CS 328 - Homework 6

Deadline
11:59 pm on Friday, March 28, 2014

How to submit
Submit your files for this homework using ~st10/328submit on nrs-projects, with a homework number of 6

Purpose
To practice writing a PL/SQL trigger, and to practice with Java and JDBC.

Important notes
• Your solutions for these problems should avoid using a hard-coded password within your Java source code. Feel free to copy and make use of the getPassword method from many of the posted Java JDBC examples.

• You are expected to use standard Java naming conventions (as discussed in class) in your Java source code.

• In your Java source code, you are expected to indent the contents of all { }’s by at least 3 spaces, and each { and } should be on its own line, even with the preceding line (as seen in posted class examples).
  – also, all Java classes and methods are expected to start with a comment that at least gives its name, and a purpose statement which explicitly describes either the purpose of the class or what the method expects and what it does and/or returns.

• Remember to also follow the style guidelines and course standards given or discussed previously for the other languages used in this homework.

• Make sure that you have executed the scripts create-bks.sql and pop-bks.sql, and that the bookstore tables are successfully created and populated.

Problem 1
Consider PL/SQL triggers! We discussed them a bit in the Week 1 Lab, and there are example triggers posted in the Week 1 Lab section of the posted in-class examples. But you haven't written one in a very long time -- and one would be of use for the bookstore scenario.

Here are a few key trigger-related reminders:
• You don't call a trigger -- instead, it is executed/triggered when the specified action occurs to the database. Consider the following trigger headers:
This trigger, named inventory_update, will be executed/fired BEFORE each insert of a row into a table named orders

```sql
create or replace trigger inventory_update
  before insert
  on orders
  for each row
```

This trigger, named add_prof_ct, will be executed/fired AFTER each insert of a row into a table named prof

```sql
create or replace trigger add_prof_ct
  after insert
  on prof
  for each row
```

This trigger, named clear_advisor, will be executed/fired BEFORE each delete of a row into a table named prof

```sql
create or replace trigger clear_advisor
  before delete
  on prof
  for each row
```

Oddly, IF you have local declarations for a trigger, you DO put declare after the trigger header and before those declarations:

```sql
create or replace trigger inventory_update
  before insert
  on orders
  for each row
declare
  -- declare section can be omitted if you do not want
  -- to declare any variables...
  amt_ordered integer;
  item_ordered integer;
  amt_in_stock integer;
begin
```

...but just proceed to begin if you don't have any local declarations:

```sql
create or replace trigger add_prof_ct
  after insert
  on prof
  for each row
begin
```

NOTE that Oracle is very concerned about preventing potential "circular" rule firings -- so it will
NOT permit you to query the table on which the trigger is fired. I would not be allowed to query prof within add_prof_ct, for example.

• Within the body of a trigger, you can obtain the attribute values in the "new" row being inserted or updated with the syntax :new. preceding the name of the attribute whose value you want. Likewise, you can obtain the attribute values in the "old" row being deleted or updated with the syntax :old. preceding the name of the attribute whose value you want.

  – For example, the expression

    :new.prof_id

    ...would be the value of the prof_id attribute that was just inserted into prof in the trigger add_prof_ct

  – And as another example, the expression

    :old.prof_id

    ...would be the value of the prof_id attribute that was just deleted from prof in the trigger clear_advisor

Now you will write a PL/SQL trigger for the bookstore scenario.

Create a SQL script 328hw6.sql, and start it off with comments including your name, CS 328 - Homework 6, and the last-modified date.

Next, add the command to run the pop-bks.sql script each time this script is run, so that you have "fresh", original versions of these tables. (Their contents are mucked with below, so it is important that these are "reset" here.)

Include the command to set serveroutput on, followed by a SQL*Plus spool command to spool the results of running this SQL script to a file named 328hw6-out.txt. Then write a SQL*Plus prompt command that says problem 1.

Now, consider the bookstore's order_sum, order_detail, and order_needed tables.

When the stock of some title falls below its order_point, a row indicating that an order of this title is needed should be added to the order_needed table. (Indeed, sell_book makes sure that this happens if necessary when a book is sold.) When a row is added to the order_needed table, the current date is inserted for the date_created attribute of the new order_needed row, and the date_placed for this new row is null (because the order needed has not been placed yet).

order_sum and order_detail hold the details of an order of titles from a publisher. If I had named these tables, I probably would have named them order and order_line_item, respectively -- each row of order_sum represents an "overall" order, including such overall details of an order as the publisher that order is from, the unique order number, the date the order was placed, and the date the order is complete. And order_detail gives the details for one of the titles being ordered as part of that order -- it indicates what order is involved, what line-number of that order this represents, which title is being ordered in this line of the order, and how many of that title are involved in this order.

So, consider -- when an order is placed in response to an order_needed row, surely the order_needed table's date_placed attribute ought to be the very value in the order_sum's...
table's `date_placed` attribute for that order. Also, consider the `on_order` attribute of the title table -- this is also the proper time to set this attribute to 'T' for each ordered title, also, since now such a title is indeed on-order.

How might we ensure that these updates are made, if necessary, to the proper rows in the `order_needed` and `title` tables?

We said very early in the semester that triggers can be used to enhance database integrity. And, indeed, we can use a trigger here for just that purpose.

What action should trigger a corresponding action? Not an insertion into `order_sum` -- that's overall information for the order, not the individual title for which an order is needed. But it might be handy, after each insert into `order_detail`, to:

• see IF there is an `order_needed` for that line item's title that has a null value for `date_placed` -- if there is a pending `order_needed` row for that line item's title -- that could now be updated to be the date that the corresponding order was placed;

• change the `on_order` attribute for that line item's title to 'T', since it is now on-order.

Important additional information:

• DON'T assume that date is the current date -- someone might be entering in this order information the next day, for example, or on Monday after a Friday order.

• Note that an order for some title might be placed even though there isn't an "open" `order_needed` row for it -- the bookstore manager may choose to simply order more of a title for strategic reasons. So no row in `order_needed` would be updated in that case, although that title's `on_order` attribute should still be set to 'T'.

Within your `328hw6.sql`, design and implement this PL/SQL trigger `order_maint`. Follow your trigger with the following testing code:

```sql
prompt ***********************
prompt demo order_maint
prompt ***********************
prompt

commit;

-- put in some "fake" old order_needed rows for '0805367829'
-- to make sure these AREN'T changed by the trigger
-- (only pending order_needed rows for a title should
-- be changed by an order, you see... 8-)

insert into order_needed
values

insert into order_needed
values
```
var results_code number;
exec :results_code := sell_book('0805367829', 11);

prompt ===============================================================
prompt title is not yet on order, although order is needed
prompt (and can see 2 fake "older" order_needed rows for this title)
prompt ===============================================================
prompt

select isbn, on_order
from title
where isbn = '0805367829';

select *
from order_needed
where isbn = '0805367829';

prompt ===============================================================
prompt simulate an order being placed for this title tomorrow
prompt ===============================================================
prompt

insert into order_sum(ord_no, pub_no, date_placed)
values
(11016, 147, sysdate+1);

insert into order_detail
values
(11016, 1, '0805367829', 10, 0);

prompt ===============================================================
prompt after order of this title, is this title now shown as on_order?
prompt ===============================================================
prompt

select isbn, on_order
from title
where isbn = '0805367829';

prompt ===============================================================
prompt ...and is JUST the LATEST order_needed date_placed now tomorrow?
prompt ===============================================================
prompt


select *
from   order_needed
where  isbn = '0805367829';

rollback;

spool off

You may add additional testing calls if you would like. Your files 328hw6.sql and 328hw6-out.txt are now ready to submit.

Problem 2

Write a little Java application that just provides a single piece of information with the help of JDBC.

Write a Java command-line application AvgOnHand.java using JDBC that queries the Oracle java account to find out the current average quantity-on-hand of all of the books, and prints that single average to the screen within a descriptive message. (Note that it does not expect any command-line arguments, and it may happily ignore any given.)

Your resulting AvgOnHand.java is now ready to submit.

Problem 3

Of course, queries that produce multiple-row results are more common. So, next you will use JDBC to obtain and display the results of such a query.

Write ShowAuthorTitle.java, a Java command-line application using JDBC that queries the Oracle java account, printing to the screen a descriptive heading, then for each book printing the following:

• the name of its author
• then a space and a dash and a space
• then its title
• then a space and a dash and a space
• then the title's quantity-on-hand (one author - title - qty trio per line)

...in alphabetical order by author, and in secondary alphabetical order by title.

They do not have to be lined up nicely (we'll do that with <table>'s when we move on to servlets and JSP). (This also does not expect any command-line arguments and it may also happily ignore any given.)

Your resulting ShowAuthorTitle.java is now ready to submit.
Problem 4

Fun fact #1: note that you can exit from a Java method "early" (say, because of inappropriate user input...) by using the statement `System.exit(0);`

Fun fact #2: the now-posted example `PrepStmtEx.java` demonstrates the use of a `PreparedStatement`, a specialized subclass of `Statement` that is a better choice than `Statement` for quite a few situations, including when you are running similar SQL statements more than once.

- As you read this example, carefully note that you create a `PreparedStatement` instance using the `Connection` method `prepareStatement` rather than `createStatement`,
- ...and that `prepareStatement` expects a `String` argument, containing a kind of "template" of the desired SQL statement with question mark characters (`?`) where you will be inserting different values for different versions of this SQL statement.
- Then, before each execution of this `PreparedStatement` instance, you use one of its `set*` methods to set each of those question marks to its value for the next execution, and
- ...then you call the appropriate `execute*` method for that SQL statement, except with NO argument.

Keeping these fun facts in mind...

...write a command-line Java application `GetPubBooks.java` that:

- expects 1 or more command-line arguments, names of publishers;
- if it doesn't get at least one command-line argument, it should complain in a descriptive message to the screen and exit;
- otherwise, for each command-line argument, assumed to be the name of a publisher, it:
  - prints a blank line followed by a nice heading noting that these are books published by `<that publisher's name>`,
  - and then lists the titles published by that publisher, one per line, in alphabetical order by title, as queried from the Oracle `java` account.

Note the following additional requirements:

- What should you do if this is called with the name of an publisher NOT in the database? Then you are permitted to just print the blank line and nice heading noting that these are books by `<that publisher's name>`, with nothing following it.
- Since there can be multiple command-line arguments, you might be doing multiple very similar queries. So, both for efficiency and to get practice for future SQL Injection avoidance, you are expected to use a `PreparedStatement` rather than a `Statement` for this application.

assumed to be the name of a publisher, it:

- prints a blank line followed by a nice heading noting that these are books published by `<that publisher's name>`,
- and then lists the titles published by that publisher, one per line, in alphabetical order by title, as
queried from the Oracle \texttt{java} account.

Note the following additional requirements:

- What should you do if this is called with the name of an publisher NOT in the database? Then you are permitted to just print the blank line and nice heading noting that these are books by <that publisher's name>, with nothing following it.

- Since there can be multiple command-line arguments, you might be doing multiple very similar queries. So, both for efficiency and to get practice for future SQL Injection avoidance, you are expected to use a \texttt{PreparedStatement} rather than a \texttt{Statement} for this application.

Your resulting \texttt{GetPubBooks.java} is now ready to submit.

\textbf{Problem 5}

In the Week 8 lab, we talked through the key JDBC aspects of an example \texttt{UpdateLog.java}, that inserts a row into a table named \texttt{log_table} that conveniently has already been created in the Oracle account with username \texttt{java}. The full version of this example is now posted.

You are going to write a variation on this that also affects the same table \texttt{log_table} on the Oracle \texttt{java} account that \texttt{UpdateLog} updates. Here are \texttt{log_table}'s columns (and note that its primary key is BOTH columns):

\begin{verbatim}
SQL> describe log_table
Name                                      Null?    Type
----------------------------------------- -------- -------------  
USERNAME                                  NOT NULL VARCHAR2(20)
TIME_LOGGED                               NOT NULL DATE

\end{verbatim}

Write a Java command-line application \texttt{AddToLog.java} using JDBC that inserts a row into \texttt{log_table} for each of its command-line arguments, treating each command-line argument as a username of a new row in \texttt{log_table}, using the current date for \texttt{time_logged}.

For example, if on March 14 you ran:

\begin{verbatim}
java AddToLog abc1 def2 ghi3
\end{verbatim}

... then the following rows would be added to \texttt{log_table}:

\begin{verbatim}
abc1                     14-Mar-14  
def2                     14-Mar-14  
ghi3                     14-Mar-14
\end{verbatim}

Note the following additional requirements:

- Since there can be multiple command-line arguments, you might be doing multiple very similar inserts. So, both for efficiency and to get practice for future SQL Injection avoidance, you are expected to use a \texttt{PreparedStatement} rather than a \texttt{Statement} for this application, also.

- Note that it should simply complain in a descriptive message to the screen and exit if called with NO command-line arguments (it should \texttt{NOT} try to connect to the database in that case).
Your resulting AddToLog.java is now ready to submit.

**Problem 6**

We are all sharing that java Oracle account - it could get pretty littered while everyone works on Problem 5!

Write a Java command-line application RemoveFromLog.java that seeks to remove all rows from log_table with a username equal to one of its command-line argument. That is,

```java
java RemoveFromLog abc1 def2 ghi3
```

...would remove all the rows added by the example call in Problem 5 (as well as any other rows with the usernames abc1, def2, or ghi3).

Note the following additional requirements:

- The concept here is to do a separate delete for each username -- so, for the same reasons as for AddToLog, you are expected to use a PreparedStatement rather than a Statement for this application.
- It also should simply complain in a descriptive message to the screen and exit if called with NO command-line arguments (it should NOT try to connect to the database in that case).

(Be polite, please - while testing, try to only remove rows you have added... 8-)

Your resulting RemoveFromLog.java is now ready to submit.

**Problem 7**

For some light meta-data playing, and as an excuse to have you peruse a bit of the Java 7 API, consider the posted example SpewTableColumns.java, which given a table name as a command-line argument, uses ResultSetMetaData to obtain and output the names of the columns in that table.

Look at the available methods for ResultSetMetaData under the Interfaces section of package java.sql at the Java 7 API (remember, there is link to it from the public course web site).

Consider the SQL*Plus describe command - it gives the names of the columns of the argument table name, along with whether they can be null (NOT NULL if they cannot be, blank if they can), and their type. But, since describe is a SQL*Plus command, you cannot call it using JDBC. But you can construct a Java version using JDBC's ResultSetMetaData...

You should find what you need amongst ResultSetMetaData's methods to create a Java command-line application JDescribe.java, which expects to take a table name as its command-line argument, and prints to the screen, for each column in that table:

- the name of each column,  
- then a blank-dash-blank,
- NOT NULL and a blank-dash-blank if it **cannot** be null,
- and a string-depiction of that column's database-specific type name (with one column-name - if-not-null - type-name combo per line).
Note the following additional requirements:

- Have it complain to the screen and exit if anything other than exactly one command line argument is given.
- Have it complain differently to the screen and exit if there exists no table with that name.

It won't be as nicely formatted as the SQL*Plus describe command, but it will do for now. (That is, you don't have to format it any further from the format described above unless you want to.)

Your resulting JDescribe.java is now ready to submit.

**Problem 8**

Consider the posted example CallHowMany.java, which demonstrates how to use a CallableStatement to call a PL/SQL stored function.

Write a Java command-line application SellBook.java that:

- expects exactly two command-line arguments, the ISBN to be sold and the quantity of that book to be sold.
- if it doesn't get exactly two command-line arguments, it should complain in a descriptive message to the screen and exit.
- if its second argument cannot be parsed as an integer, it should complain in a different descriptive message to the screen and exit.
  – Hint: what method will throw an exception -- that one could catch and handle in this way -- if given a string that cannot be parsed as an int?
- if that second argument/quantity is less than 0, it should complain in yet-another different descriptive message to the screen and exit.
- but if all is well, it should call the PL/SQL stored function sell_book to try to update the Oracle java account's database appropriately to sell that many of that book.
  – (Note that a version of sell_book and its needed auxiliary subroutines have been created within the java Oracle account; do NOT remove or replace any of these subroutines from the java Oracle account!)
- use the value that sell_book returns to print a descriptive message to the screen, indicating if the sale seems to have succeeded, and if not, why not (be as specific as you can be about what the problem is in this descriptive message printed to the screen).
- NOTE: call pop_bks.sql as needed to "restore" the poor java' account's version of these tables as the class is working on this! And be aware that you are all testing using this one java account -- I hope it won't get too bizarre, but I cannot promise it won't.

Your resulting SellBook.java is now ready to submit.