Introduction to Java Database Connectivity API

CSCE 156 - Introduction to Computer Science II

Christopher M. Bourke
cbourke@cse.unl.edu

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
  - Connection
  - PreparedStatement
  - ResultSet
  - Common Java data types (Integer, Double, String)

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 its 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 is loaded at run time through reflection
- Could be made configurable or delegated by some controller

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

JDBC

Connection

Java provides connectivity through java.sql.Connection:

```java
String url = "jdbc:mysql://cse.unl.edu/cselogin";
String username = "cselogin";
String password = "mysqlpasswd";
Connection conn = null;
try {
    conn = DriverManager.getConnection(url, username, password);
} catch (SQLException sqle) {
    ...
}
```

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 I**

**Querying – Prepared Statement**

- Always good to use PreparedStatement
- Can define parameters using `?`
- Parameters indexed by 1..n
- Can be reused (parameters reset and required)
- Parameters are safe!
- Special characters are escaped
- Potentially unsafe SQL code is sanitized

**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 name (or alias) using a String or
- Columns can be accessed via index (1,…)
- Standard getters provide functionality to get-and-cast columns

**JDBC II**

**Querying – Prepared Statement**

```
String query = "SELECT last_name AS lastName FROM user WHERE nuid = ?";
PreparedStatement ps = null;
try {
    ps = conn.prepareStatement(query);
    ps.setString(1, "35140602");
    ...}
```  

**Querying – Result Sets**

```
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**

**Querying – Updates**

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

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

**Good Practices – Rethrow Exceptions**

- Most methods explicitly throw SQLException
- This is a checked exception that must be caught and handled
- Occurs with DB errors or program bugs
- Little can be done either way
- Good to catch, log and rethrow
- Even better: use a logging utility like log4j
### 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:
  ```sql
  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:
  ```sql
  SELECT primary_email FROM user WHERE last = 'a';DROP TABLE users;
  ```
- Another example: input is `' OR '1'='1`
- Actual query:
  ```sql
  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
Additional Issues

Security Issues
- Where/how should database passwords be stored?
- Good security policy: assume an attacker has your password & take the necessary precautions (secure the server and network)
- Do not store sensitive data unencrypted

Efficiency Issues
- Repeat: close your resources
- Connection Pools
- Good normalization, design, & practice

Resources

- MySQL Community Server (http://www.mysql.com/downloads/)
- MySQL Workbench – a MySQL GUI (http://wb.mysql.com/)
- Connector/J – MySQL JDBC connector (http://www.mysql.com/downloads/connector/j/)
- Stanford’s Introduction to Databases free online course: http://db-class.com/

Log4j

- Standard output is not appropriate for deployed applications (may not exist, certainly no one is “listening” to it)
- Logging provides valuable debugging, metrics, and auditing information
- Provides runtime configuration, formatting, rolling files, etc.
- Supports granularity in different levels of logging (ERROR to DEBUG)
- Usage: give each loggable class a static logger:
  ```java
  private static org.apache.log4j.Logger log = Logger.getLogger(MyClass.class);
  ```
- Then use it: log.error("SQLException: ", e);
- Configure using a log4j.properties file (must be in the class path)
- Or: call BasicConfigurator.configure(); in your main to have a default setup

Exercise

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