CS211 Lecture: Java Database Connectivity (JDBC)

revised 11/9/2206

Objectives:

1. To introduce using JDBC to access a SQL database

Materials:

1. Projectable of registration system architecture.
2. JDBC Example program to demonstrate
3. JDBC Example handout

I. Introduction

A. Our discussion of SQL thus far has been generally applicable to any system using a relational database - OO or otherwise.

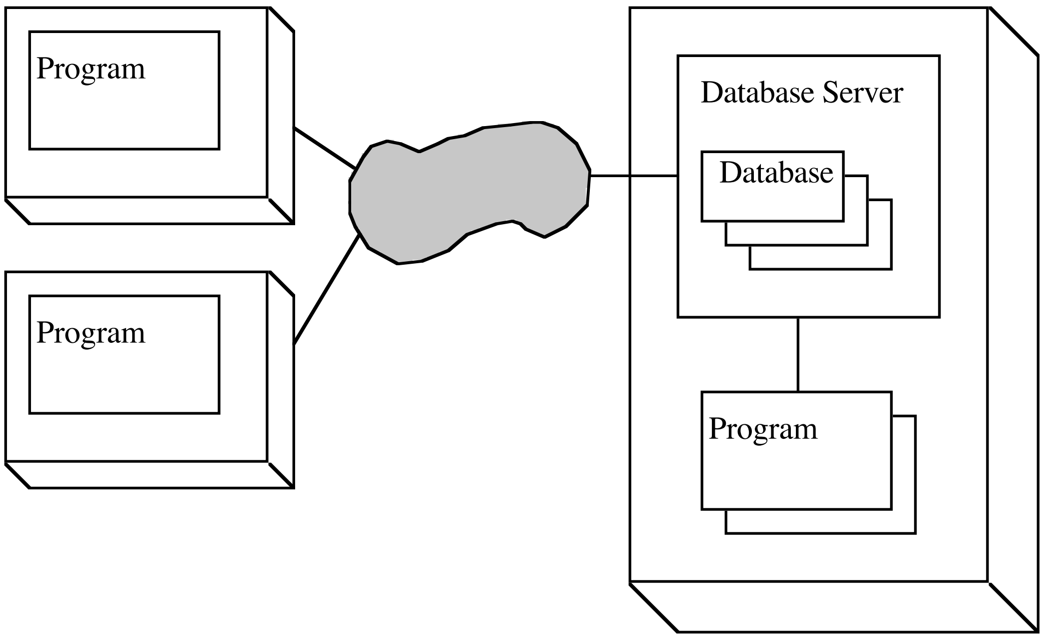
B. We now want to consider how we might access a relational database from within a Java program. Notice that this means that we are, in a sense, using a mixed approach to system architecture:

1. The GUI and business logic will be strictly object-oriented.

2. The model will be a relational database.

C. The overall architecture of such a system might look like this

PROJECT



1. All access is through a database server running on some host system.

2. A given database server may contain any number of databases

3. Any number of programs running on the same system may access the server at the same time.

4. In addition, any number of other systems may access the server over the network at a given time.

5. The programs that are accessing the database may be written in Java or any other language that the database server supports. It is common to find that database servers allow many different programming languages to be used to write programs that access databases they serve.

D. Two approaches can be used to allow a Java program to access a database server

1. One approach embeds sql statements in a Java program (so the program contains a mixture of Java and sql). A special preprocessor program is used to separate this into a sql module and "pure" Java. The resultant program must be run either on the system that contains the database server or a client version of the database server.

   a) We develop this approach in CS352

   b) We will not develop it here, because it is quite a bitmore complex (though the resultant program executes more efficiently)

2. The approach we will develop here is called JDBC - which stands for Java Database Connectivity.

   a) A system that is accessing a Java database must have an appropriate JDBC <u>driver</u> installed on it. A JDBC driver is a software component that

      (1) Supports a standardized Java API that lets Java programs access it.Knows how to communicate with the database server (typically over a network) in whatever way the server expects.

      (2) This is an example of the <u>bridge</u> design pattern. The purpose of a bridge is "to decouple an abstraction from its implementation so that the two can vary independently"

b) A JDBC driver is specific to a particular database server. JDBC drivers are often made available by the producer of the database server, though this is not necessarily the case.

(1) The JDBC driver we will be using to access our MySQL database was not developed by the developers of mysql, but rather by someone else.

(2) On the other hand, we also have a copy of IBM's database system called db2 (which we are using for CS352). It includes a JDBC driver developed by IBM

c) A JDBC driver is written in Java (and uses the facilities of java.net to actually communicate with the database server.) As a result, any JDBC driver can be run on any platform that supports Java. (It is specific to the server platform, but not to the client platform.)

E. To access a given database, a Java program must

1. Load the appropriate driver.

2. Establish a connection to it by contacting the server.

3. Issue SQL query and/or update statements

II. Basic JDBC Concepts

A. A preliminary note: JDBC has many capabilities. We will look only at a small subset.

B. The notion of a connection is fundamental. A connection is an object that implements the interface `java.sql.Connection`. It is created when the connection to the database is established. Usually a program does all of its accesses to the database through a single object. (An exception would be if the program accessed more than one database - each would need its own connection.)

C. To perform a query or update operation, one can ask the connection to create a `java.sql.Statement` object. The statement object can then be used to execute one or more queries and/or updates - after which it should be closed.

1. Thus, code like the following will appear (where connection is the connection object):

   ```
   Statement statement = connection.createStatement();
   -- use the statement object to perform one or more
      queries and/or updates
   statement.close();
   ```

2. Two methods of statement are particularly important.

   a) `executeQuery()` is used to execute a query operation. It always returns a ResultSet object - which we will discuss shortly.

   b) `executeUpdate()` is used to execute an update. It always returns an int, which is the number of rows affected by the query. (Thus, one way to see if an update was successful is to see if its result was non-zero.)

   c) Both methods take a `String` parameter, which is the SQL query or update to execute. Thus, if statement is a `Statement` object, operations like this would be legal:

   ```
   statement.executeQuery("select * from Book") or
   statement.executeUpdate("delete from Borrower where
   lastName = 'Aardvark'");
   ```

   (The strings may be constants that are part of the text of the program, or they may be created during execution just like any other strings).

   d) Both methods throw a `java.sql.SQLException` if there is any syntax error in the SQL.

D. An object that implements the `java.sql.ResultSet` interface is returned when one does a query. The reason for this is that, in general, SQL queries can return any number of rows. The `ResultSet` object is like an iterator, in that it has a `next()` method that can be executed repeatedly to access successive rows. (However, the methods are not identical to those for iterators.)

1. The `next()` method is used to advance to the next row. It returns a boolean - true if there was a row to advance to, false if there was not. (Thus, one can determine when one has seen all the results.) An important point to note is that `next()` must be called once to get to the first row of the results - failure at this point would indicate an empty result set.

2. Individual columns of the current row are accessed by using `getXXX()` methods of the `ResultSet` object.

   a) The `getString(int)` method returns a single column as a `String`. The parameter specifies what column is wanted. The first column is considered column one - not zero. (Columns are numbered in the order in which their names appear in the select statement, or in the order in which they are defined in the database if "*" is used.)

   b) If the column is known to have an appropriate type, other methods like `getInt(int)` or `getDouble(int)` or `getDate(int, Calendar)` can be used to access the column - but the method called must correspond to the type of the column. (Any type of column can be returned as a `String`.)

III. An Example Program Using JDBC

A. To illustrate JDBC, we will use a program based on the Address Book project from last semester, but which stores the address book in a SQL database.

B. DEMO

1. Show database maintained by mysql on jonah

```
a) use ADDRESS_BOOK;
b) select * from ABTABLE;
```

2. Run demo program - make some changes to address book

3. Show database maintained again - note how changes are reflected

C. **Handout:** Example JDBC Program. We've basically replaced the `AddressBook` and `Person` classes (kept in memory) with the use of a relational database on disk.

D. Walk through `AddressBookController.java`

1. methods corresponding to the Add, Edit, Delete buttons in the GUI

   a) Add: note how information to be added is generated at run time

   b) Edit: now how desired person's name is used in the initial select and later in the update, along with dynamically generated content for the various fields. (We update everything, rather than trying to figure out what has changed.)

   c) Delete: note how person's name is used in the delete statement. We could have used the returned result to know whether or not the person we attempted to delete actually existed - though we didn't in this case, since the gui presents us with a list of names or people in the book, so we couldn't specify a nonexistent person. (But we could if the user were allowed to type a name)

2. sort methods - note how we let the database server do the sorting! Note use of an instance variable in the controller to keep track of the desired sort order.

3. print method - added, since previously the `AddressBook` class (which we have eliminated) defined a method for this

4. new method `fillIn()` used to create a list of names to be displayed in the GUI. Note: if the address book were large (and hence the list were long), it would probably be better to let the user type a partial name and then create a list of matches - e.g. if the user types A he gets Aardvark, Albatross, Ant etc.