IT 4153 Advanced Database

Module 4 Triggers

|  |
| --- |
| **Introduction and Module Summary** |
| In this module, you will learn how to write PL/SQL triggers. |
| **Objectives and Outcomes** |
| This module directly supports **highlighted** course outcome(s)  Students who complete this course successfully will be able to   1. Describe current and emerging database models and technologies; 2. **Develop functions and procedures for data manipulation and database access auditing;** 3. Describe database monitoring and performance tuning; 4. Describe database security and administration issues, including backup and recovery; 5. Explain the concepts of data warehousing and data mining   **Module outcomes and activities:**   |  |  |  |  | | --- | --- | --- | --- | | After completing this module, students will be able: | To develop Oracle PL/SQL triggers | To test Oracle PL/SQL triggers | To debug Oracle triggers | | Readings | introduced | introduced | introduced | | Practice exercises | reinforced | reinforced | reinforced | | Quiz | reinforced | reinforced | reinforced | | Lab | mastered | mastered | reinforced | |
| **Assigned Reading** |
| 1. Oracle Triggers <http://docs.oracle.com/database/121/CNCPT/srvrside.htm#CNCPT218> |
| **Optional Reading** |
| 1. Oracle triggers reference <http://psoug.org/reference/table_trigger.html> |
| **Assessments and Assignments** |
| 1. Lab (10 points) 2. Quiz (10 points) |
| **Topics** |
| Open the navigation pane |
| **Module Checklist** |
| This is the suggested order of the completion of this module.  Save a copy of this file on your computer and make notes in this document while you are completing your assignments. Use the table below to keep track of your progress.   |  |  | | --- | --- | | **Activity** | **Completion** | | Read this module and assigned materials (1 hour) | NO | | Complete all exercises from the module (2 hours) | NO | | Complete the lab (2 hours) | NO | | Complete a feedback section at the end of the module | NO | | Take the quiz (30 min) | NO | | Read feedback provided for your discussion and lab. | NO | |
|  |

## Definitions

A trigger is a named PL/SQL block that is stored in the database and executed (fired) in response to a specified event that occurs in the database.

Triggers can be executed for the following database events:

* DML: DELETE, INSERT, or UPDATE
* DDL: CREATE, ALTER, or DROP
* Database operation: LOGON, SHUTDOWN, SERVERERROR, etc

Trigger can be enabled and disabled.

Procedural SQL code invoked before or after data row is selected, inserted, or updated.

Triggers

* Associated with a database table or view
* Table may have multiple triggers
* Executed as part of transaction
* Can enforce particular constraints
* Automate critical actions and provide warnings for remedial action
* Can update values, insert records, and call procedures
* Add processing power

## Purposes of Using Triggers

* Performing audits and event logging (this is their primary use)
* Preventing invalid data from being inserted into the tables (but not for data validations)
* Implementing business rules (for example, not allowing to update more than 10 rows with one update statement)
* Enforcing referential integrity rules where foreign keys can't be used (after performing denormalization)
* Generating values for columns (for example, average temperature can be calculated every time new reading is recorded)

## Oracle Triggers

Triggers are executed implicitly whenever the triggering event happens. Oracle has:

* row-level (once for each row affected)
* statement-level (once for the whole statement)

First, statement level triggers are fired, and then row level triggers are fired. Triggers of the same type fire in the object identifiers (OID) order <http://decipherinfosys.wordpress.com/2008/02/27/setting-the-execution-order-of-triggers-in-oracle-11g/> or by using the FOLLOWS clause to specify trigger execution order.

Triggers are called cascading triggers if a trigger fires another triggers.

### PL/SQL Trigger Syntax

CREATE TRIGGER <trigger-name>

{AFTER | BEFORE | INSTEAD OF } <trigger-event> ON <table-name>

[REFERENCING <references>]

[FOR EACH {ROW | STATEMENT}]

[WHEN <SQL-statements>]

<SQL-procedure-statements>

<trigger-event> ::= INSERT | DELETE | UPDATE [OF <column-names>]

<reference> ::= OLD [AS] <old-value-tuple-name> |

NEW [AS] <new-value-tuple-name> |

OLD TABLE [AS] <old-value-table-name> |

NEW TABLE [AS] <new-value-table-name>

NOTE: INSTEAD OF triggers only can be used on VIEWS.

### PL/SQL Trigger Example 1

Suppose we have a bank that does not allow overdraft, so before transaction account balance is checked for sufficient funds.

Do not forget to use SET SERVEROUTPUT ON; if you use SQL Plus or click DBMS Output ON if you use Oracle SQL developer.

CREATE TABLE account(

accountID integer primary key,

balance number (10,2)

);

/

INSERT INTO account VALUES (111, 40.50);

UPDATE account SET balance = balance + 10 WHERE accountID=111;

This query changes balance to $50.50. Suppose now account owner wants to withdraw $100. To enforce business rules, we will write a trigger that will raise application error and abort the transaction.

CREATE OR REPLACE TRIGGER withdrawTrigger

BEFORE UPDATE ON account

FOR EACH ROW

BEGIN

if :new.balance < 0 then

raise\_application\_error (-20999,

'No sufficient funds');

else

dbms\_output.put\_line('OK');

end if;

END;

/

SHOW ERRORS;

***:new.balance*** is the value used in update statement set balance = xxx

If business rule will be to process withdrawal: set account balance to zero and charge overdraft account, then the value can be modified during trigger execution.

:new.balance := 0;

add code to update overdraft\_account table to account for balance shortage.

NOTE: error\_number for custom errors has to be between -20000 and -20999.

Error message is a string up to 2048 characters long.

NOTE: When you create PL/SQL code, use new scripting window. If you have compilation errors, use SHOW ERRORS;

In case of errors, you will have error code, description and location.

Without SHOW ERRORS; this information will not be displayed. You will see only "TRIGGER WITHDRAWTRIGGER compiled

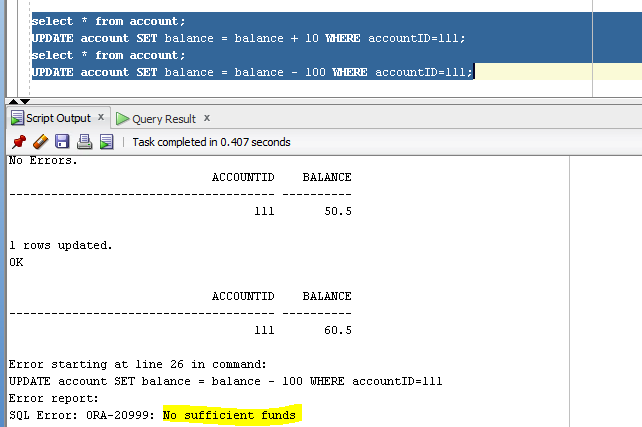
Errors: check compiler log"

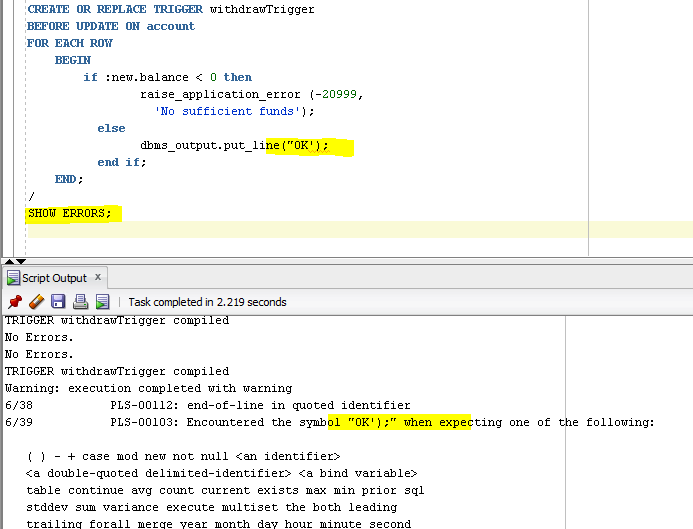
Let us try to add $10 and the withdraw $100 from account 111:

SET SERVEROUTPUT ON;

UPDATE account SET balance = balance - 100 WHERE accountID = 111;

As the result of the trigger, $100 will not be withdrawn from the account.





### Helpful Data Dictionary Views

SELECT \* FROM dba\_objects WHERE object\_name LIKE '%TRIG%';

It will return 167 objects including withdrawTrigger we created earlier.

ALL\_TRIGGERS describes the triggers on tables accessible to the current user.

DBA\_TRIGGERS describes all triggers in the database.

USER\_TRIGGERS describes the triggers owned by the current user. This view does not display the OWNER column.

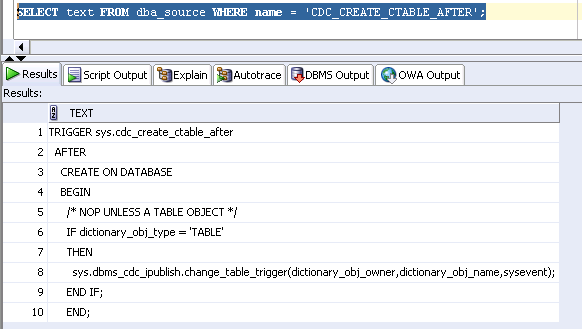
SELECT \* FROM user\_triggers;

SELECT \* FROM user\_triggers WHERE trigger\_name='WITHDRAWTRIGGER';

You will find information about triggers including type, description and trigger body

.

SELECT text FROM dba\_source WHERE name = 'CDC\_CREATE\_CTABLE\_AFTER';



NOTE:

WHERE name = 'CDC\_CREATE\_CTABLE\_AFTER'

WHERE name = ' CDC\_CREATE\_CTABLE\_AFTER'

WHERE name = ‘CDC\_CREATE\_CTABLE\_AFTER’

Are not the same. Find all differences except font face.

### PL/SQL Trigger Example 2

Triggers can be used to log successful logon events

First we need to create table that will keep track of logon events;

create table messages (messageDate date, userName varchar2(100));

Then give everybody permissions to insert rows in this table.

grant insert on messages to public;

Then create a trigger that will fire after user's successful logon

CREATE OR REPLACE

TRIGGER logonSuccess

AFTER LOGON

ON DATABASE

BEGIN

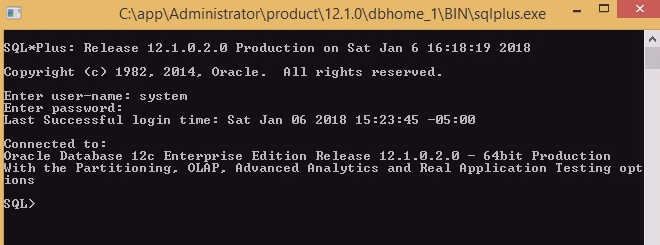
INSERT INTO SYSTEM.MESSAGES VALUES(sysdate,user);

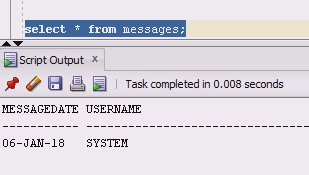
END;

/

To test use SQL\*Plus to logon as system/oracle. Two entries will be created in table Messages:

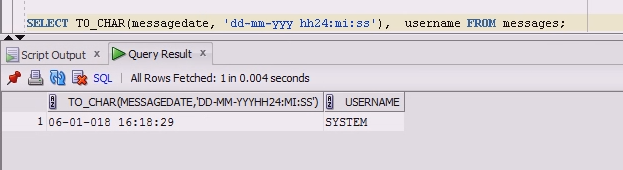
* one for SYSMAN. SYSMAN logon is used to check credentials for system/tiger
* and another one for SYSTEM, when system/tiger was successfully authenticated





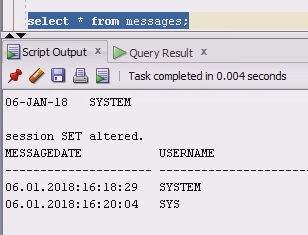
If you prefer to add time, you can either use to\_char conversion

SELECT TO\_CHAR(messagedate, 'dd-mm-yyy hh24:mi:ss'), username FROM messages;



Or alter the session date format

alter session set NLS\_DATE\_FORMAT='DD.MM.YYYY:HH24:MI:SS';



### PL/SQL trigger Example 3

Create a trigger that will be invoked before price is updated in ProductTable. The trigger should insert messages to EventTable. Use SEQUENCE to generate unique event IDs. Records should contain product id, user who updated the table and the time of the update.

For example:

1, 299, SYSTEM, 21-Apr-2015 21:18:27

2, 300, SYSTEM, 21-Apr-2015 23:20:50

Create table *EventTable*

|  |  |  |  |
| --- | --- | --- | --- |
| **EventID** | **ProductID** | **UserUpdating** | **EventTime** |
| int | int, not null | string length 250 | date |

CREATE TABLE EventTable(

EventID INTEGER,

ProductID INTEGER NOT NULL,

UserUpdating VARCHAR2(25),

EventTime DATE);

Create table *ProductTable*

|  |  |  |  |
| --- | --- | --- | --- |
| **ProductID** | **ProductName** | **ListPrice** | **ProductCategory** |
| int, not null | Not null, length 50 | $ | int, not null |

Add values to table ProductTable

|  |  |  |  |
| --- | --- | --- | --- |
| **ProductID** | **Name** | **ListPrice** | **ProductCategory** |
| 299 | Chest | $99.99 | 10 |
| 300 | Wave Cruiser | $49.99 | 11 |
| 301 | Megaland Play Tent | $59.99 | 11 |
| 302 | Wind-Up Water Swimmers | $2.00 | 11 |
| 303 | Garmin Pocket or Vehicle GPS Navigator | $609.99 | 12 |

CREATE TABLE ProductTable(

ProductID INTEGER NOT NULL,

ProductName VARCHAR2(50) NOT NULL,

ListPrice NUMBER(10,2),

Category INTEGER NOT NULL

);

/

INSERT INTO ProductTable VALUES(299,'Chest',99.99,10);

INSERT INTO ProductTable VALUES(300,'Wave Cruiser',49.99,11);

INSERT INTO ProductTable VALUES(301,'Megaland Play Tent',59.99,11);

INSERT INTO ProductTable VALUES(302,'Wind-Up Water Swimmers',2.00,11);

INSERT INTO ProductTable VALUES(303,'Garmin Pocket or Vehicle GPS Navigator',609.99,12);

First, we have to create a sequence for EventTable

CREATE SEQUENCE seq

MINVALUE 1

START WITH 1

INCREMENT BY 1;

CREATE OR REPLACE TRIGGER eventLog

BEFORE UPDATE ON ProductTable

FOR EACH ROW

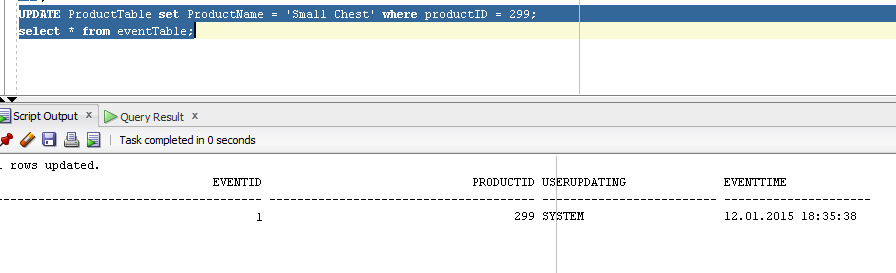
BEGIN

INSERT INTO SYSTEM.EVENTTABLE(eventID, productID, userUpdating, eventTime) values(seq.nextval,:old.productID, user, sysdate);

END;

Test the trigger:

UPDATE ProductTable set ProductName = 'Small Chest' where productID = 299;



### PL/SQL trigger Example 4

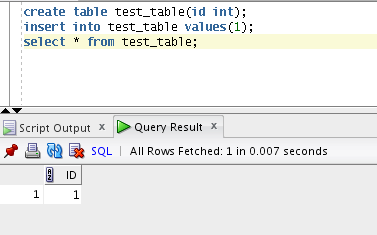
Assume we have a table

create table test\_table(id int);

with one row

insert into test\_table values(1);

select \* from test\_table;



If we create the following trigger

CREATE OR REPLACE TRIGGER test\_table\_check

BEFORE UPDATE ON test\_table

FOR EACH ROW

BEGIN

dbms\_output.put\_line('the row we are updating had value ' || :old.id);

dbms\_output.put\_line('the row we are updating will have after UPDATE a new value ' || :new.id);

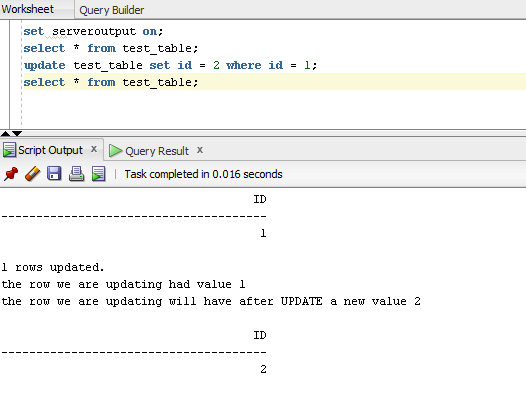
END;

/

SHOW ERRORS;

Then if we update the table

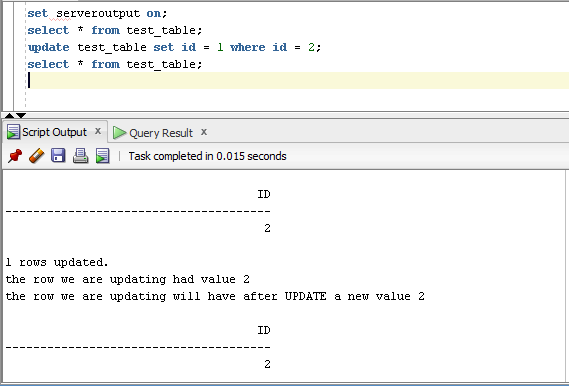
update test\_table set id = 2 where id = 1;



If we would like to prevent updating ID, we can add after begin:

:new.id := :old.id;

Then



### PL/SQL Trigger Example 5

You can write a trigger that will work with multiple DML statements and even detect what column will be updated

Create table:

create table test\_table\_two(id int, name varchar2(20));

Insert one row

insert into test\_table\_two values(10, 'ten');

The trigger can use CASE to determine what type of operation fired the trigger.

CREATE OR REPLACE TRIGGER test\_table\_two\_check

BEFORE UPDATE OR DELETE OR INSERT ON test\_table\_two

FOR EACH ROW

BEGIN

CASE

WHEN INSERTING THEN

DBMS\_OUTPUT.PUT\_LINE('Inserting ' || :new.id || ' ' || :new.name);

WHEN UPDATING('id') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating ID ' || :old.id || ' new value ' || :new.id);

WHEN UPDATING('name') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating NAME ' || :old.name || ' new value ' || :new.name);

WHEN DELETING THEN

DBMS\_OUTPUT.PUT\_LINE('Deleting');

END CASE;

END;

/

SHOW ERRORS;

To test issue SELECT, UPDATE of one column at a time, UPDATE all columns at a time, DELETE one record and DELETE all records.

insert into test\_table\_two values(20, 'twenty');



update test\_table\_two set id = 100 where id = 10;



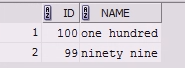
update test\_table\_two set name = 'one hundred' where id = 100;



update test\_table\_two set id = 99, name='ninety nine' where id = 20;



There is no message for update to ninety nine even though it was updated.



This is a limitation of the CASE statement To fix this, we need to change

WHEN UPDATING('id') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating ID ' || :old.id || ' new value ' || :new.id);

WHEN UPDATING('name') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating NAME ' || :old.name || ' new value ' || :new.name);

To

WHEN UPDATING THEN

IF UPDATING('id') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating ID ' || :old.id

|| ' new value ' || :new.id);

END IF;

IF UPDATING('name') THEN

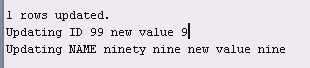
DBMS\_OUTPUT.PUT\_LINE('Updating NAME ' || :old.name

|| ' new value ' || :new.name);

END IF;

And test the code again:

update test\_table\_two set id = 9, name='nine' where id = 99;

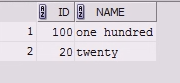


delete from test\_table\_two where id = 99;



insert into test\_table\_two values(20, 'twenty');

select \* from test\_table\_two;



delete from test\_table\_two;



The trigger was executed twice because we had two rows affected.

### Oracle trigger execution order

To demonstrate the order of execution of Oracle trigger create table

CREATE TABLE trigger\_test(

ID integer );

/

INSERT ALL

INTO trigger\_test VALUES (1)

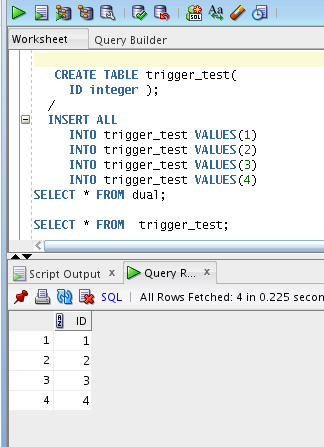
INTO trigger\_test VALUES(2)

INTO trigger\_test VALUES(3)

INTO trigger\_test VALUES(4)

SELECT \* FROM DUAL;

SELECT \* FROM trigger\_test;



First trigger will be a statement level before update trigger

CREATE OR REPLACE TRIGGER statement\_before\_update

BEFORE UPDATE ON trigger\_test

BEGIN

dbms\_output.put\_line('statement level before update trigger');

END;

/

Second trigger will be a row level before update trigger

CREATE OR REPLACE TRIGGER row\_before\_update

BEFORE UPDATE ON trigger\_test

FOR EACH ROW

BEGIN

dbms\_output.put\_line('row level before update trigger old ID= ' || :old.ID || 'new ID=' || :new.ID);

END;

/

Third trigger will be a statement level AFTER update trigger

CREATE OR REPLACE TRIGGER statement\_after\_update

AFTER UPDATE ON trigger\_test

BEGIN

dbms\_output.put\_line('statement level AFTER update trigger');

END;

/

Fourth trigger will be a row level after update trigger

CREATE OR REPLACE TRIGGER row\_after\_update

AFTER UPDATE ON trigger\_test

FOR EACH ROW

BEGIN

dbms\_output.put\_line('row level AFTER update trigger old ID= ' || :old.ID || ' new ID=' || :new.ID);

END;

/

Finally, we will create one more trigger to check OID rule.

CREATE OR REPLACE TRIGGER row\_after\_update\_second

AFTER UPDATE ON trigger\_test

FOR EACH ROW

BEGIN

dbms\_output.put\_line('row level AFTER update SECOND trigger old ID= ' || :old.ID || ' new ID=' || :new.ID);

END;

/

I expect that they will be executed in the following order:

1. before update statement level;
2. FOR EACH ROW
   1. before update row level;
   2. after update row level with highest OID (row\_after\_update\_second);
   3. after update row level with lowest OID (row\_after\_update);
3. after update statement level;

To test it run the following update statement:

UPDATE trigger\_test set ID=ID + 1;

The results are below. I used color to show output statements for rows one and three

statement level before update trigger

row level before update trigger old ID= 1new ID=2

row level AFTER update SECOND trigger old ID= 1 new ID=2

row level AFTER update trigger old ID= 1 new ID=2

row level before update trigger old ID= 2new ID=3

row level AFTER update SECOND trigger old ID= 2 new ID=3

row level AFTER update trigger old ID= 2 new ID=3

row level before update trigger old ID= 3new ID=4

row level AFTER update SECOND trigger old ID= 3 new ID=4

row level AFTER update trigger old ID= 3 new ID=4

row level before update trigger old ID= 4new ID=5

row level AFTER update SECOND trigger old ID= 4 new ID=5

row level AFTER update trigger old ID= 4 new ID=5

statement level AFTER update trigger

Now we will modify ***row\_after\_update\_second*** trigger to add FOLLOWS row\_after\_update to force its execution as a second trigger.

CREATE OR REPLACE TRIGGER row\_after\_update\_second

AFTER UPDATE ON trigger\_test

FOR EACH ROW

FOLLOWS row\_after\_update

BEGIN

dbms\_output.put\_line('row level AFTER update SECOND trigger old ID= ' || :old.ID || ' new ID=' || :new.ID);

END;

/

To test it we have to run an update statement:

UPDATE trigger\_test set ID=ID + 1 where ID=2;

The results are:

statement level before update trigger

row level before update trigger old ID= 2new ID=3

row level AFTER update trigger old ID= 2 new ID=3

row level AFTER update SECOND trigger old ID= 2 new ID=3

statement level AFTER update trigger

The use of FOLLOWING clause changed the order of execution.

# Lab

You have to run every script in this module and understand each line of code before you can start this lab assignment.

1. Post a question at the Help with Labs section of the discussion board or answer already posted question (1 point for the screenshot)
2. Use the product table from this module (do not modify table or column names) and modify the EventLog trigger that it will store all changed (deleted or updated) values from the Product table in the ProductHistory table.

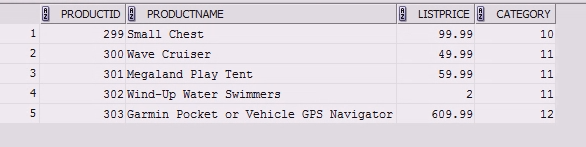
Provide the code of your working trigger (3 points), code you used to create the ProductHistory table (3 points). The code will be copied from your report and tested.

Test your code and include your test case in the report (3 points for code and screenshots of the results) (1 point for each successful test case). To receive points for testing, the **test cases must test different aspects** of your code.

Sample output:

Original data in the table:

SELECT \* from ProductTable;



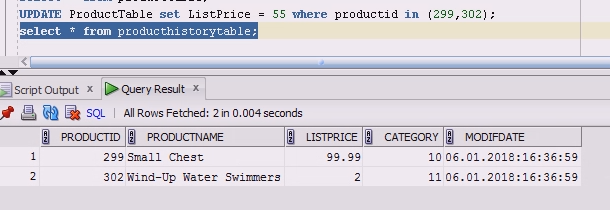
SELECT \* from ProductHistoryTable;



Test update for multiple rows: Update to $55 prices for item 299 to $55.

UPDATE ProductTable SET ListPrice = 55 where ProductID IN (299, 302);

SELECT \* from ProductHistoryTable; to show that two rows showing old prices were inserted.



**Feedback:**

Difficulty (-2 - too easy ... 0 - just right ... 2 - too hard)

Interest level (-2 - low interest ... 0 - just right ... 2 - high interest)

Time to complete (min)

Make a suggestion to improve

What to submit:

One report file that contains PL/SQL code (not screenshots of code) and screenshots of the results to prove functionality.

File name and size should conform to submission standards <http://ksuweb.kennesaw.edu/~speltsve/files/style_and_submission_guide_d2l.pdf>