Introduction to Java Database Connectivity API

CSCE 156 - Introduction to Computer Science II

Christopher M. Bourke
cbourke@cse.unl.edu

Your Database

- CSE hosts your database (MySQL)
- Database name: same as your login
- Changing your password: visit http://ponca.unl.edu
- Class database (may be used for examples, assignments): cse156 (password: cse156student)
- MySQL commandline usage:
  - mysql -u username -p dbname
- Command line redirection (careful!):
  - mysql -u username -p dbname < commands.sql
- mysqldump (database backup)

Java Database Connectivity API

Java Database Connectivity (JDBC)

- General API (using interfaces) for Java client code to connect/interact with a database
- Database providers (IBM, Oracle, etc.) provide drivers
- Driver: specific implementation of the API for interacting with a particular database
- Support for
  - Statement
  - PreparedStatement
  - CallableStatement (stored procedures)
- Common Java data types (Integer, Double, java.sql.Date)

JDBC: basic step-by-step

1. Load the database JDBC driver
   - Note: your particular driver (.jar file) must be in the class or build path of your project
2. Make a connection to the database
3. Formulate your query(ies) & prepare your statement (set parameters)
4. Execute your query
5. If it's a SELECT query:
   5.1 Get your result set
   5.2 Process your results
6. Clean up your resources (close resources, close connection)

JDBC

Reflectively loading a driver

- For portability, applications written toward JDBC API, not a particular driver
- Driver needs to be loaded at run time through reflection
- Could be made configurable or delegated by some controller

Java provides connectivity through java.sql.Connection:

```java
try {
    String url = "jdbc:mysql://cse.unl.edu/
database_name";
    String username = "cse156student";
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(url, username,
password);
    } catch (SQLException sqle) {
        ...
    }
} catch (ClassNotFoundException e) {
    ...
} catch (InstantiationException e) {
    ...
} catch (IllegalAccessException e) {
    ...
} ...
```
JDBC
Transactions

- By default, all queries are auto-commit
- To change this, use `conn.setAutoCommit(false)`
- No changes committed until `conn.commit()` is called
  - Implicitly: new transaction after each commit
- Able to explicitly rollback using `conn.rollback()`
- Some drivers may also support `conn.setReadOnly(true)`

JDBC
Querying – Prepared Statement

- Always good to use `PreparedStatement`
- Parameters denoted by `?`, indexed by `1..n`
- Can be reused (parameters reset and required)
- Parameters are safe!

```java
String query = "SELECT ... FROM user WHERE nuid = ?";
PreparedStatement ps = null;
try {
    ps = conn.prepareStatement(query);
    ps.setString(1, "35140602");
    catch (SQLException sqle) {
        ...
    }
}
```

JDBC
Querying – Updates

- Always use a prepared statement!
- Same syntax holds for INSERT statements

```java
String query = "UPDATE user SET email = ?, last_updated = ? WHERE nuid = ?";
PreparedStatement ps = null;
try {
    ps = conn.prepareStatement(query);
    ps.setString(1, "cmbourne@gmail.com");
    ps.setString(2, "2011-01-01 00:00:01");
    ps.setString(3, "35140602");
    ps.executeUpdate();
    catch (SQLException sqle) {
        ...
    }
}
```

JDBC
Querying – Result Sets

- `executeQuery()` is for read-only (select statements)
- Select statements return `results`: columns and rows
- Results are encapsulated in a Java `ResultSet` object
  - Initially a result set "points" just before the first row
  - Iterating through a `ResultSet`: `rs.next()`
- Returns a boolean: true if the iteration was successful, false otherwise
  - If successful, the "current" result row is now pointed to
- Columns can be referenced by String (alias) or index
- Standard getters provide functionality to get-and-cast columns

```java
ResultSet rs = null;
try {
    rs = ps.executeQuery();
    while (rs.next()) {
        Integer nuid = rs.getInt("nuid")
        String firstName = rs.getString("first_name");
    }
} catch (SQLException sqle) {
    ...
}
```

JDBC
Good Practices – Rethrow Exceptions

- Most methods explicitly throw `SQLException`
- Occurs with DB errors or program bugs
- Little can be done either way
- Good to catch, log and rethrow

```java
... catch (SQLException sqle) {
            System.out.println("SQLException: ");
            sqle.printStackTrace();
            throw new RuntimeException(sqle);
        }
```
JDBC
Cleaning Up
▶ Objects hold onto valuable external resources
▶ Network traffic (keep alive), limited connection pool, etc.
▶ Best practice to release resources as soon as they are no longer needed: close() method

```java
try {
    if(rs != null && !rs.isClosed())
        rs.close();
    if(ps != null && !ps.isClosed())
        ps.close();
    if(conn != null && !conn.isClosed())
        conn.close();
} catch (SQLException e) {
    ...
}
```

Good Practice Tip 1
ALWAYS use Prepared Statements

When available, in any framework or language, always use prepared statements
▶ Safer
▶ Better for batch queries
▶ Myth: no performance hit
▶ Protects against injection attacks
▶ Using just one method: more uniform, less of a chance of a mistake
▶ Unfortunately: some frameworks support named parameters, not JDBC

Injection Attack
Example
▶ Say we pull a string value from a web form (lastName)
▶ Not using a prepared statement:
  String query = "SELECT primary_email FROM user WHERE last_name = ‘" + lastName + "’;"
▶ Without scrubbing the input, say a user enters:
  a’;DROP TABLE user;
▶ Actual query run:
  SELECT primary_email FROM user WHERE last = 'a';DROP TABLE users;
▶ Another example: input is ‘’ or ‘1’='1"
▶ Actual query:
  SELECT primary_email FROM user WHERE last_name = ' OR '1'='1' ;

Good Practice Tip 2
Enumerate fields in SELECT statements

▶ Using SELECT * ... grabs all fields even if you don’t use them
▶ Be intentional about what you want/need, only the minimal set
▶ Allows the database to optimize, reduces network traffic
▶ Protects against table changes
▶ Use aliasing (first_name AS firstName) on all fields to reduce affects of changes to field names
<table>
<thead>
<tr>
<th>Additional Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Issues</td>
</tr>
<tr>
<td>- Where/how should database passwords be stored?</td>
</tr>
<tr>
<td>- Good security policy: assume an attacker has your password &amp; take the necessary precautions (secure the server and network)</td>
</tr>
<tr>
<td>- Do not store sensitive data unencrypted</td>
</tr>
<tr>
<td>Efficiency Issues</td>
</tr>
<tr>
<td>- Repeat: close your resources</td>
</tr>
<tr>
<td>- Connection Pools</td>
</tr>
<tr>
<td>- Good normalization, design, &amp; practice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>- MySQL Community Server (<a href="http://www.mysql.com/downloads/">http://www.mysql.com/downloads/</a>)</td>
</tr>
<tr>
<td>- MySQL Workbench – a MySQL GUI (<a href="http://wb.mysql.com/">http://wb.mysql.com/</a>)</td>
</tr>
<tr>
<td>- Connector/J – MySQL JDBC connector (<a href="http://www.mysql.com/downloads/connector/j/">http://www.mysql.com/downloads/connector/j/</a>)</td>
</tr>
<tr>
<td>- Stanford’s <em>Introduction to Databases</em> free online course: <a href="http://db-class.com/">http://db-class.com/</a></td>
</tr>
</tbody>
</table>

**Exercise**

Write basic CRUD methods for the Employee/Person tables by writing static methods to insert, delete, retrieve and update records in both tables.