Internet Computing Workshop

Part 1:
Introducing the course
JDBC
Behzad BORDBAR
Contents

• Introducing the module
  – Roadmap of the course
  – Lab sessions
  – Assessments

• Part 1: JDBC
  – JDBC: a review
  – Transactions
Introducing the course

Syllabus for the course at

• You can read learning outcome

Myself:
• Behzad Bordbar
• My team works on:
• Model-based software engineering: UML, Web services, fault-tolerance…
A high level view of the course

• Aim: to learn engineering of multi-tired web based systems
Road map of the course

Learn the following technologies and how to make them work alongside each other

• JDBC: connect to a database
• Hibernate: object relational mapping
• Spring framework: security of transactions
• Wicket: creating web application front end (stateful, separation of presentation from logic,…)

This is a programming course; architecture and theory will not be covered-DS module
Roadmap .. continue

Two lectures per week Monday and Friday
Four demonstrators- only available during the lab session:
• LG04- Wednesdays 2-4 and Fridays 2-5
Start working on exercises on Mon and attend ALL session labs
Exercises are not marked, they are to assist you in learning
Module page has copies of slides, source code, further explanation:
http://www.cs.bham.ac.uk/~bxb/MainPage/Teaching.html
Assessment

• Assessment sessions to be held at LG04

Five assessments:
• Assessment 1: JDBC 10% of the marks
• Assessment 2: Hibernate 20% of the marks
• Assessment 3: Hibernate and Spring (Acegi) 20% of the marks
• Assessment 4: Wicket 20% of the marks
• Assessment 5: Hibernate Spring Wicket 30% of the marks
Acknowledgement

With special thank to Alan Sexton for designing and organising this course. My slides are based on the notes provided by him in 2008-09. I strongly encourage you to make use of his notes and sample codes

Part 1 JDBC

We assume following installed on your machines:

- Java (at least 6)
- PostgreSQL

However, very little of what follows is specific to PostgreSQL and almost no changes are required, except for the connection details, for Microsoft Access or MySQL and Apache Derby (embedded database)
PostgreSQL

PostgreSQL is free, open source multi OS DBMS

If installing PostgreSQL on your own machine, the version of the JDBC driver and PostgreSQL must match. See http://jdbc.postgresql.org/download.html.
PostgreSQL

• Linux: almost always comes with the distribution. For ubuntu see
  https://help.ubuntu.com/community/PostgreSQL
• Windows: http://pginstaller.projects.postgresql.org
• Macs: Different ports of PostgreSQL to OSX:
  http://www.macports.org/
  http://www.postgresqlformac.com
  http://www.entropy.ch/software/macosx/postgresql

Not being a mac user, I have not tried any!
Any experience?
**IDEs**


Choose an IDE and learn how to use it properly

I use Eclipse (installed on the School machines)

- For your copy go to [www.eclipse.org/downloads/](http://www.eclipse.org/downloads/), look for the link for “Eclipse IDE for Java Developers”
- Lars Vogel has a good tutorial starting from basics, see [http://www.vogella.de/articles/Eclipse/article.html](http://www.vogella.de/articles/Eclipse/article.html)
- Mark Dexter has recorded six elementary lectures (each 15 mins). See [http://ecliptutorial.sourceforge.net/workbench01/lesson01.html](http://ecliptutorial.sourceforge.net/workbench01/lesson01.html) for the first lecture.
Running example:
A (seriously) simplified DB for handling on-line book orders
Running example:… continue

A Customer can have many Orders, but each Order belongs to precisely one Customer.

An Order can have many OrderDetail records, but each OrderDetail belongs to precisely one Order.

Each OrderDetail record refers to precisely one Book. However there may be many different OrderDetail records (spread over different Orders), that refer to the same Book.

All the primary keys ID fields are automatically generated integers and therefore the primary key of each table does not need to be included when inserting new records.
Referential integrity rules are maintained on all foreign keys (ID fields which not in the first position in the tables of the figure above). Hence, for example, it will cause an error to try to insert an Order record with an Orders.CustomerID for which there is no corresponding Customers record.

OrderDetail.quantity is an integer.

Books.BookPrice and OrderDetails.Charge are intended to hold monetary amounts and hence are represented in SQL as type DECIMAL. This type is handled in Java via the java.math.BigDecimal class.
Running example:...

custFirstName, custLastName, custAddress, custEmail, bookISBN, bookName, orderStatus are Strings.

- Orders.orderDate is a Timestamp object in Postgresql (use a Date/Time in Access) which is handled in Java as a java.sql.Timestamp object.

Timestamp objects can be set to the current time:

```java
long millisecs = System.currentTimeMillis();
Timestamp ts = new java.sql.Timestamp(millisecs);
```

**TODO:** check out Timestamp API
Don’t use float/double for money!

For monetary amounts **never** use *float* or *double* because of the issues with round-off and inaccuracy of translating between a floating representation and a decimal one. Remember: 2.1 in double is greater than 2.1 in float!

Instead use SQL type *DECIMAL, NUMERIC, MONEY, CURRENCY* or something similar.
Sample database

Script `shop_create.sql` is available for your use to set up the shop database in PostgreSQL (use the '\i' command in psql to import the script).

Script, `shop_drop.sql` to delete all the tables and sequences created by the creation script so that you can easily clear out your data and start again.

TODO: Review your database notes and learn how to do the above.
Java application interacting with DB via JDBC

Two steps: make connection, use it!

• Application asks for the driver to be loaded
• Application asks DriverManager for a connection to a particular database
• DriverManager asks all loaded drivers, and one of them responds with connection.
• DriverManager gives connection to the Java application… Connection is used
public class ApplicationMakesConnectionViaJDBC {
    public static void main(String[] args) {
        // first Register the driver
        try {
            Class.forName("org.postgresql.Driver");
        }
        catch (ClassNotFoundException e) {
            // Can't load JDBC drive, make sure classpath is correct
            e.printStackTrace();
        }
        System.out.println("Driver loaded- continue ");
    }
}
// Let us set the url, my localmachine
String url="jdbc:postgresql://localhost/bxb";
// If using Schools database, use the following
// String url="jdbc:postgresql://dbteach/DBNAME";
String username ="bxb";
String password ="XYZpassword";

Very naughty! Just for training purposes…
Sample code for beginners...

```java
// start the connection
Connection connection = null;
try {
    connection = DriverManager.getConnection(url, username, password);
} catch (SQLException e) {
    "We have problems with the connection"
    e.printStackTrace();
    System.exit(1);
}
//Connection seems to be OK
//follow with the code for using connection";
```
Sample code for beginners..continue

**Sample code**: available at jumpstartJDBC/ApplicationMakesConnectionViaJDBC.java for your closer inspection.

Username and passwd can be passed through a JDialog box.

**Sample code**: jumpstartJDBC/ConnectionWithUsernamePasswdFromDialogbox.java. Don't forget to modify the “url”

Database details can be read through a property file
Properties file: Database.properties

# Properties file for a local PostgreSQL JDBC connection
jdbc.drivers=org.postgresql.Driver
database.name=jdbc:postgresql://localhost/<database-name>
database.user=<username>

#Keys are in red.
# Properties file: Database.properties

```properties
jdbc.drivers=org.postgresql.Driver
database.name=jdbc:postgresql://db teach/<database-name>
database.user=<username>
```

TODO: follow instruction in [http://supportweb.cs.bham.ac.uk/servers/postgres/](http://supportweb.cs.bham.ac.uk/servers/postgres/) to get a database setup
Using properties file

```java
public static final String PROPERTIES_FILE_NAME = "database.properties";
Properties props = new Properties();
FileInputStream in = new FileInputStream(PROPERTIES_FILE_NAME);
if (in == null) // to avoid obscure errors
    throw new FileNotFoundException(PROPERTIES_FILE_NAME);
props.load(in);

String drivers =
    props.getProperty("jdbc.drivers") ;
Class.forName(drivers);
```
Use Properties file to get a Connection object from DriverManager

...
String database =
    prop.getProperty("database.name");

// start the connection
Connection connection = null;
try {
    connection =
        DriverManager.getConnection(database,
                                         username, password);
} catch (SQLException e) {
    System.out.println("We have problems with the connection");
    e.printStackTrace();
    ...
}
Using the Connection obj.

invoke these methods on the java.Sql.Connection:

```java
Statement createStatement()  // Returns a statement object that is used to send SQL to the data
PreparedStatement
    prepareStatement(String sql)  // Returns an object that can be used for sending parameterized SQL statements.
CallableStatement    preparecall(String sql)  // Returns an object that can be used for calling stored procedures.
DataBaseMetaData    getMetaData()  // Gets an object that supplies database configuration information.
```
Using the Connection obj.

```java
boolean isClosed()  // Reports whether the connection to database is currently open or not.
void setReadOnly(Boolean yn)  // Restores/removes read-only mode, allowing certain database optimizations.
void commit()  // Makes all changes permanent since the previous commit/rollback.
void setAutoCommit(Boolean yn)  // Restores/removes auto-commit mode, which does an automatic commit after each statement.
void close()  // Closes the connection and releases the JDBC resources for it.
```
Statement

A Statement is obtained from a Connection:
Statement stmt = con.createStatement();

\textbf{stmt.executeUpdate}

- with a string argument containing the text of an SQL update query (INSERT, DELETE or UPDATE).
- This returns an integer count of the number of rows updated.

\textbf{stmt.executeQuery}

- with a string argument containing the text of an SQL SELECT query.
- This returns a ResultSet object which is used to access the rows of the query results.

\textbf{stmt.execute} execute arbitrary SQL statement which may be of any type. However, extracting the results, whether an integer or a ResultSet, is less convenient. This is usually only used where you want generalized access to the database that allows programmatic generation of queries.
Example of Statement

```java
int count = stmt.executeUpdate("INSERT INTO Customers 
"+(CustomerFirstName, 
CustomerLastName, 
CustomerAddress) 
"VALUES ('Tony', 'Blair', '10 Downing Street, London')");

ResultSet rs = 
stmt.executeQuery("SELECT * FROM Customers");
// do something with count and rs
```
Example of syntax of SQL for special characters

if a name, O'Neill, is to be inserted:

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM Customers WHERE CustomerLastName = 'O''Neill'");
```
How to obtain table schema

If you do not know exactly the table structure of the ResultSet, use ResultSetMetaData object.

```java
ResultSetMetaData rsmd = rs.getMetaData() ;
int colCount = rsmd.getColumnCount() ;
for (int i = 1 ; i <= colCount ; i++)
{
    if (i > 1)
        out.print("", ");
    out.print(rsmd.getColumnLabel(i)) ;
}
out.println() ;
```
Given any ResultSet object, you can step through it to obtain its rows, or, more specifically, the fields of its rows.

Think of the ResultSet as a set of record in form of a two dimensional array on which the methods getBlob(), getBigDecimal(), getDate{}, getBytes(), getInt{}, getLong(), getString() and so on, can be invoked.
Using `getObject()` on `ResultSet`
Three points about using next()

• the column numbers start at 1, not 0
• you need to call next() to get the first record.
• It returns false when there is no further record.
Only one ResultSet can be open for given Statement.

Trying to be smart to reuse the Statement twice removes the first ResultSet.

So, if you are going to use a ResultSet a second time, create a new Statement object.
Only one ResultSet can be open for given Statement

TODO: check that the following is not working:

```java
ResultSet result1, result2
While (result1.next()) {
    /* more code */
    Result2 = myStmt.executeQuery(
        someSQLString2 ); // Ooops!
You can’t use myStmt twice and expect that the first ResultSet remains the same. You must use another Statement.
```
getObject() can take column name

This is better:

```java
while (rs.next())
{
    out.println(rs.getObject("CustomerID")
        + ", " +
    rs.getObject("CustomerFirstName") + ", "
        +
    rs.getObject("CustomerLastName") + ", "
        +
    rs.getObject("CustomerAddress") );
}
```
Which one better, getObject() or getInt()/getString()/…?

ggetObject() handles null better:
if primitive type (int/String…) you must use wasNull() method to check if the value is null IMMEDIATELY after return of value. It is easier to check for null objects!
Moreover, if printing, when using getObject(), the string “null” will be printed.
Advantages of PreparedStatements:

• Better performance: PreparedStatements are more efficient because the SQL is compile once

• Parameters are not passed as concatenated Strings, so eliminate the chance of SQL injection

TODO: study SQL injection, if you don’t know about it.
PreparedStatements

• Use SQL text with character ‘?’ for parameters
• parameters can be cleared using clearParameters()
• Parameters can be set using mutators such as setString(), setInt(),...
• Execution as usual is carried out via executeUpdate() or executeQuery()
Example of constructing a PreparedStatement

```java
PreparedStatement pstmt =
    con.prepareStatement(
    "INSERT INTO Customers " +
    "(CustomerFirstName, CustomerLastName,
    CustomerAddress) " +
    "VALUES (?, ?, ?)"");

// parameters are denoted by ? Count starts
// from 1 NOT 0
```
Executing a PreparedStatement

```java
psmt.clearParameters() ;
pstmt.setString(1, "Joan") ;
pstmt.setString(2, "D'Arc") ; // NOTE: no explicit quoting required
pstmt.setString(3, "Tower of London") ;
count = pstmt.executeUpdate() ;
System.out.println ("\nInserted " + count + " record successfully\n") ;
// don’t forget to clearParameters()
psmt.clearParameters() ;
pstmt.setString(1, "John") ;
pstmt.setString(2, "D'Orc") ;
pstmt.setString(3, "Houses of..." ) ;
count = pstmt.executeUpdate() ;
System.out.println ("\nInserted " + count + " record successfully\n") ;
```
Transactions

A mechanism for grouping operations on a database so that either all of them complete together or none of them do.

Example: YourAccount.Add(20);

Erroneous execution when withdraw fails and deposit succeed will make the bank unhappy 😊

The theory of transactions and architectures which support it is studied in Distributed Systems!
How to use JDBC Transactions?

When a Connection is obtained, by default its AutoCommit property is set to true. This means that every query execution is committed immediately after it is executed and before the next one is executed.

To enable grouping of operations into transactions, you have to switch the AutoCommit property off:

```java
con.setAutoCommit(false);
```
How to use Transactions?- continue

When all operations that you want to group together have completed, you **must** commit the updates to the database:

```java
con.commit();
```

At this point you can continue with more operations which will be grouped into a new transaction or you can switch AutoCommit back on:

```java
con.setAutoCommit(true);
```
If anything goes wrong during a transaction (e.g. an exception is thrown or an error means that you cannot complete your group of operations) then you have to undo all operations in your transaction so far:

```java
con.rollback();
```

In case that the server/machine crashed, `rollback` will be called for you automatically to clean up
**Important:** include both write and read

A common mistake is to include ONLY database update

Imagine example of a few Travel agents in geographically dispersed locations what are booking plane from a single flight. Not including READS in the transactions mean that the business logic will work with inconsistent data!

Remember the following points for having the database and Java application consistent:
**Important:** include both write and read

1) Put the relevant READ statements in the transactions before the write

2) If reading data about a transaction, include the read at the end transaction

There are methods of relaxing this requirement but they are out of the scope of the course. Some are explained in Distributed Systems module.

How about keeping a big transaction open all the time? Bad idea 😞
Minimise the length of time a transaction is open

Using transactions locks the database resources such as pull of connections and CPU, so other applications cannot use them

- Minimise the time for each transaction (less than one sec.)
- Never hold a transaction open while waiting for input from users
- In Web applications, always finish the transaction before returning the response to clients.
- Always use finally blocks to ensure connections are closed properly
Minimise the length of time a transaction is open

- if you need the data after the transaction has finished, copy it out into your own objects (not a ResultSet object) before committing the transaction.

Why can’t I use ResultSet object after the transaction completed?
Minimise the length of time a transaction is open

• if you need the data after the transaction has finished, copy it out into your own objects (not a ResultSet object) before committing the transaction.

Why can’t I use ResultSet object after the transaction completed?

Because ResultSet is a lazy cache of data from the database and can have only a few rows from the database
A word on multithreaded applications:

If AutoCommit is switched off, the connection can be shared between more than one thread.

It is important to use Synchronised threads or other mechanism to ensure appropriate usage of the connection.

Another option is pool of threads (java 1.5 and above): worker thread which picks jobs and passes them to the thread pool.

These topics are out of the scope of this course. Students who are interested in multithreaded programming can start from my last year notes in SSC2.
Correct handling Exceptions: very crucial

A SQLExceptions, thrown when something goes wrong, can be related to other SQLExceptions. Use a code similar to following to be able to inspect all of them:

catch (SQLException e)
{
    do
    {
        // do something with each SQLException
        //in the chain:
        System.out.println(e.getMessage()) ;
    }
    while ((e = e.getNextException()) != null);
}
Sample code for transaction processing

Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try
{
    conn = getConnection();
    conn.setAutoCommit(false);
    // do some JDBC calls using conn, stmt and rs
    //conn.commit();
    // if you get to here then everything has succeeded
    // and the transaction is complete.
}
catch (Exception e) {
    /* note that other exceptions might occur than SQLExceptions: if they have, we still have to rollback because the transaction has still only partially completed. Here something has gone wrong: rollback any updates to fix the database and do any other application specific corrections necessary */
    try {
        conn.rollback();
        // application specific handling of
        // exception
    }
}
Sample code for transaction processing

catch (SQLException e)
{
    // in case something goes wrong with the rollback()
    do
    {
        // do something with each SQLException in the chain.
        // possibly you want to log it rather than print it as
        // these messages are intended more for a system administrator than an end user.
        System.out.println(e.getMessage());
    }
    while ((e = e.getNextException()) != null);
}


finally
{
   // ALWAYS close the JDBC objects in a finally block to ensure that they are correctly closed even if exceptions occur.
   if (rs != null)
      try { rs.close(); } catch(SQLException e) {}
   if (stmt != null)
      try { stmt.close(); } catch(SQLException e) {}
   if (conn != null)
      try { conn.close(); } catch(SQLException e) {}
}
Remember Running example:
How to obtain automatically generated key in database?

Example: in the database of the running example, if a new *Order* is created which has a new *OrderDetails*, how do we obtain the value of *OrderID*? i.e. what is the assigned ID?

This is important as in your SQL query you may need the value of, say, a foreign key!
How to obtain automatically generated key in database?

We can do a `SELECT`,
but this is an extra over head 😞

… and sometimes not possible.

a nasty case of A reference B and B reference C.
How to obtain automatically generated key? - continue

DBMS provide solutions

• MSAccess has a field type called **Autonum**
• PostgreSQL has a general construct called a **sequence**

The sequence associated to **OrderID** filed is called **orderID_seq**, which is a normal PostgreSQL bigint field. You can obtain it by executing

```
SELECT nextval('orderID_seq') AS ID
```
How to obtain automatically generated key? continue

How does it work?

1) get a ResultSet which will contain a single row with a single column by executing

\[ \text{SELECT nextval('orderID_seq') AS ID} \]

2) get the value (of Java type long) from that to find an \textit{ID value}.

3) create the Order record using this \textit{ID value}.

4) Use the \textit{ID value} as a foreign key value in any \textit{OrderDetail} records you wish to create.
How to obtain automatically generated key? continue

This approach is simple and safe, but you still have to write an extra query.

More sophisticated solutions are available:

• see [Johnson03], [Ambler03] or the latter author's online essay
java.util.Formatter

Formatter is Java’s answer to C’s printf.
// %d Format as a decimal integer
System.out.format("Inserted %5d into database%n", 51);

Calendar cal = Calendar.getInstance();
System.out.format("%nDate: %tD %nTime: %tT %nMonth: %tB", cal, cal, cal, cal);

If you are not familiar with Formatter, see a sample
systemOutFormatTutorial\SystemOutFormat.java