IBD- Java Data-Base Connectivity

Marie-Christine Fauvet, Goran Frehse, Cyril Labbé

Université Joseph Fourier – UFR IM2AG

February 6, 2013

What is it about?[1]

How to implement an application which is:

- (not necessarily) distributed
- written in Java (or any other language)
- based on relational DBMS

[1] Thanks to Patrick Reignier and Philippe Genoud who gave me most of these slides.

Data persistence

Two opposite options:

- When programs end, all data are lost: programming language-based approaches.
- All data generated by programs are stored: database-based approach.

When a data (or an object) is made persistent it's value is stored in stable memory so it will be read in the future.

Relational DBMS

Hugely adopted...

- Relational model is based on mathematical foundations
- SQL has some good properties:
  - Declarative
  - Optimised
  - Standardised
- SQL covers data manipulation as well as data definition (including some aspects of physical data model)
- Relational technology is matured

but the relational model is sometimes too poor to meet application's needs

Object DBMS

- Rich extensible data model (composition, inheritance, operations associated with data)

but still inexpensively used...

- Object technology lacks of standard
- Technology still immatured in comparison with relational technology.

Object vs. Relational

Impedance mismatch...

- Object and Relation are two different paradigms

Relational model:

- Relation: sub-set of a cartesian product
- Relation value: tuples

Object model:

- Class: obtained by any construction of types (tuple, array, agregation, enumeration, inheritance..)
- Objects: class instances, associated with a behaviour.

Relation-Object matching could be tricky...

Outline

JDBC API

Preliminaries

JDBC Driver

Connecting to a Database

Issuing Statements

Retrieving Data with ResultSet

Exception Handling

Retrieving Meta Data

JDBC / Transaction

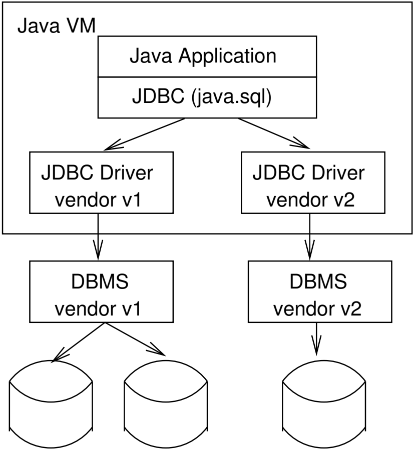
Concluding Remarks

Motivation

- Provides an API which conforms to SQL92 Standard
- Aims at:
  - Specifying data access independant from the DBMS
  - Submitting SQL queries

… and easy to use.

Principles



- Package java.sql: comes with JDK, provides classes' interface
- JDBC drivers: classes' implementation, specific to RDBMS

Outline

JDBC API

Preliminaries

JDBC Driver

Connecting to a Database

Issuing Statements

Retrieving Data with ResultSet

Exception Handling

Retrieving Meta Data

JDBC / Transaction

Concluding Remarks

JDBC Features

- Driver management: selection, load
- Database connection management: resource allocation
- Query execution: static and dynamic
- Result processing: SQL types-Java types matching
- Metadata: driver's properties, database's catalog

Loading Driver

To be used, the driver must first be registered:

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

Oracle documentation says: *When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager.*
So, it is better to dynamically load the class:

```
try {
    Class.forName("oracle.jdbc.OracleDriver").newInstance();
}
catch (ClassNotFoundException e) {
    ...
}
```

Doing so, it is possible to read the driver's name in a configuration file.

Outline

JDBC API

Preliminaries

JDBC Driver

Connecting to a Database

Issuing Statements

Retrieving Data with ResultSet

Exception Handling

Retrieving Meta Data

JDBC / Transaction

Concluding Remarks

Connecting to a Database

Piece of java code:

String user; String url; String password;
Connection conn = DriverManager.getConnection(url, user, password);

where:

- url identifies the resource (Uniform Resource Locator). It is different depending on the DBMS.
  With Oracle it looks like *jdbc:oracle:thin:@serveur:port:database*.
  For instance:
  jdbc:oracle:thin:@hopper.e.ujf-grenoble.fr:1521:ufrima
- user is the user's login name
- passwor is the user's password

/oracle/jdbc/lib/ojdbc14.jar is the driver we are using.

A Typical Session

When a connection is no longer useful, we need to close it explicitly:

```
try {
    Connection conn = DriverManager.getConnection(url, user, passwd);
    ...
    // work with the database
    ...
    // close the connection
    conn.close();
} catch (SQLException e) {
    ...
}
```

What happens when an exception is raised before conn.close() is reached?

Closing a Connection

```
Connection conn = null;
try {
    conn = DriverManager.getConnection(url, user, passwd);
    ... // work with the database
    conn.close(); // close the connection
}
catch (SQLException e) {
    ...
}
finally {
    try {
        if (conn != null) conn.close();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Outline

JDBC API

- Preliminaries
- JDBC Driver
- Connecting to a Database
- Issuing Statements
- Retrieving Data with ResultSet
- Exception Handling
- Retrieving Meta Data
- JDBC / Transaction
- Concluding Remarks

Preparing and Submitting Queries and Updates

- Assumption: a connection has been created
- 3 types of queries
  - Statement: basic query
  - PreparedStatement: pre-compiled query
  - CallableStatement: call of an embedded procedure
- 3 types of executions
  - executeQuery to submit a query which returns data
  - executeUpdate for a query which does not return date (e.g. INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE, ...)
  - execute to execute an embedded procedure

Example : A Query which Results in Data Set

```
...
Vector<String> res = new Vector<String>();
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
String query = null ;
// open the connection to define conn ...
stmt = conn.createStatement();
query = "select distinct nomS from LesSpectacles order by nomS";
rs = stmt.executeQuery(query);
while (rs.next()){
    res.addElement(rs.getString(1));
}
// close the connection: rs, stmt, conn
```

A Comprehensive Example

```
import java.sql.*;
public class TestJDBC {
    public static void main(String[] args) throws Exception {
        try {
            DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(”SELECT * from employe”);
            while (rs.next()) {
                String nom = rs.getString(”nom”);
                String prenom = rs.getString(”prenom”);
                String email = rs.getString(”email”);
            }
        } catch (...) { }
        finally { ... }
    }
}
```

Prepared Statements

- compiled and prepared beforehand, so it can be executed faster, and might even be reused,
- can take parameters,
- less error prone with data conversions.
- Use if a query is run multiple times and only the values of the same columns change.

Example : PreparedStatement with ResultSet

```
...
Vector<String> res = new Vector<String>();
Connection conn = null;
PreparedStatement pstmt = null;
ResultSet rs = null;
int number_of_seats = 500;
// open the connection to define conn ...

pstmt = conn.prepareStatement(”select distinct nomS ” +
    ”from LesSpectacles where numS > ? order by nomS”);
pstmt.setInt(1,number_of_seats); // includes type checking
rs = pstmt.executeQuery();

while (rs.next()){
    res.addElement(rs.getString(1));
}
// close the connection: rs, stmt, conn
```

Example : PreparedStatement with Update

```
...
Connection conn = null;
PreparedStatement pstmt = null;
String old_name = "Cats";
String new_name = "Hair";
// open the connection to define conn ...

pstmt = conn.prepareStatement("update LesSpectacles set " +
   "nomS = ? where nomS = ?");
pstmt.setString(1,new_name); // includes type checking
pstmt.setString(2,old_name); // includes type checking
pstmt.executeUpdate();

// close the connection: rs, stmt, conn
```

Outline

JDBC API

- Preliminaries
- JDBC Driver
- Connecting to a Database
- Issuing Statements
- Retrieving Data with ResultSet
- Exception Handling
- Retrieving Meta Data
- JDBC / Transaction
- Concluding Remarks

More about ResultSet

getXXX() methods to access values of type XXX for a row in a ResultSet:

```
ResultSet rs = stmt.executeQuery(”SELECT a, b, c FROM Table1”);
while (rs.next()) {
    int i = rs.getInt(”a”); // rs.getInt(1);
    String s = rs.getString(”b”); // rs.getString(2);
    byte b[ ] = rs.getBytes(”c”); // rs.getBytes(3);
    System.out.println(”ROW = ” + i + ” ” + s + ” ” + b[0]);
}
```

Incorrect use of ResultSet

```
Statement stmt = conn.createStatement();
ResultSet rs1 = stmt.executeQuery(myQuery1);
ResultSet rs2 = stmt.executeQuery(myQuery2);
while (rs1.next()) {
    ....
    while (rs2.next()) {
        .... }
    ...
}
```

Correct use:

```
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
ResultSet rs1 = stmt1.executeQuery(myQuery1);
ResultSet rs2 = stmt2.executeQuery(myQuery2);
while (rs1.next()) {
    ....
    while (rs2.next()) {
        .... }
```

Outline

JDBC API

- Preliminaries
- JDBC Driver
- Connecting to a Database
- Issuing Statements
- Retrieving Data with ResultSet
- Exception Handling
- Retrieving Meta Data
- JDBC / Transaction
- Concluding Remarks

SQL Exceptions

- require a try/catch block like any other Exception
- describe database or driver errors
- methods
  - getMessage(): error message
  - getSQLState(): SQLState identifier (e.g., 22012 = division by zero)
  - getErrorCode(): vendor-specific error code (e.g., ORA-01476 = division by zero)
  - getNextException(): retrieve the next SQLException or null if there are no more
  - setNextException(SQLException ex): allows the programmer to add an SQLException to the chain.

Handling Exceptions

```
try
{
   // some DB work
} // end try
catch ( SQLException ex)
{
   // do handling
   // (retry connection, abort, rollback,...)
} // end catch
```

Handling Batches of Exceptions

```
try {
    // some DB work
} // end try
catch ( SQLException ex) {
    while( ex != null) {
        // do handling

        ex = ex.getNextException();
    }
} // end catch
```

Example: A Query with Exception Handling

```
...
Vector<String> res = new Vector<String>();
Connection conn = null; Statement stmt = null;
ResultSet rs = null; String query = null ;
try {
    conn = BDConnexion.getConnexion();
    stmt = conn.createStatement();
    query = "select distinct nomS from LesSpectacles order by nomS";
    rs = stmt.executeQuery(query);
    while (rs.next()){
        res.addElement(rs.getString(1));
    }
} catch (ExceptionConnexion e) {
    throw new ExceptionSpectacles ("Erreur connexion : "+e.getMessage(),e);
} catch (SQLException e) {
    throw new ExceptionSpectacles ("Erreur oracle : "+e.getMessage(),e);
} catch (Exception e) {
    throw new ExceptionSpectacles (" BDSpectacles : autre erreur "
        + e.getMessage(),e);
```

Exceptions - Best Practices

- Do not catch Exception, since it will also catch all RuntimeExceptions (NullPointerException,...)

```
try {
    // ...
} catch (Exception ex) { // bad, too general
    // ...
}
try {
    // ...
} catch (SQLException ex) { // good, since specific
    // ...
}
```

Exceptions - Best Practices

- Do not suppress or ignore exceptions.

```
try {
    // ...
} catch (SQLException ex) {
} // bad, we did nothing
```

Exceptions - Best Practices

- If an error cannot be fixed completely, use chained exceptions to add information and pass on the exception.

```
try {
    // retrieving student names ...
} catch (SQLException ex) {
    String msg = "Could not retrieve student names";
    throw new RetrieveStudentNameException(msg,ex);
}
```

Exceptions - Best Practices

- Close ResultSet, Statement and Connection objects in finally bock.

```
Connection conn = null; Statement stmt = null; ResultSet rs = null;
try {
    // JDBC Connection/Statement/ResultSet
}
catch (SQLException sqlEx) {
    // Handle the exception
}
finally {
    try {
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Outline

JDBC API

Preliminaries

JDBC Driver

Connecting to a Database

Issuing Statements

Retrieving Data with ResultSet

Exception Handling

Retrieving Meta Data

JDBC / Transaction

Concluding Remarks

Meta Data

Metadata is data (or information) about data.

- Database Metadata: Information about the database.
- ResultSet Metadata: Information about the result of a query.

Database Meta Data

- available if there is a valid connection

```
int dver;
DatabaseMetaData dbmd = con.getMetaData();
dver = dbmd.getDriverVersion();
System.out.println(”Driver version: ” + dver);
```

Example: Database Meta Data

Discovering primary keys:

```
ResultSet primaryKeys = databaseMetaData.getPrimaryKeys(null, null,”Students”);
while (primaryKeys.next())
{
   String primaryKeyColumn = primaryKeys.getString(”COLUMN_NAME”);
   System.out.println(”Primary Key Column: ” + primaryKeyColumn);
}
```

ResultSet Meta Data

- number of columns returned,
- column names,
- column types,
- precision and scale of numbers,
- ...

Not all DBMS provide this information, so check for empty or null return values!

Example: ResultSet Meta Data

```
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(”select * from Students”);
ResultSetMetaData resultSetMetaData = resultSet.getMetaData();

int nCols = resultSetMetaData.getColumnCount();
System.out.println(”Number of columns: ” + nCols);
```

Example: Listing all Column Names and Types

```
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(”select * from Students”);
ResultSetMetaData resultSetMetaData = resultSet.getMetaData();

int nCols = resultSetMetaData.getColumnCount();
for (int i=1; i¡=nCols; i++)
{
    String columnName = resultSetMetaData.getColumnName(i);
    String columnTypeName = resultSetMetaData.getColumnTypeName(i);
    System.out.println(columnName + ”: ” + columnTypeName);
}
```

Outline

JDBC API

- Preliminaries
- JDBC Driver
- Connecting to a Database
- Issuing Statements
- Retrieving Data with ResultSet
- Exception Handling
- Retrieving Meta Data
- JDBC / Transaction
- Concluding Remarks

Commit Mode

- When a connection is created using JDBC, by default it is in auto-commit mode.
- Each SQL statement is a transaction and is automatically committed immediately after it is executed.
- To allow two or more statements to be grouped into a transaction you need to disable auto-commit mode:
  conn.setAutoCommit(false);
- no SQL statement will be committed until the commit method is called.
- The entire set of statements can be rolled back, without committing.

Start / End of a transaction

- Start / End :
  First call of
  conn.setAutoCommit(false)
  Each call of
  conn.commit()
  implicitly mark the start of a transaction.
- Transactions can be undone before they are committed by calling :
  conn.rollback()

JDBC Isolation level

Transaction isolation levels you can use in JDBC:

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	NS	-	-	-
TRANSACTION_READ_UNCOMMITTED	S	A	A	A
TRANSACTION_READ_COMMITTED	S	P	A	A
TRANSACTION_REPEATABLE_READ	S	P	P	A
TRANSACTION_SERIALIZABLE	S	P	P	P

- NS: Not Supported
- S: Supported
- P: Prevented
- A: Allowed

Set JDBC Isolation level

- The default transaction isolation level depends on your DBMS.
- To find out what transaction isolation level your DBMS is set to : conn.getTransactionIsolation()
- To set it to another level: conn.setTransactionIsolation()

Example

```
Connection connection = null;
try {
Class.forName(”...”);
connection = DriverManager.getConnection(”...”);

connection.setAutoCommit(false);

Statement statement = connection.createStatement();

statement.executeUpdate(”UPDATE Table1 SET Value = 1
WHERE Name = ’foo’”);

connection.commit();
} catch (SQLException ex) {
connection.rollback();
}
```

Outline

JDBC API

- Preliminaries
- JDBC Driver
- Connecting to a Database
- Issuing Statements
- Retrieving Data with ResultSet
- Exception Handling
- Retrieving Meta Data
- JDBC / Transaction
- Concluding Remarks

Going further...

jdbc:

- Managing connections with data sources
- JDBC Data access optimisation
- ...

Mastering multi-tiers architecture:

- Who is in charge of what?
- Good practices: MVC Models
- Frameworks can help: hibernate, struts,...

Outline

Database Design

- Approach & Concepts
- Mapping O/R

Design Approach.

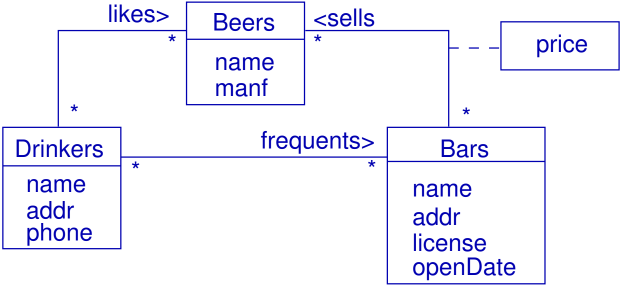
5 stages :

1. perceiving the real world
2. drawing up the conceptual schema (ER, UML,...)
3. designing the DB schema (logical)
4. refining the DB schema (logical)
5. drawing-up the physical schema

Basic concepts.

- *Property*: basic information (attributes)
- *Class*: individual/object type
  - E.g.: CUSTOMER with a name and address,
  - ITEM with a unit price, a part number.
- *Association*: represents an association between several entities/objects. Its existence depends on the related objects.

Example: conceptual schema



Outline

Database Design

Approach & Concepts

Mapping O/R

Mapping object to relational

The relational model and UML class diagram have some obvious correspondences:

Relational Model	*UML Class Diagram*
Attributes	Attributes
Relation schema	Classes and Associations
Tuple	Object value

Some major differences:

1. Relations are used to model both classes and associations
2. Relational model does not support object-oriented notions (such as subclasses, inheritance)
3. Most of object-oriented models don't offer any features to capture identifiers (apart from internal oid).

Mapping Classes

For classes that are neither abstract (nor virtual), nor associative, nor association class:

- One relation for each class
- Relation attributes are class attributes (more attributes are likely to be added in the future).

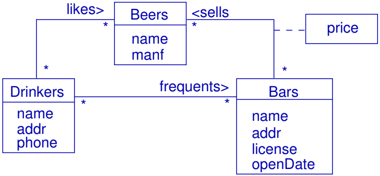
Mapping Associations

Process associations in order:

1. cardinality 1 (foreign key)
2. cardinality 0..1 (new relation)
3. cardinality * (new relation)

Example: mapping O/R

Drinkers ($\underline{\text{name}}$, addr, phone) { *name is the key.* }
Beers ($\underline{\text{name}}$, manf)
Bars ($\underline{\text{name}}$, addr, license, openDate)
Likes ($\underline{\text{drinker, beer}}$) { *drinker,beer is the key.* }
$\pi_{\text{drinker}}(\text{Likes}) \subseteq \pi_{\text{name}}(\text{Drinkers})$
$\pi_{\text{beer}}(\text{Likes}) \subseteq \pi_{\text{name}}(\text{Beers})$
Sells ($\underline{\text{bar, beer}}$, price)
price > 0
$\pi_{\text{beer}}(\text{Sells}) \subseteq \pi_{\text{name}}(\text{Beers})$
$\pi_{\text{bar}}(\text{Sells}) \subseteq \pi_{\text{name}}(\text{Bars})$
Frequents ($\underline{\text{drinker, bar}}$)
$\pi_{\text{drinker}}(\text{Frequents}) \subseteq \pi_{\text{name}}(\text{Drinkers})$
$\pi_{\text{bar}}(\text{Frequents}) \subseteq \pi_{\text{name}}(\text{Bars})$


likes>
Beers
name
manf
<sells
price
Drinkers
name
addr
phone
frequents>
Bars
name
addr
license
openDate
*
*
*
*
*
*