAR(10),
  FY1976                 INTEGER,
  TransitionQuarter      INTEGER,
  FY1977                 INTEGER,
  FY1978                 INTEGER,
  FY1979                 INTEGER,
  FY1980                 INTEGER,
  FY1981                 INTEGER,
  FY1982                 INTEGER,
  FY1983                 INTEGER,
  FY1984                 INTEGER,
  FY1985                 INTEGER,
  FY1986                 INTEGER,
  FY1987                 INTEGER,
  FY1988                 INTEGER,
  FY1989                 INTEGER,
  FY1990                 INTEGER,
  FY1991                 INTEGER,
  FY1992                 INTEGER,
  FY1993                INTEGER,
  FY1994                INTEGER,
  FY1995                INTEGER,
  FY1996                INTEGER,
  FY1997                INTEGER,
  FY1998                INTEGER,
  FY1999                INTEGER,
  FY2000                INTEGER,
  FY2001                INTEGER,
  FY2002                INTEGER,
  FY2003                INTEGER,
  FY2004                INTEGER,
  FY2005                INTEGER,
  FY2006                INTEGER,
  PRIMARY KEY (AgencyCode, BureauCode, AccountCode,
               SubfunctionCode, BEACategory, On-Off-BudgetIndicator)
);
```

The specific database I chose for this example is Microsoft Excel, mostly because it could very easily read the CSV files with which I started. Excel isn't the best example of a relational database. In fact, it isn't a relational database at all. But Excel does allow you to define a range of cells as a table, and then associate that table with an ODBC data source. This data source can then be read with SQL using JDBC through the `JdbcOdbcDriver`, which is all I really want to show here. Aside from the choice of JDBC driver, all statements will be completely database independent. The name of the ODBC data source is `budauth`. The name of the table in that source is `BudgetAuthorizationTable`.

Extracting the necessary data from the database is just a matter of SQL, which as a Java developer, you access via JDBC. To some extent you can let the database do the hard work for you by executing the right sequence of SQL commands. In this case, this simply consists of a long sequence of nested `SELECT` statements. SQL's `DISTINCT` operator will be particularly helpful. The contortions of the Muenchian method in Example 4.12 were a roundabout way of providing a distinct operation in XSLT. Example 4.14 demonstrates.

Example 4.14 A Program That Connects to a Relational Database Using JDBC and Converts the Table to Hierarchical XML

```
import java.sql.*;
import java.io.*;
```

```
public class SQLToXML {

  public static void main(String[] args ) {

    // Load the ODBC driver
    try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
    }
    catch (ClassNotFoundException e) {
      System.err.println("Could not load the JDBC-ODBC Bridge");
      return;
    }

    try {
      Writer out = new OutputStreamWriter(System.out, "UTF8");
      out.write("<?xml version=\"1.0\"?>\r\n");
      out.write("<Budget>\r\n");
      writeAgencies(out);
      out.write("</Budget>\r\n");
      out.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }


  }

  private static void writeAgencies(Writer out)
   throws IOException {

    Connection conn = null;
    Statement stmnt = null;
    try {
      conn = DriverManager.getConnection(
       "jdbc:odbc:budauth", "", "");
      stmnt = conn.createStatement();
      String query = "SELECT DISTINCT AgencyName, AgencyCode"
       + " FROM BudgetAuthorizationTable;";
      ResultSet agencies = stmnt.executeQuery( query );
      while( agencies.next() ) {

        String agencyName = agencies.getString("AgencyName");
        agencyName = escapeText(agencyName);
        String agencyCode = agencies.getString("AgencyCode");
        out.write("  <Agency>\r\n");
        out.write("    <Name>" + agencyName + "</Name>\r\n");
        out.write("    <Code>" + agencyCode + "</Code>\r\n");
        writeBureaus(out, conn, agencyCode);
        out.write("  </Agency>\r\n");

      }
    }
    catch (SQLException e) {
      System.err.println(e);
      e.printStackTrace();
    }
    finally {
      try {
        stmnt.close();
        conn.close();
      }
      catch(SQLException e) {
        System.err.println(e);
      }
    }

  }

  private static void writeBureaus(Writer out, Connection conn,
   String agencyCode) throws IOException, SQLException {

    String query
     = "SELECT DISTINCT BureauName, BureauCode "
     + "FROM BudgetAuthorizationTable WHERE AgencyCode='"
     + agencyCode + "';";
    Statement stmnt = conn.createStatement();
    ResultSet bureaus = stmnt.executeQuery(query);

    while( bureaus.next() ) {
      String bureauName = bureaus.getString("BureauName");
    bureauName = escapeText(bureauName);
    String bureauCode = bureaus.getString("BureauCode");
    out.write("    <Bureau>\r\n");
    out.write("      <Name>" + bureauName + "</Name>\r\n");
    out.write("      <Code>" + bureauCode + "</Code>\r\n");
    writeAccounts(out, conn, agencyCode, bureauCode);
    out.write("    </Bureau>\r\n");
    out.flush();
  }

}

private static void writeAccounts(Writer out, Connection conn,
 String agencyCode, String bureauCode)
 throws IOException, SQLException {

  String query = "SELECT DISTINCT AccountName, AccountCode "
   + "FROM BudgetAuthorizationTable WHERE AgencyCode='"
   + agencyCode + "' AND BureauCode='" + bureauCode + "';";
  Statement stmnt = conn.createStatement();
  ResultSet accounts = stmnt.executeQuery(query);

  while( accounts.next() ) {
    String accountName = accounts.getString("AccountName");
    accountName = escapeText(accountName);
    String accountCode = accounts.getString("AccountCode");
    out.write("      <Account>\r\n");
    out.write("        <Name>" + accountName + "</Name>\r\n");
    out.write("        <Code>" + accountCode + "</Code>\r\n");
    writeSubfunctions(
     out, conn, agencyCode, bureauCode, accountCode
    );
    out.write("      </Account>\r\n");
    out.flush();
  }

}

private static void writeSubfunctions(Writer out,
 Connection conn, String agencyCode, String bureauCode,
 String accountCode) throws IOException, SQLException {
  String query = "SELECT * FROM BudgetAuthorizationTable"
   + " WHERE AgencyCode='" + agencyCode + "' AND BureauCode='"
   + bureauCode + "' AND AccountCode='" + accountCode + "';";
  Statement stmnt = conn.createStatement();
  ResultSet subfunctions = stmnt.executeQuery(query);

  while( subfunctions.next() ) {
    String subfunctionTitle
     = subfunctions.getString("SubfunctionTitle");
    subfunctionTitle = escapeText(subfunctionTitle);
    String subfunctionCode
     = subfunctions.getString("SubfunctionCode");
    out.write("        <Subfunction>\r\n");
    out.write("          <Name>");
    out.write(subfunctionTitle);
    out.write("</Name>\r\n");
    out.write("          <Code>");
    out.write(subfunctionCode);
    out.write("</Code>\r\n");
    out.write("          <Amount year='TransitionQuarter'>");
    out.write(subfunctions.getInt("TransitionQuarter")
     + "</Amount>\r\n");
    for (int year = 1976; year <= 2006; year++) {
      String name = "FY" + year;
      long amt = subfunctions.getInt(name) * 1000L;
      out.write("          <Amount year='" + year + "'>");
      out.write(amt + "</Amount>\r\n");
    }
    out.write("        </Subfunction>\r\n");
    out.flush();
  }

}

public static String escapeText(String s) {

  if (s.indexOf('&') != -1 || s.indexOf('<') != -1
   || s.indexOf('>') != -1 || s.indexOf('"') != -1
   || s.indexOf('\'') != -1 ) {
    StringBuffer result = new StringBuffer(s.length() + 6);
    for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);
```

```
          if (c == '&') result.append("&amp;");
          else if (c == '<') result.append("&lt;");
          else if (c == '"') result.append("&quot;");
          else if (c == '\'') result.append("&apos;");
          else if (c == '>') result.append("&gt;");
          else result.append(c);
        }
        return result.toString();
      }
      else {
        return s;
      }

    }

  }
```

The basic approach here should be quite familiar by now. Tags are stored in string literals. These tags are written onto a `Writer` along with element content and attribute values that have been read from the input. The difference in this case is that because the input comes from a relational database, the program can use SQL to get the input it wants when it wants it. The Java program does not need to put itself out to accommodate the order of the input data. In essence this program is nothing more than four nested loops. The outermost loop iterates over the different agencies. This contains a loop that iterates over the bureaus within that agency. This contains a loop that iterates over the accounts within that bureau. This contains the innermost loop, which iterates over the subfunctions within the account. Compare this to the contortions the other programs had to perform. Only the XQuery solution was as straightforward as this—not surprising, because it also allows the client to specify which data it wants to receive.

■ Summary

Converting data from non-XML formats to XML is not particularly difficult, but there are no magic bullets to do it for you. You just have to roll up your sleeves and write the code. Once you've parsed the input data and organized it in the form you want, outputting it as XML is not hard. However, parsing the non-XML input data can be quite a challenge.

Unless the data is truly flat (and real-world data very rarely is), you'll generally want to arrange your XML markup to indicate the hierarchy and structure of the data. There are many different ways to perform the conversion that differ primarily in where the work is done: at the beginning, middle, or end. Thus, there are three broad approaches. If you're extracting data from a relational database using JDBC, you may be able to make multiple SQL queries, possibly joining tables, so that the data that enters your program is already in more or less the form and order you want. Alternately, if the input is coming from flat files, you might read it into a flat structure such as a list or an array, and then use Java code to rearrange it in a more hierarchical structure such as a tree. Finally, you could read the data in the most naive format possible, write it out again as almost an XML-copy of the original structure, and then post-process the initial XML document with XSLT or XQuery to get the structure you want. All three approaches produce the same document in the end. The approach you choose will depend largely on your relative comfort with SQL, Java, and XSLT.