**Database Application Development: JDBC**

**JDBCCheckup.java, continued**

Create a statement using the Connection object:

```java
Statement stmt = conn.createStatement();
```

Do DB actions using Statement –

```java
stmt.execute("drop table welcome");
stmt.execute("create table welcome(msg char(20))");
```

```java
ResultSet rset = stmt.executeQuery("select * from welcome");
while (rset.next())
    System.out.println(rset.getString(1));
```

```java
conn.close(); // Display row, close connection (and its associated objects) important to free up TCP/IP connection into the DB
```

**JdbcCheckup.java: Exceptions**

- The previous slide ignored Exceptions
- JDBC coding involves messy Exception handling
- We need to use finally as well as try and catch
- Need to review Exceptions

**Java Exceptions**

- We looked at this tutorial: [http://www.tutorialspoint.com/java/java_exceptions.htm](http://www.tutorialspoint.com/java/java_exceptions.htm)
- Another tutorial: [Java Tutorial: Exceptions](http://www.tutorialspoint.com/java/java_exceptions.htm)
- Skip coverage on The try-with-resources Statement

**Analyze JdbcCheckup for Exceptions**

```java
static void tryWelcomeExperiment(Connection conn) throws SQLException {
    // Create a statement
    Statement stmt = conn.createStatement();
    ResultSet rset = null;
    try {
        <various DB actions>
    } finally { // Note: try without catch
        stmt.close(); // clean up statement resources
    }
}
```

- This method throws any exception occurring during the DB actions
- That’s good: in Java, a method is expected to report problems via an exception
Special handling for drop table action

We treat drop table specially to allow it to fail as it will the very first time we run this program.

```java
try {
    stmt.executeUpdate("drop table welcome");
} catch (SQLException e) {
    // assume not there yet, so OK to continue
}
```

- You should always double-check any catch clause that does nothing. It's a "code smell".
- Remember that a method is supposed to throw for any failure. This is arguably not a failure.
- The other actions: create table, etc., throw from the method on any DB error.

Handling getConnection, etc.

- Note that `conn.close()` can throw, so need to put it in an inner finally to maintain control. Here's a pattern:

```java
try {
    conn = DriverManager.getConnection(connStr, user, password);
    <do-db-stuff> // this can throw SQLException
} finally {
    conn.close();
}
```

- You should always double-check any catch clause that does nothing. It's a "code smell".
- Remember that a method is supposed to throw for any failure. This is arguably not a failure.
- The other actions: create table, etc., throw from the method on any DB error.

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.

How to connect to a MySQL database with automatic driver loading

```java
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

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

Only needed if you're using older JDBC (pre-version-4)

---

**From Murach**

**JSP/Servlets: How to use JDBC to work with a database**

**Database URL syntax**

`jdbc:subprotocolName:databaseURL`

**How to connect to a MySQL database with automatic driver loading**

```java
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**

```java
try {
    Connection connection = DriverManager.getConnection("jdbc:oracle:thin:localhost/murach", "scott", "tiger");
} catch(SQLException e) {
    for (Throwable t : e) 
        t.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 database driver is loaded automatically. This is known as automatic driver loading.
- 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.
- 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.

Homework 4 Notes

- Try out your queries in SQLPlus first before coding them into Java.
- Don’t be afraid to use complex queries in Java/JDBC...
- Example: Find pilot ids (employee ids) of pilots certified for all aircraft of cruising range under 2000 miles except aircraft 16 (aircraft 16 has no certified pilots)
- This is a division query, but there’s no need for division in homework 4.

Complex query in JDBC

```java
static void tryComplexQuery(Connection conn) throws SQLException {
    Statement stmt = conn.createStatement();
    ResultSet rset = null;
    try {
        String q = "SELECT e.eid, e.salary FROM employees e WHERE " +
                    "NOT EXISTS (SELECT a.aid FROM aircraft a WHERE " +
                    "a.cruisingrange < 2000 and a.aid <> 16) " +
                    "MINUS (SELECT c.aid FROM certified c WHERE c.eid = e.eid)";
        rset = stmt.executeQuery(q);
        while (rset.next()) {
            System.out.println("eid = " + rset.getInt("eid") + "n");
            System.out.println("salary = " + rset.getInt("salary") + "n");
        }
    }
    finally {
        stmt.close(); // clean up statement resources, incl. rset
    }
}
```

Notes on example

- It’s Oracle-only because of the MINUS
- Note how the Statement variable is declared before the try { ...
- The try block localizes variables declared within it, like any other block in Java.
- As with other examples of DB-query-related methods, there is no catch here, so any exception gets thrown up to the caller (or its caller, etc.), on the theory that the caller knows better what to do with the failure.

Warning: SQL Injection Attacks

- 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.
The SQL Gateway application

- Clear case of SQL injection (invited!) that exposes the database to the world (if this webapp is on the Internet)
- Don’t be tempted to do this!

The safe way: How to use a prepared statement

To return a result set

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

To modify a row

```java
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();
```

An Email List application

This application may or may not have SQL injection problems

- This user input drives an insert into a userdb table.
- If it uses PreparedStatements to process user input, it’s safe...


- Basic idea: insert with a subquery can retrieve something (admin password for example), put it in the user table as "email"
- Later, have app print out user "email", see retrieved info
- Email entry: ' || (select max(password) from admins where user='root')||'   (note single quotes at start and end entered by user)
- Plugged into: insert into users values (' ... || (select ... || ... ');
- Yields: insert into users values ('xxx' ,''||(select ...)||'');
- Last value: nullstring||(select ...)||nullstring, equals select result
- Ends up storing admin password as user's "email"
- Note this is not using a PreparedStatement.

How can a malicious insert break into a database?

- My break in:
  Username ' or 'a' = 'a
  Password ' or 'a' = 'a
  Success login!

- This entry makes the SQL into:
  select count(*) from userpass where username=' ... || 'a' = 'a' and password=' ... || 'a' = 'a'

- The underlined parts were plugged in as uid and pwd
- This where clause evaluates to true for every row!
- Note this is not using a PreparedStatement.
- With use of a PreparedStatement, this login form can be made secure.

Admin login to a web app: depending on its implementation, this may or may not be secure...

```java
ResultSet set = stmt.executeQuery(" select count(*) from " + ADMIN_TABLE + " where username = ' ... || uid + ' ... " // username entered and password = ' ... || pwd + ' ... " // password entered //If count>0, allow access
This can be broken into easily!
```

How to break into this site as admin

- My break in:
  Username ' or 'a' = 'a
  Password ' or 'a' = 'a
  Success login!

- This entry makes the SQL into:
  select count(*) from userpass where username=' ... || 'a' = 'a' and password=' ... || 'a' = 'a'

- The underlined parts were plugged in as uid and pwd
- This where clause evaluates to true for every row!
- Note this is not using a PreparedStatement.
- With use of a PreparedStatement, this login form can be made secure.
How to use a prepared statement (continued)

To insert a row

```java
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

```java
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.

An Email List application -- now look at implementation

![Email List application implementation](image)

The top-level Java code in EmailList.java, collecting user input and calling UserDB to access the DB

```java
firstName = readEntry(in, "Enter firstName: ");
lastName = readEntry(in, "Enter lastName: ");
email = readEntry(in, "Enter email: ");
// Store data in new User object
User user = new User(firstName, lastName, email);
try {
    // Validate the parameters
    if (UserDB.emailExists(user.getEmail())) {
        message = "This email address already exists. Please enter another email address.";
    } else {
        message = "";
    }
    UserDB.insert(user);
} catch (SQLException e) { // Send back error to user }
```

The User class, a "Java Bean"

```java
import java.io.Serializable;

public class User implements Serializable {
    private String firstName;
    private String lastName;
    private String email;

    public User() {
        firstName = "";
        lastName = "";
        email = "";
    }

    public User(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
```

Actually, let's look at the corresponding line-oriented input/output program, like JdbcCheckup.java -- it's part of Homework 4

Files in emaillist subdirectory of class home page directory: from the README there:

- createuserdb.sql: create needed userdb table
- dropuserdb.sql: drop userdb table
- EmailList.java: top-level program
- User.java: Java bean for user, holds firstname, lastname, email
- UserDB.java: Database access code

To build: javac *.java
To run: java -cp ojdbc6.jar: EmailList

Hw4 #5: Write ShowUser.java
The User class (continued)

```java
public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
```

The UserDB class

```java
import java.sql.*;

public class UserDB {
    private static Connection connection;

    public static void setConnection(Connection conn) {
        connection = conn;
    }

    public static int insert(User user) throws SQLException {
        PreparedStatement ps = null;
        String query = "INSERT INTO User (Email, FirstName, LastName) VALUES (?, ?, ?)";
        try {
            ps = connection.prepareStatement(query);
            ps.setString(1, user.getEmail());
            ps.setString(2, user.getFirstName());
            ps.setString(3, user.getLastName());
            return ps.executeUpdate();
        } finally {
            closePreparedStatement(ps);
        }
    }

    public static int delete(User user) throws SQLException {
        PreparedStatement ps = null;
        String query = "DELETE FROM User WHERE Email = ?";
        try {
            ps = connection.prepareStatement(query);
            ps.setString(1, user.getEmail());
            return ps.executeUpdate();
        } finally {
            closePreparedStatement(ps);
        }
    }

    public static boolean emailExists(String email) throws SQLException {
        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();
        } finally {
            closeResultSet(rs); closePreparedStatement(ps);
        }
    }

    public static User selectUser(String email) throws SQLException {
        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"));
            }
        } finally {
            closePreparedStatement(ps);
        }
        return user;
    }
}
```

Note: the top-level code needs to get a Connection and call UserDB.setConnection(Conn)
The UserDB class (continued)

```java
    } finally { 
            closeResultSet(rs); 
            closePreparedStatement(ps); 
        } 
    } 
    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); 
        } 
    } 
```