

Java DataBase Connectivity JDBC

Chrtistian Khoury

JDBC Lifeline

- 1996 : 1st version
- API that lets users connect to a database using « sequel » (SQL)
- JDBC is platform independent and vendor independent
- Many updates since (JDBC 4)

JDBC-Database Communication Path

- Java application uses the JDBC API
- JDBC Driver Manager uses the JDBC Driver API to talk to
 - JDBC/ODBC bridge that communicates with an ODBC Driver (used for testing purposes in the beginning)
 - Vendor supplied JDBC driver (mostly used)

JDBC Basics

- `java.sql` and `javax.sql` package
- For testing : use HSQLDB
- Database URLs
 - `jdbc:subprotocol:other stuff`
 - Ex: `jdbc:postgresql:myDB`

Making a Connection (1/7)

- Find out the names of the JDBC driver classes used by your vendor
 - Typical name
 - org.postgresql.Driver
 - com.mckoi.JDBCDriver
- Find the library in which the driver is located and
 - Launch with « -classpath » command line argument
 - Modify the CLASSPATH variable
 - Copy the database into the jre/lib/ext folder

Making a Connection (2/7)

- Register driver
 - Specify the property with a command line argument
 - `java -Djdbc.drivers=org.postgresql.Driver MyProg`
 - Or the application can set the property
 - `System.setProperty(« jdbc.drivers », « org.postgresql.Driver »)`
 - Or manually register a driver by loading its class
 - `Class.forName( « org.postgresql.Driver »);`

Making a Connection (3/7)

- After registering, you open a database connection with code similar to this one

```
String url = « jdbc:postgresql:MyDB »;
```

```
String username = « dbuser »;
```

```
String password = « secret »;
```

```
Connection conn =
```

```
    DriverManager.getConnection(url, username,  
    password);
```

Making a Connection (4/7)

- Better use a properties file

jdbc.drivers=org.postgresql.Driver

jdbc.url=jdbc:postgresql:MyDB

jdbc.username=dbuser

jdbc.password=secret

- Can be an XML file (see the Properties class and the loadXML/storeXML method)

Making a Connection (5/7)

```
public static Connection getConnection()
    throws SQLException, IOException
{
    Properties props = new Properties();
    FileInputStream in = new FileInputStream("database.properties");
    props.load(in);
    in.close();

    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null)
        System.setProperty("jdbc.drivers", drivers);
    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return DriverManager.getConnection(url, username, password);
}
```

Making a Connection (6/7)

```
public static void runTest()
    throws SQLException, IOException
{
    Connection conn = getConnection();
    try
    {
        Statement stat = conn.createStatement();

        stat.execute("CREATE TABLE Greetings (Message CHAR(20))");
        stat.execute("INSERT INTO Greetings VALUES ('Hello, World!')");

        ResultSet result = stat.executeQuery("SELECT * FROM Greetings");
        result.next();
        System.out.println(result.getString(1));
        stat.execute("DROP TABLE Greetings");
    }
    finally
    {
        conn.close();
    }
}
```

Making a Connection (7/7)

```
public static void main (String args[])
{
    try
    {
        runTest();
    }
    catch (SQLException ex)
    {
        while (ex != null)
        {
            ex.printStackTrace();
            ex = ex.getNextException();
        }
    }
    catch (IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Executing SQL Commands (1/)

- First, create a Statement object using the connection object
 - Statement stat = conn.createStatement();
 - String command = « a sql command »
 - Use *executeUpdate* or *executeQuery*
 - Update for INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE SQL commands
 - Query for SELECT

Executing SQL Commands (2/)

- *execQuery* return a *ResultSet*
 - Basic loop for analyzing a result set
`while (rs.next()) { handle a row from the result set }`
- When inspecting a row, you would like to read the contents
 - Accessors exist and each has two forms (column number or column name)
 - `rs.getDouble(1);`
 - `rs.getDouble(« price »);`

Executing SQL Commands (3/)

- CHAR String getString()
- VARCHAR String getString()
- LONGVARCHAR InputStream getUnicodeStream()
- NUMERIC BigDecimal getBigDecimal()
- DECIMAL BigDecimal getBigDecimal()
- BIT boolean getBoolean()
- TINYINT byte getByte()
- SMALLINT short getShort()
- INTEGER int getInt()
- BIGINT long getLong()
- REAL float getFloat()
- FLOAT double getDouble()
- DOUBLE double getDouble()
- BINARY byte[] getBytes
- VARBINARY byte[] getBytes

Executing SQL Commands (4/)

Some other types that can be fetched

- TIME `java.sql.Time`
- TIMESTAMP `java.sql.Timestamp`
- BLOB `java.sql.Blob`
- CLOB `java.sql.Clob`

Executing SQL Commands (5/)

- Prepared Statements

- Use it when you have a parametered query that is executed many times

- `java.sql.PreparedStatement`

- String stat = « SELECT price, title from books where name = ? »;*

- PreparedStatement prepStat = conn.prepareStatement(stat);*

- prepStat.setString(1, hisName); // bind parameter to value*

- prepStat.executeQuery();*

- // prepStat.executeUpdate(); // if that is an update*

Populating a Database (1/6)

```
/**
    Executes all SQL statements in a file.
    Call this program as
    java -classpath driverPath:. ExecSQL commandFile
 */
class ExecSQL
{
    public static void main (String args[])
    {
        try
        {
            Scanner in;
            if (args.length == 0)
                in = new Scanner(System.in);
            else
                in = new Scanner(new File(args[0]));

            Connection conn = getConnection();
```

Populating a Database (2/6)

```
try
{
    Statement stat = conn.createStatement();

    while (true)
    {
        if (args.length == 0) System.out.println("Enter command or EXIT to exit:");

        if (!in.hasNextLine()) return;

        String line = in.nextLine();
        if (line.equalsIgnoreCase("EXIT")) return;
        try
        {
            boolean hasResultSet = stat.execute(line); // generic method
            if (hasResultSet)
                showResultSet(stat);
        }
    }
}
```

Populating a Database (3/6)

```
catch (SQLException e)
{
    while (e != null)
    {
        e.printStackTrace();
        e = e.getNextException();
    }
}
}
```

Populating a Database (4/6)

```
finally
{
    conn.close();
}
}
catch (SQLException e)
{
    while (e != null)
    {
        e.printStackTrace();
        e = e.getNextException();
    }
}
catch (IOException e)
{
    e.printStackTrace();
}
}
```

Populating a Database (5/6)

```
public static void showResultSet(Statement stat)
    throws SQLException
{
    ResultSet result = stat.getResultSet();
    ResultSetMetaData metaData = result.getMetaData();
    int columnCount = metaData.getColumnCount();

    for (int i = 1; i <= columnCount; i++)
    {
        if (i > 1) System.out.print(", ");
        System.out.print(metaData.getColumnLabel(i));
    }
    System.out.println();
}
```

Populating a Database (6/6)

```
while (result.next())
{
    for (int i = 1; i <= columnCount; i++)
    {
        if (i > 1) System.out.print(", ");
        System.out.print(result.getString(i));
    }
    System.out.println();
}
result.close();
}
```

Scrollable and Updatable Result Sets

(1/4)

- What if you want to move forward and backwards
- *createStatement* and *prepareStatement*
Statement **createStatement**(int resultSetType, int resultSetConcurrency)
PreparedStatement **prepareStatement**(String sql, int resultSetType, int resultSetConcurrency)

Scrollable and Updatable Result Sets

(2/4)

- *resultSetType {TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, TYPE_SCROLL_SENSITIVE}*
- *resultSetConcurrency {CONCUR_READ_ONLY, CONCUR_UPDATABLE}*
- *TYPE_SCROLL_SENSITIVE = Scrolable and sensitive to database changes*
- *CONCUR_UPDATABLE = result set can be use to update the database*

Scrollable and Updatable Result Sets

(3/4)

- Some methods from the *ResultSet* class

next();

previous();

absolute(int);

relative(int);

updateXXX(int/String, value);

updateRow();

...

Scrollable and Updatable Result Sets

(4/4)

```
rs.absolute(5); // moves the cursor to the fifth row of rs
rs.updateString("NAME", "AINSWORTH"); // updates the NAME column of
    row 5 to be AINSWORTH
rs.updateRow(); // updates the row in the data source
```

```
rs.moveToInsertRow(); // moves cursor to the insert row
rs.updateString(1, "AINSWORTH"); // updates the first column of the insert
    row to be AINSWORTH
rs.updateInt(2,35); // updates the second column to be 35
rs.updateBoolean(3, true); // updates the third column to true
rs.insertRow();
rs.moveToCurrentRow();
```

Transactions (1/3)

- Set of database commands that form a single UNIT
 - should be committed when all has gone well
 - Rolled back otherwise
- Why transactions ?
 - Data integrity
 - Example : money transfer from one account to another

Transactions (2/3)

- By default
 - Each SQL command is a transaction
 - *autocommit mode*
- Turn off autocommit
 - `conn.setAutoCommit(false);`
- Commit at the end
 - `conn.commit();`
- If an error occurred
 - `conn.rollback();`
 - All commands until the last commit are automatically reversed

Transactions (3/3)

- Save points
 - `Savepoint svpt = conn.setSavePoint(« save point 1 »);`
 - `conn.rollback(svpt);`