Section 1. Introduction

Should I take this tutorial?

This tutorial is designed to assist JavaTM developers who need to extract information from a database and place it into an XML document.

The tutorial assumes that you are already familiar with Java and XML in general, and the Document Object Model (DOM) in particular. You should be familiar with Java programming, but prior knowledge of database connections using JDBCTM is not required to master the techniques described in this tutorial. The tutorial briefly covers the basics of SQL. GUI programming knowledge is not necessary because application input/output is handled from the command line. The links in Resources on page 26 include referrals to tutorials on XML and DOM basics, and to a detailed SQL backgrounder.

What is this tutorial about?

XML works so well for storing data that it's inevitable that at some point someone will ask you to pull information from a database and manipulate it as though it were XML. JDBC is a vendor-independent method for accessing databases using Java. This tutorial explains how to instantiate and use a JDBC driver to connect to a database in order to retrieve information. It also explains the basics of SQL, and how to create and use the results of a JDBC query.

The goal of the tutorial is to extract data from a database and create a DOM document. The structure of the DOM document is determined by an XML mapping file, which demonstrates one way that XML files can be used for this purpose.

Tools

This tutorial will help you understand the topic even if you read the examples rather than trying them out. If you do want to try the examples as you go through this tutorial, make sure you have the following tools installed and working correctly:

* A text editor: XML and Java source files are simply text. To create and read them, a text editor is all you need.
* A Java environment such as the Java 2 SDK, which is available at http://java.sun.com/j2se/1.3/.
* Java APIs for XML Processing: Also known as JAXP 1.1, this is the reference implementation that Sun provides. You can download JAXP from http://java.sun.com/xml/xml_jaxp.html.
* Any database that understands SQL, as long as you have an ODBC or JDBC driver. You can find a searchable list of more than 150 JDBC drivers at http://industry.java.sun.com/products/jdbc/drivers. (If you have an ODBC driver, you can skip this step, and use The JDBC-ODBC bridge on page 5.) This tutorial uses JDataConnect, available at http://www.j-netdirect.com/Downloads.htm.

Conventions used in this tutorial
There are several conventions used to reinforce the material in this tutorial:

* Text that needs to be typed is displayed in a **bold monospace** font. In some code examples bold is used to draw attention to a tag or element being referenced in the accompanying text.
* **Emphasis/Italics** is used to draw attention to windows, dialog boxes, and feature names.
* A **monospace** font presents file and path names.
* Throughout this tutorial, code segments irrelevant to the discussion have been omitted and replaced with ellipses ( . . . )

---

**About the author**

Nicholas Chase has been involved in Web site development for companies including Lucent Technologies, Sun Microsystems, Oracle Corporation, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Florida. He is the author of three books on Web development, including *Java and XML From Scratch* (Que). He loves to hear from readers and can be reached at nicholas@nicholaschase.com.
Section 2. Accessing a database through JDBC

What is JDBC?

If you have already been through the "Using JDBC to insert XML data into a database" tutorial, feel free to skip ahead to.

It was not very long ago that in order to interact with a database, a developer had to use the specific API for that database. This made creating a database-independent application difficult, if not impossible.

JDBC is similar to ODBC (or Open Database Connectivity), which provides a standard API intermediary for accessing a database. As seen on the left, standard JDBC commands can be used, and the JDBC driver translates them into the native API for the database.

Nowhere in this tutorial is a specific database mentioned, because the choice is largely irrelevant. All commands are standard JDBC commands, which are translated by the driver into native commands for whatever database you choose. Because of this API independence, of sorts, you can easily use another database without changing anything in your application except the driver name, and possibly the URL of the connection, used when you Create the connection on page 6.

See Resources on page 26 for information on downloading the appropriate JDBC drivers for your database. Over 150 JDBC drivers are available, and for virtually any database.

There are even options for databases that do not have an available JDBC driver.
The JBDC-ODBC bridge

It is not necessary to have a specific JDBC driver for a database as long as an ODBC driver is available; you can use a JDBC-ODBC bridge instead. The application calls the bridge, which translates the commands into ODBC, and the ODBC driver translates them into the native API.

The JDBC-ODBC bridge is not the recommended way to access a database for various reasons, involving both performance and configuration -- commands must pass through two API's, and the ODBC driver must be installed and configured on every client -- but it is acceptable for testing and development if there is no pure JDBC driver available.

If you choose to use the bridge, create a System Data Source Name (DSN) on a Windows system by choosing Start>Settings>Control >ODBC Data Sources. Make note of the name of the DSN, as it will be referenced when you Create the connection on page 6.

Set up the database and driver

First, make sure that whatever database you choose to use is installed and running, and that the driver is available. JDBC drivers can be downloaded from http://industry.java.sun.com/products/jdbc/drivers.

Once you have created the database, create the necessary tables. This tutorial uses only one table, products. Its structure is at the left. Create the table with the appropriate steps for your database.

Note: Under normal circumstances this data would be broken out into at least two related tables; for simplicity's sake, the example represents only one.

The process of accessing a database

Interacting with a database using Java usually consists of the following steps:

1. Load the database driver. This can be a JDBC driver or the JDBC-ODBC bridge.
2. Create a Connection to the database.
3. Create a **Statement** object. This object actually executes the SQL or stored procedure.
4. Create a **ResultSet** and populate it with the results of an executed query (if the goal is to retrieve or directly update data).
5. Retrieve or update the data from the **ResultSet**.

The `java.sql` package contains the JDBC 2.0 Core API for accessing a database as part of the JavaTM 2 SDK, Standard Edition distribution. The `javax.sql` package that is distributed as part of the Java 2 SDK, Enterprise Edition, contains the JDBC 2.0 Optional Package API.

This tutorial uses only classes in the JDBC 2.0 Core API.

### Instantiate the driver

To access the database, first load the JDBC driver. A number of different drivers may be available at any given time; it is the **DriverManager** that decides which one to use by attempting to create a connection with each driver it knows about. The first one that successfully connects will be used by the application. There are two ways that the **DriverManager** can know a driver exists.

The first way is to load it directly using `Class.forName()`, as shown in this example. When the driver class is loaded, it registers with the **DriverManager**, as shown here:

```java
public class Pricing extends Object {

    public static void main (String args[]) {

        // For the JDBC-ODBC bridge, use
        // driverName = "sun.jdbc.odbc.JdbcOdbcDriver"
        String driverName = "JData2_0.sql.$Driver";

        try {
            Class.forName(driverName);
        } catch (ClassNotFoundException e) {
            System.out.println("Error creating class: "+e.getMessage());
        }
    }
}
```

With a successfully loaded driver, the application can connect to the database.

The second way the **DriverManager** can locate a driver is to cycle through any drivers found in the `sql.drivers` system property. The `sql.drivers` property is a colon-delimited list of potential drivers. This list is always checked prior to classes being loaded dynamically, so if you want to use a particular driver, make sure that the `sql.drivers` property is either empty or starts with your desired driver.

### Create the connection

Once you load the driver, the application can connect to the database.

The **DriverManager** makes a connection through the static `getConnection()` method, which takes the URL of the database as an argument. The URL is typically referenced as:
jdbc:<sub-protocol>:databasename

However, the reference URL can be written in any format the active driver understands. Consult the documentation for your JDBC driver for the URL format.

One occasion where the subprotocol comes into play is in connecting via ODBC. If the sample database, with its DSN pricing, were accessed directly via ODBC the URL might be:

odbc:pricing

This means that to connect via JDBC, the URL would be:

jdbc:odbc:pricing

The actual connection is created below:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class Pricing extends Object {
    public static void main (String args[]){
        //For the JDBC-ODBC bridge, use
        //driverName = "sun.jdbc.odbc.JdbcOdbcDriver"
        //and
        //connectURL = "jdbc:odbc:pricing"
        String driverName = "JData2_0.sql.$Driver";
        String connectURL = "jdbc:JDataConnect://127.0.0.1/pricing";
        Connection db = null;
        try {
            Class.forName(driverName);
            db = DriverManager.getConnection(connectURL);
            catch (ClassNotFoundException e) {
                System.out.println("Error creating class: "+e.getMessage());
            } catch (SQLException e) {
                System.out.println("Error creating connection: "+e.getMessage());
            }
        }
    }
}
```

Once a connection is successfully made, it is possible to perform any required database operations (for example, inserting or updating data).

---

### Closing the connection

Because the `Statement` and `Connection` are objects, Java will garbage collect them, freeing up the database resources they take up. This may lull you into thinking that this means you don't have to worry about closing these objects, but it's not true.

It is entirely possible that the Java application itself has plenty of resources available, which means less-frequent garbage collection. It is also possible that while the Java application has plenty of resources, the available database resources are limited. Many of the database resources may be taken up by Java objects that could just as easily have been closed by the application.

It's important to make sure that these objects are closed whether or not there are any errors, so add a `finally` block to the `try-catch` block that's already in place:
Connection db = null;
try {
    Class.forName(driverName);
    db = DriverManager.getConnection(connectURL);
} catch (ClassNotFoundException e) {
    System.out.println("Error creating class: "+e.getMessage());
} catch (SQLException e) {
    System.out.println("Error creating connection: "+e.getMessage());
} finally {
    System.out.println("Closing connections...");
    try {
        db.close();
    } catch (SQLException e) {
        System.out.println("Can't close connection.");
    }
}

Ironically, the close() method itself can throw a SQLException, so it needs its own try-catch block.
Section 3. Extracting information from a database via JDBC

Anatomy of a SELECT statement

Once you have a connection to the database, the application can begin the process of retrieving data. In SQL databases data is (usually) retrieved using a SELECT statement. A SELECT statement has several basic parts. For example:

```
SELECT product_id, product_name FROM products WHERE product_id < 1000 ORDER BY product_id
```

This statement can be broken down as follows:

* `SELECT product_id, product_name`: The **select clause**, which defines which columns from a table (or group of tables) will be returned. To return all columns, use `SELECT *`.
* `FROM products`: The **from clause**, which defines the tables containing data to be returned. You can select from more than one table. This is known as a **join**, and requires a carefully crafted **where clause**.
* `WHERE product_id < 1000`: The **where clause**, which determines which of the available rows will be returned.
* `ORDER BY product_id`: The **order by clause** determines the order in which the data will be returned.

A SELECT statement can also group rows together and return an aggregate value. For example, the following statement returns the total revenue for all products that generated more than $1000 of revenue on September 15, 2001:

```
SELECT product_id, sum(quantity*price) as revenue FROM orders WHERE order_date = '9/15/01' GROUP BY product_id HAVING revenue > 1000
```

The SELECT statement is executed by the **Statement** object.

Create the statement

Creating the **Statement** object is straightforward; simply use the **Connection**'s `createStatement()` method, making sure to catch the potential **SQLException**. Most of the classes used in this section throw **SQLException**, so the rest of the code goes in this try-catch block.

```java
import java.sql.Statement;

public class Pricing extends Object {
    ...
    } catch (SQLException e) {
        System.out.println("Error creating connection: "+e.getMessage());
    }

    //Create the Statement object, used to execute the SQL statement
    Statement statement = null;
    try {
        statement = db.createStatement();
    } catch (SQLException e) {
```
System.out.println("SQL Error: "+e.getMessage());
} finally {
    System.out.println("Closing connections...");
    try {
        db.close();
    } catch (SQLException e) {
        System.out.println("Can't close connection.");
    }
}

This is similar to creating PreparedStatements on page 14 and Calling stored procedures with CallableStatement on page 15.

Execute the statement

To actually retrieve the data, the Statement must be executed. This typically involves passing a SELECT statement, which creates a set of data returned as a ResultSet as shown below.

```java
import java.sql.ResultSet;

public class Pricing extends Object {
...
    //Create the Statement object, used to execute the SQL statement
    Statement statement = null;
    //Create the ResultSet object, which ultimately holds the data retrieved
    ResultSet resultset = null;
    try {
        statement = db.createStatement();
        //Execute the query to populate the ResultSet
        resultset = statement.executeQuery("SELECT * FROM products");
    } catch (SQLException e) {
        System.out.println("SQL Error: "+e.getMessage());
    }
    finally {
...

If there is a problem with the SQL statement, such as a reference to a nonexistent table, the application throws the SQLException. Otherwise, the application proceeds, whether or not any data was found.
Testing for data

When a ResultSet is created, it has a "pointer" that references relative position within a dataset. Immediately following the return of the ResultSet statement (even if table is empty), this pointer is located just "above" the first row.

To get to the first row of actual data, the application calls the next() method. This method returns a Boolean value indicating whether a row exists at the new location. If no data is found, next() returns false.

```java
resultset = statement.executeQuery("select * from products");
//Check for data by moving the cursor to the first record (if there is one)
if (resultset.next()) {
    System.out.println("Data exists.");
} else {
    System.out.println("No data exists.");
}
```

A similar technique is used later in Looping through the data on page 12.

Data types

Once you establish that there is data, you can retrieve it using ResultSet's getXXX() methods. There is no actual method called getXXX(), but there is a group of methods referred to in this way because the ResultSet can return data as many types. For example, if the unit of data was an integer in the database, ResultSet can return it to the application as a number (with methods such as getDouble(), getInt() and getFloat()); as a String (with getString()); or as an array (with getArray()). Large units of data can even be retrieved as an InputStream (with getBinaryStream()) or Reader (with getCharacterStream()).

The important things to remember are that Java can closely match the data types found in most databases, and that only certain data types can be retrieved as something other than what they are. For example, getString() can retrieve data that started life as any database type, but date and time values can only be retrieved with getDate() (or getTime()), getObject(), or getAsString().

Retrieving data by name

You can retrieve the data itself two ways: by name and by index. To retrieve by name, use one of the previously discussed getXXX() methods. These take arguments of the form int or String. Retrieving data by index on page 12 is detailed in a moment; for now, the application will retrieve the fields by column name.
Ultimately, all of this data is destined for inclusion in an XML file where it will be all text, so use `getString()` for all values:

```java
if (resultset.next()) {
    //Output data by referencing the ResultSet columns by name
    System.out.print(resultset.getString("product_id") + "|" );
    System.out.print(resultset.getString("product_name") + " | ");
    System.out.print(resultset.getString("base_price") + " | ");
    System.out.print(resultset.getString("size") + " | ");
    System.out.print(resultset.getString("unit") + " | ");
    System.out.print(resultset.getString("lower") + " | ");
    System.out.print(resultset.getString("upper") + " | ");
    System.out.println(resultset.getString("unit_price"));
} else {
    System.out.println("No data exists.");
}
```

Compiling and running the application shows the first row of data -- quantity pricing information for an online gourmet food retailer.

```
 1 | Filet Mignon | $8 | 1 lb | 1 lb | 1 lb | 1 lb | $8
```

Looping through the data

Displaying one row at a time is useful, but an even better approach is to loop through the data and show each record on a separate line.

The application accomplishes this by advancing to the next row and outputting the data, then advancing to the next row again. When the pointer advances past the last row, `next()` returns `false` and the loop ends.

```java
resultset = statement.executeQuery("select * from products");
//Execute the loop as long as there is a next record
while (resultset.next()) {
    //Output data by referencing the ResultSet columns by name
    System.out.print(resultset.getString("product_id") + "|" );
    System.out.print(resultset.getString("product_name") + " | ");
    System.out.print(resultset.getString("base_price") + " | ");
    System.out.print(resultset.getString("size") + " | ");
    System.out.print(resultset.getString("unit") + " | ");
    System.out.print(resultset.getString("lower") + " | ");
    System.out.print(resultset.getString("upper") + " | ");
    System.out.println(resultset.getString("unit_price"));
}
```...

Running this application returns all the rows in the table.

```
1 | Filet Mignon | $8 | 1 lb | 1 lb | 1 lb | 1 lb | $8
```

Retrieving data by index

Using JDBC to extract data into XML Page 12
Of course, retrieving data by its column name can be rather unwieldy, particularly if there are many columns involved. It also makes generalizing and customizing the retrieval difficult. To retrieve the data faster and in a more easily customized way, use the index number of each column.

The first column has an index of 1, the second 2, and so on. Knowing that there are eight columns, the example in the previous panel can be rewritten as follows:

```java
while (resultset.next()) {
    for (int i=1; i <= 8; i++) {
        //Output each column by its index
        System.out.print(resultset.getString(i)+" | ");
    }
    //Output a line feed at the end of the row
    System.out.println("\n");
}
```

Using the above technique makes the data retrieval process completely generic, thus portable to other similar tasks and/or applications.

---

**Generic retrieval**

The ultimate goal of this tutorial is to create a data retrieval and manipulation procedure that is based on a mapping file completely independent of the application. Therefore, the application needs to be able to retrieve the data without prior knowledge of how it's structured. In other words, the application needs a way to independently find out how many columns are in the `ResultSet`.

To accomplish this, make use of the `ResultSetMetaData` class. Like the metadata of a database, it has access to the number and types of each column, as well as to the names of the columns and other information.

The `ResultSetMetaData` can be used to output all information about a record even when the only information it has is the `ResultSet` function itself.

```java
import java.sql.ResultSetMetaData;

public class Pricing extends Object {

    Statement statement = null;
    ResultSet resultset = null;
    //Create the ResultSetMetaData object, which will hold information about
    //the ResultSet
    ResultSetMetaData resultmetadata = null;
    try {
        statement = db.createStatement();
        resultset = statement.executeQuery("select * from products");
        //Get the ResultSet information
        resultmetadata = resultset.getMetaData();
        //Determine the number of columns in the ResultSet
```
int numCols = resultmetadata.getColumnCount();

while (resultset.next()) {
    for (int i=1; i <= numCols; i++) {
        // For each column index, determine the column name
        String colName = resultmetadata.getColumnName(i);
        // Get the column value
        String colVal = resultset.getString(i);
        // Output the name and value
        System.out.println(colName+"="+colVal);
    }
    // Output a line feed at the end of the row
    System.out.println(" ");
}

Running the application now gives not only the data, but also the column names, which you’ll need to create an XML document for the data:

```
upper 10
unit_price 88
product_id 2
product_name Filet Mignon
base_price 88
size 1
unit lb
lower 11
upper 50
unit_price 75
product_id 3
product_name Filet Mignon
```

---

### ResultSet options

To this point, the application has used a ResultSet created with default properties, but you can adjust these properties to control things such as the direction the data is fetched, and whether the ResultSet shows changes made by other users.

For example, the ResultSet can be created in such a way that it can be scrolled or used to update data directly. It fetches data from top to bottom, and is not affected by changes made by other database users.

The properties are actually set on the Statement that ultimately creates the ResultSet:

```
Statement statement = db.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.FETCH_FORWARD
);
```

Note that passing multiple custom properties to the ResultSet method can affect the performance of your application.

---

### PreparedStatements

One way to improve the performance of your application is to use a PreparedStatement.

A PreparedStatement is like a Statement, except that it is created with a SQL statement as a parameter. In most cases, the SQL statement will be precompiled (or at least cached) by the database. For example, the SELECT statement could have been precompiled if the application were structured like so:
import java.sql.PreparedStatement;

public class Pricing extends Object {

    //Create the PreparedStatement
    PreparedStatement statement = null;
    ResultSet resultset = null;
    ResultSetMetaData resultmetadata = null;
    try {
        //Compile or cache the SQL within the database and prepare it for execution
        statement = db.prepareStatement("select * from products");
        //Execute the SQL above to populate the ResultSet
        resultset = statement.executeQuery();
        //Get the ResultSet Information
        resultmetadata = resultset.getMetaData();
    }

    //A PreparedStatement is probably most useful when a particular query will be executed
    //over and over. In such a situation, it is useful to be able to set parameters at runtime.

    //Setting IN parameters
    In most cases where a Statement or PreparedStatement will be reused, each use
    //involves slight differences such as a different range of records. In these situations, you need
    //IN parameters. For example, to retrieve a range of product_ids to be determined at
    //runtime, use:

    statement = db.prepareStatement("select * from products
    //where product_id < ? and product_id > ?");
    statement.setInt(1, 5);
    statement.setInt(2, 10);
    resultset = statement.executeQuery();

    //The setXXX() methods match the getXXX() methods, except they include the parameter
    //number and the value to be set for that parameter. For example, setInt(2, 10) replaces
    //the second ? in the statement with the integer 10 so the statement is executed as though it
    //were written:

    select * from products where product_id < 5 and product_id > 10

    //You can use a similar technique when calling stored procedures with CallableStatement.

    //Calling stored procedures with CallableStatement
    Most modern databases allow a developer to create a stored procedure within the database.
    //This stored procedure can be as simple as a single SQL statement, or as complex as a
    //miniapplication. In either case, it is sometimes necessary to call these procedures from Java
    //to produce a set of data to be retrieved.

    //The CallableStatement class extends PreparedStatement and allows a developer to
    //specify parameters to a database query. CallableStatement then returns a ResultSet
    //(or ResultSets).
Creating a CallableStatement is much like creating a PreparedStatement, except a call statement is used instead of an SQL statement. The call statement is then translated, by the driver, into a native call.

```java
statement = db.prepareCall("{call product_listing }");
```

**Note:** One difference between CallableStatement and PreparedStatement is a CallableStatement can also provide OUT parameters in addition to the ResultSet normally created.

---

### Detecting null values

The data in this tutorial involves quantity pricing for items. Because of this, the upper limit for some quantity ranges is null, indicating that no further quantity discounts are available.

This is all well and good, but it makes building the eventual XML document difficult as the empty values can cause problems. To solve this problem, you can use the `wasNull()` method to determine if a particular piece of data is null, and if it is replace it with something else. This example shows the replacement of null values with the words "and up".

```java
... while (resultset.next()) {
    //Output each row
    for (int i=1; i <= numCols; i++) {
        //For each column, get name and value information
        String colName = resultmetadata.getColumnName(i);
        String colVal = resultset.getString(i);
        //Determine if the last column accessed was null
        if (resultset.wasNull()) {
            colVal = "and up";
        }
        //Output the information
        System.out.println( colName + " = " + colVal);
    }
    System.out.println( " ");
... 
```

Notice that there is no argument for `wasNull()`. The method acts on the last column that was retrieved from the ResultSet, so the call to `getXXX()` has to be made first.

At this point, the application is retrieving the appropriate data and column names. You're ready to start building the XML document.
Section 4. Setting up the mapping

How mapping works

The goal of this tutorial is to show how to create an XML document out of data from a database using a mapping file. In other words, the map file determines what data is retrieved and how it is ultimately represented in the XML file.

One way to do this is to create a temporary XML document out of the data as it's extracted from the database, then massage that data into the new format according to the map file. The map file determines which data is extracted, the name and structure of the new file, and what data is stored where.

The structure

The mapping file contains several pieces of information:

* The original data, in the form of a data element. For maximum flexibility, this is in the form of an SQL statement. In this way, you can use the mapping file to specify that data should be drawn from more than one table.
* The overall structure of the new document. This is in the form of the root element, which, through attributes, specifies the name of the destination root element and the name of the elements that are to represent database rows.
* The names and contents of data elements. These are contained in a series of element elements. The elements include the name of the new element and any attribute or content elements. These two elements designate the data that should be added, and, in the case of attributes, what it should be called. For example, if the description element should have a product_number attribute that represents the product_id column and the product_name as content, the map file would represent it as:

```xml
<element name="description">
    <attribute name="product_number">product_id</attribute>
    <content>product_name</content>
</element>
```

The mapping file

The final mapping file is as follows:

```xml
<?xml version="1.0"?>
<mapping>
    <data sql="select * from products" />
    <root name="pricingInfo" rowName="product">
        <element name="description">
            <attribute name="product_number">product_id</attribute>
            <content>product_name</content>
        </element>
        <element name="quantity">
            <content>lower</content>
        </element>
        <element name="size">
            <content>size</content>
        </element>
        <element name="sizeUnit">
            <content>unit</content>
        </element>
    </root>
</mapping>
```
Section 5. Creating an XML document using SQL results

The algorithm

The process for creating the new XML document is as follows:

1. **Parse the map file** on page 19 to make the necessary information, including the data to be retrieved, available.
2. **Retrieve the source query** on page 20. This allows for the dynamic retrieval of data based on the map file.
3. **Store the data in a Document object** on page 20. This temporary document will then be available to pull the data from to create the destination document according to the mapping.
4. **Retrieve the data mapping** on page 22 to make it available to the application.
5. **Loop through the original data** on page 22. Each row of the data is analyzed and re-mapped to the new structure.
6. **Retrieve element mappings** on page 22. The mapping file determines what data is pulled from the temporary document, and in what order.
7. **Add elements to the new document** on page 23. Once the data is retrieved, add it to the new document under new names.
8. **Add attributes to the new document** on page 24. Finally, add any attributes to the appropriate elements.

Parse the map file

The first step in creating the new document is to retrieve the mapping, which can only be accomplished by parsing the map file. Note that because this file also contains references to the data that will be eventually retrieved, you must parse it before any database operations can be performed.

```java
import org.w3c.dom.Document;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

public class Pricing extends Object {

    public static void main(String args[]) {
        //Create the Document object
        Document mapDoc = null;
        try {
            //Create the DocumentBuilderFactory
            DocumentBuilderFactory dbfactory = DocumentBuilderFactory.newInstance();
            //Create the DocumentBuilder
            DocumentBuilder docbuilder = dbfactory.newDocumentBuilder();
            //Parse the file to create the Document
            mapDoc = docbuilder.parse("mapping.xml");
        } catch (Exception e) {
            System.out.println("Problem creating document: "+e.getMessage());
        }

        //For the JDBC-ODBC bridge, use
        //driverName = "sun.jdbc.odbc.JdbcOdbcDriver"
        //and
        //connectURL = "jdbc:odbc:pricing"
        String driverName = "JData2_0.sql.$Driver";
        String connectURL = "jdbc:JDataConnect://127.0.0.1/pricing";
        Connection db = null;
```

Retrieve the source query

Next retrieve the source query stored in the sql attribute of the data element.

```java
import org.w3c.dom.Element;
import org.w3c.dom.Node;
...
System.out.println("Problem creating document: "+e.getMessage());
}

//Retrieve the root element
Element mapRoot = mapDoc.getDocumentElement();
//Retrieve the (only) data element and cast it to Element
Node dataNode = mapRoot.getElementsByTagName("data").item(0);
Element dataElement = (Element)dataNode;
//Retrieve the sql statement
String sql = dataElement.getAttribute("sql");

//Output the SQL statement
System.out.println(sql);
//For the JDBC-ODBC bridge, use
//driverName = "sun.jdbc.odbc.JdbcOdbcDriver"
//and
//connectURL = "jdbc:odbc:pricing"
String driverName = "JData2_0.sql.$Driver";
String connectURL = "jdbc:JDataConnect://127.0.0.1/pricing";
Connection db = null;
...
```

First determine the root element, then retrieve the data node. Because there is only one data element, you can retrieve it directly. Similar techniques can also be utilized to build a document from several queries run in sequence.

Finally, cast the Node to an Element so the Attribute value is available.

Remove the previous output statements and run the application to show the SQL statement as output.

Store the data in a Document object

Once the data is successfully extracted from the database, it is stored in a temporary Document. The generic method is to create a row element for each row of data, with each column represented as an element named after that column, and with the data itself as the content of the element.

```java
public static void main (String args[]){
    Document mapDoc = null;
    //Define a new Document object
    Document dataDoc = null;
    try {
        //Create the DocumentBuilderFactory
        DocumentBuilderFactory dbfactory = DocumentBuilderFactory.newInstance();
//Create the DocumentBuilder
DocumentBuilder docBuilder = dbfactory.newDocumentBuilder();
//Parse the file to create the Document
mapDoc = docBuilder.parse("mapping.xml");

//Instantiate a new Document object
dataDoc = docBuilder.newDocument();
}

try {
    statement = db.createStatement();
    resultset = statement.executeQuery("select * from products");
    resultmetadata = resultset.getMetaData();
    int numCols = resultmetadata.getColumnCount();
    while (resultset.next()) {
        //For each row of data
        Element rowEl = dataDoc.createElement("row");
        for (int i=1; i <= numCols; i++) {
            //For each column, retrieve the name and data
            String colName = resultmetadata.getColumnName(i);  
            String colVal = resultset.getString(i);
            //If there was no data, add "and up"
            if (resultset.wasNull()) {
                colVal = "and up";
            }
            //Create a new element with the same name as the column
            Element dataEl = dataDoc.createElement(colName);
            //Add the data to the new element
            dataEl.appendChild(dataDoc.createTextNode(colVal));
            //Add the new element to the row
            rowEl.appendChild(dataEl);
        }
        //Add the row to the root element
        dataRoot.appendChild(rowEl);
    }
}

} catch (SQLException e) {
    System.out.println("SQL Error: "+e.getMessage());
} finally {
    try {
        db.close();
    } catch (SQLException e) {
        System.out.println("Can't close connection.");
    }
}

//Add the root element to the document
dataDoc.appendChild(dataRoot);

Specifically, and referring to the code example above, first create an empty document along with the root element, data. For each row in the database, create a row element, and for each column, create an individual element and add it to the row. Finally, add each row element to the root, and the root to the Document.
Retrieve the data mapping

Once you have the data, it's time to work on mapping it to the new structure: Retrieve the mapping from the parsed map document. First retrieve the information on the root and row elements, then the element mappings themselves.

```java
import org.w3c.dom.NodeList;
...
dataDoc.appendChild(dataRoot);

//Retrieve the root element (also called "root")
Element newRootInfo = (Element)mapRoot.getElementsByTagName("root").item(0);
//Retrieve the root and row information
String newRootName = newRootInfo.getAttribute("name");
String newRowName = newRootInfo.getAttribute("rowName");
//Retrieve information on elements to be built in the new document
NodeList newNodesMap = mapRoot.getElementsByTagName("element");

```

Armed with this information, you can build the new Document.

Loop through the original data

Each of the original rows is stored in the temporary document as a row element. You will want to loop through these elements, so retrieve them as a NodeList.

```java
NodeList newNodesMap = mapRoot.getElementsByTagName("element");

//Retrieve all rows in the old document
NodeList oldRows = dataRoot.getElementsByTagName("row");
for (int i=0; i < oldRows.getLength(); i++){
    //Retrieve each row in turn
    Element thisRow = (Element)oldRows.item(i);
}
```

Retrieve element mappings

Now that you have the data and the mapping information, it's time to begin building the new Document. For each row, cycle through the map to determine the order in which the data columns should be retrieved from the temporary Document, and to determine what they should be called when added to the new Document.

```java
for (int i=0; i < oldRows.getLength(); i++){
    //Retrieve each row in turn
    Element thisRow = (Element)oldRows.item(i);
    for (int j=0; j < newNodesMap.getLength(); j++) {
        //For each node in the new mapping, retrieve the information
        //First the new information...
        Element thisElement = (Element)newNodesMap.item(j);
        String newElementName = thisElement.getAttribute("name");
    }
}
```
For each element in the `newNodesMap`, the application retrieves the new element name, then the name of the old element to retrieve.

### Add elements to the new document

Adding the new elements to the document is a simple matter of creating a new element with the proper name, then retrieving the appropriate data and setting it as the content of the element.

```java
public static void main (String args[]){
  Document mapDoc = null;
  Document dataDoc = null;
  //Create the new Document
  Document newDoc = null;
  try {
    //Create the DocumentBuilderFactory
    DocumentBuilderFactory dbfactory = DocumentBuilderFactory.newInstance();
    //Create the DocumentBuilder
    DocumentBuilder docbuilder = dbfactory.newDocumentBuilder();
    //Parse the file to create the Document
    mapDoc = docbuilder.parse("mapping.xml");
    //Instantiate a new Document object
    dataDoc = docbuilder.newDocument();
    //Instantiate the new Document
    newDoc = docbuilder.newDocument();
  } catch (Exception e) {
    System.out.println("Problem creating document: "+e.getMessage());
  }
  //Retrieve the root element (also called "root")
  Element newRootInfo = (Element)mapRoot.getElementsByTagName("root").item(0);
  //Retrieve the root and row information
  String newRootName = newRootInfo.getAttribute("name");
  String newRowName = newRootInfo.getAttribute("rowName");
  //Retrieve information on elements to be built in the new document
  NodeList newNodesMap = mapRoot.getElementsByTagName("element");
  //Create the final root element with the name from the mapping file
  Element newRootElement = newDoc.createElement(newRootName);
  NodeList oldRows = dataRoot.getElementsByTagName("row");
  for (int i=0; i < oldRows.getLength(); i++) {
    //For each of the original rows
    Element thisRow = (Element)oldRows.item(i);
    //Create the new row
    Element newRow = newDoc.createElement(newRowName);
    for (int j=0; j < newNodesMap.getLength(); j++) {
      //Get the mapping information for each column
      Element thisElement = (Element)newNodesMap.item(j);
      String newElementName = thisElement.getAttribute("name");
      Element oldElement = (Element)thisElement.getElementsByTagName("content").item(0);
      //Then the old information
      Element oldElement = (Element)thisElement.getElementsByTagName("content").item(0);
      String oldField = oldElement.getFirstChild().getNodeValue();
    }
  }
}
```
First, create the new Document, then create the new root element, which takes its name from the mapping information. For each row in the temporary Document, create an element for a new row using the newRowName specified in the map.

For each row, loop through each new element specified in the mapping and retrieve the original data. In the previous panel, you retrieved the text of the content elements in order. Use this information to determine which nodes to retrieve from the temporary rows, and in what order. Once you have the old data and the new name, create the new element and add it to the row.

Finally, add the new row to the root, and the root to the Document. The only thing left to add is any attributes.

**Add attributes to the new document**

The new Document is almost complete. You've added the new elements, but not any attributes that may have been specified. Adding them is similar to adding the new elements themselves. The possibility exists, however, that there is more than one attribute for an element, so this must be accounted for in the code. The easiest way to accomplish this is to retrieve the attribute elements into a NodeList, then iterate through this list to deal with each one. For each desired attribute, determine the field name from the original Document, and the name in the new Document.

```java
Element newElement = newDoc.createElement(newElementName);
ewElement.appendChild(newDoc.createTextNode(oldValue));

//Retrieve list of new elements
NodeList newAttributes = thisElement.getElementsByTagName("attribute");
for (int k=0; k < newAttributes.getLength(); k++) {
  //For each new attribute
  //Get the mapping information
  Element thisAttribute = (Element) newAttributes.item(k);
  String oldAttributeField = thisAttribute.getFirstChild().getNodeValue();
  String newAttributeName = thisAttribute.getAttribute("name");

  //Get the original value
  oldValueElement = (Element) thisRow.getElementsByTagName(oldAttributeField).item(0);
  String oldAttributeValue = oldValueElement.getFirstChild().getNodeValue();

  //Create the new attribute
  newElement.setAttribute(newAttributeName, oldAttributeValue);
}
The final document

At the end of this process, newDoc holds the old information in the new format. From here, it can be used in another application or transformed further using XSLT or other means.

```
<pricingInfo>
  <product>
    <description product_number="1">Filet Mignon</description>
    <quantity>1</quantity>
    <size>1</size>
    <sizeUnit>item</sizeUnit>
    <quantityPrice>40</quantityPrice>
  </product>
  <product>
    <description product_number="2">Filet Mignon</description>
    <quantity>11</quantity>
    <size>1</size>
    <sizeUnit>item</sizeUnit>
    <quantityPrice>30</quantityPrice>
  </product>
  <product>
    <description product_number="3">Filet Mignon</description>
    <quantity>101</quantity>
    <size>1</size>
    <sizeUnit>item</sizeUnit>
    <quantityPrice>20</quantityPrice>
  </product>
  <product>
    <description product_number="4">Prime Rib</description>
    <quantity>1</quantity>
    <size>1</size>
    <sizeUnit>lb</sizeUnit>
    <quantityPrice>20</quantityPrice>
  </product>
  <product>
    <description product_number="5">Prime Rib</description>
    <quantity>101</quantity>
    <size>1</size>
    <sizeUnit>lb</sizeUnit>
    <quantityPrice>15</quantityPrice>
  </product>
  ...
</pricingInfo>
```
Section 6. JDBC data extraction summary

Summary

This tutorial discussed the details of connecting to a database using JDBC and extracting data, which was then used to create an XML file. Discussion included making the application generic so that it could be used with any query and any structure. This was accomplished partly through the use of metadata.

The structure of the XML destination file was determined by an XML mapping file. The tutorial also covered the details of reading the mapping file and using it to massage the temporary XML file into the desired structure.

Resources

JDBC and XML are both powerful data tools in their own right. Some good places to find additional information are:

* Download a zip archive of the sample code presented in this tutorial.
* For information on using JDBC to insert data from an XML document into a database, read the companion tutorial Using JDBC to insert XML data into a database.

**XML Information**

* For a basic grounding in XML read through the Introduction to XML tutorial.
* For information on the Document Object Model, read the Understanding DOM tutorial.
* For an intro to programming XML, try Doug Tidwell's XML programming in Java.
* Order XML and Java from Scratch, by Nicholas Chase. While it doesn't have more details on the use of JDBC, it does cover the use of XML and Java in general. It also covers other data-centric views of XML, such as XML Query, and other uses for XML, such as SOAP. And besides, it's written by the author of this fine tutorial.

**JDBC information**

* See Sun's Sun Microsystems' information on JDBC.
* Review recent developerWorks articles on working with JDBC:
  Use JDBC for industrial-strength performance by Lennart Jorelid.
  A July 2001 look at the latest JDBC: What's new in JDBC 3.0 by Josh Heidebrecht
  An easy JDBC wrapper by Greg Travis

* Find complete background on JDBC at Sun Microsystems' JDBC page.
* Learn the basics of SQL with an online tutorial.
* For pointers to details on working with DB2 and JDBC, see:
* Find resources for developers on the AS/400 platform who work with JDBC on the JDBC for iSeries page.

Downloads

* Download the Java 2 SDK, Standard Edition version 1.3.1.
* Download JAXP 1.1, the JavaTM APIs for XML Processing.
* Download JDBC drivers for your database.
* Download JDataConnect, a JDBC driver that supports DB2, Oracle, Microsoft SQL Server, Access, FoxPro, Informix, Sybase, Ingres, dBase, Interbase, Pervasive and
others.

* For an open-source database, download MySQL and a MySQL JDBC driver.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.