Chapter 12

How to use JDBC to work with a database

Objectives

Applied

1. Develop data access classes that use JDBC to provide all of the methods that your servlets need to work with a database.
2. Develop a utility class that allows you to get a connection from a connection pool.
3. Develop servlets that use the methods of your data classes.

Objectives (continued)

Knowledge

1. Describe how a web application can use the DriverManager, Connection, Statement, PreparedStatement, and ResultSet classes to get data from a database.
2. Explain how prepared statements can improve the performance and security of database operations.
3. Describe the use of a ResultSetMetaData object.
4. Explain how connection pooling can improve the performance of a web application.
5. Describe O/R (object-relational) mapping.

The four types of JDBC database drivers

Type 1	A *JDBC-ODBC bridge driver* converts JDBC calls into ODBC calls that access the DBMS protocol.
Type 2	A *native protocol partly Java driver* converts JDBC calls into calls in the native DBMS protocol.
Type 3	A *net protocol all Java driver* converts JDBC calls into a net protocol that’s independent of any native DBMS protocol.
Type 4	A *native protocol all Java driver* converts JDBC calls into a native DBMS protocol.

How to make a database driver available to an application

- Before you can use a database driver, you must make it available to your application. To do this, use your IDE to add the JAR file for the driver to your application.
- To add the MySQL JDBC driver to a NetBeans project, right-click on the Libraries folder, select Add Library, and use the resulting dialog box to select the MySQL JDBC Driver library.
- To add any JDBC driver to a NetBeans project, right-click on the Libraries folder, select Add JAR/Folder, and select the JAR file for the driver.

Database URL syntax

```
jdbc:subprotocolName:databaseURL
```

How to connect to a MySQL database with automatic driver loading

```
try {
    String dbURL = "jdbc:mysql://localhost:3306/murach";
    String username = "root";
    String password = "sesame";
    Connection connection = DriverManager.getConnection(
        dbURL, username, password);
} catch(SQLException e) {
    for (Throwable t : e)
        t.printStackTrace();
}
```

How to connect to an Oracle database with automatic driver loading

```
    Connection connection = DriverManager.getConnection(
        "jdbc:oracle:thin@localhost/murach", "scott", "tiger");
```

How to explicitly load a database driver

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch(ClassNotFoundException e) {
    e.printStackTrace();
}
```

How to connect to a database

- Before you can get or modify the data in a database, you need to connect to it. To do that, use the getConnection method of the DriverManager class to return a Connection object.
- When you use the getConnection method of the DriverManager class, you must supply a URL for the database, a username, and a password. This method throws an SQLException.
- With JDBC 4.0, the SQLException class implements the Iterable interface. As a result, use an enhanced for statement to loop through any nested exceptions.
- With JDBC 4.0, the database driver is loaded automatically. This is known as *automatic driver loading*.
- Typically, you only need to connect to one database for an application. However, it's possible to load multiple database drivers and establish connections to multiple types of databases.

How to create a result set with 1 row and 1 column

```
Statement statement = connection.createStatement();
ResultSet userIDResult = statement.executeQuery(
    "SELECT UserID FROM User " +
    "WHERE Email = 'jsmith@gmail.com'");
```

How to create a result set with multiple columns and rows

```
Statement statement = connection.createStatement();
ResultSet products = statement.executeQuery(
    "SELECT * FROM Product ");
```

How to move the cursor to the first row

```
boolean userIDExists = userIDResult.next();
```

How to loop through a result set

```
while (products.next()) {
    // statements that process each row
}
```

ResultSet methods for forward-only, read-only result sets

Method	Description
`next()`	Moves the cursor to the next row in the result set.
`last()`	Moves the cursor to the last row in the result set.
`close()`	Releases the result set's resources.
`getRow()`	Returns an int value that identifies the current row of the result set.

How to return a result set and move the cursor through it

- To return a *result set*, use the createStatement method of a Connection object to create a Statement object. Use the executeQuery method of the Statement object to execute a SELECT statement that returns a ResultSet object.
- By default, the createStatement method creates a forward-only, read-only result set. You can only move the *cursor* through it from the first row to the last and you can't update it. This is appropriate for most web applications.
- When a result set is created, the cursor is positioned before the first row.
- Use the methods of the ResultSet object to move the cursor.
- To move the cursor to the next row, call the next method. If the row is valid, this method moves the cursor to the next row and returns a true value.

Methods of a ResultSet object that return data

Method	Description
`getXXX(int columnIndex)`	Returns data from the specified column number.
`getXXX(String columnName)`	Returns data from the specified column name.

Code that uses indexes to return columns

```
String code = products.getString(1);
String description = products.getString(2);
double price = products.getDouble(3);
```

Code that uses names to return columns

```
String code = products.getString("ProductCode");
String description = products.getString("ProductDescription");
double price = products.getDouble("ProductPrice");
```

Code that creates a Product object from the products result set

```
Product product = new Product(products.getString(1),
                              products.getString(2),
                              products.getDouble(3));
```

The getXXX methods

- The getXXX methods can be used to return all eight primitive types. For example, the getInt method returns the int type and the getLong method returns the long type.
- The getXXX methods can also be used to return strings, dates, and times. For example, the getString method returns any object of the String class, and the getDate, getTime, and getTimestamp methods return objects of the Date, Time, and Timestamp classes of the java.sql package.

How to use the executeUpdate method to…

Add a row

```
String query =
   "INSERT INTO Product (ProductCode, ProductDescription, ProductPrice) " +
   "VALUES ('" + product.getCode() + "', " +
           "'" + product.getDescription() + "', " +
           "'" + product.getPrice() + "')";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

Update a row

```
String query = "UPDATE Product SET " +
   "ProductCode = '" + product.getCode() + "', " +
   "ProductDescription = '" + product.getDescription() + "', " +
   "ProductPrice = '" + product.getPrice() + "' " +
   "WHERE ProductCode = '" + product.getCode() + "'";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

How to use the executeUpdate method to…(cont.)

Delete a row

```
String query = "DELETE FROM Product " +
               "WHERE ProductCode = '" + productCode + "'";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

How to insert, update, and delete data

- The executeUpdate method is an older method that works with most JDBC drivers. Some newer methods require less SQL code, but they may not work properly with all JDBC drivers.
- The executeUpdate method returns an int value that identifies the number of rows affected by the SQL statement.

Warning

- If you build an SQL statement from user input and use a method of the Statement object to execute that SQL statement, you may be susceptible to an SQL injection attack.
- An *SQL injection attack* allows a hacker to bypass authentication or execute SQL statements against your database that can read data, modify data, or delete data.
- To prevent most types of SQL injection attacks, use prepared statements.

How to use a prepared statement

To return a result set

```
String preparedSQL = "SELECT ProductCode, ProductDescription, "
                   + "       ProductPrice "
                   + "FROM Product WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(preparedSQL);
ps.setString(1, productCode);
ResultSet product = ps.executeQuery();
```

To modify a row

```
String preparedSQL = "UPDATE Product SET "
                   + "   ProductCode = ?, "
                   + "   ProductDescription = ?, "
                   + "   ProductPrice = ?"
                   + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(preparedSQL);
ps.setString(1, product.getCode());
ps.setString(2, product.getDescription());
ps.setDouble(3, product.getPrice());
ps.setString(4, product.getCode());
ps.executeUpdate();
```

How to use a prepared statement (continued)

To insert a row

```
String preparedQuery =
    "INSERT INTO Product "
    + "(ProductCode, ProductDescription, ProductPrice) "
    + "VALUES "
    + "(?, ?, ?)";
PreparedStatement ps = connection.prepareStatement(preparedQuery);
ps.setString(1, product.getCode());
ps.setString(2, product.getDescription());
ps.setDouble(3, product.getPrice());
ps.executeUpdate();
```

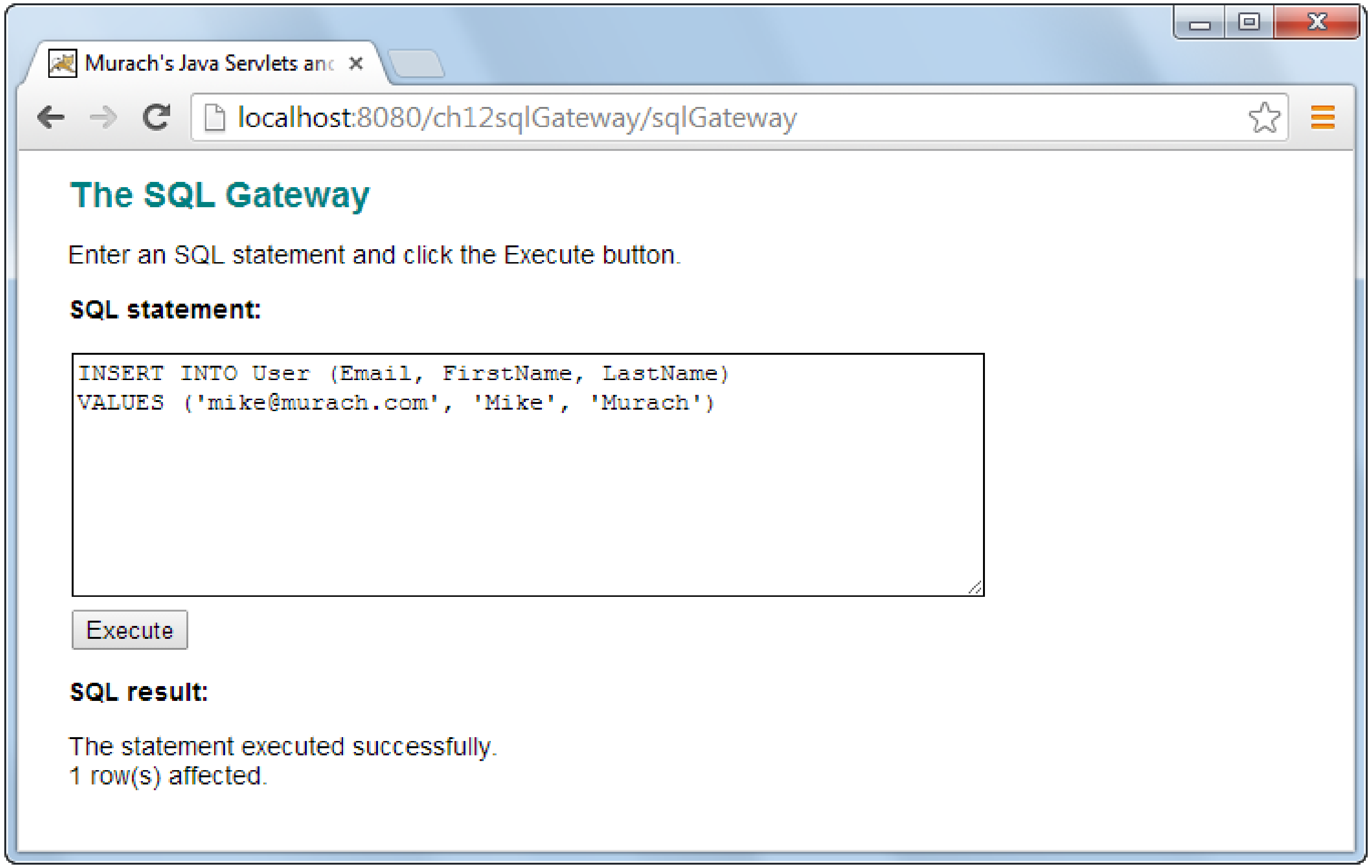
To delete a row

```
String preparedQuery = "DELETE FROM Product "
                     + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(preparedQuery);
ps.setString(1, productCode);
ps.executeUpdate();
```

How to work with prepared statements

- When you use *prepared statements* in your Java programs, the database server only has to check the syntax and prepare an execution plan once for each SQL statement. This improves the efficiency of the database operations and prevents most types of SQL injection attacks.
- To specify a parameter for a prepared statement, type a question mark (?) in the SQL statement.
- To supply values for the parameters in a prepared statement, use the set methods of the PreparedStatement interface.
- To execute a SELECT statement, use the executeQuery method. To execute an INSERT , UPDATE, or DELETE statement, use the executeUpdate method.

The SQL Gateway application after executing an INSERT statement


Murach's Java Servlets and
localhost:8080/ch12sqlGateway/sqlGateway

The SQL Gateway

Enter an SQL statement and click the Execute button.

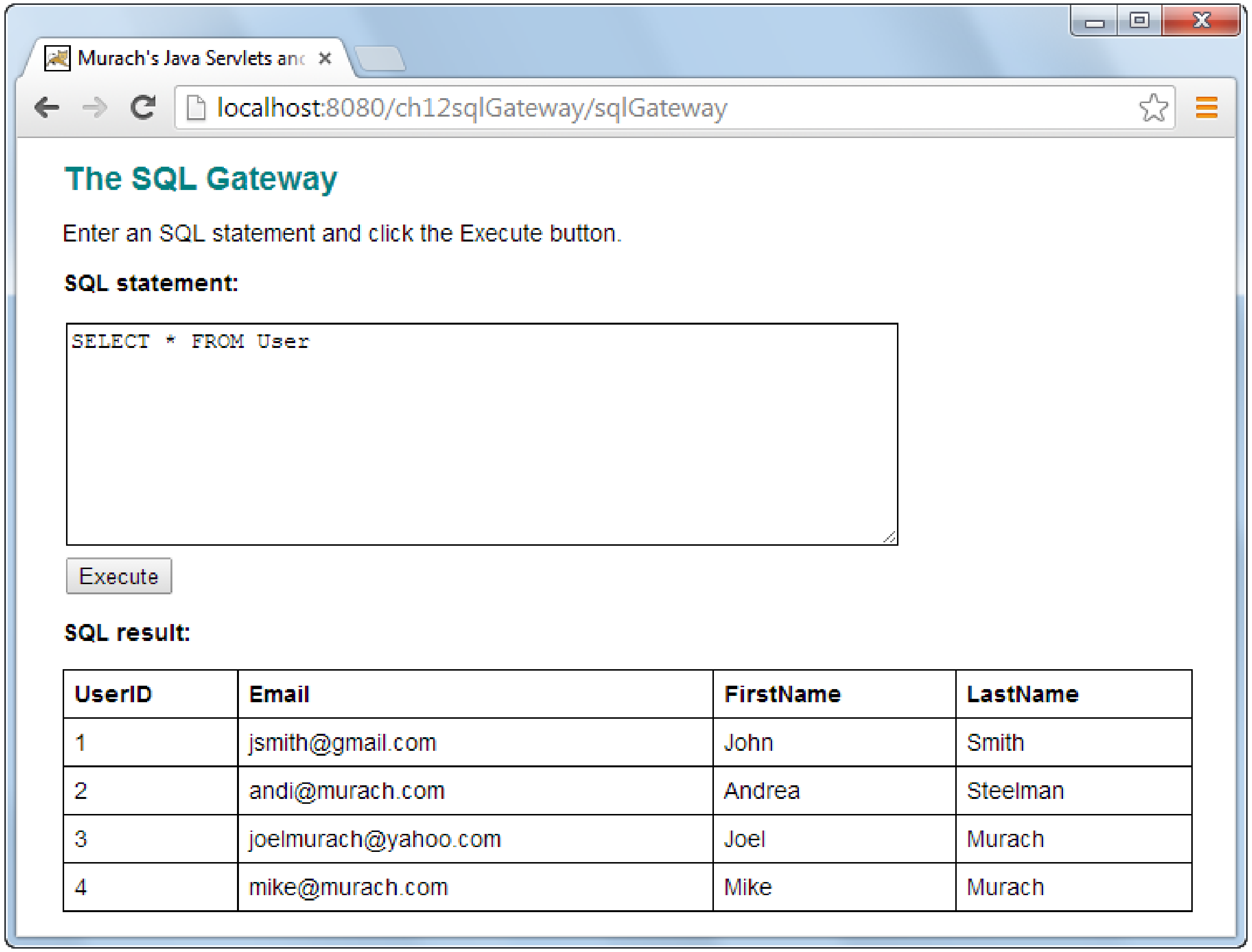
SQL statement:

```
INSERT INTO User (Email, FirstName, LastName)
VALUES ('mike@murach.com', 'Mike', 'Murach')
```

SQL result:

The statement executed successfully.
1 row(s) affected.

The SQL Gateway application after executing a SELECT statement


Murach's Java Servlets anc
localhost:8080/ch12sqlGateway/sqlGateway

The SQL Gateway

Enter an SQL statement and click the Execute button.

SQL statement:

```
SELECT * FROM User
```

SQL result:

UserID	Email	FirstName	LastName
1	jsmith@gmail.com	John	Smith
2	andi@murach.com	Andrea	Steelman
3	joelmurach@yahoo.com	Joel	Murach
4	mike@murach.com	Mike	Murach

The code for the JSP

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Murach's Java Servlets and JSP</title>
    <link rel="stylesheet" href="styles/main.css" type="text/css"/>
</head>
<body>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:if test="${sqlStatement == null}">
    <c:set var="sqlStatement" value="select * from User" />
</c:if>

<h1>The SQL Gateway</h1>
<p>Enter an SQL statement and click the Execute button.</p>
```

The code for the JSP (continued)

```
<p><b>SQL statement:</b></p>
<form action="sqlGateway" method="post">
   <textarea name="sqlStatement" cols="60"
rows="8">${sqlStatement}</textarea>
   <input type="submit" value="Execute">
</form>

<p><b>SQL result:</b></p>
${sqlResult}

</body>
</html>
```

The SQLGatewayServlet class

```
package murach.sql;

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

import java.sql.*;

public class SqlGatewayServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        String sqlStatement = request.getParameter("sqlStatement");
        String sqlResult = "";
        try {
            // load the driver
            Class.forName("com.mysql.jdbc.Driver");
```

The SQLGatewayServlet class (continued)

```
// get a connection
            String dbURL = "jdbc:mysql://localhost:3306/murach";
            String username = "murach_user";
            String password = "sesame";
            Connection connection = DriverManager.getConnection(
                    dbURL, username, password);

            // create a statement
            Statement statement = connection.createStatement();

            // parse the SQL string
            sqlStatement = sqlStatement.trim();
            if (sqlStatement.length() >= 6) {
                String sqlType = sqlStatement.substring(0, 6);

                if (sqlType.equalsIgnoreCase("select")) {
                    // create the HTML for the result set
                    ResultSet resultSet
                            = statement.executeQuery(sqlStatement);
                    sqlResult = SQLUtil.getHtmlTable(resultSet);
                    resultSet.close();
```

The SQLGatewayServlet class (continued)

```
        } else {
            int i = statement.executeUpdate(sqlStatement);
            if (i == 0) { // a DDL statement
                sqlResult =
                  "<p>The statement executed successfully.</p>";
            } else { // an INSERT, UPDATE, or DELETE statement
                sqlResult =
                  "<p>The statement executed successfully.<br>"
                        + i + " row(s) affected.</p>";
            }
        }
    }
    statement.close();
    connection.close();
} catch (ClassNotFoundException e) {
    sqlResult = "<p>Error loading the database driver: <br>"
            + e.getMessage() + "</p>";
} catch (SQLException e) {
    sqlResult = "<p>Error executing the SQL statement: <br>"
            + e.getMessage() + "</p>";
}
```

The SQLGatewayServlet class (continued)

```
        HttpSession session = request.getSession();
        session.setAttribute("sqlResult", sqlResult);
        session.setAttribute("sqlStatement", sqlStatement);

        String url = "/index.jsp";
        getServletContext()
                .getRequestDispatcher(url)
                .forward(request, response);
    }
}
```

Note

- The web.xml file for this application maps the SQLGatewayServlet class to the /sqlGateway URL.

The SQLUtil class

```
package murach.sql;

import java.sql.*;

public class SQLUtil {

    public static String getHtmlTable(ResultSet results)
            throws SQLException {

        StringBuilder htmlTable = new StringBuilder();
        ResultSetMetaData metaData = results.getMetaData();
        int columnCount = metaData.getColumnCount();

        htmlTable.append("<table>");
```

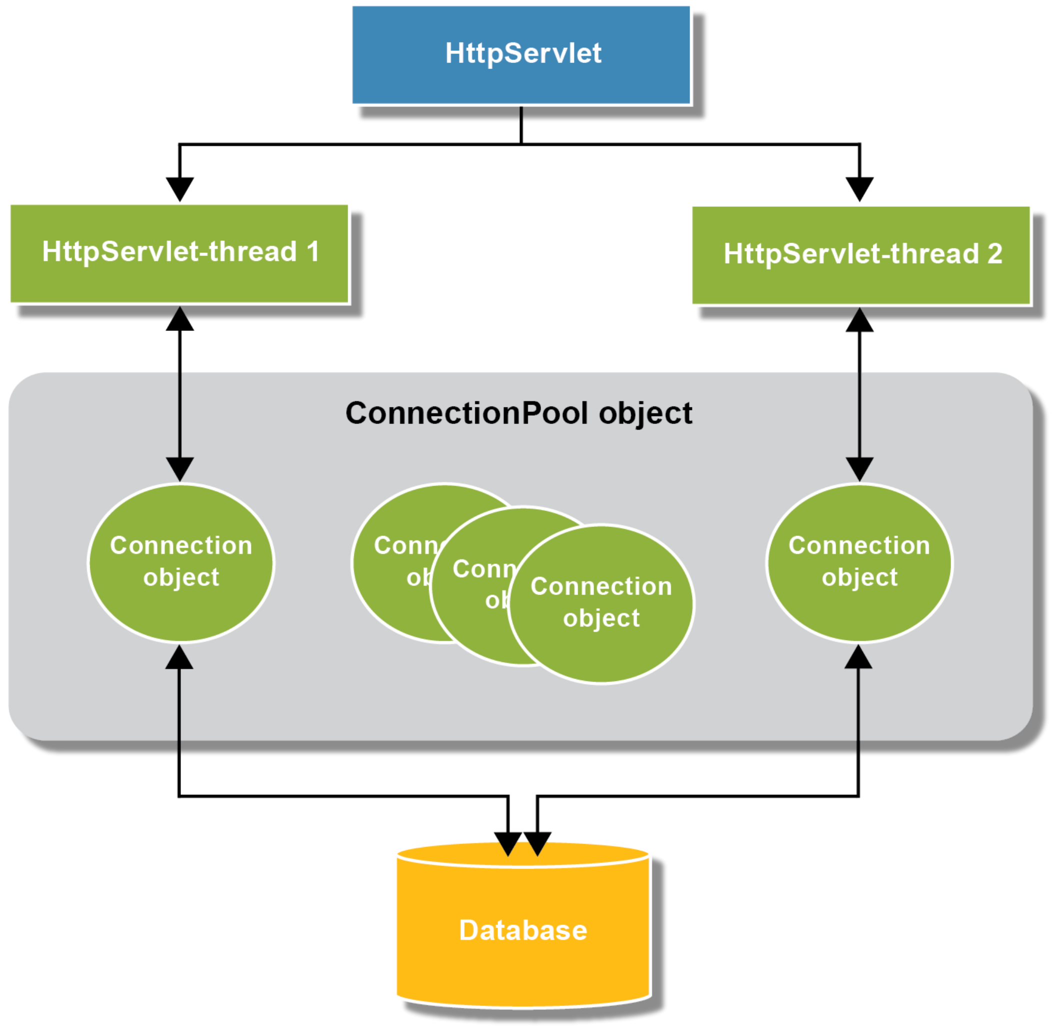
The SQLUtil class (continued)

```
            // add header row
            htmlTable.append("<tr>");
            for (int i = 1; i <= columnCount; i++) {
                htmlTable.append("<th>");
                htmlTable.append(metaData.getColumnName(i));
                htmlTable.append("</th>");
            }
            htmlTable.append("</tr>");

            // add all other rows
            while (results.next()) {
                htmlTable.append("<tr>");
                for (int i = 1; i <= columnCount; i++) {
                    htmlTable.append("<td>");
                    htmlTable.append(results.getString(i));
                    htmlTable.append("</td>");
                }
                htmlTable.append("</tr>");
            }

            htmlTable.append("</table>");
            return htmlTable.toString();
        }
    }
```

The SQLUtil class (continued)

- The getHtmlTable method in this class accepts a ResultSet object and returns a String object that contains the HTML code for the result set so it can be displayed by a browser.
- The getMetaData method of a ResultSet object returns a ResultSetMetaData object.
- The getColumnCount method of a ResultSetMetaData object returns the number of columns in the result set.
- The getColumnName method of a ResultSetMetaData object returns the name of a column in the result set.

How connection pooling works



How connection pooling works (continued)

- When *database connection pooling* (*DBCP*) is used, a limited number of connections are opened for a database and are shared by the users who connect to the database. This improves the performance of the database operations.
- When one of the threads of a servlet needs to perform a database operation, the thread gets a Connection object from the ConnectionPool object and uses that Connection object to do the operation.
- When a thread finishes using a Connection object, it returns the object to the pool.
- Before you can use the connection pool, you must make it available to your application. Use your IDE to add the JAR file for the connection pool to your application.

A context.xml file that configures a connection pool

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/ch12email">

    <Resource name="jdbc/murach" auth="Container"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/murach?autoReconnect=true"
        username="murach_user" password="sesame"
        maxActive="100" maxIdle="30" maxWait="10000"
        logAbandoned="true" removeAbandoned="true"
        removeAbandonedTimeout="60" type="javax.sql.DataSource" />

</Context>
```

A class that defines a connection pool

```
package murach.data;

import java.sql.*;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class ConnectionPool {

    private static ConnectionPool pool = null;
    private static DataSource dataSource = null;

    private ConnectionPool() {
        try {
            InitialContext ic = new InitialContext();
            dataSource = (DataSource)
                ic.lookup("java:/comp/env/jdbc/murach");
        } catch (NamingException e) {
            System.out.println(e);
        }
    }
```

A class that defines a connection pool (continued)

```
    public static synchronized ConnectionPool getInstance() {
        if (pool == null) {
            pool = new ConnectionPool();
        }
        return pool;
    }

    public Connection getConnection() {
        try {
            return dataSource.getConnection();
        } catch (SQLException e) {
            System.out.println(e);
            return null;
        }
    }

    public void freeConnection(Connection c) {
        try {
            c.close();
        } catch (SQLException e) {
            System.out.println(e);
        }
    }
}
```

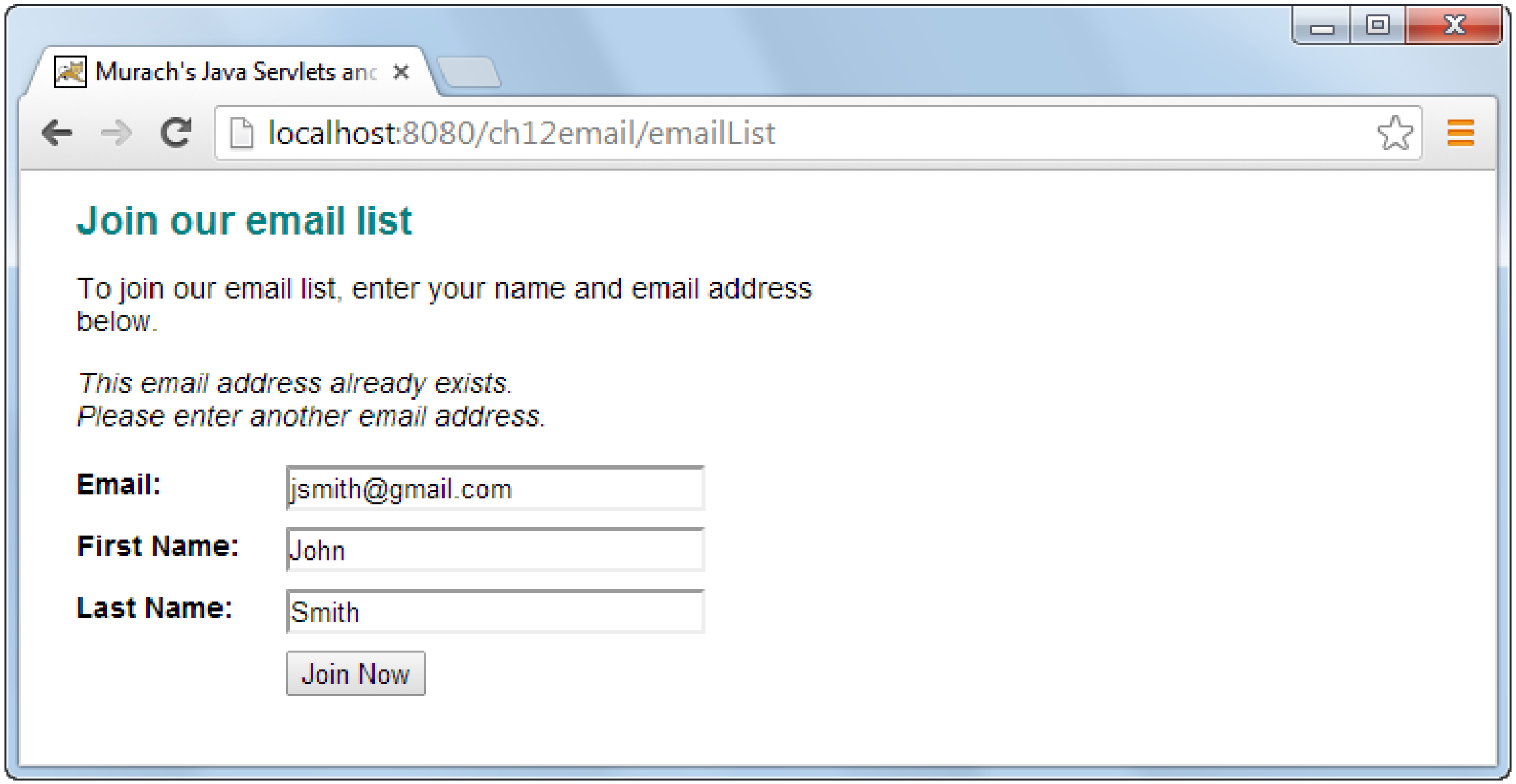
Code that uses the connection pool

```
ConnectionPool pool = ConnectionPool.getInstance();
Connection connection = pool.getConnection();

// code that uses the connection to work with the database

pool.freeConnection(connection);
```

How to implement and use a connection pool

- With Tomcat 6 and later, you can use the context.xml file to configure connection pooling for an application.
- The ConnectionPool class provides the getConnection and freeConnection methods that make it easy for programmers to get connections and to return connections to the connection pool.

An Email List application that displays an error message


Murach's Java Servlets anc
localhost:8080/ch12email/emailList

Join our email list

To join our email list, enter your name and email address below.

This email address already exists.
Please enter another email address.

Email:	jsmith@gmail.com
First Name:	John
Last Name:	Smith

The code for the JSP

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Murach's Java Servlets and JSP</title>
    <link rel="stylesheet" href="styles/main.css" type="text/css"/>
</head>
<body>
    <h1>Join our email list</h1>
    <p>To join our email list, enter your name and
       email address below.</p>
    <p><i>${message}</i></p>
    <form action="emailList" method="post">
        <input type="hidden" name="action" value="add">

        <label class="pad_top">Email:</label>
        <input type="email" name="email" value="${user.email}"
               required><br>
```

The code for the JSP (continued)

```
        <label class="pad_top">First Name:</label>
        <input type="text" name="firstName" value="${user.firstName}"
               required><br>

        <label class="pad_top">Last Name:</label>
        <input type="text" name="lastName" value="${user.lastName}"
               required><br>

        <label>&nbsp;</label>
        <input type="submit" value="Join Now" class="margin_left">
    </form>
</body>
</html>
```

The code for the servlet

```
package murach.email;

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

import murach.business.User;
import murach.data.UserDB;

public class EmailListServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        String url = "/index.html";

        // get current action
        String action = request.getParameter("action");
        if (action == null) {
            action = "join";  // default action
        }
```

The code for the servlet (continued)

```
        // perform action and set URL to appropriate page
         if (action.equals("join")) {
             url = "/index.jsp";    // the "join" page
         }
         else if (action.equals("add")) {
             // get parameters from the request
             String firstName = request.getParameter("firstName");
             String lastName = request.getParameter("lastName");
             String email = request.getParameter("email");

             // store data in User object
             User user = new User(firstName, lastName, email);
```

The code for the servlet (continued)

```
            // validate the parameters
            if (UserDB.emailExists(user.getEmail())) {
                message = "This email address already exists.<br>" +
                          "Please enter another email address.";
                url = "/index.jsp";
            }
            else {
                message = "";
                url = "/thanks.jsp";
                UserDB.insert(user);
            }
            request.setAttribute("user", user);
            request.setAttribute("message", message);
        }
        getServletContext()
                .getRequestDispatcher(url)
                .forward(request, response);
    }
}
```

The UserDB class

```
package murach.data;

import java.sql.*;

import murach.business.User;

public class UserDB {

    public static int insert(User user) {
        ConnectionPool pool = ConnectionPool.getInstance();
        Connection connection = pool.getConnection();
        PreparedStatement ps = null;

        String query
                = "INSERT INTO User (Email, FirstName, LastName) "
                + "VALUES (?, ?, ?)";
```

The UserDB class (continued)

```
        try {
            ps = connection.prepareStatement(query);
            ps.setString(1, user.getEmail());
            ps.setString(2, user.getFirstName());
            ps.setString(3, user.getLastName());
            return ps.executeUpdate();
        } catch (SQLException e) {
            System.out.println(e);
            return 0;
        } finally {
            DBUtil.closePreparedStatement(ps);
            pool.freeConnection(connection);
        }
    }
```

The UserDB class (continued)

```
public static int update(User user) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    PreparedStatement ps = null;

    String query = "UPDATE User SET "
            + "FirstName = ?, "
            + "LastName = ? "
            + "WHERE Email = ?";
    try {
        ps = connection.prepareStatement(query);
        ps.setString(1, user.getFirstName());
        ps.setString(2, user.getLastName());
        ps.setString(3, user.getEmail());

        return ps.executeUpdate();
    } catch (SQLException e) {
        System.out.println(e);
        return 0;
    } finally {
        DBUtil.closePreparedStatement(ps);
        pool.freeConnection(connection);
    }
}
```

The UserDB class (continued)

```
public static int delete(User user) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    PreparedStatement ps = null;

    String query = "DELETE FROM User "
            + "WHERE Email = ?";
    try {
        ps = connection.prepareStatement(query);
        ps.setString(1, user.getEmail());

        return ps.executeUpdate();
    } catch (SQLException e) {
        System.out.println(e);
        return 0;
    } finally {
        DBUtil.closePreparedStatement(ps);
        pool.freeConnection(connection);
    }
}
```

The UserDB class (continued)

```
public static boolean emailExists(String email) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    PreparedStatement ps = null;
    ResultSet rs = null;

    String query = "SELECT Email FROM User "
            + "WHERE Email = ?";
    try {
        ps = connection.prepareStatement(query);
        ps.setString(1, email);
        rs = ps.executeQuery();
        return rs.next();
    } catch (SQLException e) {
        System.out.println(e);
        return false;
    } finally {
        DBUtil.closeResultSet(rs);
        DBUtil.closePreparedStatement(ps);
        pool.freeConnection(connection);
    }
}
```

The UserDB class (continued)

```
public static User selectUser(String email) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    PreparedStatement ps = null;
    ResultSet rs = null;

    String query = "SELECT * FROM User "
            + "WHERE Email = ?";
    try {
        ps = connection.prepareStatement(query);
        ps.setString(1, email);
        rs = ps.executeQuery();
        User user = null;
        if (rs.next()) {
            user = new User();
            user.setFirstName(rs.getString("FirstName"));
            user.setLastName(rs.getString("LastName"));
            user.setEmail(rs.getString("Email"));
        }
        return user;
```

The UserDB class (continued)

```
} catch (SQLException e) {
            System.out.println(e);
            return null;
        } finally {
            DBUtil.closeResultSet(rs);
            DBUtil.closePreparedStatement(ps);
            pool.freeConnection(connection);
        }
    }
}
```

The DBUtil class

```
package murach.data;

import java.sql.*;

public class DBUtil {

    public static void closeStatement(Statement s) {
        try {
            if (s != null) {
                s.close();
            }
        } catch (SQLException e) {
            System.out.println(e);
        }
    }
```

The DBUtil class (continued)

```
    public static void closePreparedStatement(Statement ps) {
        try {
            if (ps != null) {
                ps.close();
            }
        } catch (SQLException e) {
            System.out.println(e);
        }
    }

    public static void closeResultSet(ResultSet rs) {
        try {
            if (rs != null) {
                rs.close();
            }
        } catch (SQLException e) {
            System.out.println(e);
        }
    }
}
```