CS422 Principles of Database Systems
JDBC and Embedded SQL

Chengyu Sun
California State University, Los Angeles

Applications and Databases

Users  Applications  Databases

C/C++, Java, PHP, Perl...  SQL

Call-level Interface (CLI)
◆ A library of functions that allow applications to
  ◆ connect to the database
  ◆ send SQL statements to the database
  ◆ get the results back from the database

Connection c = DriverManager.getConnection( url );
Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery( "select * from items" );
while( rs.next() ) {
    // some processing here
}

Embedded SQL
◆ Embed SQL statements directly in host language
  ◆ Variables shared by SQL and the host language
  ◆ Check SQL syntax at compilation time
  ◆ Part of the SQL standard

double price;
#sql {
select price into :price from items where name = 'milk';
}
System.out.println( "The price of milk is " + price + "." );

JDBC
◆ An interface between Java programs and SQL databases

Java program  JDBC Driver  Oracle

JDBC API

oracle.jdbc.driver.OracleDriver

Setup Oracle JDBC Driver
◆ Download instantclient-basic (see http://sun.calstatela.edu/~cysun/www/teaching/cs422/extras/server.html)
◆ Add one of following JAR files to your class path:
  ◆ classes12.jar – for JDK 1.2
  ◆ ojdbc14.jar – for JDK 1.4 or above
**JDBC Basics** …

- import java.sql.*;
- Load driver
  - Class.forName("oracle.jdbc.driver.OracleDriver")
- Create connection
  - Connection c = DriverManager.getConnection( url, username, password );
  - URL
    - jdbc:oracle:thin:@host:port:service
    - E.g. jdbc:oracle:thin:@cs.calstatela.edu:1521:orc

**... JDBC Basics**

- Create statement
  - Statement stmt = c.createStatement();
  - stmt.executeQuery()
  - stmt.executeUpdate()
- Get result back
  - ResultSet rs

http://java.sun.com/j2se/1.5.0/docs/api/java/sql/package-summary.html

---

**DB Query Results**

- In a program, we want to
  - Access each row
  - Access column in a row
  - Access column names

```sql
select * from items;
```

<table>
<thead>
<tr>
<th>name</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk</td>
<td>3.99</td>
<td>2</td>
</tr>
<tr>
<td>beer</td>
<td>6.99</td>
<td>3</td>
</tr>
</tbody>
</table>

**ResultSet: Row Access**

- next() – move cursor down one row
  - true if the current row is valid
  - false if no more rows
  - Cursor starts from before the 1st row

---

**ResultSet: Column Access**

- Access the columns of current row
- getXXxx( String columnName )
  - E.g. getString( "user" );
- getXXxx( int columnIndex )
  - columnIndex starts from 1
  - E.g. getString( 1 );

---

**ResultSet: Column Names**

```java
ResultSetMetaData meta = rs.getMetaData();
```
ResultSet: Size

- No size() method?
- Something about `FetchSize`
  - `getFetchSize()`
  - `setFetchSize(int nrows)`

Prepared Statements

- Statements with parameters
  ```java
  String sql = "insert into items values (?, ?, ?)";
  PreparedStatement pstmt = c.prepareStatement(sql);
  pstmt.setString(1, "orange");
  pstmt.setDouble(2, 1.59);
  pstmt.setInt(3, 4);
  pstmt.executeUpdate();
  ```

Benefits of Using Prepared Statements

- Easier to create the query string
- Much more secure if part of the query string is provided by user
- Better performance (maybe)
  ```java
  // without PreparedStatement, you need to worry
  // about quotations
  String sql = "select salary from employees where " +
  "username ="" + username + "";
  // and somebody may try to pass a username like
  // "cysun" or username <> "cysun"
  ```

Call Stored Procedures

- Different DBMS have different implementations of stored procedures.
  ```java
  CallableStatement cstmt = c.prepareCall("{call sum2p(?,?,?)}");
  cstmt.setInt(1, 12);
  cstmt.setInt(2, 22);
  cstmt.registerOutParameter(3, Types.INTEGER);
  cstmt.executeUpdate();
  System.out.println( cstmt.getInt(3) );
  ```

Transactions in JDBC

- `c.setAutoCommit(false)`
- `stmt.executeUpdate("insert into turnins values (1,1, 'Foo.java')")`;
- `stmt.executeUpdate("insert into turnins values (1,2, 'Bar.java')")`;
- `c.commit()`

A Transaction Example

- `TxnExample.java`
  ```java
  Connection c = DriverManager.getConnection( url, user, pass );
  Statement stmt = c.createStatement();
  c.setAutoCommit( false );
  stmt.executeUpdate("insert into turnins values (1,1,'Foo.java')");
  stmt.executeUpdate("insert into turnins values (1,2,'Bar.java')");
  c.commit();
  ```
Rollback upon Exception

- `rollback()` should be called explicitly if a SQLException is raised during a transaction.

```java
try {
    System.err.println( e.getMessage() );
    if( c != null ) {
        c.rollback();
    } catch( SQLException ex ) {
        System.err.println( ex.getMessage() );
    }
} catch( SQLException e ) {
    System.err.println( e.getMessage() );
}
```

SQLJ – Embed SQL in Java

- Part of SQJ-99 standard
- Oracle’s SQLJ implementation is built on top of JDBC
  - requires JDBC driver (`classes12.jar`, not `ojdbc14.jar`)

Setting Up SQLJ

- Download from http://sun.calstatela.edu/~cysun/documentation/oracle/sqlj/
  - runtime12.jar
  - translator.jar
- Add the two jar files to your CLASSPATH
- java sqlj.tools.Sqlj <filename.sqlj>

An SQLJ Example

```java
Oracle.connect {
    "jdbc:oracle:thin:@cs.calstatela.edu:1521:orc",
    "cs422stu31",
    "abcd"
};
double price;
#sql {
    select price into :price from items
    where name = 'milk'
};
System.out.println("The price of milk is " + price + ".");
```

SQLJ Basics

- Making connection
- Variables shared by Java and SQL
- SELECT...INTO
- Handling NULL value

Iterators

- An iterator must be declared to handle queries that return multiple rows.
```sql
#sql <modifier> iterator IteratorName( column_list );
```
An Iterator Example

```java
#sql iterator Results ( String name, double price );
Results r;
#sql r = { select name, price from items };
while( r.next() )
    System.out.println( r.name() + ", " + r.price() );
```

Note that based on the iterator declaration, two methods are automatically created:
- `String name()`
- `double price()`

Dynamic SQL

- When SQL statements are generated by the host language at runtime.
- It’s easy to have dynamic SQL in JDBC, but not so easy in embedded SQL:
  - SQL statements have to be enclosed in the `#sql` block
  - SQL statements are checked at compile time.
- Solution
  - `IN` variables or `EXECUTE IMMEDIATE`

About SQLJ

- The good
  - simple syntax
  - SQL statements are checked at compile time
- The bad
  - Creating dynamic SQL statements is not as easy as in JDBC
  - More library dependency
- The Ugly
  - Oracle seems to be phasing out support for SQLJ, so you shouldn’t use it for any new applications