CHAPTER 8

Bridging Application Data Structure and XML

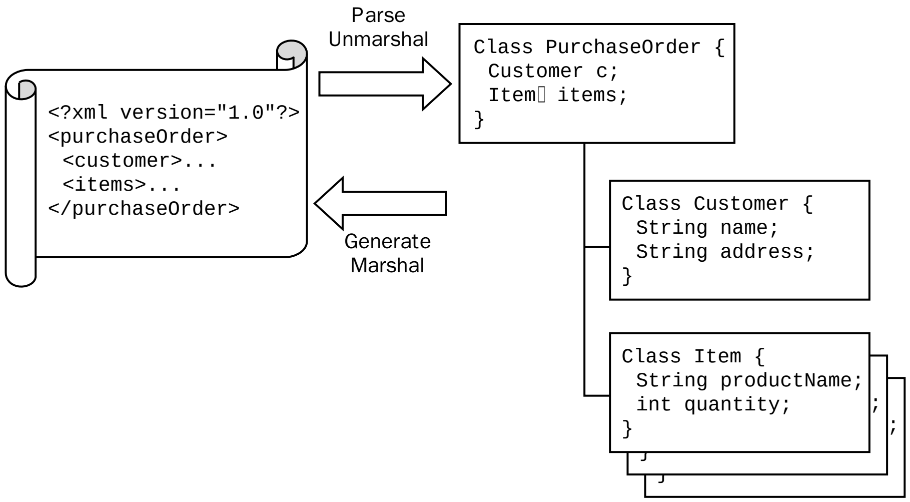
8.1 Introduction

The theme of this book is how XML and Java interact with each other. In Chapter 2 (parsing) we explained how to transform an XML document into a Java data structure based on DOM and SAX. Chapter 3 (generation) showed how to generate an XML document from a Java program. Chapter 4 (DOM/DOM2) and Chapter 5 (SAX/SAX2) dealt with standard APIs to access an XML document from a Java program. Common to these techniques is the concept of mapping between XML documents and Java data structures. However, these are not the only ways to do mapping between XML and Java. This chapter introduces various mapping patterns and techniques.

As we discussed in Chapter 1, and as we will see in Chapters 12 (messaging) and 13 (Web services), XML is a data format suitable for *data exchange* and is not necessarily suitable for *processing*. From an application programmer's point of view, XML documents exist in an external data format only, and once they are read into memory, the programmer deals with the internal data structure—that is, Java objects for implementing application-specific logic. XML processors are responsible for converting XML documents into Java data in the form of DOM or SAX, but these data structures rarely represent your application's data structure. For example, suppose that you parse a purchase order document and receive a DOM structure. You need the customer's name and the serial number to process the data. From a `<customer>` element, you may need to scan its child nodes to find a `<name>` node and a `<serialNumber>` node and then convert them into appropriate Java data types. Instead of a DOM tree, what the application programmer really wants is Java objects reflecting the application data structure, such as class `Customer`. This class has the `name` and `serialNumber` fields, and these fields are to be filled with the data extracted from the XML document. This eliminates the extra code of scanning a DOM tree and simplifies the application code. Therefore, it is common that application programmers convert a DOM tree or a SAX event stream into an application-specific data representation before any application-specific process is executed.

In the programming language literature, the concept of mapping between internal data structures and external octet sequences is common, and the terms marshal and unmarshal are used for describing the mapping processes (see Figure 8.1). An XML document is an octet stream. Therefore, parsing an XML document can be considered to be unmarshaling, while generating an XML document can be considered as marshaling.

In this chapter, we explain that there are certain patterns in mappings between XML documents and application data. In Section 8.2, we consider mappings where the application data structure and the XML document structure are isomorphic. If the application data structure is slightly different from the input XML document structure, the use of XSLT to adjust the structure is a standard technique. We explain this technique in Section 8.3. Two-dimensional arrays, or tables, are also a common data structure. In Section 8.4, we briefly discuss tables as the application data structure. The general technique of mapping between XML documents and relational tables is covered in detail in Chapter 11, XML and Databases. However, we explain mapping for one special type of table—*hash tables,* in this chapter. Section 8.5 shows a useful technique of representing an XML document as a hash table. In more complex cases, the application data structure may be represented



```
<?xml version="1.0"?>
<purchaseOrder>
 <customer>...
 <items>...
</purchaseOrder>
```

```
Class PurchaseOrder {
 Customer c;
 Item[] items;
}
```

```
Class Customer {
 String name;
 String address;
}
```

```
Class Item {
 String productName;
 int quantity;
}
```

Figure 8.1 Marshaling and unmarshaling

as a graph. We give an example of mapping an XML document into a graph structure in Section 8.6. In Chapter 15, Data Binding, we revisit mappings and explore how to automate mappings between the application data structure and the XML document structure.

8.2 Mapping to Almost Isomorphic Tree Structures

The most typical mapping occurs when the structure of XML documents reflects the application data structure and thus the mapping is almost isomorphic.[1] We use the hypothetical purchase order document, `po.xml`, in Listing 8.1 as our example.

Listing 8.1 Purchase order document, `chap08/isomorphic/po.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE purchaseOrder SYSTEM "PurchaseOrder.dtd">
<purchaseOrder>
    <customer>
        <name>Robert Smith</name>
        <customerId>788335</customerId>
        <address>8 Oak Avenue, New York, US</address>
    </customer>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <shipDate>2002-09-03</shipDate>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>2002-08-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

Our application reads in this purchase order document, calculates the price, and generates an invoice. One natural modeling of this program is to represent each of the concepts, such as purchase order, customer, and item, as a Java object. Hence, the three classes shown in Listings 8.2, 8.3, and 8.4 are to be prepared.

[1] The data binding tools that we describe in Chapter 15 are most useful in such cases.

Listing 8.2 `PurchaseOrder` class, `chap08/isomorphic/PurchaseOrder.java`

```
public class PurchaseOrder {

    Customer customer;
    String comment;
    Vector items = new Vector();
   ...
    }
```

Listing 8.3 `Customer` class, `chap08/isomorphic/Customer.java`

```
public class Customer {

    String name;
    int customerId;
    String address;
   ...
}
```

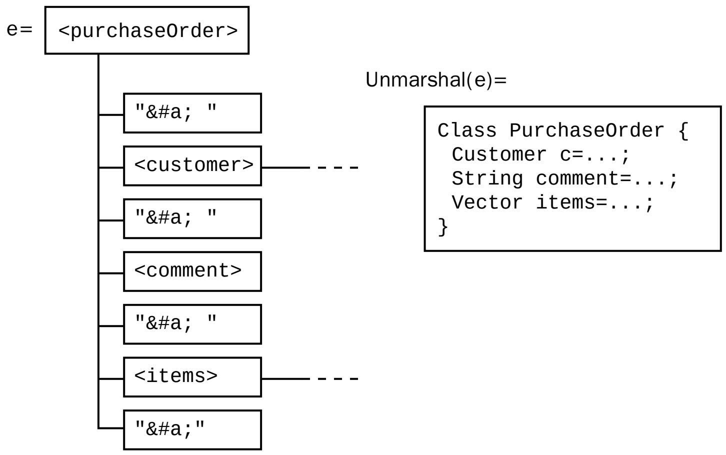
Listing 8.4 `Item` class, `chap08/isomorphic/Item.java`

```
public class Item {

    String partNum;
    String productName;
    int quantity;
    float usPrice;
    String shipDate;
   ...
}
```

How can we map an input XML document into instances of these classes? For simplicity, let us assume that we have already parsed an XML document and have a DOM tree. What we need to do is recursively scan this DOM tree, and for each element that represents a concept to be represented as a Java object, generate an instance of the corresponding class. For example, upon encountering a `<purchaseOrder>` element during the DOM tree scan, create a new instance of the `PurchaseOrder` class, as shown in Figure 8.2.

To do this, we provide a static method called `unmarshal()` in the class `PurchaseOrder` (see Listing 8.5). This method takes a DOM node representing a `<purchaseOrder>` element as an input parameter and returns a new `PurchaseOrder` instance.



```
Class PurchaseOrder {
 Customer c=...;
 String comment=...;
 Vector items=...;
}
```

Figure 8.2 Creating a Java object from a DOM element

Listing 8.5 `unmarshal()` method for `PurchaseOrder` class, `chap08/isomorphic/PurchaseOrder.java` (continued)

```
static PurchaseOrder unmarshal(Element e) {
   PurchaseOrder po = new PurchaseOrder();
   for (Node c1=e.getFirstChild(); c1!=null; c1=c1.getNextSibling()) {
      if (c1.getNodeType()==Node.ELEMENT_NODE) {

         if (c1.getNodeName().equals("customer")) {
            // <customer> subelement
            po.setCustomer(Customer.unmarshal((Element)c1));

         } else if (c1.getNodeName().equals("comment")) {
            // <comment> subelement
            po.setComment(TypeConversion.toString(c1));

         } else if (c1.getNodeName().equals("items")) {
            // <items> subelement
            for (Node c2 = c1.getFirstChild();
               c2 != null;
               c2 = c2.getNextSibling()) {
               if (c2.getNodeType()==Node.ELEMENT_NODE) {
                  Element childElement2 = (Element)c2;
                  if (c2.getNodeName().equals("item")) {
                     po.addItem(Item.unmarshal((Element) c2));
                  }
               }
            }
         }
```

```
            }
        }
        return po;
    }
```

Parameter e of this method points to a <purchaseOrder> element in a DOM tree. This method first creates an instance of the PurchaseOrder class and then scans the child nodes of the element e to fill in the fields of the PurchaseOrder instance. For example, when the method encounters a <customer> element, it creates a Customer object by calling the static method unmarshal() of the class Customer and sets the created object to the customer field of the PurchaseOrder object (lines 48–50).

When seeing a <comment> element, the method converts the contents of the element into a String object and assigns it to the comment field (lines 52–54).

Child nodes of an <items> element are a repetition of <item> elements, so the method goes further down to its subelements and for each <item> element, the method creates an instance of the Item class and adds it to the item field of the PurchaseOrder (lines 56–68). As a helper class for handling a type conversion from a DOM node to a Java primitive type, we write a simple class called TypeConversion, shown in Listing 8.6.

Listing 8.6 TypeConversion class, chap08/isomorphic/TypeConversion.java

```
package chap08.isomorphic;

 import org.w3c.dom.Node;

 public class TypeConversion {

    static String toString(Node n) {
        String content = n.getFirstChild().getNodeValue();
        return content;
    }

    static int toInteger(Node n) {
        String content = n.getFirstChild().getNodeValue();
        return Integer.parseInt(content);
    }

    static float toFloat(Node n) {
        String content = n.getFirstChild().getNodeValue();
        return Float.parseFloat(content);
    }

}
```

What about unmarshaling of the `Customer` class? As we did for our `PurchaseOrder` class, we write a static method called `unmarshal()` that is to be called whenever a `<customer>` element is found. The implementation of this method is shown in Listing 8.7. You may notice that it has exactly the same pattern as the `unmarshal()` method of our `PurchaseOrder` class.

Listing 8.7 `unmarshal()` method for `Customer` class, `chap08/isomorphic/Customer.java` (continued)

```
    static Customer unmarshal(Element e) {
        Customer co = new Customer();
        for (Node c1=e.getFirstChild(); c1!=null; c1=c1.getNextSibling()) {
            if (c1.getNodeType()==Node.ELEMENT_NODE) {

                if (c1.getNodeName().equals("name")) {
                    // <name> subelement
                    co.setName(TypeConversion.toString(c1));

                } else if (c1.getNodeName().equals("customerId")) {
                    // <customreId> subelement
                    co.setCustomerId(TypeConversion.toInteger(c1));

                } else if (c1.getNodeName().equals("address")) {
                    // <address> subelement
                    co.setAddress(TypeConversion.toString(c1));
                }
            }
        }
        return co;
    }
```

Similarly, we can build the `unmarshal()` method for the class `Item`. In this code, shown in Listing 8.8, we also extract the value of the attribute `partNum` (line 57).

Listing 8.8 `unmarshal()` method for `Item` class, `chap08/isomorphic/Item.java` (continued)

```
    static Item unmarshal(Element e) {
        Item item = new Item();
[57]    item.setPartNum(e.getAttribute("partNum"));
        for (Node c1=e.getFirstChild(); c1!=null; c1=c1.getNextSibling()) {
            if (c1.getNodeType()==Node.ELEMENT_NODE) {

                if (c1.getNodeName().equals("productName")) {
                    // <productName> subelement
                    item.setProductName(TypeConversion.toString(c1));
                } else if (c1.getNodeName().equals("quantity")) {
                    // <quantity> subelement
                    item.setQuantity(TypeConversion.toInteger(c1));

                } else if (c1.getNodeName().equals("USPrice")) {
                    // <USPrice> subelement
                    item.setUSPrice(TypeConversion.toFloat(c1));

                } else if (c1.getNodeName().equals("shipDate")) {
                    // <shipDate> subelement
                    item.setShipDate(TypeConversion.toString(c1));
                }
            }
        }
        return item;
    }
```

Now we are ready to convert a whole DOM tree into our application data model. We need to parse an input XML document and give the resulting DOM tree to one of the `unmarshal()` methods that we programmed. The `main()` method, shown in Listing 8.9, does exactly that.

Listing 8.9 `main()` method for `PurchaseOrder` class, `chap08/isomorphic/PurchaseOrder.java` (continued)

```
    public static void main(String[] argv) throws Exception {
        if (argv.length < 1) {
            System.err.println("Usage: java chap08.PurchaseOrder file");
            System.exit(1);
        }
[79]    DocumentBuilderFactory factory =
[80]        DocumentBuilderFactory.newInstance();
[81]    DocumentBuilder builder = factory.newDocumentBuilder();
[82]    builder.setErrorHandler(new MyErrorHandler());
[83]    Document doc = builder.parse(argv[0]);

        Element root = doc.getDocumentElement();
        if (root.getNodeName().equals("purchaseOrder")) {
[87]        PurchaseOrder po = unmarshal(root);
            Customer co = po.getCustomer();
            System.out.println("************ Invoice *************  ");
            System.out.println("Date:"+ new java.util.Date());
            System.out.println("");
            System.out.println("To: "+co.getName());
            System.out.println("    "+co.getAddress());
            System.out.println("    Customer#:"+co.getCustomerId());
            System.out.println("");
            System.out.println("---------------------");
            int i=0;
            float total = (float)0.0;
            for (Iterator itr=po.getItems(); itr.hasNext(); i++) {
                Item item = (Item)itr.next();
                System.out.println("Item #"+i+" : "+item.getPartNum()+
                                   ", Quantity="+item.getQuantity()+
                                   ", UnitPrice="+item.getUSPrice()+
                                   ", ShipDate="+item.getShipDate());

                total += item.getUSPrice();
            }
            System.out.println("---------------------");
            System.out.println("total = $"+total);
        }
    }
```

Lines 79–83 parse the input and obtain a DOM tree. After checking that the root element is in fact a `purchaseOrder` element, we create a `PurchaseOrder` instance by calling the static method `unmarshal()` in line 87. The rest of the code is pure application logic that is responsible for generating an invoice.

What can we learn from this manual mapping? We have seen that if the mapping preserves the structural similarity between the XML document and the Java data structure, writing unmarshaling codes can be a fairly automatic task. You prepare one class for one complex element type (for example, `<purchaseOrder>`, `<customer>`, or `<item>`) that has a static method, `unmarshal()`, for scanning a DOM tree and creating a corresponding Java instance.

The question is, then, is it possible to automatically generate such mapping codes from a schema of input XML documents? The answer is yes. In Chapter 15, we describe a few of these tools.

8.3 Structure Adjustment by XSLT

In the previous section, we assumed that our XML documents and application data share the same structure. Sometimes this assumption does not hold in the real world, and the technique we just saw cannot be directly applied.

Consider our purchase order application. We use the same XML document, `po.xml`, (see Listing 8.1) as the input to our application. In this format, one purchase order can have multiple items in it. Suppose that we are asked to feed the XML documents into a legacy application whose data model allows only one

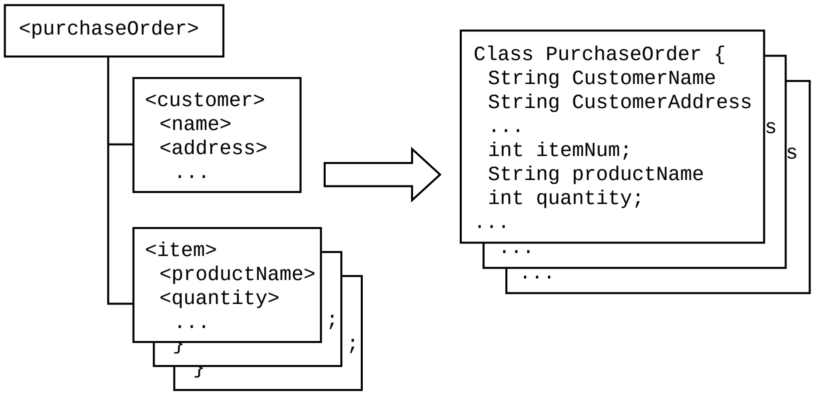


Figure 8.3 Structural mismatch

item per purchase order.[2] The mapping we need is something like that shown in Figure 8.3.

There is an apparent mismatch between the input XML structure and the application data structure, so the technique we used in the previous section cannot be directly applied. How can we write a program that creates the desired Java data structure, as shown in Figure 8.3, from this input? Modifying the program in the previous section is one possibility. To do that, however, you need to temporarily store the customer data that will later be assigned to the `PurchaseOrder` objects. This is not a big deal in terms of the number of lines of additional code, but you may need either a global variable or additional parameters in the `unmarshal()` methods.

Another way to solve this nonstraightforward mapping is to use XSLT to adjust the XML structure *before* processing. Suppose that the input XML has the form shown in Listing 8.10 instead of Listing 8.1. This time, the mapping is isomorphic and the technique in the previous section can be directly applied.

Listing 8.10 Transformed purchase order, `chap08/mismatch/po1.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrders>
    <purchaseOrder>
        <name>Robert Smith</name>
        <customerId>788335</customerId>
        <address>8 Oak Avenue, New York, US</address>
        <partNum>872-AA"</partNum>
        <productName>Lawnmower</productName>
```

[2] We encountered such an application when we were building an order entry system for IBM.

```
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <shipDate>2002-09-03</shipDate>
        </purchaseOrder>
        <purchaseOrder>
            <name>Robert Smith</name>
            <customerId>788335</customerId>
            <address>8 Oak Avenue, New York, US</address>
            <partNum>926-AA</partNum>
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>2002-08-21</shipDate>
        </purchaseOrder>
    </purchaseOrders>
```

How can we obtain Listing 8.10 from Listing 8.1? The answer is to use XSLT. XSLT is designed for this sort of transformation. It is a powerful language to do such things as:

- Extracting `<item>` elements from inside a `purchaseOrder/items` element
- Adding child elements of the `<customer>` element to each `<item>` element

The XSLT script `mismatch/flatten.xsl` (see Listing 8.11) does this transformation. Note that this transformation can be called from a Java program using the JAXP API, as we discussed in Section 7.2.2 of Chapter 7, so the extra generation and parsing are eliminated.

Listing 8.11 XSLT transformation script, `chap08/mismatch/flatten.xsl`

```
    <?xml version="1.0" encoding="UTF-8"?>
    <xsl:stylesheet
        version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:output indent="yes" encoding="UTF-8"/>

        <xsl:template match="/">
            <purchaseOrders>
                <xsl:apply-templates select="purchaseOrder/items"/>
            </purchaseOrders>
        </xsl:template>

        <xsl:template match="item">
            <purchaseOrder>
                <xsl:copy-of select="/purchaseOrder/customer/name"/>
                <xsl:copy-of select="/purchaseOrder/customer/customerId"/>
            <xsl:copy-of select="/purchaseOrder/customer/address"/>
            <partNum><xsl:value-of select="@partNum"/></partNum>
            <xsl:copy-of select="productName"/>
            <xsl:copy-of select="quantity"/>
            <xsl:copy-of select="USPrice"/>
            <xsl:copy-of select="shipDate"/>
        </purchaseOrder>
    </xsl:template>

</xsl:stylesheet>
```

Can you see the usefulness of XSLT in mapping from XML to an application-specific data structure? This technique also applies to applications that need to process logically equivalent but different DTD documents. For example, if your application is required to process both types of purchase orders as in Listings 8.1 and 8.10, a single application program, combined with an appropriate XSLT stylesheet, will be enough for processing both. This situation frequently occurs when there are multiple versions of DTDs for essentially the same set of documents.

So far we focused on cases where mappings are relatively straightforward. In other words, your application data structure has essentially a tree shape, and it reflects the structure of marshaled XML documents fairly well. Some developers argue that 80% of all application development is indeed of this type, so 80% of the time you can have a straightforward mapping as we saw in Section 8.2, especially when we use the XSLT technique for adjusting the structure. Having a straightforward mapping is even more important if you use a software tool that does the mapping automatically. In Chapter 15, we will see such software tools.

Of course, there is always the remaining 20%. When we are told to develop such an application, we need to write our own custom mapping code. Even in such cases, there are certain patterns that may be useful to you. The next three sections show some of these patterns that we encounter most frequently.

8.4 Mapping to Tables

Mapping to tables is one of the most common patterns. Even in our purchase order application, the content of an `<items>` element is a repetition of the same element type, `item`, so this part can be naturally represented as a table (see Figure 8.4). From an application programming point of view, a table is a data structure that is suitable for operations such as searching, sorting, and totaling specific columns. In addition, table-represented data can be easily exported into and imported from applications such as databases and spreadsheets.

```
<items>
  <item partNum="872-AA">
    <productName>Lawnmower</productName>
    <quantity>1</quantity>
    <USprice>148.95</USPrice>
    <shipDate>2002-09-03</shipDate>
  </item>
  <item partNum="926-AA">
    <productName>BabyMonitor</productName>
    <quantity>1</quantity>
    <USPrice>39.98</USPrice>
    <shipDate>2002-08-21</shipDate>
  </item>
</items>
```

partNum	productName	Quantity	USPrice	shipDate
872-AA	Lawnmower	1	148.95	2002-09-03
926-AA	Baby Monitor	1	39.98	2002-08-21
:	:	:	:	:

Figure 8.4 Table data

In Chapter 11, we discuss how to store XML documents into a relational database and how to retrieve relational database contents as XML documents. In Section 11.5, refer to Listing 11.6, which decomposes XML documents and stores them into relational tables, and Listing 11.8, which generates XML documents from these tables. These programs correspond to unmarshaling and marshaling of table data. In both cases, we define mapping based on the database schema and the schema of XML documents. Some commercial database management systems have tools to generate such mappings automatically.

Mapping to tables is most suitable when your data consists of a set of records that all have a common structure. If your data is semistructured—that is, it is essentially tree-structured—mapping to hash tables (described next) is another candidate to consider.

8.5 Mapping to Hash Tables

Another interesting mapping is to map XML documents to hash tables. Frequently, XML is used as the format of software configuration files. For example, Tomcat, the servlet container we explain in Chapter 10, uses a number of configuration files (such as `server.xml`) in XML. You may think that many configuration files are simple enough so that flat text files are good enough. However, in our experience, we know that configuration files keep growing as program development continues. At some point we are forced to segment a configuration file into several logical sections. We must also define a basic syntax of delimiter characters and escape characters. We will also face deciding what international character encoding to use. Considering all this, it is worth using XML for configuration files in the first place.

Let us consider a configuration file (see Listing 8.12) of a hypothetical application program for generating sales reports.

Listing 8.12 Configuration file, `chap08/hashtable/config.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
   <productData locationType="file">
      <defaultFileName>c:/productMarketing/products.xml
      </defaultFileName>
   </productData>
   <customerData locationType="db">
      <databaseURL>jdbc:db2:sales.ibm.com/customer</databaseURL>
      <userId>maruyama</userId>
      <passWord>montelac</passWord>
   </customerData>
   <reportFormat locationType="file">
      <defaultFileName>c:/CEO/monthlyReport.xml</defaultFileName>
      <reportTo>M. Murata</reportTo>
      <reportTo>K. Kosaka</reportTo>
   </reportFormat>
</config>
```

How can our application program access this configuration file? It is not likely that we want to scan the entire file and do some specific task (for example, summing numbers). Instead, each piece of data in the configuration file will be accessed whenever a specific component of our application needs that particular piece of data, using the name of the configuration parameter as the key. Therefore, a hash table with configuration parameters as its keys is a natural choice of data structure to hold the configuration data. If our program needs the value of a configuration parameter called `defaultFileName`, we can efficiently look up the hash table with this name as the key. In XML-based configuration files, parameter names can be expressed as path expressions, such as `/config/productData/defaultFileName`.[3]

Our configuration file can be expressed as the following hash table.

KEY	**VALUE**
/config/productData/@locationType	[file]
/config/productData/defaultFileName	[c:/productMarketing/products.xml]
/config/customerData/@locationType	[db]
/config/customerData/databaseURL	[jdbc:db2:sales.abc.com/customer]
/config/customerData/userId	[maruyama]

[3] In Section 11.5.2, we discuss storing XML data as a pair of an XPath expression and its value. We do the same here, except that instead of using XPath's positional parameter to ensure the uniqueness of a path's value (as in `/purchaseOrder[1]/shipTo[1]/street[1]`), we use path expressions without positional predicates (as in `/config/productData/defaultFileName`) and allow their values to have multiple values in a `Vector` object.

/config/customerData/passWord	[montelac]
/config/reportFormat/@locationType	[file]
/config/reportFormat/defaultFileName	[c:/CEO/monthlyReport.xml]
/config/reportFormat/reportTo	["M. Murata", "K. Kosaka"]

Now let us consider how we can map an XML file into such a hash table. We use a common technique of generating path expressions using SAX. Look at the SAX handler in Listing 8.13.[4]

Listing 8.13 `Config` class, `chap08/hashtable/Config.java`

```
    package chap08.hashtable;
    import java.io.IOException;
    import org.xml.sax.SAXException;
    import org.xml.sax.Attributes;
    import org.xml.sax.helpers.DefaultHandler;
    import javax.xml.parsers.SAXParser;
    import javax.xml.parsers.SAXParserFactory;
    import javax.xml.parsers.ParserConfigurationException;
    import java.util.Hashtable;
    import java.util.Vector;
    import java.util.Enumeration;

    public class Config extends DefaultHandler {

        private StringBuffer path;
        private StringBuffer textContent;
[17]    private Hashtable hashtable;

        public Config(String fn) throws SAXException, IOException,
        ParserConfigurationException {
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser parser = factory.newSAXParser();

            this.path = new StringBuffer();
[24]        this.hashtable = new Hashtable();
            parser.parse(fn, this);
        }

        public void startElement(String uri, String local, String qname,
        Attributes atts) throws SAXException {
            // Update path
            path.append('/');
```

[4] In a similar program in Section 11.5.2, we first parse the input to obtain a DOM tree and then compute a path expression for each node.

```
                path.append(qname);
                int nattrs = atts.getLength();
                for (int i=0; i<nattrs; i++) {
[34]                addValue(path.toString()+"/@"+atts.getQName(i), atts.
                    getValue(i));
                }
[36]            this.textContent = new StringBuffer();
            }

            public void endElement(String uri, String local, String qname)
            throws SAXException {
                if (this.textContent != null) {
[41]                addValue(path.toString(), this.textContent.toString());
                    this.textContent = null;
                }
                // Restore path
                int pathlen = path.length();
[46]            path.delete(pathlen-qname.length()-1,pathlen);
            }

            public void characters(char[] ch, int start, int length) throws
            SAXException {
                if (this.textContent != null) {
                    this.textContent.append(ch, start, length);
                }
            }

            Hashtable getHashtable() {
                return this.hashtable;
            }

            void addValue(String key, String value) {
                Vector v = (Vector)this.hashtable.get(key);
                if (v == null) {
                    v = new Vector();
                    this.hashtable.put(key,v);
                }
                v.add(value);
            }

            public static void main(String[] args) throws Exception {
                if (args.length < 1) {
                    System.err.println("Usage: java chap08.hashtable.Config
                    file");
                    System.exit(1);
                }
            Config theConfig = new Config(args[0]);
            Hashtable ht = theConfig.getHashtable();
            for (Enumeration e = ht.keys(); e.hasMoreElements(); ){
                String key = (String)e.nextElement();
                System.out.println(key+"="+ht.get(key));
            }
        }

    }
```

The hash table that we are going to build is declared in line 17 and initialized in line 24. A new entry is added to the hash table when an attribute (line 34) or an element with some text content (line 41) is found. The text content of an element is accumulated in a `StringBuffer` in a variable named `textContent`. This buffer is initialized when a start tag is found (line 36) and discarded when an end tag is found. Therefore, any characters in SAX events that occur before a start element or after an end element are ignored. This is OK because our XML documents have no MIXED content models.

Another interesting point of this program is the way to build path expressions. During the parsing process, the current path expression is kept in a `StringBuffer` in a variable named `path`. We do not need to keep track of this path expression in a stack because when we see an end tag, we can always recover the parent path expression by removing the element name plus one character (for the separator "/"), as we do in line 46.

Once an XML configuration file is mapped into a hash table, configuration parameters can be efficiently accessed from any part of our program. A great advantage of this approach is that because our mapping code does not hardcode any particular element names or attribute names, there is no need to modify the code when new configuration parameters are added. In fact, this mapping code works universally regardless of the schema of input XML documents.

This mapping is optimized for keyed access by path expressions. Mapping to hash tables does not make sense for applications that have different access patterns, such as traversing the entire document.

8.6 Mapping to Graph Structures

Trees, tables, and hash tables are optimized for different access patterns. Here, we consider another frequently used data structure: graphs. Graphs are convenient when the access pattern of your program involves following links between shared data.

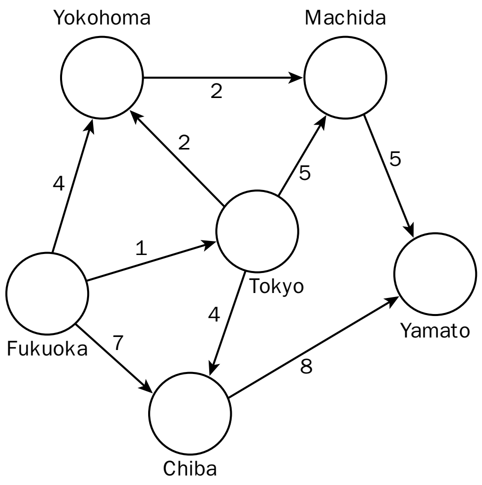


Figure 8.5 Graph data

Let us take an application that computes the shortest path between two cities as our example. Figure 8.5 shows our problem. Circles represent cities, and arrows between them represent connections between cities. The numbers associated with the arrows are the cost of moving to one city from another. What is the shortest path from Fukuoka to Yamato? What is the cost of the shortest path? There is a well-known, simple but efficient algorithm known as *Dijkstra's algorithm* to solve this problem.

We want to express this problem as an XML document and map it into a graph data structure, as shown in Figure 8.5, that is needed for implementing Dijkstra's algorithm efficiently. One natural XML encoding of the problem is something like `cities.xml`, shown in Listing 8.14.

Listing 8.14 Graph represented as XML, `chap08/graph/cities.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<FindShortestPath xmlns="urn:shortest-path">
   <StartCity>Fukuoka</StartCity>
   <TargetCity>Yamato</TargetCity>
   <Paths>
      <Path from="Fukuoka" to="Tokyo" cost="1"/>
      <Path from="Fukuoka" to="Yokohama" cost="4"/>
      <Path from="Fukuoka" to="Chiba" cost="7"/>
      <Path from="Tokyo" to="Chiba" cost="4"/>
      <Path from="Tokyo" to="Yokohama" cost="2"/>
      <Path from="Tokyo" to="Machida" cost="5"/>
      <Path from="Yokohama" to="Machida" cost="2"/>
      <Path from="Machida" to="Yamato" cost="5"/>
```

```
        <Path from="Chiba" to="Yamato" cost="8"/>
    </Paths>
</FindShortestPath>
```

We will develop code to convert this XML document into a graph structure, whose vertexes are represented by `Vertex` objects and whose edges are represented by `Edge` objects.

Part of the `Vertex` class is shown in Listing 8.15. A `Vertex` instance has a string, `label`, for the city's name and a list, `outgoingEdges`, that holds all the edges pointing to other cities (lines 12–13). In addition, it has two variables, `bestScore` and `bestPath`, to keep track of working values during the computation. All the vertexes are kept in a single hash table named `vertexList`. When a vertex is needed, we call the static method `register()` instead of creating a new instance. This method checks whether there is already a vertex for the city and returns it if there is.

Listing 8.15 `Vertex` class, `chap08/graph/Vertex.java`

```
    class Vertex  {

[12]    private String label;
[13]    private List outgoingEdges;
        private int bestScore;
        private Edge bestPath;
        private static Hashtable vertexList = new Hashtable();

        static Vertex register(String label) {
            Vertex v = (Vertex)vertexList.get(label);
            if (v==null) {
                v = new Vertex(label);
                vertexList.put(label,v);
            }
            return v;
        }

        Vertex(String label) {
            this.label = label;
            this.outgoingEdges = new LinkedList();
            this.bestScore = Integer.MAX_VALUE;
            this.bestPath = null;
        }

        void addEdge(Edge e) {
            this.outgoingEdges.add(e);
        } /* end excerpt1 */
        ...
    }
```

The Edge class has fromVertex and toVertex to keep both ends of the connection (see Listing 8.16). When a new edge is created, it registers itself in outgoingEdges of fromVertex.

Listing 8.16 Edge class, chap08/graph/Edge.java

```
class Edge {
   private Vertex fromVertex;
   private Vertex toVertex;
   private int cost;

   Edge(Vertex from, Vertex to, int c) {
      this.fromVertex = from;
      this.toVertex = to;
      this.cost = c;
      this.fromVertex.addEdge(this);
   }
   ...
}
```

Now that we have both our application data structure and the input XML structure, we are ready to consider the mapping between them. Our strategy is this. We will scan an input XML document. Whenever we encounter a city, whether it appears in a Path element or in TargetCity or StartCity, we register it using Vertex.register(). When we see a Path element, we also need to create an Edge instance. Because the constructor of the Edge class takes care of updating the appropriate variables in the structure being built, we do not need to do anything other than create a new instance of Edge.

We implement our strategy in a SAX handler, ShortestPath.java, shown in Listing 8.17.

Listing 8.17 ShortestPath class, chap08/graph/ShortestPath.java

```
package chap08.graph;
import java.io.IOException;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.util.*;

public class ShortestPath extends DefaultHandler {

   static final String SHORTEST_PATH_URI = "urn:shortest-path";
   StringBuffer buffer = null;
```

```
       static Vertex startCity;
       static Vertex targetCity;

       public void startElement(String uri, String local, String qname,
       Attributes atts) throws SAXException {
           if (!uri.equals(SHORTEST_PATH_URI)) return;
           if ( local.equals("StartCity") || local.equals("TargetCity")) {
               this.buffer = new StringBuffer();
           } else if (local.equals("Path")) {

[23]           String fromString = atts.getValue("from");
[24]           if (fromString==null) throw new SAXException("Attribute
               'from' missing");
[25]           Vertex from = Vertex.register(fromString);
[26]
[27]           String toString = atts.getValue("to");
[28]           if (toString==null) throw new SAXException("Attribute 'to'
               missing");
[29]           Vertex to = Vertex.register(toString);

               String costString = atts.getValue("cost");
               if (costString==null) throw new SAXException("Attribute
               'cost' missing");
[33]           new Edge(from,to,Integer.parseInt(costString));
           }
       }

       public void endElement(String uri, String local, String qname)
       throws SAXException {
           if (!uri.equals(SHORTEST_PATH_URI)) return;
           if (local.equals("StartCity")) {
[40]           this.startCity = Vertex.register(new String(this.buffer));
               buffer = null;
           } else if (local.equals("TargetCity")) {
[43]           this.targetCity = Vertex.register(new String(this.buffer));
               buffer = null;
           }
       }

       public void characters(char[] ch, int start, int length) throws
       SAXException {
           if (this.buffer != null) {
               this.buffer.append(ch, start, length);
           }
       }
       /* end excerpt1 */
```

For the `StartCity` and `TargetCity` elements, we need to accumulate possibly fragmented characters in text content using a `StringBuffer`. This is the same technique we have seen repeatedly in this book. These city names are registered when end tags are processed (lines 40 and 43).[5] When a `path` element is found, its three attributes are extracted, the from city and the to city are registered (lines 23–29), and an `Edge` object is created (line 33). In this way, we can map an input XML document into our application data structure.

For completeness, we show the rest of our main class, `ShortestPath`, in Listing 8.18.

Listing 8.18 `main()` method for `ShortestPath` class, `chap08/graph/ShortestPath.java` (continued)

```
public static void main(String[] argv) throws Exception {
    if (argv.length < 1) {
        System.err.println("Usage: java chap08.graph.ShortestPath
        file");
        System.exit(1);
    }

    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setNamespaceAware(true);
    SAXParser parser = factory.newSAXParser();
    ShortestPath handler = new ShortestPath();

    parser.parse(argv[0],handler);

    SortedSet open = new TreeSet(new VertexComparator());
    startCity.update(0,null,open);

    while (!open.isEmpty()) {
        Vertex v = (Vertex) open.first();
        open.remove(v);
        for (Iterator i = v.getEdges(); i.hasNext(); ) {
            Edge e = (Edge)i.next();
            e.getToVertex().update(v.getBestScore()+e.getCost(),e,
            open);
        }
    }

    Vertex.showAll();
}
```

[5] Dijkstra's algorithm computes the shortest paths from a given vertex to *all* vertexes simultaneously. Therefore, the value of `TargetCity` is not used in this program.

Lines 68–78 implement Dijkstra's algorithm. It uses a helper class, `VertexComparator`, to compare the current cost values between two `Vertex` objects. Its source is included on the CD-ROM.

As we have done in this example, if you need to map XML documents into a graph structure, you need to give labels to each node and refer to these labels to express the links between nodes. Having a hash table to record the mapping between node labels and node objects is essential in this process.

8.7 Summary

In this chapter, we have considered the general problem of mapping between XML documents and application data structures. In particular, we looked at mapping to trees, tables, and graphs. These are not the only techniques for mapping—if you have written more than a few XML applications, you should have already accumulated a list of useful "patterns" of mapping. If and when these patterns become sufficiently frequent, it will be worth developing a tool for automating the mapping. Chapter 15 covers such tools.