SYBEX Sample Chapter

Mastering™ JSP™

Todd Cook

Chapter 7: Database Access

Copyright © 2002 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-2940-4

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

Chapter 7

Database Access

ALMOST ALL APPLICATIONS MUST deal with saving and preserving data. Although some information can be stored in property files and the system environment, using a database is a natural way to ease development and increase the scalability of an application. Even if an application isn't primarily designed for storing and cataloging information, it can still benefit from having easy access to data storage and retrieval.

Databases are the lifeblood of many systems and applications, and being able to understand them and manipulate them comfortably is important to the career of a software engineer. Data management is such a large task that a database team is an integral part of any medium- to large-sized organization, and an organization can have a number of enterprise database systems.

In this chapter, we'll discuss how to connect to a database using JDBC (Java Database Connectivity). We'll use the MySql database for the chapter examples, and we'll try to keep the code as portable as possible. (We'll discuss portability and how to use complicated relational designs in Chapters 13 and 14.)

Featured in this chapter:

- Understanding database connectivity
- Using Java Database Connectivity
- Basic connection pooling
- A sample application illustrating basic database interactions

Database Connectivity and JDBC

Before we look in depth at JDBC, let's look at the big picture. When a database starts, it usually services requests by listening to a network port, often using TCP/IP (Transmission Control Protocol/Internet Protocol). The situation is analogous to a web server that starts, listens to port 80, and then responds to requests. However, unlike a web server, no two databases speak exactly the same language. Even a complex application that deals with raw streams of data needs a translator to facilitate the connection to the database. That translator mechanism is a database driver.

The translation layer can be a call interface that is compiled into a program, or it can be an ODBC (open database connectivity) or a JDBC driver. Compiling a call interface into a program is the most

fossilized approach; however, it is suitable for rare occasions when the interfaces can be determined in advance and when the extra time it takes to load a driver is unacceptable.

Loadable drivers were first made available with ODBC. ODBC drivers were written in C or C++, and an individual driver had to be created and distributed for each operating system. Writing good C/C++ code that twists the bits of customized objects and handles huge amounts of data is no easy task. Therefore, new ODBC drivers were released when developers could get them out, and if your operating system wasn't supported or supported quickly, all you could do was get in line to complain. Writing drivers for everybody was a lot of work before Java.

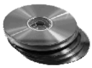
JDBC takes advantage of Java's virtual machine layer and Java's ability to produce write-once-run-anywhere code. Early on, there was a hodgepodge of JDBC drivers, many of which were hybrid ODBC/JDBC drivers. Today, every major database maker provides pure Java database drivers. The JDBC classes and objects related to the use of JDBC are found in the `java.sql` and `javax.sql` packages.

How JDBC Starts

You must explicitly load a JDBC driver into the DriverManager in the Java runtime environment. You can do so by calling the `registerDriver` method on the `java.sql.DriverManager` object like this:

```
DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver());
```

Or you can use the more general dynamic class loading method `java.lang.Class.forName()` like this:



```
Class.forName("org.gjt.mm.mysql.Driver")
```

In both cases, the name in parentheses is the packaged class name of the driver. The second example is used in this chapter's sample code, and you'll find the actual file `mm.mysql-2.0.4-bin.jar` on the CD.

`Class.forName` forces the class to be dynamically loaded into the Java runtime environment; `registerDriver` forces the `java.sql.DriverManager` to explicitly load the named class. The success of `Class.forName` merely ensures that the Java runtime environment can find the class named; the success of `registerDriver` presumably signals the runtime presence of the `java.sql.*` classes. In either case, a flawed or corrupted driver can still throw an exception on attempting a connection, or perhaps a faulty driver will cause an error because it relies on a deprecated method. At any rate, `registerDriver` performs more rigorous checking, and it's more likely to throw an exception.

You establish a connection by calling the `getConnection` method on the `java.sql.DriverManager` object. There are several overloaded function call signatures for getting a connection. Each takes a URL-based connection string, a URL string, a properties object, a URL-based connection string, and two additional strings for username and password.

Let's look at an example of the connection string and see how it's composed:

```
cn = DriverManager.getConnection
   ("jdbc:mysql://localhost:3306/mjsp:mjsp:mjsp");
```

This string includes the connection method (`jdbc/jdbc:odbc`), the database driver type, the machine name with the database followed by a colon and the port number the database is listening to, the default startup database, and another colon separating the username and password.

To prevent confusion, the examples in this chapter's code use the most explicit separation of the elements.

Listing 7.1, the JdbcSetupTest JSP, tests the plain setup of the MySql database and the JDBC driver that should be placed in the classpath. Figure 7.1 shows the result of a successful test.

LISTING 7.1: JDBCSETUPTEST.JSP

```
<%@ page import="java.sql.*"%>
<% //@ page import="org.gjt.mm.mysql.*"
 //uncomment to force "compile time" checking %>
<%@ page errorPage="JdbcConnectError.jsp" %>
<HTML>
<HEAD>
<TITLE>
JDBC Setup Test
</TITLE>
</HEAD>
<BODY>
<H1>
JDBC Setup Test
</H1>
<%
java.sql.Connection cn ;
  Class.forName("org.gjt.mm.mysql.Driver");
  cn = DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/test", "root", "");
  cn.close();
%>
<P>
If this message displays in the browser,
<br>
 your MySql JDBC connection is set up properly.
</P>
<BR>
</BODY>
</HTML>
```

WARNING *As soon as possible, change the default passwords on your database installation. Using the default passwords opens a security hole that's too great to ignore, even though many enterprise installations forget to change these passwords. For more information on security, see Chapter 6.*

FIGURE 7.1
The JDBC Setup Test page

To help you troubleshoot your JDBC installation, the JDBC setup test page references an error page if an Exception is thrown. The error page, which is created in Listing 7.2, provides some troubleshooting information, and it's shown in Figure 7.2.

LISTING 7.2: JDBCCONNECTERROR.JSP

```
<%@page isErrorPage="true" %>
<HTML>
<HEAD>
<TITLE>
JDBC Connect Error
</TITLE>
</HEAD>
<BODY>
<H3>
JDBC Connect Error
</H3>
<table width="600">
<tr><td>
There are several reasons a JDBC connection can fail:
</td></tr>
<tr><td>
<UL>
<LI>the MySql database is not installed</LI>
<LI>the MySql database is not running</LI>
<LI>the JDBC driver is not installed</LI>
<LI>the JDBC driver is not in the classpath
     of the JavaServer Engine</LI>
<LI>the naming scheme of the downloaded
  JDBC driver is different <br>
    compared with the name found
    in the Java code that invokes the driver by name.</LI>
</UL>
</td></tr>
<tr><td>
Unfortunately, the above explanations
  are probably more lucid
<br> than the usually
  cryptic exception message.
```

```
</td></tr>
<tr><td>
JdbcSetupTest.jsp threw this exception report:<BR>
<%= exception %>
</td></tr>
<tr><td>
Stack Trace:
</td></tr>
<tr><td>
<PRE>
<% exception.printStackTrace(new PrintWriter (out));%>
</PRE>
</td></tr>
</table>
</BODY>
</HTML>
```

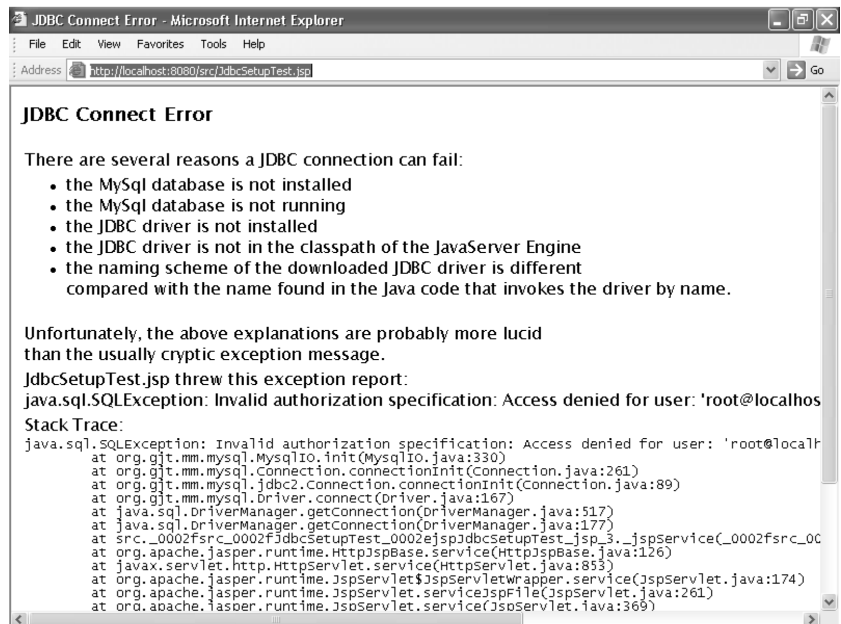


FIGURE 7.2
The JDBC Error page

Using JDBC: A Simple Product, Account, and Order Application

All the sample pages in this chapter require that you create one database and four SQL tables in MySQL. The setup of MySQL is detailed in Appendix A. All you need to generate the tables required to run the examples in this chapter is the SQL script shown in Listing 7.3.

LISTING 7.3: DATABASEANDTABLES.SQL

```
create database mjsp;
use mjsp;
create table accounts
(
 account_id int(11) auto_increment NOT NULL,
 biz_name varchar(50) NOT NULL,
 rep_fname varchar(50) NOT NULL,
 rep_lname varchar(50) NOT NULL,
 rep_phone varchar(15) NULL,
  PRIMARY KEY (account_id)
);

create table products
(
 product_id int(11) auto_increment NOT NULL,
 product_name varchar(99) NOT NULL,
 price double(9, 2) NOT NULL,
  PRIMARY KEY (product_id)
);

create table orders
(
 order_id int(11) auto_increment NOT NULL,
 account_id int(11) NOT NULL,
 timeordered date NOT NULL,
 timeshipped date  NULL,
 paid char(1) NULL,
  PRIMARY KEY (order_id),
  FOREIGN KEY (account_id) REFERENCES accounts (account_id)
);

create table orderdetail
(
 order_detail_id int(11) auto_increment NOT NULL,
 order_id int(11) NOT NULL,
 product_id int(11) NOT NULL,
 price double (9, 2) NOT NULL,
  PRIMARY KEY (order_detail_id),
  FOREIGN KEY (order_id) REFERENCES orders (order_id),
  FOREIGN KEY (product_id) REFERENCES products (product_id)
);
/* Adding a user. See MySql Documentation for more info */
GRANT ALL PRIVILEGES ON *.* TO mjsp@localhost
  IDENTIFIED BY 'mjsp' WITH GRANT OPTION;
```

Without too much effort, you should be able to create these tables in another database. We deliberately kept the SQL in these chapter examples simple to avoid undue hassles during setup. PostgreSQL is another open-source database that could also be used for the examples in this book. Advanced—and usually expensive—database systems offer the ability to use stored procedures, which can be a tremendous help in factoring out SQL manipulations and building a more robust object-oriented application.

TIP *Don't neglect proper database design and sane practices. If you're interested in further study, we recommend Joe Celko's* SQL for Smarties *(Morgan Kaufmann, 1999) and C. J. Date's* An Introduction to Database Systems *(Addison-Wesley 1999) for a thorough explanation of theoretical underpinnings of database design.*

Once you establish a JDBC connection, you must negotiate interactions with the database using statements (`java.sql.Statement`). A JDBC statement encapsulates a single database call, a packaging of a SQL statement. If the object already exists in the database, such as a stored procedure, use a CallableStatement (`java.sql.CallableStatement`). Stored procedures require an advanced understanding of SQL, and we'll look at the advantages of stored procedures in Chapter 14.

Statements ferry commands to the database, but query results are returned only in ResultSets (*java.sql.ResultSet*). The result set object is a full-fledged Java object with lots of functionality. Each result set packages the rows returned by a statement or a CallableStatement as an item. To access data in the next row of a result, call the `next()` member function.

NOTE *To access the data, you must call the* `next` *member function of a ResultSet:* `next` *returns boolean* `true` *if there's data inside the ResultSet, and if the ResultSet is empty,* `next` *returns* false*. You must call* `next` *before attempting to read any of the data.*

Once `next() is` called on the ResultSet, the data values for each column are accessed by requesting them in the header field named in the SQL statement. For example, if the SQL query is `Select account_id...`, the data value is accessed by calling `rs.getString("account_id")`.

Figure 7.3 illustrates these principles. To get these results, the SimpleConnect JSP (see Listing 7.4) requires users to first enter data in the database tables, something we cover later in this chapter.

LISTING 7.4: SIMPLECONNECT.JSP

```
<%@ page import="org.gjt.mm.mysql.*"%>
<%@ page import="java.sql.*"%>
<HTML>
<HEAD>
<TITLE>
Simple Connect
</TITLE>
</HEAD>
<BODY>
<H1>
Simple Connect
</H1>
```

```
Accounts Listing:
<BR>
<%
java.sql.Connection cn ;
try
 {
  Class.forName("org.gjt.mm.mysql.Driver");
  cn = DriverManager.getConnection(
                         "jdbc:mysql://localhost:3306/mjsp",
                         "mjsp", "mjsp");

  java.sql.Statement stm = cn.createStatement ();
  java.sql.ResultSet rs = stm.executeQuery("Select
                             account_id, biz_name,
                             rep_fname, rep_lname,
                             rep_phone from accounts");

    while (rs.next())
    {
    out.print("account_id: ");
    out.print( rs.getString("id"));
    out.print( " biz_name: ");
    out.print( rs.getString("biz_name"));
    out.print( " rep_fname: ");
    out.print( rs.getString("rep_fname"));
    out.print( " rep_lname: ");
    out.print( rs.getString("rep_lname"));
    out.print( " rep_phone: ");
    out.print( rs.getString("rep_phone"));
    out.print( "<BR>");
    }
     rs.close();
     if (stmt != null)
        {
         stmt.close();
        }
     if (cn != null)
        {
         cn.close();
        }
 }
catch (SQLException e)
 {
  e.printStackTrace();
 }
%>
</BODY>
</HTML>
```

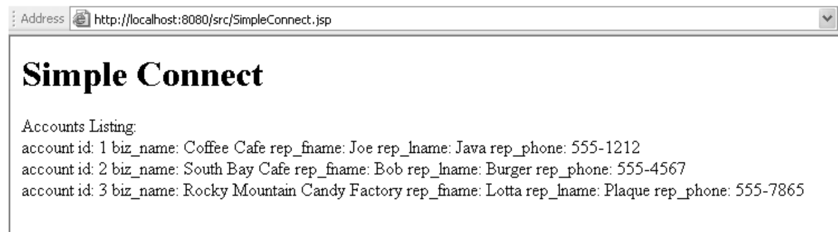


FIGURE 7.3
Connecting, querying, and printing the values

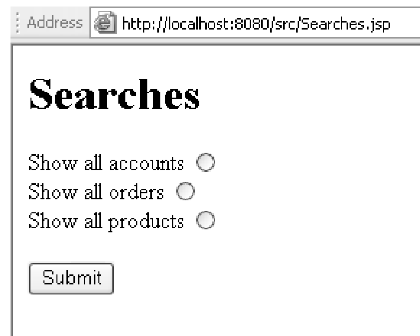
The ResultSet is created, cycled through, printed out, and closed. Also, the connection is created and closed. Calling `close` on a connection that has already been closed throws an exception, so check before calling `close()`. If an exception occurs during connection or while using the ResultSet, an SQLException is thrown. An SQLException is similar to a regular Exception except that additional database information can be packaged in the Exception. (We'll look at exceptions and error handling in detail in the next chapter.)

But there's a lot of code in the page in Figure 7.3, so let's see how we can thin things out and make it more flexible. A simple printout of a query isn't very helpful, so let's see how this can be incorporated into some basic searches. First, let's look at our setup page, shown in Listing 7.5, which could have been done in plain old, static HTML, illustrated in Figure 7.4.

LISTING 7.5: SEARCHES.JSP

```
<HTML>
<HEAD>
<TITLE>
Searches
</TITLE>
</HEAD>
<BODY>
<H1>
Searches
</H1>
<FORM action="SearchResultDisplay.jsp" method="GET">
Show all accounts
<input type="radio" name="search" value="accounts">
<BR>
Show all orders
<input type="radio" name="search" value="orders">
<BR>
Show all products
<input type="radio" name="search" value="products">
<BR>
<BR>
<input type="Submit" value="Submit">
</FORM>
</BODY>
</HTML>
```

FIGURE 7.4
A simple search form



This page feeds into SearchResultDisplay JSP, shown in Listing 7.6, which performs several operations to format the data.

LISTING 7.6: SEARCHRESULTDISPLAY.JSP

```
<%@ page import="org.gjt.mm.mysql.*"%>
<%@ page import="java.sql.*"%>
<HTML>
<HEAD>
<TITLE>
SearchResultDisplay
</TITLE>
</HEAD>
<BODY>
<H1>
Search Result Display
</H1>
<%
String[] mArFields = {""};
 //default initialization
String[] mArHeaderNames = {""};
String mSQL = "";

java.sql.Connection cn ;
// use if switch and arrays for field & header allocation

if ("products".equalsIgnoreCase(
                             request.getParameter("search")
                                ))
{
  mArFields = new String[] { "product_id",
                             "product_name",
                             "price" };
```

```
  mArHeaderNames = new String[] { "Product Id",
                                  "Product Name",
                                  "Price" };
  mSQL = " select product_id, product_name, price
          from products ";
}
if ((“accounts”").equalsIgnoreCase(
                              request.getParameter("search")
                                  ))
{
 mArFields = new String[] { "account_id",
                             "biz_name",
                             "rep_fname",
                             "rep_lname",
                             "rep_phone" };
 mArHeaderNames = new String[] {"Account Id",
                                 "Business Name",
                                 "Representative First Name",
                                 "Last name",
                                 "Phone"};
 mSQL = " SELECT account_id, biz_name, rep_fname, rep_lname,
         rep_phone from accounts  ";
}
if ("orders".equalsIgnoreCase(
                            request.getParameter("search")
                              ))
{
 mArFields = new String [] { "order_id",
                             "biz_name",
                             "timeordered",
                             "timeshipped",
                             "paid" };
 mArHeaderNames = new String [] { "Order Id",
                                  "Business Name",
                                  "Time ordered",
                                  "Time shipped",
                                  "Paid" };
 mSQL = " select o.order_id, a.biz_name, o.timeordered,
          o.timeshipped,  o.paid from orders o, accounts a
          where o.account_id = a.account_id  ";
}

try
 {
  Class.forName("org.gjt.mm.mysql.Driver");
  cn = DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/mjsp",
                        "mjsp", "mjsp");
```

```
    java.sql.Statement stm = cn.createStatement ();
    java.sql.ResultSet rsSearch = stm.executeQuery( mSQL );

  %>
   <table bgcolor="lightblue" border="1">
   <TR>
       <%     for( int i=0; i < mArHeaderNames.length; i++ )
                { %>
       <TD><% out.print(mArHeaderNames[i]); %></TD>
       <%       } %></TR><%
      while ( rsSearch.next() )
            {
       %><TR><%
              for ( int ii = 0; ii < mArFields.length; ii++ )
             { %><TD><%
             try
              {
              if (rsSearch.getString(mArFields[ii]) == null)
                  out.print ( " " );
              else
               out.print( rsSearch.getString(mArFields[ii]) );
              }
             catch (Exception ee)
              {
              out.print(" ");
              }
                 %></TD><%
             } %></TR><%
            } %>
  </table>
  <%
  cn.close();
   }
  catch (SQLException ee)
   {
    ee.printStackTrace();
   }
  %>
  </BODY>
  </HTML>
```

A lot is going on with this page besides the usual creating a connection and making a ResultSet. First, the search parameters are checked, and an appropriate set of array objects is initialized. The `mArFields` array object contains the SQL column header names returned in the ResultSet, and another array `mArHeaderNames` contains the formatted column names as you would like them to

appear in the HTML table that is generated in the latter section of the page. The table is built by printing out the HTML-ready header name (`mArHeaderNames`). Each row of the table is then built by cycling the array of SQL fields (`mArFields`) against the ResultSet. When the array reaches the end, the loop cycles over, the ResultSet is advanced, and another row is repeated until it's finished. The page display with data is shown in Figure 7.5.

Avoiding an Extra Function Call to Check Null Value Parameters

If `java.lang.String.equals()` is passed a variable that's initialized to a null value, an exception is thrown. Frequently, to prevent these exceptions, developers write code that tests for a null value and then makes a comparison, as in the following:

```
If (request.getParameter("searches") != null)
{
If (request.getParameter("searches").equals("products")
{
// do products processing
}
If (request.getParameter("searches").equals("accounts")
{
// do accounts processing
}
```

This approach, although common, is somewhat inappropriate: `java.lang.String.equals()` is inherited from `java.lang.Object.equals()`, and the member function compares two objects to see if they are the same. It throws an exception if the object compared is a null value. More important, on a practical level it creates code that breaks too easily if another developer changes the parameter string value to `"Products"` instead of `"products"`.

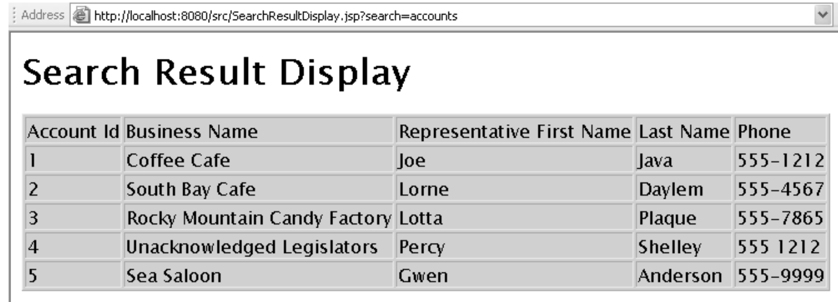
A better way to check for nulls and robustly check for string values is to use the `java.lang.String .equalsIgnoreCase()` function. You can prevent the code from breaking in the future because of capitalization by using this function, like:

```
If (request.getParameter("searches") != null)
{
If (request.getParameter("searches").equalsIgnoreCase("products")
{ …
```

To avoid that extra function call required to check for the null value, you can use the somewhat unexpected functionality afforded by the `java.lang.String.equalsIgnoreCase()` function. Comparing a string case insensitively to a null value returns false, instead of throwing an exception. So the proper and economical idiom is:

```
If ("products".equalsIgnoreCase(request.getParameter("searches")) )
{ …
```

FIGURE 7.5
Basic search results



Account Id	Business Name	Representative First Name	Last Name	Phone
1	Coffee Cafe	Joe	Java	555-1212
2	South Bay Cafe	Lorne	Daylem	555-4567
3	Rocky Mountain Candy Factory	Lotta	Plaque	555-7865
4	Unacknowledged Legislators	Percy	Shelley	555 1212
5	Sea Saloon	Gwen	Anderson	555-9999

In Chapter 14, you'll see how to refine the presentation of data on a page. But first let's look at how to remove some of the obvious repetitive code using connection pooling.

Connection Pooling

It takes valuable time to connect to a database—often it's the longest operation—and as an application grows, more connections and more users can exact a heavy toll on performance. Fortunately, there's a way to economize and share one connection: connection pooling. The concept is so widely accepted that it has been incorporated into the enterprise features of JDBC 2.*x*. However, since not all JDBC drivers implement the full 2.*x* features, and because it requires using the `javax.sql.DataSource` object and some other complications, we'll use a simple custom class here to introduce the concept.

A Connection Manager Class

Our connection class, shown in Listing 7.7, is used to create and share a connection across a user's browser session (instantiated and referenced on pages as a session bean).

LISTING 7.7: CONNECTIONMANAGER.JAVA

```
package c7;

import java.sql.*;
import javax.servlet.http.*;

public class ConnectionManager implements
                             HttpSessionBindingListener
{
  private Connection connection;
  private Statement statement;
  // In Chapter 13 this will be moved
  // to a Properties file assignment
  private String driver = "org.gjt.mm.mysql.Driver";
  private String dbURL =
     "jdbc:mysql://localhost:3306/mjsp";
```

```
 private String login = "mjsp";
 private String password = "mjsp";

public ConnectionManager()
 {
 }

public void setDriver (String sDriver)
 {
  if (sDriver != null)
      driver = sDriver;
 }

public String getDriver ()
 {
  return driver;
 }

public void setDbURL (String sDbURL)
 {
   if (sDbURL != null)
    dbURL = sDbURL;
 }

public String getDbURL()
 {
  return dbURL;
 }

public void setLogin (String sLogin)
 {
  if (sLogin != null)
     login = sLogin;
 }

public String getLogin()
 {
  return login;
 }

public void setPassword (String sPassword)
 {
  if (sPassword != null)
     password = sPassword;
 }

private String getPassword()
```

```
 {
  return password;
 }

private void getConn ()
 {
  try
   {
     Class.forName(driver);
     connection =
      DriverManager.getConnection(dbURL,login,password);
     statement=connection.createStatement();
   }
  catch (ClassNotFoundException e)
   {
     System.out.println(
       "ConnectionManager: driver unavailable");
     connection = null;
   }
  catch (SQLException e)
   {
     System.out.println(
       "ConnectionManager: driver not loaded");
     connection = null;
   }
 }

public Connection getConnection()
 {
  if (connection == null)
     getConn();
  return connection;
 }

public void commit() throws SQLException
 {
   connection.commit();
 }

public void rollback() throws SQLException
 {
   connection.rollback();
 }

public void setAutoCommit(boolean autoCommit)
   throws SQLException
 {
```

```
   connection.setAutoCommit(autoCommit );
 }

 /**
  *  Passing ResultSets can be hazardous;
  *  for this demo it's okay, but see Chapter 14 for
  *  an explanation and workarounds
  */

public ResultSet executeQuery(String sql)
                              throws SQLException
 {
   if (connection == null || connection.isClosed())
       getConn();
   return statement.executeQuery(sql);
 }

public int executeUpdate(String sql) throws SQLException
 {
   if (connection == null || connection.isClosed())
       getConn();
   return statement.executeUpdate(sql);
 }

  /**
   * Required implementation for session binding
   */

public void valueBound(HttpSessionBindingEvent event)
 {
   System.err.println(
            "ConnectionBean: in the valueBound method");
   try
    {
     if (connection == null || connection.isClosed())
      {
       connection =
       DriverManager.getConnection(dbURL,login,password);
       statement = connection.createStatement();
      }
    }
   catch (SQLException e)
    {
     e.printStackTrace();
     connection = null;
```

```
        }
    }

   public void valueUnbound(HttpSessionBindingEvent event)
    {
     try
      {
        if ( connection != null
             || !connection.isClosed())
           connection.close();
          }
     catch (SQLException e)
      {
        e.printStackTrace();
      }
     finally
      {
        connection = null;
      }
    }

   public void close()
    {
     try
      {
         if ( connection != null
              || !connection.isClosed())
          connection.close();
      }
     catch (SQLException e)
      {
        e.printStackTrace();
      }
    }
  }
```

The ConnectionManager class creates a connection, holds it, and hands out a copy when requested by a JSP or JavaBean. The class implements the `HttpSessionBindingListener` interface and the methods `valueBound()` and `valueUnbound()`—which is a fancy way of forcing the class to be instantiated at session scope. It also forces the class to connect when it is bound/rebound and to release the connection as soon as the session is terminated. The Connection object disappears with the destruction of the class and the session, but by calling `Connection.close()`, the ConnectionManager class politely informs the database that resources are no longer needed.

We kept the ConnectionManager class simple to facilitate understanding, and you could easily improve it by making the connection values (the database name, user, and password) come from a properties file (which of course should be protected by file permissions and security measures).

We'll use this ConnectionManager class in the rest of the code examples in this chapter, so it's important to get it working. In Listing 7.8, the ConnectionManagerTest JSP tests the ConnectionManager class.

LISTING 7.8: CONNECTIONMANAGERTEST.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM" scope="session" />
<HTML>
<HEAD>
<TITLE>
Connection Manager Test
</TITLE>
</HEAD>
<BODY>
<H1>
Connection Manager Test
</H1>
<%
try
{
 CM.executeQuery("select user from mysql.user");
 %><BR>Success! Connection Manager is set up properly<%
}
catch (Exception e)
{
 %><BR>Connection Manager is not set up properly;
        compiled and in the classpath<%
 out.print( new PrintWriter (e.printStackTrace()) );
}
%>
</BODY>
</HTML>
```

Figure 7.6 shows the successful results of the Connection Manager Test.

FIGURE 7.6
A successful test of the Connection Manager Setup page

Improving the Chapter's Sample Application Using the Connection Manager and JavaScript

Now let's see what else we can do to thin the server-side code for handling ResultSets. We'll change the name of the search page to avoid confusion. While we're at it, let's change some of the code to make it easier for users to interact with the page. If a page has a simple search, such as a Show All Orders, the search can be activated by using a radio button that's activated by some JavaScript to submit when the user clicks the radio button, for example, `onClick="javascript:submit();"`. The JavaScript enabled search page can be seen in Listing 7.9 and in Figure 7.7.

LISTING 7.9: SEARCHESWITHJAVASCRIPT.JSP

```
<HTML>
<HEAD>
<TITLE>
Searches With JavaScript
</TITLE>
</HEAD>
<BODY>
<H1>
Searches With JavaScript
</H1>
<FORM action="SearchResultsWithJavaScript.jsp" method="GET">
Show all accounts
<input type="radio" onClick="javascript:submit();"
      name="search" value="accounts">
<BR>
Show all orders
<input type="radio" onClick="javascript:submit();"
      name="search" value="orders">
<BR>
Show all products
<input type="radio" onClick="javascript:submit();"
       name="search" value="products">
<BR>
<BR>
</FORM>
<BR>
<BR>
Click <a href="AddProduct.jsp">here </a> to add a product.
<BR>
<BR>
Click <a href="AddAccount.jsp">here </a> to add an account.
<BR>
<BR>
Click <a href="CreateOrder.jsp">here </a> to create an order.
</BODY>
</HTML>
```

FIGURE 7.7
Enabling searches with JavaScript

The SearchResultsWithJavaScript JSP, shown in Listing 7.10, improves our design and presentation by using the ConnectionManager class and by using JavaScript arrays to organize the search category headers. Additionally, to improve the functionality, we've added code to insert hyperlinks to other pages using unique values that are primary keys for tables used in other lookups. For example, the `order_id` value writes out a hyperlinked pop-up window that will display details from the `order_details` table, which is shown in Figure 7.8.

LISTING 7.10: SEARCHRESULTSWITHJAVASCRIPT.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
      scope="session" />
<HTML>
<HEAD>
<TITLE>
Search Results With JavaScript
</TITLE>
</HEAD>
<BODY>
<H1>
Search Results With JavaScript
</H1>
<%
String [] mArFields = {""};
 // default instantiation/initialization
String [] mArHeaderNames = {""};
String mSQL = "";

// use if switch and arrays for field & header allocation
```

```
if ("products".equalsIgnoreCase(
                            request.getParameter("search")
                                ) )
{
  mArFields = new String [] { "product_id", "product_name",
                             "price" };
  mArHeaderNames = new String [] { "Product Id",
                                   "Product Name",
                                   "Price" };
  mSQL = " select product_id, product_name, price
           from products ";
  %>Click <a href="AddProduct.jsp">here</a>
            to add another product<%
}
if ("accounts".equalsIgnoreCase(
                            request.getParameter("search")
                                ) )
{
  mArFields = new String [] { "account_id",
                              "biz_name",
                              "rep_fname",
                              "rep_lname",
                              "rep_phone" };
  mArHeaderNames = new String [] { "Account Id",
                                   "Business Name",
                               "Representative First Name",
                                   "Last name",
                                   "Phone" };
  mSQL = " select account_id, biz_name, rep_fname,
   rep_lname, rep_phone from accounts  ";
  %>Click <a href="AddAccount.jsp">here</a> to add another
   account<%
}
if ("orders".equalsIgnoreCase(
                            request.getParameter("search")
                                ) ){
  mArFields = new String [] { "order_id",
                              "biz_name",
                              "timeordered",
                              "timeshipped",
                              "paid" };
  mArHeaderNames = new String [] { "Order Id",
                                   "Business Name",
                                   "Time ordered",
                                   "Time shipped",
                                   "Paid" };
  mSQL = " select o.order_id, a.biz_name, o.timeordered,
           o.timeshipped,  o.paid from orders o, accounts a
```

```
          where o.account_id = a.account_id
          order by order_id ";
  %><BR>Click on the Order id to see the line items or to
   add new items<BR>
  <BR>Click <a href="CreateOrder.jsp">here</a> to create an
   order for an established account<BR>
  <%
}
try
 {
  java.sql.ResultSet rsSearch = CM.executeQuery( mSQL );
 %>
 <table bgcolor="lightblue" border="1">
 <TR>
     <%     for( int i=0; i < mArHeaderNames.length; i++ )
        { %>
      <TD><% out.print(mArHeaderNames[i]); %></TD>
     <%     } %></TR><%
    while ( rsSearch.next() )
    {
     %><TR><%
          for ( int ii = 0; ii < mArFields.length; ii++ )
        {%><TD><%
     try
      {
         if (rsSearch.getString(mArFields[ii]) == null)
              out.print (" ");
        else if (mArFields[ii].equals("order_id"))
                {
                %><a target="_blank"
                href="ShowOrderDetails.jsp?order_id=<%=
                rsSearch.getString( "order_id") %>"><%=
                rsSearch.getString("order_id") %></a><%
                }
       else
            out.print( rsSearch.getString(mArFields[ii]) );
      }
     catch (Exception ee)
      {
          out.print(" ");
      }
               %></TD><%
         } %></TR><%
     } %>
</table>
<%
 }
catch (Exception ee)
```

```
  {
   ee.printStackTrace();
  }
%>
<BR>
<BR>
Click <a href="SearchesWithJavaScript.jsp">here</a>
     to return to searches
</BODY>
</HTML>
```

FIGURE 7.8

Search results displayed with JavaScript hyperlinking

Address http://localhost:8080/src/SearchResultsWithJavaScript.jsp?search=accounts

Search Results With JavaScript

Click here to add another account

Account Id	Business Name	Representative First Name	Last Name	Phone
1	Coffee Cafe	Joe	Java	555-1212
2	South Bay Cafe	Lorne	Daylem	555-4567
3	Rocky Mountain Candy Factory	Lotta	Plaque	555-7865
4	Unacknowledged Legislators	Percy	Shelley	555 1212
5	Sea Saloon	Gwen	Anderson	555-9999

Click here to return to searches

ADDING PRODUCTS

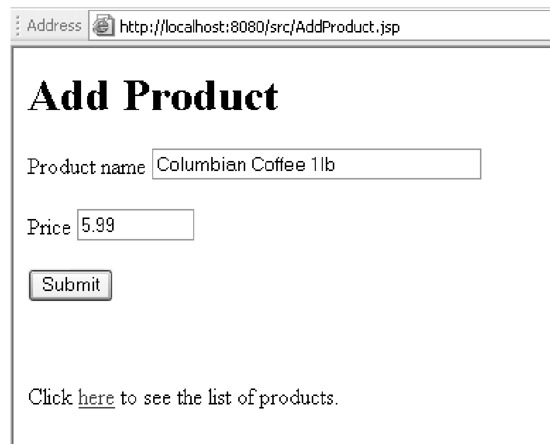
Before we can add items to an order, we need to add products to the products table. The AddProducts JSP, shown in Figure 7.9, is so plain that it could be an HTML file, but, of course, you could easily add "smart" functionality to this page. For example, you could add a table or a drop-down menu that lists previously entered products and prices or price validation code. Listing 7.11 shows the web page that allows users to enter data into the products table.

LISTING 7.11: ADDPRODUCT.JSP

```
<HTML>
<HEAD>
<TITLE>
Add Product
</TITLE>
</HEAD>
<BODY>
<H1>
Add Product
</H1>
<form action="InsertProduct.jsp" method="POST">
Product name <input name="prod_name" type="text" size="35">
```

```
<BR>
<BR>
Price <input name="prod_price" type="text" size="10">
<BR>
<BR>
<input type="submit" value="Submit">
</form>
<BR>
<BR>
Click <a href=
       "SearchResultsWithJavaScript.jsp?search=products">
        here</a> to see the list of products.
</BODY>
</HTML>
```

FIGURE 7.9
The Add Product page



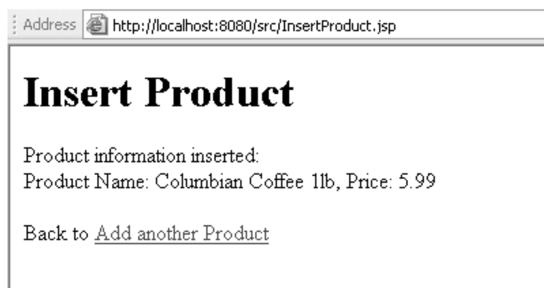
In Listing 7.12, the InsertProduct JSP checks the passed parameters and inserts the user's input into the database. If `prod_name` and `prod_price` are not passed in, nothing is executed. The code is slightly ugly in that the testing of parameters and the subsequent execution are wrapped by a large, bracketed `if` statement. A more elegant solution is to redirect the user to an error or a status page that details the deficiency or sends the user back to the AddProduct JSP. Successful execution is shown in Figure 7.10.

LISTING 7.12: INSERTPRODUCT.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
       scope="session" />
<HTML>
<HEAD>
<TITLE>
Insert Product
```

```
</TITLE>
</HEAD>
<BODY>
<H1>
Insert Product
</H1>
<%
if ( (request.getParameter("prod_name") != null)
     && (request.getParameter("prod_price") != null) )
 {
  String mSQL= "INSERT INTO products (product_name, price)
              VALUES ('" + request.getParameter("prod_name")
              + "', "+ request.getParameter("prod_price")
              + " )" ;
   try
    {
     System.out.println("SQL=" + mSQL);
      CM.executeQuery (mSQL);
     %>Product information inserted: <BR>Product Name: <%
     out.print(request.getParameter("prod_name"));
     %>, Price: <%
     out.print(request.getParameter("prod_price") ) ;
     }
   catch (Exception e)
    {
    %>There was a problem with your insert.
      Please go back and check your values.<%
    e.printStackTrace();
    }
 }
%>
<BR>
<BR>
Back to <a href="AddProduct.jsp">Add another Product</a>
</BODY>
</HTML>
```

FIGURE 7.10
The Insert Product page



Adding Accounts

Adding an account is done in a similar fashion using a plain old enter values page, shown in Listing 7.13 and depicted in Figure 7.11.

Listing 7.13: AddAccount.jsp

```
<HTML>
<HEAD>
<TITLE>
Add Account
</TITLE>
</HEAD>
<BODY>
<H1>
Add Account
</H1>
<FORM method="post" action="InsertAccount.jsp">
<BR>
Business Name
<BR>
<input type="text" name="biz_name">
<BR>
Representative's First Name
<BR>
<input type="text" name="rep_fname">
<BR>
Representative's Last Name
<BR>
<input type="text" name="rep_lname">
<BR>
Phone
<BR>
<input type="text" name="phone">
<BR>
Note: all fields are required.
<BR>
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

FIGURE 7.11
The Add Account page

InsertAccount JSP, shown in Listing 7.14, doesn't display any significant variation on the same approach used in InsertProduct JSP, and a successful execution is shown in Figure 7.12.

LISTING 7.14: INSERTACCOUNT.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
       scope="session" />
<HTML>
<HEAD>
<TITLE>
Insert Account
</TITLE>
</HEAD>
<BODY>
<H1>
Insert Account
</H1>
<%
if ( (request.getParameter("biz_name") != null)
     && (request.getParameter("rep_fname") != null)
     && (request.getParameter("rep_lname") != null)
     && (request.getParameter("phone") != null)
   )
  {
   String mSQL= "INSERT INTO accounts ( biz_name, rep_fname,
               rep_lname, rep_phone) VALUES ('"
               + request.getParameter("biz_name") + "', '"
               + request.getParameter("rep_fname") + "', '"
```

```
                + request.getParameter("rep_lname") + "', '"
                + request.getParameter("phone") + "' )" ;
    try
     {
      System.out.println("SQL=" + mSQL);
       CM.executeQuery(mSQL);
      %>Account information inserted: <BR>Account Name: <%
      out.print(request.getParameter("biz_name"));
      %>, Representative Name: <%
      out.print(request.getParameter("rep_fname") + " "
                 + request.getParameter("rep_lname") );
      %>, Phone: <%
      out.print(request.getParameter("phone") ) ;
     }
    catch (Exception e)
     {
      %>There was a problem with your insert.
        Please go back and check your values.<%
      e.printStackTrace();
     }
  }
%>
<BR>
<BR>
Back to <a href="AddAccount.jsp">Add another Account</a>
<BR>
<BR>
Click <a
      href="SearchResultsWithJavaScript.jsp?search=accounts"
      >here</a> to see a listing of accounts
</BODY>
</HTML>
</HTML>
```

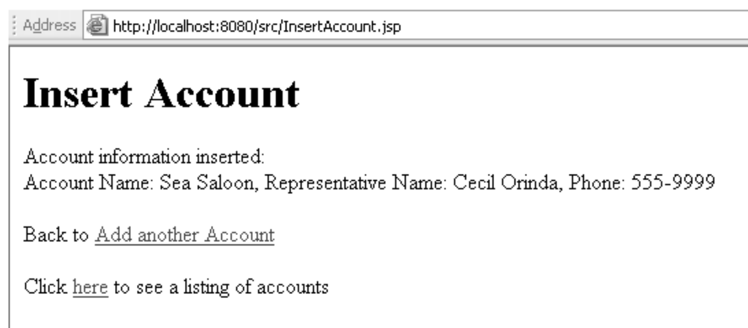


FIGURE 7.12
The Insert Account page

CREATING ORDERS

The CreateOrder JSP, shown in Listing 7.15, adds a dynamic element to the setup of an entry form: the JSP queries the accounts table and uses the results to populate a select list. This allows new entries to the accounts table to be integrated on the fly into the other pages. The page is shown in Figure 7.13.

LISTING 7.15: CREATEORDER.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
     scope="session" />
<HTML>
<HEAD>
<TITLE>
Create Order
</TITLE>
</HEAD>
<BODY>
<H1>
Create Order
</H1>
<FORM method="post" action="ProcessNewOrder.jsp">
<%
 try
{
  String mSQL = "select account_id, biz_name
                 from accounts ";
  System.out.println("SQL=" + mSQL);
  java.sql.ResultSet rsAccounts = CM.executeQuery(mSQL);
  %>
  Business Name:
  <select name="account_id"><%
  while (rsAccounts.next())
  {
   %><option value="<%=
   rsAccounts.getString("account_id")%>"><%=
   rsAccounts.getString("biz_name")%></option>
   <%
  }
  %></select><%
}
 catch (Exception e)
 {
  %><BR>Problem with listing accounts<BR><%
 }
%>
<BR>
<BR>
```

```
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Create New
   Order">
</FORM>
</BODY>
</HTML>
```

FIGURE 7.13
The Create Order page

The information is then passed to ProcessNewOrder JSP, shown in Listing 7.16, which simply adds a new order and prints out a hyperlink that sends the user to a page that lists the orders, as depicted in Figure 7.14.

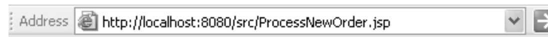
LISTING 7.16: PROCESSNEWORDER.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
        scope="session" />
<HTML>
<HEAD>
<TITLE>
Process New Order
</TITLE>
</HEAD>
<BODY>
<H1>
Process New Order
</H1>
<%
if ( (request.getParameter("account_id") != null) )
 {
  String mSQL= "INSERT INTO orders
                (account_id, timeordered) VALUES ('"
               + request.getParameter("account_id")
               + "', SYSDATE() )";
    try
    {
     System.out.println("mSQL=" + mSQL);
     CM.executeQuery(mSQL);
```

```
            %>New order created, click
            <a href="SearchResultsWithJavaScript.jsp?search=orders"
            >here</a> here to see all the orders. To add items to
            an order, select it from the search results page.<%
            }
          catch (Exception e)
          {
            %>There was a problem with your insert.
               Please go back and check your values.<%
            e.printStackTrace();
          }
     }
   %>
   <BR>
   <BR>
   Back to <a href="AddAccount.jsp">Add another Account</a>
   <BR>
   <BR>
   Click <a
         href="SearchResultsWithJavaScript.jsp?search=accounts"
         >here</a> to see a listing of accounts
   </BODY>
   </HTML>
```

FIGURE 7.14

The Process New Order page



ADDING ITEMS TO AN ORDER

Selecting an order from a search allows users to add additional order items, commonly referred to as line items because they are the separate lines that make up an order. AddOrderItem JSP, created in Listing 7.17, performs the process of adding to the order. The page is a little unusual in that it submits to itself. (In Chapter 2, you learned that the default `action` behavior of a form is to submit a `GET` request to itself, in effect, passing parameters to itself.) To discourage the accidental insertion of line items, we made the form action into a `POST`. The advantage of the self-reloading form is that users

often demand that an application allow them to enter line item values as quickly as possible. Just think of all the supermarket checkouts that now use scanners because they can ring up a bill (order) faster. The working page is shown in Figure 7.15.

LISTING 7.17: ADDORDERITEM.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
     scope="session" />
<HTML>
<HEAD>
<TITLE>
Add Order Item
</TITLE>
</HEAD>
<BODY>
<H1>
Add Order Item
</H1>
<%  //insert code
String mSQL ="";
if ( (request.getParameter("item") != null)
     && (request.getParameter("price") != null)
     && (request.getParameter("order_id") != null)
   )
 {
    mSQL = "INSERT INTO orderdetail ( order_id, product_id,
            price) VALUES ("
           + request.getParameter("order_id") + ", "
           + request.getParameter("item") + ", "
           + request.getParameter("price")+ " )" ;
    System.out.println("SQL=" + mSQL);
   try
     {
     CM.executeQuery( mSQL);
     }
    catch (Exception e)
    {
     %>There was a problem with your insert.
        Please go back and check your values.<%
     e.printStackTrace();
    }
 }
%>
<FORM method="POST">
<BR>
<BR>
```

```
Product: <select name="item">
<%
  mSQL = " select product_id, product_name from products ";
     try
    {
     System.out.println("SQL=" + mSQL);
     java.sql.ResultSet rsProd;
     rsProd = CM.executeQuery( mSQL);
     while (rsProd.next())
    {
     %>
     <option value="<%= rsProd.getString ("product_id")
     %>"><%= rsProd.getString("product_name" ) %></option>
     <%
     }
     %></select><%
     rsProd.close();
     }
    catch (Exception e)
    {
     %>There was a problem with your connection.
        Please check the database and try again.<%
     e.printStackTrace();
    }
%>
<BR>
<BR>
Price: <input type="text" name="price" size="8">
<BR>
<BR>
<input type="submit" value="Submit">
</FORM>
<BR>
<BR>
Click <a href="ShowOrderDetails.jsp?order_id=<%=
       request.getParameter("order_id") %>">here</a>
       to see the Order Details.
<BR>
<BR>
Click <a href="AddProduct.jsp">here</a> to add a product.
</BODY>
</HTML>
```

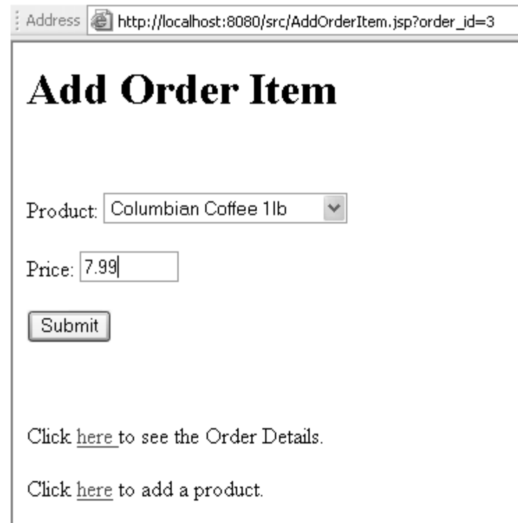


FIGURE 7.15
The Add Order Item page

Last, the AddOrderItem JSP prints out a hyperlink to show all the line items in the order. A revised version of the page might show this on the page where the order entries take place.

NOTE *The examples in this chapter are designed to acquaint you with the basic data manipulations required to build a simple application. The code is for illustration purposes and in the real world would be suitable only for a quick-and-dirty prototype. In Chapter 14, we'll examine some advanced techniques for producing a robust database application with a proper separation of the data manipulation tier from the Presentation layer.*

VIEWING THE DETAILS OF AN ORDER

The ShowOrderDetails JSP, shown in Listing 7.18, uses the basic concept of the search results pages that we looked at earlier in this chapter. The SQL query is probably the most complicated part of the code.

LISTING 7.18: SHOWORDERDETAILS.JSP

```
<jsp:useBean class="c7.ConnectionManager" id="CM"
     scope="session" />
<HTML>
<HEAD>
<TITLE>
ShowOrderDetails
</TITLE>
</HEAD>
<BODY>
<H1>
Show Order Details
</H1>
```

```
<% if (request.getParameter("order_id") != null)
{
 %>Click <a href="AddOrderItem.jsp?order_id=<%=
  request.getParameter("order_id") %>">here</a>
   to add an item to this order.
 <BR>
 <%
   String [] mArFields = new String [] {"order_detail_id",
                               "product_name", "price" };
   String [] mArHeaderNames = new String [] {
          "Line Item Number", "Product", "Price Charged" };
   String mSQL = " select o.order_detail_id, o.product_id,
                   p.product_name, o.price
                   from orderdetail o, products p
                   where p.product_id = o.product_id
                   and order_id ="
                   + request.getParameter("order_id");
   System.out.println("SQL" + mSQL);
  try
    {
    java.sql.ResultSet rsSearch = CM.executeQuery( mSQL );
    %>
     <table bgcolor="lightblue" border="1">
     <TR>
     <%     for( int i=0; i < mArHeaderNames.length; i++ )
        {
        %>
     <TD><% out.print(mArHeaderNames[i]); %></TD>
          <% } %></TR><%
     int lineCount =1;
    while ( rsSearch.next() )
    {
     %><TR><%
     for ( int ii = 0; ii < mArFields.length; ii++ )
          {%><TD><%
           try
             {
             if (rsSearch.getString(mArFields[ii])
                    == null)
                  {
                  out.print ( " " );
                  }
               else if
               (mArFields[ii].equals("order_detail_id"))
                  {
                   out.print(lineCount );
                  }
               else
                  out.print( rsSearch.getString(
                                mArFields[ii]) );
```

```
                }
            catch (Exception ee)
                {
                    out.print(" ");
                }
                  %></TD><%
            } %></TR><%
             lineCount++;
        } %>
        </table>
       <%
       }
     catch (Exception e)
       {
       %>There was a problem with your insert.
          Please go back and check your values.<%
         e.printStackTrace();
       }
   }
   else
   {
   %><P>Search not found, please try again.</P>
   <%
   }
   %>
   <BR>
   <BR>
   Click <a href="SearchesWithJavaScript.jsp">here</a>
      for searches.
   </BODY>
   </HTML>
```

Figure 7.16 illustrates ShowOrderDetails JSP with data.

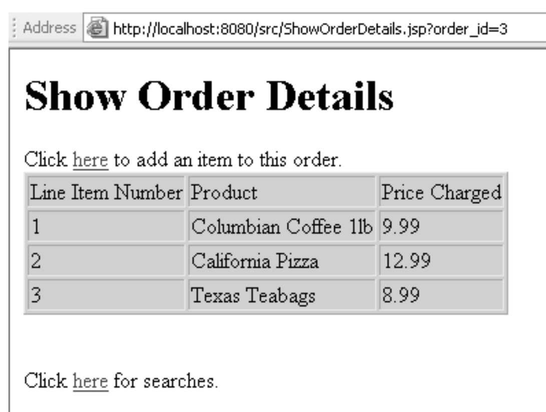


FIGURE 7.16
The Show Order Details page

The pages we looked at in this chapter show the basic prototype skeleton of most web applications. Certainly you could improve the design. You could add foreign keys to enhance the data integrity; you could add pages that would allow users to update and delete entries; and you could improve the navigation by separating headers and footers into included files, among other things. In Chapter 14, we'll focus on remedying the design elements that would hinder this prototype's scalability and reliability if it were deployed to a large user environment.

Summary

Using a database greatly enhances the flexibility of designing an application. Databases keep track of orders, accounts, and inventories, but they can also be used to record and retrieve configuration information, user settings, and security permissions. New developments in database storage allow the archiving of images, sounds, and even XML.

Connecting to a database and executing SQL statements that allow you to insert, update, delete, and view data are the building blocks for many prototypes and applications. JSP encourages code reuse, and you can save some time and effort by sharing connections across a user's session. Later, in Chapters 13 and 14, when we look at enterprise application development, you'll see that there can be a need for further balance on the server side so that a web application runs smoothly.

A fair amount of smart management of the Presentation layer can improve a user's experience with a database-driven application: JavaScript can check for required fields or submit simple searches using a hyperlink click; judicious use of arrays can generate different HTML table displays depending on the search criteria; and information identifying an order can be written right into the query string parameter values of a hyperlink on a page. Databases enhance the flexibility of applications, but JSP must work hard to present this flexibility to end users.

Enterprise demands for scalability and robustness will require additional programming concepts that were out of the current scope of complexity for this chapter; we'll cover them in Chapter 14.