Introduction to JDBC

A framework for accessing and manipulating (tabular) data stored in a relational database

- The API is independent of
  - machine architecture
  - database used
  - Java virtual machine

- The API is not independent of the database access language
  - JDBC relies on SQL (SQL-92)

- JDBC does not provide totally transparent database access

JDBC Architecture

- Client Application
  - the application that is accessing the DB

- Driver
  - the “bridge” between the client and the DB
  - vendor-specific
  - sends the client requests to the server (after possibly some processing) and presents the results to the client

- DriverManager
  - manages the different drivers that can co-exist in the same client

- Database Server
  - the DB engine that supports the application
  - located most likely on a different machine than the client

JDBC Architecture Types (i)

Type 3
- Pure Java JDBC Driver
- DB Server

Type 4
- Pure Java JDBC Driver
- DB Server

Application / Java Applet
- JDBC API
- JDBC Driver Manager
- DB Middleware
JDBC Architecture Types (ii)

Write Once, Run Everywhere?

- As we've seen
  - Java is platform independent, and
  - JDBC is also platform and database independent
- Then, it follows
  - code that uses JDBC is also platform and database independent,
  - ... Right?
- Well...
  - SQL is not totally standardised over all platforms
    - lots of vendor-specific features and extensions
  - to be JDBC-compliant, a driver should implement the whole of the ANSI SQL-92 standard
  - this does not prevent it to understand vendor-specific extensions
  - lowest common denominator (SQL-92) should be re-usable

JDBC Versions — JDBC 1.0

- One of the very first defined Java APIs
- Simple facilities
  - connect to a database via an appropriate driver
    - Driver, DriverManager, Connection
  - construct SQL statements to query or update the database
    - Statement
  - retrieve and extract the results
    - ResultSet

JDBC Versions — JDBC 2.0

- 2.1 is the latest “official” release
- Split into
  - Core API
  - Optional Package
- Compared to 1.0, the Core API has been extended to include
  - scrollable result sets
  - batch updates
  - performance hints
  - support for Unicode characters
  - etc.
- The JDBC Optional Package (or Standard Extension API) includes new facilities targetted for high-performance, heavy-duty, server-side applications
JDBC Versions — JDBC 3.0

- Currently in draft form (4th draft), under review
  - planned to be included in Java 1.4
- Unifies Core API and Optional Package and adds more functionality

JDBC API

- JDBC classes/interfaces are included in the `java.sql` package
- Any errors are indicated by an `SQLException`
- For clarity, all `try/catch` blocks are omitted from most of the code in this lecture
  - This does not mean you do not need to use them!
- You do not need to run on the machine that has the Oracle system on (i.e. `crooked`) to use JDBC and access the DB
  - you can run the Java applications on any lab machine
  - client-server model, remember?

Concrete Example

- The following table will be used to illustrate the basic facilities of JDBC

<table>
<thead>
<tr>
<th>MID</th>
<th>Title</th>
<th>Year</th>
<th>Explosions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>StarWars</td>
<td>1977</td>
<td>3,653,543</td>
</tr>
<tr>
<td>56</td>
<td>BladeRunner</td>
<td>1982</td>
<td>3,203</td>
</tr>
<tr>
<td>75</td>
<td>Aliens</td>
<td>1986</td>
<td>343,400</td>
</tr>
<tr>
<td>98</td>
<td>Junior</td>
<td>1994</td>
<td>0</td>
</tr>
<tr>
<td>123</td>
<td>Pocahontas</td>
<td>1995</td>
<td>0</td>
</tr>
</tbody>
</table>

Creating a Database

- Creating a database is not a standard feature of JDBC
- Some drivers support it, some not
- Typically, the database is created by the database administrator
  - This refers to initialising the database structures
    - not creating the table...
Connecting to the Database — Step 1

- **Step 1:** Load the appropriate JDBC driver
  - the driver is vendor-specific
  - therefore its name is also vendor-specific
- For Oracle in the Dept
  
  ```java
  Class.forName("oracle.jdbc.driver.OracleDriver");
  ```

  (remember to deal with the `ClassNotFoundException`)

  - Your (runtime) `CLASSPATH` should include
    `/users/students4/software/ojdbc14.jar`

Connecting to the Database — Step 2 (i)

- **Step 2:** Establish the connection
  - requires a specific type of URL to find the DB server
  - this URL is JDBC and vendor specific
- We establish the connection with the `getConnection` static method of the `DriverManager` class
  ```java
  Connection conn = DriverManager.getConnection(connectionString);
  ```
- It returns an object that implements the `Connection` interface
  - this object represents this particular connection

Connecting to the Database — Step 2 (ii)

- For Oracle in the Dept, `connectionString` is something like
  ```
  "jdbc:oracle:thin:" + USER_NAME + "/" + PASSWD + 
  "@crooked.dcs.gla.ac.uk:1521:lev3"
  ```

  - where `USER_NAME` and `PASSWD` are your Oracle user name and password, NOT the Unix ones!
  - the `DriverManager` will determine from the connection string which driver to use
    - as multiple drivers can be loaded at the same time
- The format of the connection string might be different for other drivers

Creating a Table (i)

- We now want to execute the following SQL statement
  ```sql
  CREATE TABLE Movies (
      MID INTEGER NOT NULL,
      Title VARCHAR(30) NOT NULL,
      Year INTEGER NOT NULL,
      Explosions INTEGER NOT NULL,
      PRIMARY KEY (MID)
  );
  ```

  that creates the table `Movies` in the database
Creating a Table (ii)

- First, we need to create a new `Statement` object, associated with the connection we have already established.

```java
Statement stmt = connection.createStatement();
```

- Then, we can execute the statement by invoking the `executeUpdate` method (creating a table actually updates the database).

```java
stmt.executeUpdate("CREATE TABLE Movies (" +
                  "MID INTEGER NOT NULL," +
                  "Title VARCHAR(30) NOT NULL," +
                  "Year INTEGER NOT NULL," +
                  "Explosions INTEGER NOT NULL," +
                  "PRIMARY KEY (MID)" +
                ")");
```

Executing Updates — Some Notes

- Depending on the SQL statement used, `executeUpdate` performs any update, not only table creation.
- The string containing the SQL statement was broken up for clarity.
  - It is not necessary to break it up.
  - It is good practice though as it looks tidier.
- Notice that no terminator is included at the end of the statement.
  - This is vendor-specific.
  - The JDBC driver deals with it.

Populating the Table (i)

- We now want to populate the table with some values.

```java
INSERT INTO Movies VALUES ( 1, 'StarWars', 1977, 3653543 );
INSERT INTO Movies VALUES ( 56, 'BladeRunner', 1982, 3203 );
INSERT INTO Movies VALUES ( 75, 'Aliens', 1986, 343400 );
INSERT INTO Movies VALUES ( 98, 'Junior', 1994, 0 );
INSERT INTO Movies VALUES ( 123, 'Pocahontas', 1995, 0 );
```

Populating the Table (ii)

- Again, we can do it in a simple way with `executeUpdate`.

```java
stmt.executeUpdate("INSERT INTO Movies VALUES " +
                   ":( 1, 'StarWars', 1977, 3653543 )" +
                   
stmt.executeUpdate("INSERT INTO Movies VALUES " +
                   ":( 56, 'BladeRunner', 1982, 3203 )" +
                   
    ...;
```

This is a bit tedious though!
Populating the Table (iii)

Why, don’t we create a method to add a movie?

```java
void addMovie(Statement stmt,
              int mid, String title,
              int year, int explosions) {
    stmt.executeUpdate("INSERT INTO Movies VALUES " +
                      "(" + mid + ", " +
                      ", " + title + ", " +
                      ", " + year + ", " +
                      ", " + explosions + ")");
}
```

We can then call it after we read user input, iterate over an array, read a file containing the data, etc.

Querying the Table (i)

We now want to perform a query on the Movies table

“We which movies have more than 100 explosions?”

The SQL for it is

```sql
SELECT * FROM Movies WHERE Explosions > 100;
```

We now need to use `executeQuery` to perform the query (no updates this time!)

```java
ResultSet results =
    stmt.executeQuery("SELECT * " +
                      "FROM Movies " +
                      "WHERE Explosions > 100");
```

Querying the Table (ii)

Notice that `executeQuery` returns an object that implements the `ResultSet` interface

- this contains the results of the query

The facilities (methods) that `ResultSet` provides are quite elaborate

- read the API documentation!

However, a few useful ones are

- `int getInt (String columnName)`
- `String getString (String columnName)`
  - return the value of the specified column for the current row in the specified format
- `boolean next ()`
  - determines whether the result set has another row and, if it does, it moves to it

Querying the Table (iii)

Example usage of `ResultSet`

```java
ResultSet results =
    stmt.executeQuery("SELECT * " +
                      "FROM Movies " +
                      "WHERE Explosions > 100");
    while (results.next()) {
        String title = results.getString("Title");
        int year = results.getInt("Year");
        System.out.println(title + " " + year);
    }
```
Handling Errors (i)

- Any JDBC call will throw an SQLException to indicate an error.
- Such exceptions must be caught and dealt with.

```java
try {
    // do some JDBC calls
} catch (SQLException e) {
    e.printStackTrace();
    System.exit(-1);
}
```

...or show the error in a window, in the case of a GUI!

Handling Errors (ii)

Connection could not be established

```
java.sql.SQLException: The Network Adapter could not establish the connection
    at java.lang.Throwable.fillInStackTrace(Native Method)
    at java.lang.Throwable.fillInStackTrace(Compiled Code)
```

Duplicate primary key

```
java.sql.SQLException: ORA-00001:
    unique constraint (L32001_TONY.SYS_C00216118) violated
    at java.lang.Throwable.fillInStackTrace(Native Method)
    at java.lang.Throwable.fillInStackTrace(Compiled Code)
```

etc.

Putting It All Together

```java
static public void main (String args[]) {
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) { /* deal with it */ }
    try {
        String connString = "jdbc:oracle:thin:" + USER_NAME + "/" + PASSWD + "@crooked.dcs.gla.ac.uk:1521:lev3";
        Connection conn = DriverManager.getConnection(connString);
        Statement stmt = conn.createStatement();
        ResultSet results = stmt.executeQuery("SELECT * FROM Movies WHERE Explosions > 100");
        while (results.next()) {
            String title = results.getString("Title");
            int year = results.getInt("Year");
            System.out.println(title + " " + year);
        }
    } catch (SQLException e) { /* deal with it */ }
}
```

More On Statements

- Statements executed with `executeUpdate` and `executeQuery` on the Statement interface are parsed and checked dynamically, e.g.

  ```java
  ResultSet results = stmt.executeQuery("SELECT * FROM Movies");
  ```

- Every time this will be invoked, the statement will be parsed, checked (for syntax, consistency, etc.), and executed

- This is why you can generate the SQL string at runtime

```java
int target;
...
ResultSet results = stmt.executeQuery("SELECT * FROM Movies " +
    "WHERE Explosions > " + target);
```
Prepared Statements

- Sometimes, we want to perform the same query several times
  - parsing and checking complex queries is not very efficient
  - why do we need to have to parse them every time?
- Prepared Statements
  - PreparedStatement is a subinterface of Statement
  - created with the prepareStatement method on Connection
  - the SQL statement is registered with the DB once
    - i.e. compiled or prepared by the DB
  - then, it can be used without needing to be parsed again
  - less dynamic compared to Statement
    - after it's been registered, the SQL cannot change
  - both updates and queries are supported

Using Prepared Statements — Queries

- Using a "standard" Statement
  ```java
  Statement stmt = connection.createStatement();
  ResultSet results = stmt.executeQuery
  ("SELECT * FROM Movies WHERE Explosions > 100");
  ```
- Using a PreparedStatement
  ```java
  PreparedStatement pstmt = connection.prepareStatement
  ("SELECT * FROM Movies WHERE Explosions > 100");
  ```
  ... ResultSet results = pstmt.executeQuery();
- The two approaches are equivalent
  - Statement will be parsed by every executeQuery
  - PreparedStatement will be parsed once by prepareStatement
    and then only executed by every executeQuery

Using Prepared Statements — Updates

- Same idea as queries
  ```java
  PreparedStatement pstmt = connection.prepareStatement
  ("INSERT INTO Movies VALUES " +
  "( 1, 'StarWars', 1977, 3653543 )");
  ```
  ... pstmt.executeUpdate();
- Notice however that executing a prepared statement that always adds
  the same row to a database is not particularly useful!
  - it would be nice if we could parametarise it...

Parameterised Prepared Statements (i)

- It turns out that prepared statements can be parameterised
- If you want to introduce "arguments", introduce a ? inside the SQL statement
  - before executing the statement you have to explicitly specify what the
    values of the "arguments" will be
    - i.e. what the ?s should be replaced with
  - there are methods on PreparedStatement that do this
    - setInt to set an int argument
    - setString to set a String argument
    - ...
- You can have more than one ? inside the same statement
Parameterised Prepared Statements (ii)

- Let's revisit the "add a movie to the DB" example

```java
void addMovie(PreparedStatement pstmt,
        int mid, String title,
        int year, int explosions) {
    pstmt.setInt(1, mid);
    pstmt.setString(2, title);
    pstmt.setInt(3, year);
    pstmt.setInt(4, explosions);
    pstmt.executeUpdate();
}
...
PreparedStatement pstmt = connection.prepareStatement
        ("INSERT INTO Movies VALUES ( ?, ?, ?, ? )");
addMovie(pstmt, 1, "StarWars", 1977, 3653543);
addMovie(pstmt, 56, "BladeRunner", 1982, 3203);
...
```

Parameterised Prepared Statements (iii)

- Same for queries

```java
PreparedStatement pstmt = conn.prepareStatement
        ("SELECT * FROM Movies WHERE Explosions > ?");
pstmt.setInt(1, 0);
ResultSet results1 = pstmt.executeQuery();
// do something with results1
pstmt.setInt(1, 5000);
ResultSet results2 = pstmt.executeQuery();
// do something with results2
pstmt.setInt(1, 1000000);
ResultSet results3 = pstmt.executeQuery();
// do something with results3
```

JDBC Resources on the WWW

- Sun's JDBC Homepage
- JDBC Overview
- Getting Started with the JDBC API
- JDBC API Documentation
- Links to these (and a few other) sites here:
  [http://www.dcs.gla.ac.uk/~tony/teaching/db3](http://www.dcs.gla.ac.uk/~tony/teaching/db3)

Books


- Both cover JDBC 2.0. They are not required. But do have a look at them if you happen to come across them.