Basic JDBC

Java Developer

StayAhead Training; May, 2022

JDBC

Basic JDBC

- JDBC (Java DB Connectivity) is an API that describes the classes and methods that must be used to interact with a relational DB
- JDBC is realised through the interfaces in the java.sql package, e.g.
  - Connection
  - Statement
  - ResultSet
- Each DB vendor (MySQL, Postgres, Oracle, etc.) provides its own implementations of the java.sql interfaces
- We can code to the interfaces, ignorant of the specific implementations

3rd Party Libraries*

Basic JDBC

- Whilst the java.sql package of classes is a part of the standard library, the vendor-specific code is not
- To use 3rd party code it must be added to the classpath
- Recall that the classpath is the directory/directories in which both the compiler and interpreter look for classes (.class files)
- Whilst you can download the 3rd party code and add it to the classpath manually, it is easier to have the editor do it for you
- Build tools, like Maven, can also be used to manage such dependencies

Class Loading*

Basic JDBC

- Confusingly, Java classes are themselves objects
- You can obtain a reference to the class object in a number of ways, e.g.

```
var classObject1 = new Movie().getClass();
var classObject2 = Movie.class;
```

- The class must be loaded into memory before it can be instantiated
- Sometimes it's necessary to load a class without instantiating it, e.g.

```
Class.forName(Movie.class); // rare!
```

Loading the Driver

Basic JDBC

- A <u>Driver</u> is the fundamental JDBC object and provides for obtaining connections to the DB
- To "load the Driver" is to load the Driver class into memory whereupon, weirdly, it will instantiate itself and register itself with the DriverManager
- Prior to Java 6 it was necessary to manually load the Driver using, e.g. Class.forName or DriverManager.registerDriver
- Since Java 6 the Driver will be loaded automatically provided the vendor library is on the classpath

Establishing a Connection

Basic JDBC

- Obtain a Connection by calling the DriverManager's getConnection method (internally getConnection calls the Driver's connect method)

- getConnection accepts a connection String which takes the form:

```
<protocol>:<subprotocol>://<hostname>:<port>/<dbname>
```

- The protocol is always jdbc

- The subprotocol is typically the name of the vendor, e.g. postgresql

Establishing a Connection

Basic JDBC

- Obtain a Connection, e.g.

```
var connection = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/mydb");
```

- The inclusion of a username (x) and password (y) can vary, e.g.

```
jdbc:postgresql://x:y@localhost:5432/mydb

jdbc:postgresql://localhost:5432/mydb?user=x&password=y
```

Transactions*

Basic JDBC

- A transaction (tx) is a DB device to ensure that a unit of work composed of n operations either succeeds in full or not at all (is never partially successful)
- For example, a bank transfer involves a withdrawal from one account and a deposit into another - both operations must succeed; if one fails then the other must fail too
- If all the operations are successful then the transaction is committed - all the changes are persisted to the DB
- If one of the operations fails then the transaction is rolled back - **none** of the changes are persisted to the DB

Transactions*

Basic JDBC

- Transactions must pass the ACID test:
  - A atomic; all or nothing
  - C consistent; the DB must be left in a valid state
  - I isolated; changes must not be visible to other txs until committed
  - D durable; post commit the changes will persist on DB restart

Auto Commit

Basic JDBC

- In auto commit mode the DB will execute every op. in its own transaction
- This is likely to be undesirable, particularly if we're performing units of work that comprise two or more operations
- Typically the JDBC Driver will enable auto commit for new Connections
- Turn off auto commit, e.g.

```
connection.setAutoCommit(false);
```

Transaction Management

Basic JDBC

- To manage transactions manually ensure that auto commit mode is disabled
- A transaction begins automatically when the first operation is performed
- Each subsequent operation is included in the ongoing transaction
- If an exception is thrown then it's likely that one or more operations has failed and the transaction should be rolled back
- If no exception is thrown then the transaction is committed after all of the operations that make up the unit of work

Transaction Management

Basic JDBC

```
try {
  // TODO operation #1
  // TODO operation #2
  connection.commit();
} catch (SQLException e) {
  connection.rollback();
}
```

Creating a Statement

Basic JDBC

- A Statement is used to execute some SQL against the DB
- Obtain a Statement, e.g.

```
var statement = connection.createStatement();
```

- Execute a query, e.g.

```
var resultSet =
    statement.executeQuery("select * from users");
```

SQL Injection Attacks*

Basic JDBC

- An SQL injection attack occurs when a hacker exploits poorly written code to inject malicious values into some SQL, e.g.

```
var sql = "select * from users where username = '" +
    userInput + "'";
```

- Instead of inputting a valid username, the hacker will input something like: `' or 'a' = 'a` which will yield SQL that reads:

```
select * from users where username = '' or 'a' = 'a';
```

- **Never** use concatenation to insert user values into the SQL

PreparedStatement

Basic JDBC

- A PreparedStatement is a child of Statement and enables the separating of user input (parameters) from SQL so that the DB can check for illegal values
- Construct a PreparedStatement, e.g.

```
var statement = connection.prepareStatement(
    "select * from users where username = ?");

// the first ? is to be substituted for userInput
// there are many such set methods, e.g. setInt, setFloat
statement.setString(1, userInput);
```

CallableStatement

Basic JDBC

- A CallableStatement is a child of PreparedStatement and is used to execute stored procedures, e.g.

```
var statement = connection.prepareCall(
    "{call myProcedure()}");
```

- CallableStatement deals with parameters like PreparedStatement, e.g.

```
var statement = connection.prepareCall(
    "{call myProcedureWithParam(?)}");

statement.setString(1, userInput);
```

Executing a Query

Basic JDBC

- Statement's **executeQuery** method should be used when the SQL is expected to return one or more results

```
var resultSet = statement.executeQuery(
    "select * from users");
```

- The return type is ResultSet (more later)

Executing an Update

Basic JDBC

- Statement's **executeUpdate** method should be used when the SQL is an insert, update, or delete statement, or any DDL statement*

```
var statement = connection.prepareStatement(
    "insert into users (username, password) values(?, ?))";
...
var rowsAffected = statement.executeUpdate();
```

- The return type is int
- *A DDL (Data Definition Language) statement is one that deals with the structure of a table, e.g. create, alter, drop etc.

Batch Updates

Basic JDBC

- Statement supports the executing of updates in batches, e.g.

```
var statement = connection.createStatement();

statement.addBatch("delete from users where id = 12");
statement.addBatch("delete from users where id = 27");
statement.addBatch("delete from users where id = 51");

var rowsAffected = statement.executeBatch();
```

Batch Updates

Basic JDBC

- PreparedStatement supports the executing of updates in batches, e.g.

```
var statement = connection.prepareStatement(
    "insert into users (username, password) values(?, ?))";

for (var user : users) {
  statement.setString(1, user.username);
  statement.setString(2, user.password);
  statement.addBatch();
}

var rowsAffected = statement.executeBatch();
```

Processing a ResultSet

Basic JDBC

- A ResultSet is a table of data representing the result of some query
- It maintains a cursor which is initially positioned *before* the first row
- The ResultSet's next method moves the cursor to the next row and returns false if there are no more records
- Due to the nature of the ResultSet (and its cursor) and the behaviour of the next method, a ResultSet is typically processed using a while loop

Processing a ResultSet

Basic JDBC

- Processing a ResultSet, e.g.

```
while (resultSet.next()) {
  var id = resultSet.getInt("id");
  var username = resultSet.getString("username");
  var password = resultSet.getString("password");
  users.add(new User(id, username, password));
}
```

- Each of the get methods is overridden to accept either a column index or column name (column indexes begin at 1)

ResultSet Type

Basic JDBC

- ResultSet Type dictates whether you can navigate forwards and backwards between the rows in the ResultSet table
- There are three ResultSet Types:
    - FORWARD_ONLY (the default) means you can move forwards only
    - SCROLL_INSENSITIVE means you can move forwards and backwards; changes made by other threads/processes are **not** reflected in the table
    - SCROLL_SENSITIVE means you can move forwards and backwards; changes made by other threads/processes are reflected in the table

ResultSet Type

Basic JDBC

- A ResultSet Type of something other than FORWARD_ONLY enables the use of other ResultSet methods including:
  - absolute (accepts a row number)
  - beforeFirst (moves the cursor to before the first row)
  - first
  - last
  - previous

ResultSet Concurrency

Basic JDBC

- ResultSet Concurrency dictates whether the ResultSet can be read only, or whether it can be updated (used to update the associated DB table)
- There are two ResultSet Concurrency types:
  - CONCUR_READ_ONLY (the default) means the ResultSet can only be read
  - CONCUR_UPDATABLE means the ResultSet can be used to update rows and even to insert new ones

ResultSet Concurrency

Basic JDBC

- Using a ResultSet to update the DB table, e.g.

```
resultSet.updateString("password", "newPassword");
resultSet.updateRow();

resultSet.moveToInsertRow();
resultSet.updateString("username", "Dave");
resultSet.updateString("password", "evaD");
resultSet.insertRow();
```

- **DON'T** store plain text passwords in your DB!

ResultSet Holdability

Basic JDBC

- ResultSet Holdability dictates whether the ResultSet is closed when the transaction is committed
- There are two ResultSet Holdability types:
    - CLOSE_CURSORS_OVER_COMMIT (the default) means the ResultSet is closed when the transaction is committed
    - HOLD_CURSORS_OVER_COMMIT means the ResultSet is kept open after the transaction is committed (you might exploit to this to perform updates in another transaction thereby reducing the total number of DB operations)

ResultSet Options

Basic JDBC

- ResultSet Type, Concurrency, and Holdability options may be configured at the point at which you create the Statement, e.g.

```
var statement = connection.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE,
    ResultSet.HOLD_CURSORS_OVER_COMMIT);
```

- NB: not all JDBC Drivers support all of these features - check with your vendor before attempting to make use of them

Closing Resources

Basic JDBC

- ResultSets, Statements, and Connections should be closed (in that order) when you are done with them
- Exploit try with resources to do so, e.g.

```
try (var conn = getConnection();
     var stmt = conn.createStatement();
     var resultSet = stmt.executeQuery(sql)) {

  // TODO process resultSet
} catch (SQLException e) { ... }
```

A Note On Exceptions*

Basic JDBC

- Practically every line of JDBC code may throw an SQLException
- This can make writing JDBC code tedious and result in more exception handling code than anything else
- Avoid the temptation to ignore/swallow exceptions
- If coding in an object-oriented manner then the class that houses the JDBC code is unlikely to be able to deal with SQLExceptions and so should either throw them to the caller, or catch them and re-throw more meaningful exceptions