JDBC
(Java Database Connectivity)

March 11, 2019
Announcements

• PS 1 due 3/15 (KST)
• TA: Mayukh Maitra, mayukh.maitra@stonybrook.edu
  • Office hours: T 6:30-10:30pm, H 6:30-8:30pm (CS Commons)
  • When you send an email with questions, send it to me and TA together
• Today
  • JDBC (Java Database Connectivity)
• Reading
  • Sections 6.2 and 6.3 of Cowbook
  • JDBC tutorial (http://docs.oracle.com/javase/tutorial/jdbc/)
  • Other recommended reading
    • Java Reflection (https://docs.oracle.com/javase/tutorial/reflect/)
• Break around 4:15pm
JDBC: Plan

- Overview of JDBC technology
- JDBC drivers
- Seven basic steps in using JDBC
- A simple database retrieval example
- Some useful classes from java.sql.*
- Precompiled queries
- Transactions
Programmer’s view

- Interface between a programming language and a database system
  - Java program running in one address space
  - Database management system running in another address space (not necessarily on two different machines)
  - Data being transferred between the two address spaces
    - That is where the complication/mismatch comes from
Why JDBC?

- Provides a standard library for accessing relational databases
- Can access different SQL databases with exactly the same Java syntax
- Standardizes
  - the mechanism for connecting to databases
  - the syntax for sending queries and committing transactions
  - the data structures representing the result
    - table: columns and rows (actual data)
    - metadata (description of data)
- But, does not standardize the SQL syntax
  - so, you can use any SQL extensions your database vendor supports
  - differences among vendors are minimal
References on JDBC, MySQL, MariaDB

- For JDBC, see:
  

  to find a list of available JDBC drivers among other things

- For MySQL, see:
  
  http://www.mysql.com/

- For MariaDB, see:
  
  http://mariadb.org/
JDBC drivers

- JDBC API, purely Java-based
- JDBC Driver Manager communicates with vendor-specific drivers that perform the real communication with the database
- JDBC driver types
  - I. JDBC-ODBC Bridge driver
  - II. Direct translation to native API via non-Java driver
  - III. Network bridge to a middleware server that translates JDBC into DBMS-specific methods
  - IV. Direct translation to native API via Java driver, e.g., MySQL Connector/J
### JDBC data types

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>FLOAT</td>
<td>double</td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY</td>
<td></td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>CLOB</td>
<td>Clob*</td>
</tr>
<tr>
<td>BLOB</td>
<td>Blob*</td>
</tr>
<tr>
<td>ARRAY</td>
<td>Array*</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>mapping of underlying type</td>
</tr>
<tr>
<td>STRUCT</td>
<td>Struct*</td>
</tr>
<tr>
<td>REF</td>
<td>Ref*</td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td>underlying Java class</td>
</tr>
</tbody>
</table>

*SQL3 data type supported in JDBC 2.0
Seven basic steps in using JDBC

1. Load the JDBC driver
2. Define the connection URL
3. Establish the connection
4. Create a statement object
5. Execute a query or update
6. Process the results
7. Close the connection
1. Load the JDBC driver

- `import java.sql.*`
- Driver
  - is on the client
  - software that knows how to talk to the database server
  - translates calls written in Java into the format understood by the server
  - you have to use a driver specific to the database server
    - see the link given in an earlier slide for a list of available drivers

- Load the driver by loading the appropriate class
  - a `static` block in the class automatically
    - creates a driver instance
    - registers it with the JDBC driver manager
Initialization block

- is a block of statements that appear within the class declaration, outside of any member, constructor, or declaration
- initializes the fields: static or non-static
  - executed at the time the class is loaded
- non-static initialization block
  - executed as if it were placed at the beginning of every constructor in the class
- static initialization block
  - initializes static members
Example: static initialization block

class Primes {

    static int[] knownPrimes = new int[4];

    static {
        knownPrimes[0] = 2;
        for (int i = 1; i < knownPrimes.length; i++)
            knownPrimes[i] = nextPrime();
    }

    ...

}
Code for loading the driver

// Learn the Class class (Java Reflection reference)

Try {
    // to register drivers with the Driver Manager
    Class.forName("connect.microsoft.MicrosoftDriver");
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Class.forName("com.mysql.jdbc.Driver");
    Class.forName("org.mariadb.jdbc.Driver");
}

catch (ClassNotFoundException cnfe) {
    System.err.println("Error loading driver: "+ cnfe);
}
2. Define the connection URL

- With the driver loaded, specify the location of the database server

- URL referring to a database contains:
  - protocol (jdbc:)
  - server host name
  - port
  - database name

- E.g., "jdbc:mysql://localhost:3306/bankdb"
Code for connection URLs

String host = “dbhost.yourcompany.com”;
String dbName = “someName”;
int port = 1234;
String oracleURL = “jdbc:oracle:thin:@” + host + “:” + port + “:” + dbName;
String sybaseURL = “jdbc:sybase:Tds:” + host + “:” + port + “:” + “?SERVICENAME=“ + dbName;
String msAccessURL = “jdbc:odbc:” + dbName;

// see this for more details on Oracle:
//
// http://docs.oracle.com/cd/E11882_01/appdev.112/e13995/oracle/jdbc/OracleDriver.html
3. Establish a connection

- To make the network connection to a database server, pass the following to the `getConnection` method of `DriverManager`:
  - URL
  - database username
  - password

```java
String username = "jay_debesee";
String password = "secret";
Connection connection =
    DriverManager.getConnection(oracleURL, username, password);
```
Establish a connection (cont.)

- Can get more information about the database through the metadata:
  ```java
  DatabaseMetaData dbMetaData = connection.getMetaData();
  String productName = dbMetaData.getDatabaseProductName();
  System.out.println("Database: “ + productName);
  String productVersion = dbMetaData.getDatabaseProductVersion();
  System.out.println("Version: “ + productVersion);
  ```
- Other methods available in the **Connection** class:
  - `prepareStatement` - to create a **PreparedStatement** to create precompiled query to submit to the database
  - `prepareCall` - to create a **CallableStatement** to access stored procedure in the database
  - `rollback` - to undo statements since the last `commit`
  - `commit` - to finalize operations since the last `commit`
  - `close` - to terminate connection
4. Create a statement

- Send queries and commands to the database using a `Statement` object

- Created as follows using `Connection`:

```java
Statement statement = connection.createStatement();
```
5. Execute a query

- With the `Statement` object, you can send SQL queries by using the `executeQuery` method.

- This method returns a `ResultSet` object.

```java
String query = "SELECT col1,col2,col3 FROM tablename";
ResultSet resultSet = statement.executeQuery(query);
```

- To modify the database, use `executeUpdate` instead of `executeQuery`, and supply a query string that uses `UPDATE`, `INSERT`, or `DELETE`.

- Other methods supported by `Statement` class:
  - `execute` - to execute an arbitrary command
  - `setQueryTimeout` - to set a maximum delay to wait for results
6. Process the results

- Simplest way
  - process the results one row at a time
  - use the `next` method of `ResultSet` to move a row at a time within a table
  - within a row, use `getXxx` methods with a column index or name
    - e.g., `getInt`, `getString`
    - Note: index is 1-based, not 0-based

- To print some columns in all rows in a `ResultSet` object:
  ```java
  while (resultSet.next()) {
      System.out.println(
          resultSet.getString(1) + " " +
          resultSet.getString(2) + " " +
          resultSet.getString("firstname") + " " +
          resultSet.getString("lastname"));
  }
  ```
Process the results (cont.)

- **ResultSet** is used for **fixed-format queries**, and supports:
  - `getXxx`
  - `next/previous`
  - `findColumn` - to get the index of the named column
  - `getMetaData` - to retrieve information about the **ResultSet** in a **ResultSetMetaData** object
Process the results (cont.)

- **ResultSetMetaData** is used for *ad hoc queries* to *discover* dynamically high-level information about the query result

- **Methods on ResultSetMetaData:**
  - `getColumnCount` - to get the number of columns
  - `getColumnName(colNumber)` - to get a column name
  - `getColumnType` - returns the SQL type, to compare with entries in `java.sql.Types`
  - `isReadOnly` - is the entry read only?
  - `isSearchable` - can it be used in a *WHERE* clause?
  - `isNullable` - is a *null* value permitted?
  - etc.

- Note: it does not include the number of rows - use `next` instead
7. Close the connection

- To close it explicitly:
  ```java
  connection.close();
  ```

- Do not close it if you expect to perform additional database operations

- The overhead of opening a connection is usually large

- More later on this issue . . . (‘connection pool’)
A simple JDBC example

• See Bank1.java, DB.java from the course web
Using **Statement**

- Through the `Statement` object, SQL statements are sent to the database.

- Three types of statement objects are available:
  - `Statement`:
    - For executing a **simple SQL statement**.
    - Useful for static SQL statements at runtime. Can’t accept parameters.
  - `PreparedStatement`:
    - For executing a **precompiled SQL statement** passing in parameters.
    - Useful when you use a SQL statement multiple times.
  - `CallableStatement`:
    - For executing a database **stored procedure**.
    - Can also accept runtime parameters.
Useful **Statement** methods

- **executeQuery**
  - Executes the SQL query (typically `SELECT` statement) and returns the data in a table (`ResultSet`)
  - The resulting table may be empty but never **null**
    
    ```java
    ResultSet results = 
    statement.executeQuery("SELECT a, b FROM atable");
    ```

- **executeUpdate**
  - Used to execute for **INSERT**, **UPDATE**, or **DELETE** SQL statements
  - Returns the number of rows that were affected in the database
  - Supports Data Definition Language (DDL) statements `CREATE TABLE`, `DROP TABLE` and `ALTER TABLE`
    
    ```java
    int rows = 
    statement.executeUpdate("DELETE FROM EMPLOYEES" + 
    "WHERE STATUS=0");
    ```
Useful **Statement** methods (cont.)

- **getMaxRows/setMaxRows**
  - Determines the maximum number of rows a ResultSet may contain
  - Unless explicitly set, the number of rows is unlimited (return value of 0)

- **getQueryTimeout/setQueryTimeout**
  - Specifies the amount of a time a driver will wait for a Statement to complete before throwing a SQLException
Prepared statements (precompiled queries)

• Idea
  • If you are going to execute similar SQL statements multiple times, using “prepared” (parameterized) statements can be more efficient
  • Create a statement in standard form that is sent to the database for compilation before actually being used
  • Each time you use it, you simply replace some of the marked parameters using the setXxx methods

• PreparedStatement inherits Statement: use the following
  • executeUpdate if the query does not return a ResultSet
  • executeQuery if it returns only one ResultSet, e.g., SELECT
  • execute if it might return more than one ResultSet
Connection conn =
    DriverManager.getConnection(url, user, password);
PreparedStatement statement =
    conn.prepareStatement("UPDATE employees "+
    "SET salary = ? " +
    "WHERE id = ?");
int[] newSalaries = getSalaries();
int[] employeeIDs = getIDs();
for (int i=0; i<employeeIDs.length; i++) {
    statement.setInt(1, newSalaries[i]);  // salary
    statement.setInt(2, employeeIDs[i]);  // id
    statement.executeUpdate();
}
Useful *PreparedStatement* methods

- **setXxx**
  - Sets the indicated parameter (?) in the SQL statement to the value

- **clearParameters**
  - Clears all set parameter values in the statement
Transactions

• Idea
  • By default, after each SQL statement is executed the changes are automatically committed to the database
  • Turn auto-commit off to group two or more statements together into a transaction
    `connection.setAutoCommit(false)`
  • Call `commit` to permanently record the changes to the database after executing a group of statements
  • Call `rollback` if an error occurs
Transactions, example

Connection connection = DriverManager.getConnection(url, username, passwd);

boolean autocommit = connection.getAutoCommit();

Statement statement;

try {
    connection.setAutoCommit(false);
    statement = connection.createStatement();
    statement.executeUpdate(...);
    statement.executeUpdate(...);
    ...
    connection.commit();
} catch (SQLException e) {
    connection.rollback();
}

finally {
    connection.close();
    connection.setAutoCommit(autoCommit);
}
Useful Connection methods (for transactions)

- `getAutoCommit/setAutoCommit`
  - By default, a connection is set to auto-commit
  - Retrieves or sets the auto-commit mode

- `commit`
  - Force all changes since the last call to commit to become permanent
  - Any database locks currently held by this `Connection` object are released

- `rollback`
  - Drops all changes since the previous call to commit
  - Releases any database locks held by this `Connection` object
More JDBC options

- Find a reference on the following:
  - Stored procedures
    - See [http://www.tutorialspoint.com/jdbc/jdbc-statements.htm](http://www.tutorialspoint.com/jdbc/jdbc-statements.htm) for some sample code
  - Connection pooling
    - See [http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/pool.html](http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/pool.html)
Next topic

- Entity-Relationship Data Model (Chapter 2 of Cowbook)