Outline

- Quiz 1
- JBDC review
- Update queries
- Scrollable / updatable result sets
- Prepared Statements
- Stored Procedures
Quiz 1

- Ave: 9.25
- Mode: 10
- Min: 3
- Max: 15
JDBC

- **Connection**
  - two-way interaction with DB

- **Statement**
  - holder for sql statement
  - `executeQuery/executeUpdate`

- **ResultSet**
  - result of executing a query
  - access contents via `getXXX` methods
INSERT statement

INSERT INTO `{table}`
VALUES ( {value1, ..., valuen} );

INSERT INTO Reservations
VALUES ( 1212, #1/23/2003#, 'Sosa', 'Sammy', 2, 14, 2 );
UPDATE statement

UPDATE {table}
SET {column} = {value}
WHERE { criteria }

UPDATE Reservations
SET RoomType = 4
WHERE ID = 1234;
DELETE statement

DELETE FROM {table}
WHERE { criteria }

DELETE FROM Reservations
WHERE ID = 9998;
JDBC Update queries

- **No ResultSet returned**

- **Example 1**
  ```java
  String sql = "INSERT INTO Reservations " +
               "VALUES (100, #12/11/2003#, 'L.L.', 'Bean', 2, 2, 2);";
  int rows = stmt.executeUpdate(sql);  // always 1
  ```

- **Example 2**
  ```java
  String sql = "DELETE FROM Reservations WHERE (Date = #1/21/2003#);";
  int rows = stmt.executeUpdate(sql);  // how many rows deleted?
  ```
Assembling queries

- Use String operators (+) to assemble queries

String sql = "INSERT INTO Reservations " + 
    "VALUES (100, #12/11/2003#, " + firstName + ", " + 
    lastName + ", 2, 2, 2);";
int rows = stmt.executeUpdate(sql);
Scrollable / Updatable Result Sets
Cursors

- bundle of information
  - captures the current state of interaction with a database

- cursor types
  - simplest = forward-only
  - otherwise tradeoff
    - flexibility / power
    - speed / server resources
JDBC Cursor Types

- Two parameters
  - ResultSet type
  - ResultSet concurrency
ResultSet Type

- Two dimensions
  - forward-only vs scrollable
  - insensitive vs sensitive
- But
  - forward-only = insensitive
- Three possibilities
  - forward-only
  - scrollable, sensitive
  - scrollable, insensitive
ResultSet Concurrency

- **Updatable**
  - Can the ResultSet be used to modify the database?
- **Two possibilities**
  - read-only
  - updatable
JDBC Cursors

Forward-Only

Scrollable

Updatable

Database
Using ResultSet Types

- **Statement**
  - RS types are assigned to a Statement
  - ResultSet created by that Statement will have the given properties

```java
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery (querySQL);
```
Scrollable ResultSet

- Can move both directions in results
- Can jump around

Methods
- previous ()
- absolute ()
- relative ()
- last ()
- first ()

Note
- not subclass of ResultSet
Sensitive vs Insensitive

- If insert happens first, query will return new row
- If select happens first?
  - insensitive cursor will not return new row
  - sensitive cursor will/should return new row
Updatable ResultSet

- Use results of SELECT query
  - perform INSERT, DELETE, and UPDATE
- Useful for interactive DB editing
Update Process

1. Connect to database
2. Create an updatable statement
3. Call executeQuery to get ResultSet
4. Navigate to a row
5. Call update methods to modify contents
6. Call updateRow () to commit changes
Update Methods

- **Examples**
  - `rs.updateString (1, "Jack");`
  - `rs.updateDate ("Arrival_Date", m_date);`

- **Like “get” methods**
  - `update[Type_name]`
  - always the Java type
  - column name or number
  - new value
public void upgradeCar (String firstName, String lastName)  
{  
    Statement dateStmt = conn.createStatement (  
        ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);  
    ResultSet rs = dateStmt.executeQuery (  
        "SELECT * FROM Reservations WHERE (FirstName='" + firstName +  
        ") AND (LastName='" + lastName + ");");  
    if (rs.next())  
    {  
        int carType = rs.getInt ("Car_Type");  
        if (carType < MAXIMAL_CAR_TYPE)  
        {  
            rs.updateInt ("carType", carType + 1);  
            rs.updateRow ();  
        }  
    }  
    rs.close ();  
}
Inserting

- Insert Row
  - Separate location for insertion
Insertion Process

1. Connect to database
2. Create an updatable statement
3. Call executeQuery to get ResultSet
4. Move to the insert row
5. Call update methods to fill row
6. Call insertRow to commit changes
7. Continue inserting or
   Move back to other results
Example

Statement stmt = conn.createStatement (  
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery ("SELECT * FROM Reservations");
rs.moveToInsertRow ();
rs.updateInt ("Id", newId);
rs.updateString ("FirstName", newFirstName);
rs.updateString ("LastName", newLastName);
rs.updateInt ("Car_Type", newCarType);
... etc ...
rs.insertRow ();
rs.close ();
Delete

- Simple
  
  `rs.deleteRow();`
Updates vs SQL

Statement stmt = conn.createStatement (  
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery (  
    "SELECT * FROM Books WHERE (Publisher='MS Press');");
while (rs.next()){
    double price = rs.getDouble ("Price");
    rs.updateDouble ("Price", price * 0.9);
    rs.updateRow ();
}

Or

Statement stmt = conn.createStatement ();
ResultSet rs = stmt.executeQuery (  
    "UPDATE Books SET Price=(Price*0.9) WHERE (Publisher='MS Press');");
Note

- Not every ResultSet is updatable
- Joins are often (usually) not
Summary – Updatable ResultSet

- Use Updatable RS instead of INSERT, UPDATE and DELETE

- Benefits
  - integrates with querying
  - single SQL expression

- But
  - SQL may be more efficient
Query Answering Process

1. Parse SQL string
2. Construct a query plan
3. Execute plan
4. Deliver results
Efficiency Gains?

- Steps 1 & 2 will be similar for similar queries
- Programs often issue many similar queries
  - empirical fact
- String manipulation is slow
Example

```java
public void upgradeCar (String firstName, String lastName)
{
    Statement dateStmt = conn.createStatement (  
        ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = dateStmt.executeQuery (  
        "SELECT * FROM Reservations WHERE (FirstName='" + firstName +  
        "') AND (LastName='" + lastName + ");");
    if (rs.next())
    {
        int carType = rs.getInt ("Car_Type");
        if (carType < MAXIMAL_CAR_TYPE)
        {
            rs.updateInt ("carType", carType + 1);
            rs.updateRow ();
        }
    }
    rs.close ();
}
```
Repeated Costs

- String assembly
- SQL Parsing identical
  - queries the same except for names
- Query planning probably identical
  - depends on exact structure of indices
- Standard computational solution
  - do the work once
  - cache the result
Maintenance Problems

- Hard to verify / debug the SQL
- Hard to modify the query
Prepared Statement

- SQL statement
  - with parameters
- Parameters filled programmatically
  - not with string operations
Find By Name

SELECT * FROM Reservations WHERE (FirstName='"' + firstName + '"') AND (LastName='"' + lastName + '"');"
Prepared Statement

“SELECT * FROM Reservations
WHERE (FirstName=?) AND (LastName=?);”
Prepared Statements in JDBC

```java
PreparedStatement stmt =
    con.prepareStatement (  
        "SELECT * FROM Reservations WHERE (FirstName=? AND (LastName=?));";  
    );
stmt.setString (1, "Jack");
stmt.setString (2, "Spratt");
rs = stmt.executeQuery();
```
Steps

1. Call prepareStatement with parameterized SQL String
2. Set parameters values using setter methods
   - `setXxx (index, value)`
   - parameters are numbered from 1!
3. Call `executeQuery` or `executeUpdate`
   - no arguments
Summary – Prepared Statement

- Use parameterized SQL instead of assembling queries with Strings

- benefits
  - much easier to read / maintain
  - significant performance gain for multiple queries
  - standard practice in JDBC coding
Stored Procedures
2-Tier Application

Client program

JDBC

Database

Driver
Where to put computation?

- All client
  - use database just to store
  - copy to client to compute

- All server
  - use client just to display
  - use database for computation
## Client-Server Trade-off

<table>
<thead>
<tr>
<th>Condition</th>
<th>Data small</th>
<th>Data large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server available</td>
<td>Either OK</td>
<td>Server</td>
</tr>
<tr>
<td>Server busy</td>
<td>Client</td>
<td>It depends</td>
</tr>
</tbody>
</table>
Stored Procedures

- Database-resident code
- Serve many purposes
  - computations too complex for SQL
  - triggers
  - move computation to the data
Steps

1. Write stored procedure
2. Create a callable statement
3. Set the parameters
4. Call the procedure
5. Possibly get return values
public static void updateAvailableRoomData
  (int hotelId, String roomType, int mumOfRooms, String updateFlag)
{
  Connection connection = null; // Database Connection Object
  try {
    connection = new OracleDriver().defaultConnection();
    String addBookedSeats = " + ";
    String subTotalSeats = " - ";
    if (updateFlag.equals("COUT")) {
      addBookedSeats = " - ";
      subTotalSeats = " + ";
    }
    if (roomType.equals("ORCL")) roomType = "OTHR";
    PreparedStatement pstmt =
        connection.prepareStatement("UPDATE ROOM_AVAILABILITY " + " SET BOOKED_" + roomType + " = BOOKED_" + roomType + addBookedSeats + mumOfRooms + ", TOTAL_" + roomType + " = TOTAL_" + roomType + subTotalSeats + mumOfRooms + " WHERE HOT_ID = ? AND BOOKING_DATE = " + " ( SELECT MAX(BOOKING_DATE) FROM ROOM_AVAILABILITY " + " WHERE HOT_ID = ? ) " ");
    pstmt.setInt(1,hotelId); // Bind the Hotel ID input parameter
    pstmt.setInt(2,hotelId); // Bind the Hotel Id input parameter int noRecordsUpdated = pstmt.executeUpdate(); // Execute the Statement ...
    ... et cetera ...
}
From client

```java
CallableStatement stmt = con.prepareCall ("Call Hotel_HotelBookingsSample_updateAvailableRoomData (?, ?, ?, ?)");
stmt.setInt (1, 5);
stmt.setString (2, "Single");
stmt.setInt (3, 1);
stmt.setString (4, "COUT");
stmt.execute();
```
Return values?

```java
stmt.registerOutParameter (3, Types.INTEGER);
stmt.execute();
int result = Stmt.getInt(3);
```
Summary – Stored Procedures

- Embed computation in database using database-specific code

  - benefits
    - move computation to data

  - drawbacks
    - SP code not portable
    - maintenance problematic

- still frequently used for performance benefits