

Using JDBC to insert data from XML into a database

XML data can find a permanent home

Skill Level: Intermediate

[Nicholas Chase](mailto:nicholas@nicholaschase.com) (nicholas@nicholaschase.com)

Writer

Freelance

03 Oct 2001

As flexible as XML is with regards to storing data, it is perfect for holding data that is ultimately destined for a database. This tutorial will show you how to access a database using the JDBC API and use SQL to insert data that has been retrieved from an XML file using a predetermined mapping. It also demonstrates the use of updatable ResultSets.

Section 1. Introduction

Should I take this tutorial?

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

The tutorial assumes that you are already familiar with the Java and XML languages in general, and the Document Object Model (DOM) in particular. You should be familiar with Java programming, but prior JDBC knowledge 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](#) include referrals to tutorials on XML and DOM basics and a detailed SQL backgrounder.

What is this tutorial about?

This tutorial demonstrates a method for retrieving XML data and inserting it into a database using the example of orders stored in an XML file. A second XML file is used to determine which elements in the orders file correspond to which tables and columns in the database.

The JDBC API is a vendor-independent method for accessing databases using the Java language. This tutorial explains how to instantiate and use a JDBC driver to connect to a database in order to store information. It also explains the basics of the required SQL and how to execute it using JDBC methods. Finally, this tutorial demonstrates techniques for updating data directly through the `ResultSet` for a query.

Tools

This tutorial will help you understand the topic even if you only read through the examples without trying them out. If you do want to try the examples as you go through the tutorial, however, 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 API 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](#).) This tutorial uses JDataConnect, available at <http://www.jnetdirect.com/products.php?op=jdataconnect>.

Conventions used in this tutorial

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

- 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 (. . .)

Section 2. Accessing a database through JDBC

What is JDBC?

(If you have already been through the "Using JDBC to extract data into XML" tutorial, feel free to skip ahead to [The orders.xml file](#).)

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.

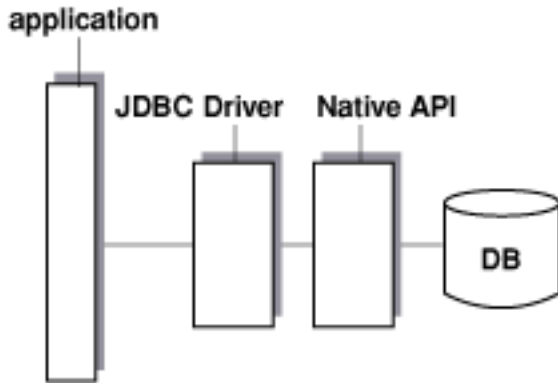
JDBC is similar to ODBC, or Open Database Connectivity, in that it provides a standard API intermediary for accessing a database. As seen on the left, the JDBC driver translates standard JDBC commands into the native API for the database.

You will notice that 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 (which you will use when you [Create the connection](#)).

See [Resources](#) for information on downloading the appropriate JDBC driver for your database. Over 150 JDBC drivers are available, covering virtually every database.

There are even options for databases that do not have an available JDBC driver, as you will see in the next panel.

Figure 1. Example of database without a JDBC driver



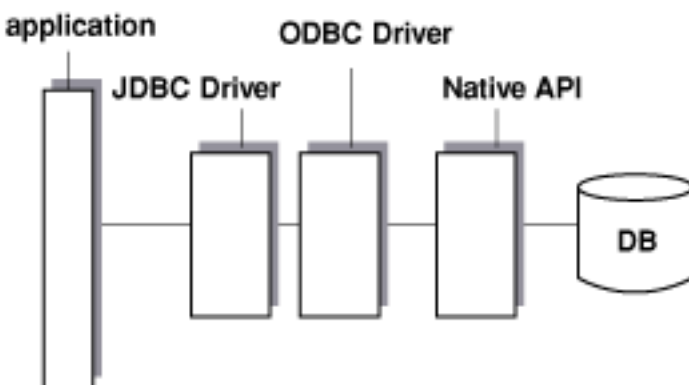
The JBDC-ODBC bridge

It is not necessary to have a specific JDBC driver for a database as long as an ODBC driver is available; a JDBC-ODBC *bridge* can be used instead. As you can see on the left, the application accesses the bridge via JDBC calls. The bridge then translates the commands into ODBC, whereupon the ODBC driver translates them into the native API.

The JDBC-ODBC bridge is not the recommended way to access a database for reasons involving both performance and configuration -- commands must pass through two APIs, 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 on a Windows system, create a System Data Source Name (DSN) by choosing `Start>Settings>Control Panel>ODBC Data Sources`. Note the name of the DSN, as it will be referenced when you [Create the connection](#).

Figure 2. Example of database with JDBC and drivers



Set up the database and driver

```
create table orders (
  orderid      numeric
              primary key,
  userid       varchar(50),
  productid    varchar(10),
  quantity     numeric,
  unitprice    numeric )

create table productOrders (
  productid    varchar(10),
  quantity     numeric,
  revenue      numeric )
```

First, make sure your database of choice is installed and running, and that the JDBC driver is available. You can download drivers from <http://industry.java.sun.com/products/jdbc/drivers>.

Once you have created the database, you will need to create the necessary tables. Ultimately, the structure will be unimportant because it can be determined by the XML mapping file. This tutorial assumes the presence of two tables, `orders` and `productOrders`, detailed at the left. Create the tables using the appropriate steps for your database.

The process of accessing a database

Using the Java language to interact with a database 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 Java 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 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:

```
public class SaveOrders 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 `orders`, were accessed directly via ODBC, the URL might be:

```
odbc:orders
```

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

```
jdbc:odbc:orders
```

The actual connection is created below:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class SaveOrders extends Object {

    public static void main (String args[]){

        //For the JDBC-ODBC bridge, use
        //driverName = "sun.jdbc.odbc.JdbcOdbcDriver"
        //and
        //connectURL = "jdbc:odbc:orders"
        String driverName = "JData2_0.sql.$Driver";
        String connectURL = "jdbc:JDataConnect://127.0.0.1/orders";
        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 the database resources they take up. This may lull you into thinking you do not have to worry about closing these objects, but that is not true.

It is entirely possible that the Java application itself has plenty of resources available, which means less-frequent garbage collection. At the same time, there

may be limited database resources available, many of which may be taken up by Java objects that can easily be closed by the application.

It is 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 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. Reading the XML file

The orders.xml file

The orders data for this tutorial is listed in the `orders.xml` file. Each order has information specific to the order itself, such as the order number, user id, and total. Each order also has information on individual items in the order such as the product id, unit price, and quantity.

The `orders.xml` file is below:

```
<?xml version="1.0" encoding="UTF-8"?>
<orders>
  <order orderid="1">
    <customerid>12341</customerid>
    <status>pending</status>
    <product productid="SA15">
      <name>Silver Show Saddle, 16 inch</name>
      <price>825.00</price>
      <qty>1</qty>
    </product>
  </order>
</orders>
```

```

    </product>
    <product productid="C49">
      <name>Premium Cinch</name>
      <price>49.00</price>
      <qty>1</qty>
    </product>
  </order>
  <order orderid="2">
    <customerid>251222</customerid>
    <status>pending</status>
    <product productid="WB78">
      <name>Winter Blanket (78 inch)</name>
      <price>20</price>
      <qty>10</qty>
    </product>
    <product productid="C49">
      <name>Premium Cinch</name>
      <price>49.00</price>
      <qty>5</qty>
    </product>
  </order>
</orders>

```

Parsing the file

Reading the XML data starts with parsing the file and creating a Document object:

```

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

public class SaveOrders extends Object {

    public static void main (String args[]){

//Determine the file location
String ordersFile = "orders.xml";

//Create the XML document
Document ordersDoc = null;
try {
//Instantiate the parser and parse the file
DocumentBuilderFactory docbuilderfactory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder docbuilder =
    docbuilderfactory.newDocumentBuilder();
ordersDoc = docbuilder.parse(ordersFile);
} catch (Exception e) {
System.out.println("Cannot read the orders file: "+e.getMessage());
}

Connection db = null;
...

```

This example shows the creation of an XML Document object using the Java API for XML Processing (JAXP). First, create the DocumentBuilderFactory, then use it to create the DocumentBuilder. This DocumentBuilder is the actual parser, which takes the orders.xml file and reads each element to create a Document object.

Once you create the `Document`, loop through the data.

Looping through the elements

Before you can build an application that reads the XML file based on a mapping file, you first need to read the XML without a mapping file.

In this example, the `order` element contains each order, and a `product` element contains each item in the order. To access this data, a loop is constructed:

```
...
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;

public class SaveOrders extends Object {

...
    Connection db = null;
    try {
        Class.forName(driver);
        db = DriverManager.getConnection(connectURL);

        //Get the root element and all order elements
        Element ordersRoot = ordersDoc.getDocumentElement();
        NodeList orders = ordersRoot.getElementsByTagName("order");
        for (int i = 0; i < orders.getLength(); i++) {
            //For each order, get the order element
            Element thisOrder = (Element)orders.item(i);

            //Loop through each product for the order
            NodeList products =
                thisOrder.getElementsByTagName("product");
            for (int j=0; j < products.getLength(); j++) {
                }
            }
        } catch (ClassNotFoundException e) {
            System.out.println("Error creating class: "+e.getMessage());
        }
    }
...
}
```

In order to retrieve information from the `Document`, you need to retrieve the *root element*, which contains all of the data. Once you have this element, the application can retrieve data by element name using `getElementsByTagName()`. This method returns a `NodeList`, which can then be used to access each element.

Because each `order` contains `product` elements, repeat the process to create a second loop for each order.

Next, print out the actual data.

Retrieving the data

Retrieving the actual data involves a combination of `getElementsByTagName()` and careful access of the children of nodes, as seen here:

```

...

Element ordersRoot = ordersDoc.getDocumentElement();
NodeList orders = ordersRoot.getElementsByTagName("order");
for (int i = 0; i < orders.getLength(); i++) {
    //For each order, get the order element
    Element thisOrder = (Element)orders.item(i);

    //Get order information
    String thisOrderid = thisOrder.getAttribute("orderid");
    System.out.println("Order: " + thisOrderid);

    //Get customer information
    String thisCustomerid =
        thisOrder.getElementsByTagName("customerid")
            .item(0)
            .getFirstChild().getNodeValue();
    System.out.println("Customer: " + thisCustomerid);

    //Loop through each product for the order
    NodeList products = thisOrder.getElementsByTagName("product");
    for (int j=0; j < products.getLength(); j++) {
        //Retrieve each product
        Element thisProduct = (Element)products.item(j);

        //Get product information from attributes and child
        //elements
        String thisProductid = thisProduct.getAttribute("productid");
        System.out.println(" Product: " + thisProductid);

        String priceStr = thisProduct.getElementsByTagName("price")
            .item(0)
            .getFirstChild().getNodeValue();
        System.out.println(" Price: " + priceStr);

        String qtyStr = thisProduct.getElementsByTagName("qty")
            .item(0)
            .getFirstChild().getNodeValue();
        System.out.println(" Quantity: " + qtyStr);
        System.out.println();
    }
}
} catch (ClassNotFoundException e) {
    System.out.println("Error creating class: "+e.getMessage());
}
...

```

First, retrieve the `orderid` attribute from the `order` element. This value is conveniently returned by `getAttribute()`.

Retrieving the values of child elements is slightly more complex. First retrieve a `NodeList` of all the child elements with the desired name, such as `customerid`, `productid`, and `price`. This way, you avoid any potential text nodes. But because there is only one of the desired children, choose `item(0)`. This provides the desired elements; to get the actual value, the value of the text node that is the element's first child is needed.

Now that you have the data, you are ready to start adding it to the database.

Section 4. Updating the database using SQL

Creating the Statement

The most straightforward way to add any data to a database is using an SQL statement. The most common object used to execute a SQL statement on a database is the `Statement`.

The `Statement` object is created by the `Connection`:

```
...
import java.sql.Statement;

public class SaveOrders extends Object {
    ...
    Connection db = null;
    try {
        Class.forName(className);
        db = DriverManager.getConnection(connectURL);

        //Create the Statement object
        Statement statement = db.createStatement();

        Element ordersRoot = ordersDoc.getDocumentElement();
    }
    ...
}
```

The `Statement` executes the `INSERT` statement.

Anatomy of an `INSERT` statement

Inserting data via SQL involves the use an `INSERT` statement. Consider this example:

```
insert into orders (orderid, userid) values (2333, 'nchase')
```

`insert into orders`: This indicates the table that the data will be added to.

`(orderid, userid)`: Although optional in certain circumstances, it is always a good idea to list the columns in the database that will be taking the inserted data, in the same order in which they appear in the *values clause*.

`values (2333, 'nchase')`: The values clause lists the actual values that are to be added to the database. If no column names have been included in the statement, then the values clause must list a value for every column in the table, in the order in

which they appear in the table's definition in the database.

You can use the `Statement` object to execute this statement.

Executing the INSERT statement

In general, one of four methods is used to execute an SQL statement using the `Statement` object:

- `executeQuery()`: Used to execute select statements that will return a single `ResultSet`.
- `executeUpdate()`: Used to execute statements that make changes to the database. Returns an integer that represents the number of records that were affected.
- `executeBatch()`: Used to send multiple statements to the database at once. Returns an array of result counts.
- `execute()`: Used for more complex database operations that may return more than one result or `ResultSet`, or a combination of both.

This tutorial will use `executeUpdate()`:

```
...
        String qtyStr = thisProduct.getElementsByTagName("qty")
            .item(0)
            .getFirstChild()
            .getNodeValue();

        //Create and execute the SQL statement
        String sqlTxt = "insert into orders "+
            "(orderid, userid, productid, unitprice, quantity) "+
            "values (" + thisOrderid + ", " + thisCustomerid + ", '" +
                thisProductid + "', " + priceStr + ", " + qtyStr + ")";
        int results = statement.executeUpdate(sqlTxt);

    }
} catch (ClassNotFoundException e) {
...

```

Create the INSERT statement based on the `orders` table and the values taken from the `orders.xml` file.

Ultimately, this action will be "genericized" to use an XML mapping file, which is introduced in the next section.

Section 5. Using an XML mapping file

Structure of the mapping file

Now that you have successfully entered data into the database, it is time to generalize this information so that it can be used in an XML mapping file.

The purpose of the mapping file is to specify all aspects of the data transfer, including the structure and name of the file containing the orders and the location and structure of the database.

A mapping file can become quite complex, so certain assumptions will be made for the purposes of this tutorial:

- **Database tables:** There are only two tables involved in this project. One hosts the overall order information, including a record for each product ordered; and one hosts the aggregate information, with overall revenue per product. In most circumstances, the aggregate information -- considered to be *derived data* -- would be retrieved programmatically from the first table. But for the purposes of this tutorial, it has its own table.
- **Database types:** While different databases typically have different names for specific data types, they generally fall into the overall categories of text, numbers, and dates. This tutorial makes the assumption that even if the actual data types differ slightly, they remain within these overall classifications.
- **XML structure:** Although an XML file can have any number of parent-child layers and can be organized in many ways, this tutorial assumes that products will always be the first-level children of orders, even if the names differ. Data can be either attributes or elements.

Keeping these restrictions in mind, you can build the mapping file based on the existing orders.xml file and database tables.

The mapping file: database information

The mapping file needs to designate two different types of information: the database information and the XML file information. This can be done with two child elements of the root. The first deals with information on the database:

```
<?xml version="1.0"?>
```

```

<map>
  <!-- For the JDBC-ODBC bridge, use
        driver = "sun.jdbc.odbc.JdbcOdbcDriver"
        and
        url = "jdbc:odbc:orders" -->
  <database url="jdbc:JDataConnect://127.0.0.1/orders"
        driver="JData2_0.sql.$Driver">
    <order_table>orders</order_table>
    <order_orderid>orderid</order_orderid>
    <order_customer>userid</order_customer>
    <order_productid>productid</order_productid>
    <order_quantity>quantity</order_quantity>
    <order_price>unitprice</order_price>
    <products_table>productOrders</products_table>
    <products_productid>productid</products_productid>
    <products_quantity>quantity</products_quantity>
    <products_total>revenue</products_total>
  </database>
  ...

```

The `database` element holds information on the URL of the database, and on the driver to use. This can be useful if you are not sure what kind of database will be used to hold the information. Adding it to the mapping file prevents you from having to recompile the application to change databases.

Database columns and tables will be treated as `String` variables, since they will be used only for building SQL strings. The map file lists each variable and the table or column to which it corresponds in the database. Note that in the existing tables, the `orders` and `products` tables share column names, but the mapping file makes this unnecessary.

The mapping file also specifies the XML information.

The mapping file: XML information

The XML portion of the file has a bit more information to convey; data can be stored in either elements or attributes:

```

...
  <xmlfile url="orders.xml">
    <order>order</order>
    <orderid type="attribute">orderid</orderid>
    <customer type="element">customerid</customer>
    <products>product</products>
    <productid type="attribute">productid</productid>
    <quantity type="element">qty</quantity>
    <unitprice type="element">price</unitprice>
  </xmlfile>
</map>

```

Because of the way the application is structured, there is no need to indicate the parents, or in the case of attributes, the elements they belong to. The reference point, so to speak, is already available to the application. All you need to know is the name of the item, and whether it is an element or an attribute.

Retrieving the basic file information

The mapping file contains even the information about which file contains the orders and which database to connect to. As a result, this file must be parsed and analyzed before you can accomplish anything else.

This example shows how to retrieve the order file information:

```
...
//Create Elements for later use by methods
static Element databaseEl = null;
static Element xmlfileEl = null;

public static void main (String args[]){

    //Declare mapping elements and document
    String mappingFile = "map.xml";
    Document mapDoc = null;
    Element mapRoot = null;
    Document ordersDoc = null;
    try {

        DocumentBuilderFactory docbuilderfactory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder docbuilder = docbuilderfactory.newDocumentBuilder();

        //Parse map file and get root element
        mapDoc = docbuilder.parse(mappingFile);
        mapRoot = mapDoc.getDocumentElement();

        //Retrieve database and XML file elements and information
        databaseEl =
            (Element)mapRoot.getElementsByTagName("database").item(0);
        xmlfileEl =
            (Element)mapRoot.getElementsByTagName("xmlfile").item(0);

        System.out.println("Reading orders file...");
        String ordersFile = xmlfileEl.getAttribute("url");
        ordersDoc = docbuilder.parse(ordersFile);
    } catch (Exception e) {
        System.out.println("Cannot read the orders file:"+e.getMessage());
    }
}
...
```

The first thing to notice is that `databaseEl`, which will hold the main database mapping element, and `xmlfileEl`, which will hold the main XML file mapping element, are declared as `static` outside the `main` method so they can be used in the methods created later.

In the previous examples, `ordersFile` was the first main variable set in stone by the application. Now that variable is `mappingFile`, which points to the actual map file where the value of `ordersFile` can be found. Also you'll need to create the major XML structures, such as `mapRoot`, outside the `try-catch` block, because you'll need them later as well.

Inside the `try-catch` block, the `DocumentBuilder` first parses the map file rather

than the orders file, and populates the two major mapping elements, `databaseEl` and `xmlfileEl`. You can then use `xmlfileEl` to determine the name of the orders file, and finally parse it.

You also need the database information.

Retrieving the basic database information

Getting the basic database information is straightforward, because the driver and URL are attributes of the `databaseEl` element:

```
...
    } catch (Exception e) {
        System.out.println("Cannot read the orders file:"+e.getMessage());
    }

    String driverName = databaseEl.getAttribute("driver");
    String connectURL = databaseEl.getAttribute("url");
    Connection db = null;
    try {
        Class.forName(driverName);
        db = DriverManager.getConnection(connectURL);

        //Create the Statement object
        Statement statement = db.createStatement();

        Element ordersRoot = ordersDoc.getDocumentElement();
        NodeList orders = ordersRoot.getElementsByTagName("order");
    }
    ...
```

Note that this is one of the advantages of using dynamic class loading. If you change JDBC drivers, you don't need to recompile; just change the mapping file.

The rest of the database information, such as table and column names, is repetitive and best suited to a separate method.

Using the database variables

All of the database variables are stored as children of the same element, so retrieving them is well suited to a separate method:

```
...
public class SaveOrders extends Object {

    static Element databaseEl = null;
    static Element xmlfileEl = null;

    private static String getDatabaseInfo (String varname) {
        //Retrieve text node of specified element
        String tempStr = databaseEl.getElementsByTagName(varname)
            .item(0)
            .getFirstChild().getNodeValue();
        return tempStr;
    }
}
```

```

    }

    public static void main (String args[]){
...
        } catch (Exception e) {
            System.out.println("Cannot read the orders file:"+e.getMessage());
        }

        String driverName = databaseEl.getAttribute("driver");
        String connectURL = databaseEl.getAttribute("url");

        //Retrieve column names
        String order_table = getDatabaseInfo("order_table");
        String order_orderid = getDatabaseInfo("order_orderid");
        String order_customer = getDatabaseInfo("order_customer");
        String order_productid = getDatabaseInfo("order_productid");
        String order_quantity = getDatabaseInfo("order_quantity");
        String order_price = getDatabaseInfo("order_price");
        String products_table = getDatabaseInfo("products_table");
        String products_productid = getDatabaseInfo("products_productid");
        String products_quantity = getDatabaseInfo("products_quantity");
        String products_total = getDatabaseInfo("products_total");

        Connection db = null;
        try {
...

```

The `getDatabaseInfo()` method simply gets the child element with the specified name, and returns the value of its child text node. Use this method to populate the other database variables so they can be used in the application.

To use the variables, simply replace their hard-coded equivalents in the SQL statement:

```

...
    String qtyStr = thisProduct.getElementsByTagName("qty")
        .item(0)
        .getFirstChild().getNodeValue();

    //Create and execute the SQL statement
    String sqlTxt = "insert into "+order_table+
        " (" +order_orderid+", "+order_customer+", "+
        "order_productid+", "+order_price+", "+order_quantity+") "+
        "values (" +thisOrderid+", "+thisCustomerId+", '"+
        thisProductid+"', "+priceStr+", "+qtyStr+")";

    int results = statement.executeUpdate(sqlTxt);
...

```

That takes care of the column names. The column values are already coming from `orders.xml`, but they are not yet retrieved based on the mapping file. That is a bit more complicated.

Using the basic XML variables

Getting the basic XML file mappings is straightforward because of the assumptions made earlier. The overall structure of the document is predetermined, so all you need to start is the names of the elements holding orders and products. This

information is in the mapping file as the text of the order and product elements. All of this makes the basic mappings easy to use:

```

...
    try {
        Class.forName(driverName);
        db = DriverManager.getConnection(connectURL);
        Statement statement = db.createStatement();

        //Get the name of the product and order elements
        String orderElName =
            xmlfileEl.getElementsByTagName("order").item(0)
                .getFirstChild().getNodeValue();
        String productElName =
            xmlfileEl.getElementsByTagName("products").item(0)
                .getFirstChild().getNodeValue();

        //Get the root element and all order elements
        Element ordersRoot = ordersDoc.getDocumentElement();
        NodeList orders = ordersRoot.getElementsByTagName(orderElName);
        for (int i = 0; i < orders.getLength(); i++) {
            //For each order, get the order element
            Element thisOrder = (Element)orders.item(i);

            //Get order information
            String thisOrderid = thisOrder.getAttribute("orderid");
            System.out.println("Order: " + thisOrderid);

            //Get customer information
            String thisCustomerid = thisOrder.getElementsByTagName("customerid")
                .item(0)
                .getFirstChild().getNodeValue();
            System.out.println("Customer: " + thisCustomerid);

            //Loop through each product for the order
            NodeList products = thisOrder.getElementsByTagName(productElName);
            for (int j=0; j < products.getLength(); j++) {

                Element thisProduct = (Element)products.item(j);

                String thisProductid = thisProduct.getAttribute("productid");
            }
        }
    }
...

```

With the basics established, it is time to look at the element mappings.

Using element mappings

Element mappings themselves should also be separated out into a method for two reasons. First, the logic will be used multiple times. Second, because there is logic involved, it is to your advantage to modularize the operation. This way, should you decide to change the structure of the mapping or orders files, changing the logic of how the map file is read will be easier.

The logic itself takes into account the fact that both elements and attributes can hold the data you're looking for:

```

...
public class SaveOrders extends Object {

```

```

...
private static String getDatabaseInfo (String varname) {
    String tempStr = databaseEl.getElementsByTagName(varname).item(0)
        .getFirstChild().getNodeValue();
    return tempStr;
}

private static String getXMLInfo (Element start, String varname) {
    //Get specified element and type
    Element mapEl = (Element)xmlfileEl.getElementsByTagName(varname).item(0);
    String type = mapEl.getAttribute("type");
    String mappedName = mapEl.getFirstChild().getNodeValue();

    //Determine whether to return the attribute or text node
    String tempStr = null;
    if (type.equals("attribute")) {
        tempStr = start.getAttribute(mappedName);
    } else if (type.equals("element")) {
        tempStr = start.getElementsByTagName(mappedName)
            .item(0)
            .getFirstChild().getNodeValue();
    }

    return tempStr;
}

public static void main (String args[]){
...
    Element ordersRoot = ordersDoc.getDocumentElement();
    NodeList orders = ordersRoot.getElementsByTagName(orderElName);
    for (int i = 0; i < orders.getLength(); i++) {
        Element thisOrder = (Element)orders.item(i);

        //Get order and customer information
        String thisOrderid = getXMLInfo(thisOrder, "orderid");
        String thisCustomerid = getXMLInfo(thisOrder, "customer");

        NodeList products = thisOrder.getElementsByTagName(productElName);
        for (int j=0; j < products.getLength(); j++) {
            //Retrieve each product
            Element thisProduct = (Element)products.item(j);

            //Get product information
            String thisProductid = getXMLInfo(thisProduct, "productid");
            String priceStr = getXMLInfo(thisProduct, "unitprice");
            String qtyStr = getXMLInfo(thisProduct, "quantity");

            //Create and execute the SQL statement
            String sqlTxt = "insert into "+order_table+
                " (" +order_orderid+", "+order_customer+", "+
                order_productid+", "+order_price+", "+order_quantity+") "+
                "values (" +thisOrderid+", "+thisCustomerid+", '"+
                thisProductid+"', "+priceStr+", "+qtyStr+")";
...

```

Previous examples showed data that is in a fixed place, but this example retrieves data from multiple nodes. Fortunately, you always have a reference node, either `thisOrder` or `thisProduct`. Pass this node to the method, and once you determine whether you are looking for an attribute or an element, retrieve the data relative to this node.

At this point, you have made the operation generic. Using the mapping file, you can change the table or column names, or even the database itself. It is also possible to change the structure of the orders.xml file. As long as you update map.xml, the application will still work as expected.

Combining XML files and a database involves more than just a mapping file, however. Database applications frequently involve grouping operations into transactions that can be cancelled or rolled back if there are errors with the data or with the XML file itself. The next section explains transactions and batch operations.

Section 6. Using transactions and batches

What is a transaction?

Now that the application is assembled, there are several relatively simple things that can be done to make it more robust.

Often in database manipulations there exists what is commonly known as an "all-or-nothing" mentality. When a record is updated, partial changes are undesirable; it is difficult to reverse a partially changed record to a "known-good" state. So when a database record is updated, the application should either save all changes, or have the means to reverse the record to a pre-existing state. In the context of our example, if you are inserting the orders and one of the statements causes an error, you want the means to cancel the other changes that have been made.

The ability to do this is known as using *transactions*. In transactions, all statements that are executed against the database exist in a state of "limbo" until they are either *committed* and made permanent, or *rolled back* and undone. An actual transaction begins with either the first SQL statement, or immediately after a commit or rollback.

Normally, a `Statement` automatically commits each SQL statement immediately, but it is possible to change this setting using the `Connection` object. Certain actions, such as making a change to the structure of the database or closing a `Statement` object, will also commit the transaction immediately -- even without the execution of the `commit()` method.

You can use transactions to create an application that will include all INSERTS (if there are no problems) or none of them (if something goes wrong).

Grouping inserts into a transaction

Grouping the individual insert statements into a transaction is straightforward. In this example, the application simply keeps track of whether there were any errors and, if so, rolls back the transaction:

```
...
    Connection db = null;
    //Set error flag
    boolean dbErrors = false;
    try {
...
    } catch (ClassNotFoundException e) {
        System.out.println("Error creating class: "+e.getMessage());
    } catch (SQLException e) {
        System.out.println("Error creating connection: "+e.getMessage());

        //An error occurred, so set the error flag
        dbErrors = true;
    } finally {

        System.out.println("Closing connections...");
        try {
            //Determine whether there were errors and commit or
            //rollback appropriately
            if (dbErrors) {
                db.rollback();
            } else {
                db.commit();
            }

            db.close();
        } catch (SQLException e) {
            System.out.println("Can't close connection.");
        }
    }
...

```

To keep track of whether there are errors, create a Boolean variable outside the `try-catch-finally` block and set it to `false`. If all statements complete without errors, the application will get to the `finally` block with Boolean variable still set to `false`. At this point, the transaction can be committed. If, on the other hand, errors are encountered, the `catch` block will set `dbErrors` to `true` and the application will roll back any operations.

You can also include other operations in making the decision on whether the transaction is committed.

Incorporating file actions into the transaction

Because you can directly control whether the transaction is committed, any action that can throw an exception can affect the decision. For example, you only want to add each order to the database once, so you might want to delete the `orders.xml` file when you are done with it. If that deletion fails for some reason, you will want to roll

back the transaction.

In fact, you can set up the application so that any error causes a rollback:

```

...
    } catch (SQLException e) {
        System.out.println("Error creating connection: "+e.getMessage());

        //An error occurred, so set the error flag
        dbErrors = true;

    } catch (Exception e) {
        //Display the error message and
        //set the error flag for any errors
        System.out.println(e.getMessage());
        dbErrors = true;
    } finally {

        System.out.println("Closing connections...");
        try {
            //Determine whether there were errors and commit or
            //rollback appropriately
            if (dbErrors) {
                db.rollback();
            } else {
                db.commit();
            }
        }
    }
...

```

This way, any exception will set `dbErrors` to true. You can also set this variable programmatically for any reason related to business logic.

Before counting on transactions, however, you should make sure that the database and driver are capable of supporting them.

Testing for transaction support

Not all databases and drivers support transactions, so before building this particular capability into your application, you should check to make sure that yours do.

You can determine support for transactions and other advanced features by checking the metadata for the database. Retrieve this information from the `Connection`:

```

...
import java.sql.DatabaseMetaData;

public class SaveOrders extends Object {
    ...
    Connection db = null;
    DatabaseMetaData dbmeta = null;
    //Set error flag
    boolean dbErrors = false;
    try {
        Class.forName(driverName);
        db = DriverManager.getConnection(connectURL);
    }
    ...
}

```

```

    //Get the database meta data
    dbmeta = db.getMetaData();

    //If transactions are supported, turn off autocommit
    if (dbmeta.supportsTransactions()) {
        db.setAutoCommit(false);
    }

    Statement statement = db.createStatement();

    Element ordersRoot = ordersDoc.getDocumentElement();
    NodeList orders = ordersRoot.getElementsByTagName(orderElName);
...
} catch (Exception e) {
    System.out.println(e.getMessage());
    dbErrors = true;
} finally {

    System.out.println("Closing connections...");

    try {
        //Determine whether there were errors and commit or
        //rollback appropriately
        if (dbmeta.supportsTransactions()) {
            if (dbErrors) {
                db.rollback();
            } else {
                db.commit();
            }
        }
        db.close();
    } catch (SQLException e) {
        System.out.println("Can't close connection.");
    }
}
...

```

In this application, there are really only two places where it matters whether transactions are supported: when the autocommit is turned off, and when the transaction is either committed or rolled back. Everything else proceeds as normal, except that without transactions the operations are committed immediately.

If transactions are not supported, the application proceeds on its merry way without them. But in some cases, you will need to write two alternate paths depending on whether a capability is supported. One of those capabilities, which can be used to improve performance, is *batch processing*.

Using batch updates

As long as you are committing all or none of the inserts, why use the bandwidth to go to the database for every single operation? Instead, you can create a *batch* of operations, which are all executed on a single trip to the server:

```

...
    Element ordersRoot = ordersDoc.getDocumentElement();
    NodeList orders = ordersRoot.getElementsByTagName(orderElName);
    for (int i = 0; i < orders.getLength(); i++) {
        Element thisOrder = (Element)orders.item(i);
...

```

```
NodeList products = thisOrder.getElementsByTagName(productElName);
for (int j=0; j < products.getLength(); j++) {
...
    //If transactions are supported, add to the batch.
    //Otherwise, execute the statement
    if (dbmeta.supportsBatchUpdates()) {
        statement.addBatch(sqlTxt);
    } else {
        int results = statement.executeUpdate(sqlTxt);
    }
}

//If there is a batch, execute it
if (dbmeta.supportsBatchUpdates()) {
    System.out.println("Executing batch ...");
    statement.executeBatch();
}
} catch (ClassNotFoundException e) {
...
}
```

If the database and driver support batch updates, add the statements to the batch rather than executing them. When the loops are complete, if support exists, execute the batch. If there is no support, the statements will already have been executed.

As you can see, the process so far has been relatively straightforward; all of the data being added to the database has been independent of data that may or may not already exist. The situation is not always that simple.

Section 7. Updatable ResultSets

Using ResultSets

In order to aggregate the order information, you'll need to integrate with information that is already in the database rather than simply creating and executing SQL statements. One way to do this is to use *updatable* ResultSets.

A `ResultSet` is an object returned by a database query. It is a structure of data that allows you to loop through it, accessing the data by index or by column name. For example, you could loop through the contents of the products table:

```
ResultSet resultset =
    statement.executeQuery("select * from productorders");
while (resultset.next()) {
    System.out.print(resultset.getString("productid"));
    System.out.print("|");
}
```

```
System.out.print(resultSet.getString("quantity"));
System.out.print("|");
System.out.println(resultSet.getString("revenue"));
}
```

(The [Resources](#) include a link to an in-depth tutorial on using read-only `ResultSet`s with XML.)

This code snippet demonstrates the steps required to use a `ResultSet`. First, create it by populating it with the results of a query. Then, loop through each record and output the results. Use the `next()` method to move to the next record; the "pointer" starts before the first record, so the first time you invoke `next()`, it goes to the first record (if there is one). As long as `next()` finds another record, it will return `true`.

There are methods for retrieving data as any of the available types. Like `getString()` here, they each can take either the name of the column (more convenient, but less efficient) or the position of the column in the `ResultSet` (more efficient, but less convenient).

The real power of a `ResultSet` comes when you use it to update the database directly.

Preparing the Statement for updatable `ResultSet`s

Created with default settings, a `ResultSet` is a one-time-through, forward-only, read-only object. You get one shot at the data, and if you need it again, you must query the database again.

It does not, however, have to be this way. By setting parameters on the `Statement` object, you can control the `ResultSet`s it produces. For example:

```
...
Class.forName(driverName);
db = DriverManager.getConnection(connectURL);
Statement statement = db.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                         ResultSet.CONCUR_UPDATABLE);

String orderElName = xmlfileEl.getElementsByTagName("order").item(0)
                    .getFirstChild().getNodeValue();
...
```

This `Statement` will now produce `ResultSet`s that can be updated, and will pick up changes made by other database users. You can also move backward and forward through this `ResultSet`.

The first parameter specifies the type of `ResultSet`. The options are:

- `TYPE_FORWARD_ONLY`: The default type. Allows only one forward pass and will not be affected by changes other users make to the database.
- `TYPE_SCROLL_INSENSITIVE`: Allows movement forward or backward in the list, and even specific positioning such as going to the fourth record in the list, or back two records from the current position. Will not be affected by changes other users make to the database.
- `TYPE_SCROLL_SENSITIVE`: Like `TYPE_SCROLL_INSENSITIVE`, allows positioning within the records. This type *is* affected by changes other users make. If a user deletes a record after the query has been executed, that record will disappear from the `ResultSet`. Similarly, changes to the values of data will be reflected in the `ResultSet`.

The second parameter sets the concurrency of the `ResultSet`, which determines whether or not it may be updated. The options are:

- `CONCUR_READ_ONLY`: The default value, specifying that a `ResultSet` may not be updated
- `CONCUR_UPDATABLE`: Specifies that the `ResultSet` may be updated

Note that just because you have updated the `ResultSet` does not necessarily mean that the changes are reflected in the database.

Updating the ResultSet

Creating and updating the `ResultSet` is much like creating and reading the `ResultSet` in the earlier example:

```

...
import java.sql.ResultSet;

public class SaveOrders extends Object {
...
    int results = statement.executeUpdate(sqlTxt);
}

    //Select this product from the productOrders table
    String updateSqlTxt = "select * from productOrders where productid = '"+
        thisProductid +"'";
    ResultSet updateRs = statement.executeQuery(updateSqlTxt);

    //Get new quantities and update the ResultSet
    int oldQty = updateRs.getInt("quantity");
    updateRs.updateInt("quantity", getSum(oldQty, qtyStr));

    double oldRev = updateRs.getDouble("revenue");
    updateRs.updateInt("revenue", getRev(oldRev, qtyStr, priceStr));

    //Send the changes to the database
    updateRs.updateRow();
}

```

```

    }
  } catch (ClassNotFoundException e) {
    ...
  }
}

```

Create the `ResultSet` as usual, but instead of reading the columns using methods such as `getString` and `getInt`, update the `ResultSet` fields with methods such as `updateString` and `updateInt` (depending on the type within the database). The first parameter is the name of the field (though you can also use the field index) and the second is the value to be set. (`getSum()` and `getRev()` simply do the type conversions and math.)

These changes do not affect the database until the `updateRow()` method is called. They do, however, affect the values stored in the `ResultSet`. (You can also cancel the changes before they are sent to the database using `cancelRowUpdates()`.)

From here, you can easily apply the mapping information.

Mapping the updates

One advantage of using updatable `ResultSet`s is the ease with which they can be converted to use the variables that came from the mapping file:

```

...
import java.sql.ResultSet;

public class SaveOrders extends Object {
  ...
    int results = statement.executeUpdate(sqlTxt);

    //Select this product from the productOrders table
    String updateSqlTxt = "select * from " + products_table
        + " where " + products_productid
        + " = '" + thisProductid + "'";
    ResultSet updateRs = statement.executeQuery(updateSqlTxt);

    //Get new quantities and update the ResultSet
    int oldQty = updateRs.getInt(products_quantity);
    updateRs.updateInt(products_quantity, getSum(oldQty, qtyStr));

    double oldRev = updateRs.getDouble(products_total);
    updateRs.updateDouble(products_total, getSum(oldRev, qtyStr, priceStr));

    //Send the changes to the database
    updateRs.updateRow();
  }
} catch (ClassNotFoundException e) {
  ...
}

```

In this case, the SQL statement is the only place where `Strings` have to be combined in a complex way. The variables representing the names of the columns are simply referenced in the `updateXXX()` method.

A serious problem still exists, however: what if the record doesn't exist yet?

Inserting a new record

If there is no record for a given product, you'll need to insert a new record into the database. You do this with a special row in the `ResultSet` called the *insert row*. The insert row is a special row created as a temporary space to hold changes -- similar to the changes of updating a record -- until those changes are committed to the database.

This example shows how to test for the existence of a current record, and how to create a new one if that test fails:

```
...
String updateSqlTxt = "select * from " + products_table
                    + " where " + products_productid
                    + " = '" + thisProductid + "'";
ResultSet updateRs = statement.executeQuery(updateSqlTxt);
if (updateRs.next()) {

    //Get new quantities and update the ResultSet
    int oldQty = updateRs.getInt(products_quantity);
    updateRs.updateInt(products_quantity, getSum(oldQty, qtyStr));

    double oldRev = updateRs.getDouble(products_total);
    updateRs.updateDouble(products_total, getRev(oldRev, qtyStr, priceStr));

    //Send the changes to the database
    updateRs.updateRow();
} else {

    //Create new placeholder row
    updateRs.moveToInsertRow();
    //Add data to placeholder
    updateRs.updateInt(products_productid, getSum(0, thisProductid));
    updateRs.updateInt(products_quantity, getSum(0, qtyStr));
    updateRs.updateDouble(products_total, getRev(0, qtyStr, priceStr));
    //Send new row to the database
    updateRs.insertRow();

}
...

```

Just as in updating, no changes are actually made to the database when the `updateXXX` methods are called. When you execute `insertRow()`, the database receives the new data. (In this case, `getSum()` and `getRev` are used to convert types.)

You now have a completely generic way to both update and insert data into the database.

Section 8. JDBC/XML data insertion summary

Summary

Although XML files are great for storing small amounts of data, eventually you will want to use XML with a database. This tutorial has discussed the basics of connecting to a database via JDBC, as well as inserting data and creating database transactions.

Data was entered into the database based on an external XML mapping file that determined the database, tables, column names, and element and attribute names in the XML file holding the data to be inserted into the database.

The tutorial also discussed updatable `ResultSet`s, which let you directly alter the data programmatically, based on the predetermined mappings.

Resources

Learn

- For information on using JDBC to insert data from an XML document into a database, read the companion tutorial [Using JDBC to extract data into an XML document](#).
- 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 introduction to programming XML, try Doug Tidwell's [XML programming in Java](#) on developerWorks.
- 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.
- See Sun's [Sun Microsystems' information on JDBC](#).
- 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
- For pointers to details on working with DB2 and JDBC, see the [Java Enablement with DB2](#) summary page.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- Download a [zip archive of the sample code](#) presented in this tutorial.
- Download [the Java 2 SDK](#), Standard Edition version 1.3.1.
- Download [JAXP 1.1](#), the Java 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](#).
- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content.](#)

About the author

Nicholas Chase



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.