

LAB BOOK 5

COMP41090 SQL PROGRAMMING

ANDREW DOYLE

STUDENT NUMBER: 12252388

MSc COMPUTER SCIENCE (CONVERSION)

23rd April 2013

TABLE OF CONTENTS

TABLE OF CONTENTS.....	1
LIST OF FIGURES.....	1
INTRODUCTION.....	2
QUESTION 1	3
QUESTION 2	4
QUESTION 3	5
QUESTION 4	6
QUESTION 5	8
QUESTION 6	10
QUESTION 7	11
QUESTION 8	14

LIST OF FIGURES

Figure 1.1 – Simple Loop.....	3
Figure 1.2 – Simple Loop (Remaining output)	3
Figure 2.1 – Nested Loop	4
Figure 3.1 – Prime Number Loop	5
Figure 4.1 - Table before Sequence	6
Figure 4.2 - Sequence Implementation.....	7
Figure 5.1 - Adding Salary attribute to employee table.....	8
Figure 5.2 – Adding Values to Salary Column	9
Figure 5.3 – Implementing and testing the trigger	9
Figure 6.1 - Employee Table after 1000 employees inserted.	10
Figure 7.1 – Creating Procedure	12
Figure 7.2 – Implementation of Exception and Cursor	12
Figure 8.1 – Implementation of Exception Catch	14

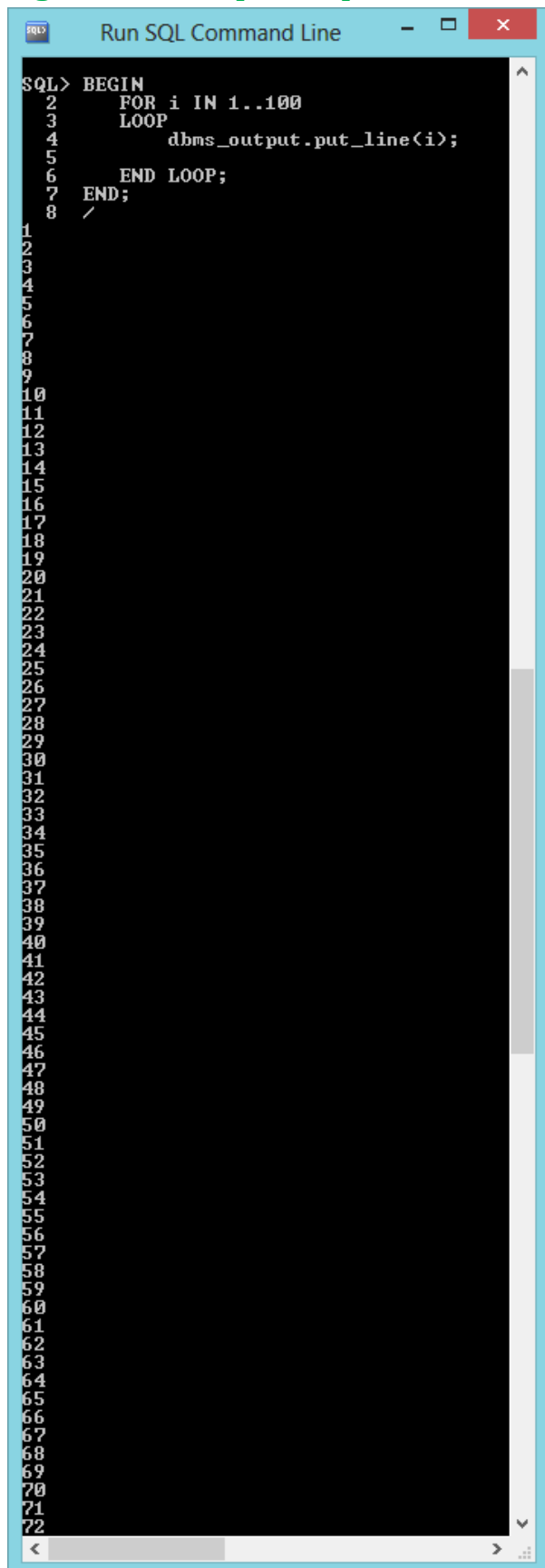
INTRODUCTION

This introduction serves sets out the format of the author's submission for this Lab Book. For each question, the SQL code is provided in this document accompanied by explanatory notes which serve to demonstrate and reinforce knowledge of the topics covered. The notes may prove useful to the author as a reference point for future Lab Book or Project submissions.

In each section screenshots are also provided which provide evidence of the SQL code being implemented in the database together with the actual output to the screen. Additionally, separate .sql files are included with the submission (one for each question), which the Examiner may wish to utilize to test the code.

QUESTION 1

Figure 1.1 – Simple Loop



```
SQL> BEGIN
2   FOR i IN 1..100
3   LOOP
4       dbms_output.put_line(i);
5
6   END LOOP;
7 END;
8 /
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72

SQL USED:

```
BEGIN
    FOR i IN 1..100
    LOOP
        dbms_output.put_line(i);

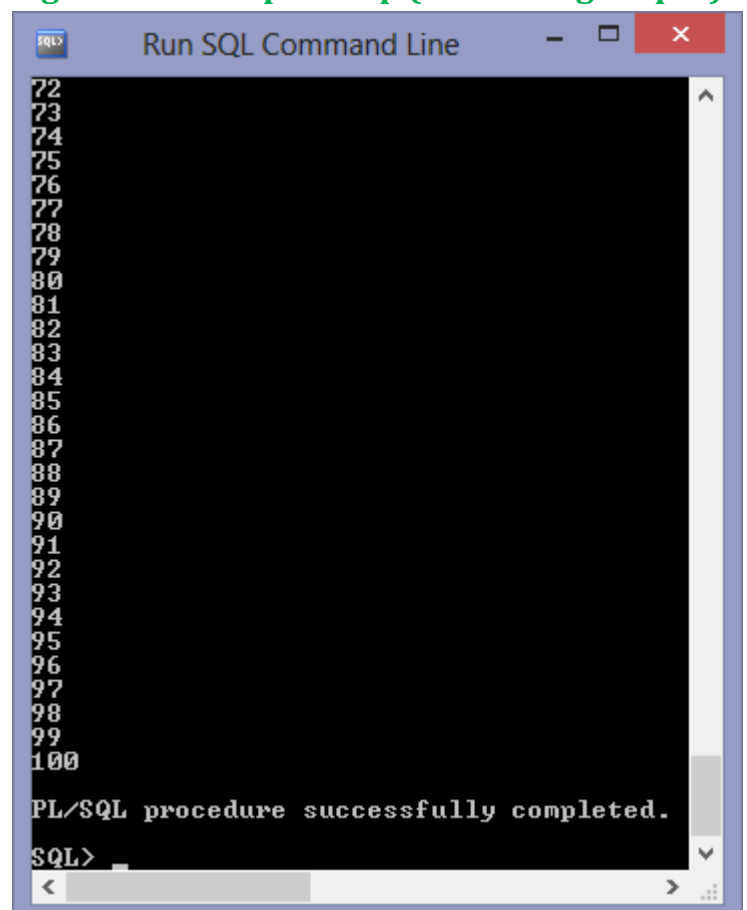
    END LOOP;
END;
/
```

Explanatory Notes

In the line **FOR i IN 1..100**, **i** represents the variable that will be used to output the number (**i** does not need to be separately declared). **IN 1..100** defines the range of values (you could also have a tedious comma separated list of values).

Within the loop **i** is output to the screen using **DBMS_OUTPUT.PUT_LINE**.

Figure 1.2 – Simple Loop (Remaining output)



```
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

PL/SQL procedure successfully completed.
SQL>
```

QUESTION 2

SQL USED:

```
BEGIN

<<outer>>
FOR i IN 1 .. 10 LOOP
    <<inner>>
    FOR j IN 1 .. 10 LOOP
        dbms_output.put_line(I || . ||
        j);
    END LOOP inner;
END LOOP outer;

END;
/
```

Explanatory Notes

The nested loop is labelled using the triangle brackets <<label>>. The author has used the alias <<outer>> to label the outer loop – it could be any label you wish.

The outer loop loops from 1 to 10 for i. The inner loop loops from 1 to 10 for j. In order to output in the required format i.e. 1.1,1.2,1.3 etc., within the **DBMS_OUTPUT** statement, I is joined (using the || characters) to j. A '.' is also added in the join.

Both the inner and outer loops are then ended.

Figure 2.1 – Nested Loop

```
SQL> BEGIN
2
3  <<outer>>
4  FOR i IN 1 .. 10 LOOP
5      <<inner>>
6      FOR j IN 1 .. 10 LOOP
7          dbms_output.put_line(I || '.' || j);
8      END LOOP inner;
9  END LOOP outer;
10
11  END;
12  /
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
1.10
2.1
2.2
2.3
2.4
2.5
2.6
2.7
2.8
2.9
2.10
3.1
3.2
3.3
3.4
3.5
3.6
3.7
3.8
3.9
3.10
4.1
4.2
4.3
4.4
4.5
4.6
4.7
4.8
4.9
4.10
5.1
5.2
5.3
5.4
5.5
5.6
5.7
5.8
5.9
5.10
6.1
6.2
6.3
6.4
6.5
6.6
6.7
6.8
6.9
6.10
7.1
7.2
7.3
7.4
7.5
7.6
7.7
7.8
7.9
```

QUESTION 3

SQL USED:

```

DECLARE encountered NUMBER;

BEGIN
<<outer>>
FOR i IN 1 .. 30 LOOP
    encountered := 0;
    <<inner>>
        FOR j In 2..i-1 LOOP
            IF MOD(i,j) = 0 THEN
                dbms_output.put_line(i);
                encountered := 1;
            END IF;
            EXIT WHEN encountered =1;
        END LOOP inner;
    END LOOP outer;

END;
/

```

Explanatory Notes

Encountered is declared which will be used as an exit condition later in the SQL code. The outer loop loops from 1 to 30 for i. **Encountered** is set to zero.

In the inner loop, j loops from 2 to i-1; so in the first iteration j is 2 and i is 1. An **if-statement** tests if the **modulus** of i and j is zero (**IF MOD (i,j) = 0**, and if this is true then i is **DBMS_OUTPUT** to the screen.

In the third iteration, for example, j is 4 and i is 3, the modulus of j and i is not zero so i is not output to the screen.

Figure 3.1 – Prime Number Loop

```

SQL> DECLARE encountered NUMBER;
2
3 BEGIN
4 <<outer>>
5 FOR i IN 1 .. 30 LOOP
6     encountered := 0;
7     <<inner>>
8         FOR j In 2..i-1 LOOP
9             IF MOD(i,j) = 0 THEN
10                 dbms_output.put_line(i);
11                 encountered := 1;
12             END IF;
13             EXIT WHEN encountered =1;
14         END LOOP inner;
15     END LOOP outer;
16
17 END;
18 /
4
6
8
9
10
12
14
15
16
18
20
21
22
24
25
26
27
28
30
PL/SQL procedure successfully completed.

```

QUESTION 4

SQL USED:

```
CREATE SEQUENCE s2
  START WITH 310
  INCREMENT BY 1;

CREATE OR REPLACE TRIGGER
q4_trigger
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
  :NEW.EMP_ID:=S2.NEXTVAL;
END;
/

BEGIN
INSERT INTO EMPLOYEE (FNAME,
LNAME, MANAGER_EMP_ID)
VALUES('Panda', 'Bear', 304);
END;
/
```

Explanatory Notes

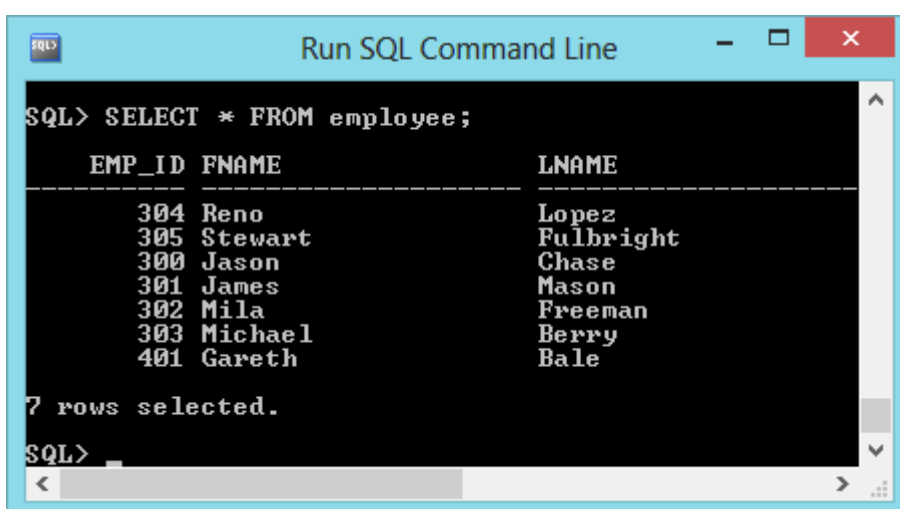
A **sequence** is used in this question to generate values for the **emp_id** primary key. The sequence is named **s2** and is set to start with 310 (emp_id 310). You could also set a min and max value but it is not necessary for this exercise.

Finally, in the sequence, the value is incremented by 1, so the next incremented value after 310 would be 311, 312 etc.

A trigger is then created, named **q4_trigger**, with a **BEFORE INSERT** statement (**ON** the employee table) which specifies the new value of the **emp_id** (using **:NEW.EMP_ID** syntax) is equal to the next value from the **s2** sequence (using **S2.NEXTVAL** syntax). **FOR EACH ROW** was previously specified to ensure the action could be executed multiple times.

To test that the trigger works, an employee with an **FNAME** of 'Panda', **LNAME** of 'Bear' and **MANAGER_EMP_ID** of 304 is inserted into the employee table. Selecting all data from the table as shown in Figure 4.2 demonstrates that the trigger has been implemented successfully.

Figure 4.1 - Table before Sequence

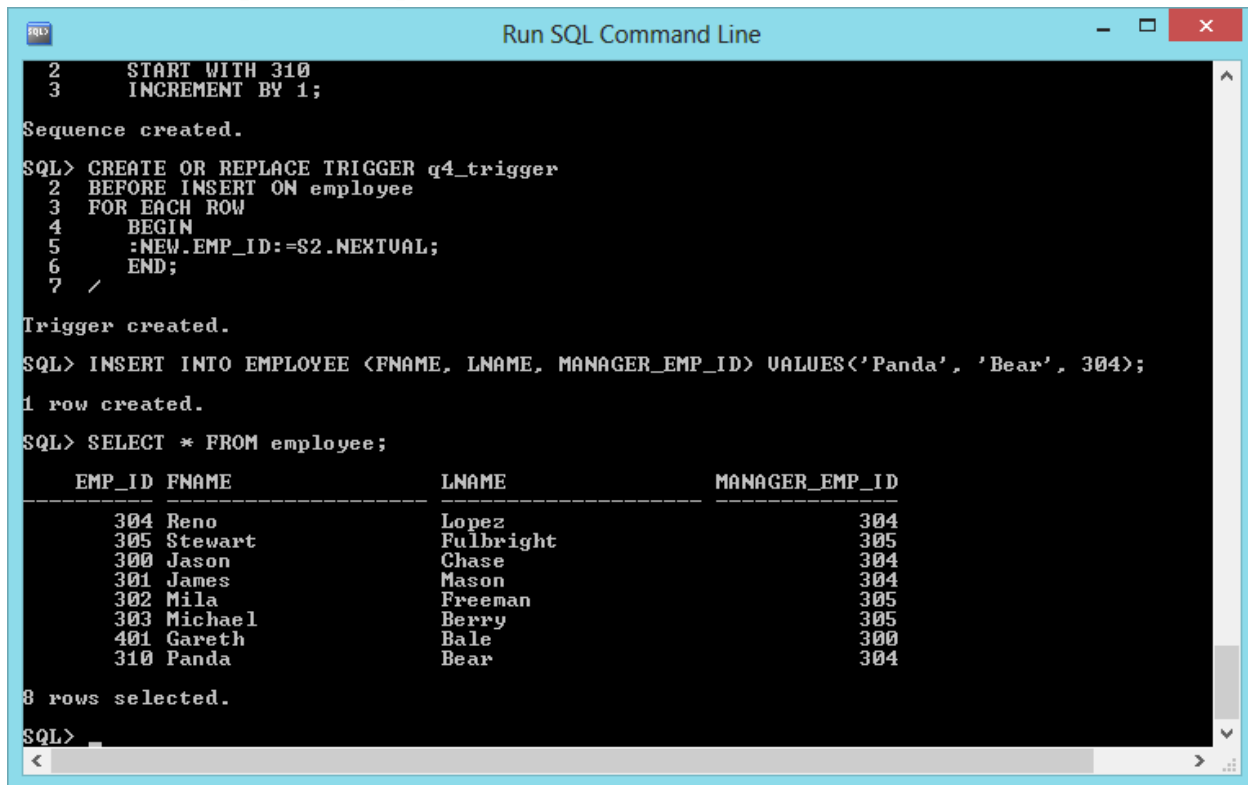


```
SQL> SELECT * FROM employee;
```

EMP_ID	FNAME	LNAME
304	Reno	Lopez
305	Stewart	Fulbright
300	Jason	Chase
301	James	Mason
302	Mila	Freeman
303	Michael	Berry
401	Gareth	Bale

7 rows selected.

```
SQL>
```

Figure 4.2 - Sequence Implementation

The screenshot shows a SQL Command Line window with the following text:

```
2 START WITH 310
3 INCREMENT BY 1;
Sequence created.
SQL> CREATE OR REPLACE TRIGGER q4_trigger
2 BEFORE INSERT ON employee
3 FOR EACH ROW
4 BEGIN
5 :NEW.EMP_ID:=S2.NEXTVAL;
6 END;
7 /
Trigger created.
SQL> INSERT INTO EMPLOYEE (FNAME, LNAME, MANAGER_EMP_ID) VALUES('Panda', 'Bear', 304);
1 row created.
SQL> SELECT * FROM employee;
```

EMP_ID	FNAME	LNAME	MANAGER_EMP_ID
304	Reno	Lopez	304
305	Stewart	Fulbright	305
300	Jason	Chase	304
301	James	Mason	304
302	Mila	Freeman	305
303	Michael	Berry	305
401	Gareth	Bale	300
310	Panda	Bear	304

8 rows selected.

```
SQL>
```


QUESTION 5

SQL USED:

```
ALTER TABLE employee ADD salary NUMBER(10);
DESCRIBE employee;
SELECT * FROM employee;

UPDATE employee SET salary='10000' WHERE emp_id=304;
UPDATE employee SET salary='20000' WHERE emp_id=305;
UPDATE employee SET salary='30000' WHERE emp_id=300;
UPDATE employee SET salary='40000' WHERE emp_id=301;
UPDATE employee SET salary='50000' WHERE emp_id=302;
UPDATE employee SET salary='60000' WHERE emp_id=303;
UPDATE employee SET salary='90000' WHERE emp_id=401;
UPDATE employee SET salary='110000' WHERE emp_id=310;

CREATE OR REPLACE TRIGGER q5trigger
  AFTER INSERT OR UPDATE ON employee
  FOR EACH ROW
BEGIN
  IF :NEW.salary > 100000 THEN
    dbms_output.put_line(:NEW.emp_id || ' ' || :NEW.salary);
  END IF;
END;
/
INSERT INTO EMPLOYEE (FNAME, LNAME, MANAGER_EMP_ID, SALARY) VALUES ('Pandaaa', 'Bearaa', 304, 160000);
```

Explanatory Notes

The aim of this question is to create a trigger which checks if an employee's salary is **set to be** more than 100,000, and if it is, the salary value and employee ID are output to the screen.

Because there was no salary field initially, it was added and values were inserted.

The trigger is created with an **AFTER INSERT OR UPDATE** clause. In other words, after an insert or an update is carried out on the employee table, the actions in the trigger body are carried out.

The body contains an **if-statement** which checks if **the** new row inserted contains a salary field with a value greater than 100,000, and if so, **THEN** the employee id and salary values are **DBMS_OUTPUT** to the screen.

A new employee is inserted into the table with a salary value greater than 100,000. As shown in Figure 5.3, this demonstrates that the trigger has been successfully implemented.

Figure 5.1 - Adding Salary attribute to employee table

The screenshot shows a 'Run SQL Command Line' window. The command prompt shows the following sequence of commands and their outputs:

```
SQL> ALTER TABLE employee ADD salary NUMBER(10);
Table altered.
SQL> DESCRIBE employee;
```

Name	Null?	Type
EMP_ID	NOT NULL	NUMBER(10)
FNAME		NVARCHAR2(20)
LNAME		NVARCHAR2(20)
MANAGER_EMP_ID		NUMBER(10)
SALARY		NUMBER(10)

The command prompt ends with 'SQL>'.

Figure 5.2 – Adding Values to Salary Column

```

SQL> UPDATE employee SET salary='10000' WHERE emp_id=304;
1 row updated.
SQL> UPDATE employee SET salary='20000' WHERE emp_id=305;
1 row updated.
SQL> UPDATE employee SET salary='30000' WHERE emp_id=300;
1 row updated.
SQL> UPDATE employee SET salary='40000' WHERE emp_id=301;
1 row updated.
SQL> UPDATE employee SET salary='50000' WHERE emp_id=302;
1 row updated.
SQL> UPDATE employee SET salary='60000' WHERE emp_id=303;
1 row updated.
SQL> UPDATE employee SET salary='90000' WHERE emp_id=401;
1 row updated.
SQL> UPDATE employee SET salary='110000' WHERE emp_id=310;
1 row updated.
SQL> SELECT * FROM employee;

```

EMP_ID	FNAME	LNAME	MANAGER_EMP_ID	SALARY
304	Reno	Lopez	304	10000
305	Stewart	Fulbright	305	20000
300	Jason	Chase	304	30000
301	James	Mason	304	40000
302	Mila	Freeman	305	50000
303	Michael	Berry	305	60000
401	Gareth	Bale	300	90000
310	Panda	Bear	304	110000

```

8 rows selected.
SQL>

```

Figure 5.3 – Implementing and testing the trigger

```

SQL> CREATE OR REPLACE TRIGGER q5trigger
2   AFTER INSERT OR UPDATE ON employee
3   FOR EACH ROW
4   BEGIN
5     IF :NEW.salary > 100000 THEN
6       dbms_output.put_line(:NEW.emp_id || ' ' || :NEW.salary);
7     END IF;
8   END;
9   /
Trigger created.
SQL> INSERT INTO EMPLOYEE (FNAME, LNAME, MANAGER_EMP_ID, SALARY) VALUES('Pandaa', '00000');
311 160000
1 row created.
SQL>

```

QUESTION 6

SQL USED:

```

DECLARE
    rand_salary NUMBER;
BEGIN
    FOR i IN 1..1000 LOOP
        SELECT DBMS_RANDOM.VALUE(12000,350000) INTO
        rand_salary FROM DUAL;
        INSERT INTO EMPLOYEE (FNAME, LNAME,
        MANAGER_EMP_ID, SALARY) VALUES ('GIANT', 'PANDA',
        304, rand_salary);
    END LOOP;
END;
/

```

Explanatory Notes

A loop is carried out from 1 to 1000. Within the loop **SELECT DBMS_RANDOM.VALUE** is used to insert random values into the temporary variable **rand_salary** (which was declared earlier). The first figure represents the lowest number and the second figure is one number higher than the highest number to be inserted (i.e. 349999 in this scenario).

Then, an insert is carried out, with **rand_salary** being the distinguishing input. The **emp_id** is automatically generated by the trigger and sequence from

Figure 6.1 - Employee Table after 1000 employees inserted.

EMP_ID	FNAME	LNAME	MANAGER_EMP_ID	SALARY
1351	GIANT	PANDA	304	168593
1352	GIANT	PANDA	304	335153
1353	GIANT	PANDA	304	158731
1354	GIANT	PANDA	304	87230
1355	GIANT	PANDA	304	256129
1356	GIANT	PANDA	304	238105
1357	GIANT	PANDA	304	270312
1358	GIANT	PANDA	304	78848
1359	GIANT	PANDA	304	206199
1360	GIANT	PANDA	304	340024
1361	GIANT	PANDA	304	64493
1362	GIANT	PANDA	304	176610
1363	GIANT	PANDA	304	246048
1364	GIANT	PANDA	304	184703
1365	GIANT	PANDA	304	212611
1366	GIANT	PANDA	304	195838
1367	GIANT	PANDA	304	124145
1368	GIANT	PANDA	304	166014
1369	GIANT	PANDA	304	223084
1370	GIANT	PANDA	304	206912
1371	GIANT	PANDA	304	76669
1372	GIANT	PANDA	304	305827
1373	GIANT	PANDA	304	319124
1374	GIANT	PANDA	304	301906
1375	GIANT	PANDA	304	110818
1376	GIANT	PANDA	304	318457
1377	GIANT	PANDA	304	344787
1378	GIANT	PANDA	304	27046
1379	GIANT	PANDA	304	129478
1380	GIANT	PANDA	304	217110
1381	GIANT	PANDA	304	247336
1382	GIANT	PANDA	304	235616
1383	GIANT	PANDA	304	165384
1384	GIANT	PANDA	304	177485
1385	GIANT	PANDA	304	217271
1386	GIANT	PANDA	304	277236
1387	GIANT	PANDA	304	109155
1388	GIANT	PANDA	304	298947
1389	GIANT	PANDA	304	61954
1390	GIANT	PANDA	304	220602
1391	GIANT	PANDA	304	171504
1392	GIANT	PANDA	304	136825
1393	GIANT	PANDA	304	180393
1394	GIANT	PANDA	304	61937
1395	GIANT	PANDA	304	219483
1396	GIANT	PANDA	304	267463
1397	GIANT	PANDA	304	335638
1398	GIANT	PANDA	304	168138
1399	GIANT	PANDA	304	279334
1400	GIANT	PANDA	304	163878
1401	GIANT	PANDA	304	222397

1000 rows selected.
SQL>

QUESTION 7

SQL USED

```
-- Create Procedure
CREATE OR REPLACE PROCEDURE q7_salary_over_ninety_k
IS
BEGIN
    UPDATE employee SET salary = '90000';
END q7_salary_over_ninety_k;
/

--Declare the Exception

DECLARE
q7_exception EXCEPTION;
--Begin the cursor
BEGIN
DECLARE
CURSOR lab_5_q7 IS -- Declaration of cursor name
-- Below, the columns are selected from the employee table
SELECT emp_id, fname, sname, manager_emp_id, salary FROM employee;
-- Below, a variable is declared which represents every column in a
row
q7_cursor lab_5_q7 %ROWTYPE;

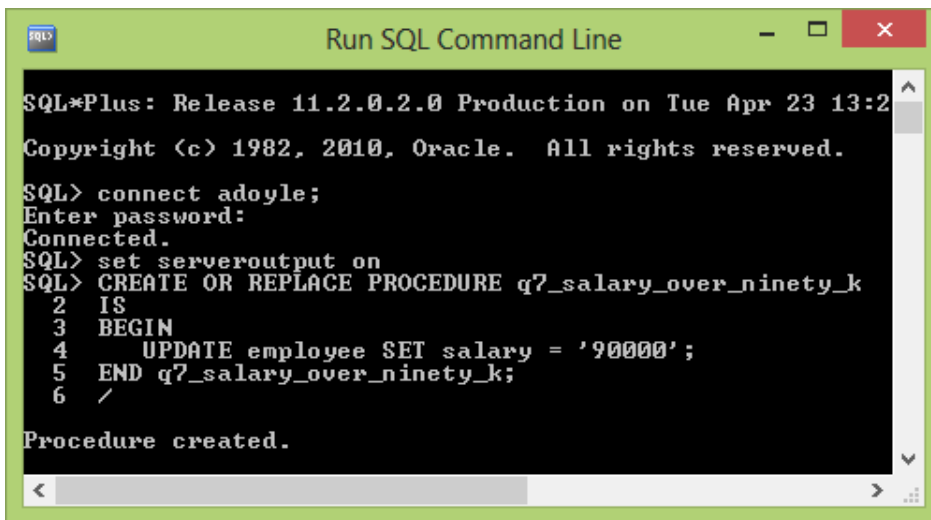
BEGIN -- sub-block begins
OPEN lab_5_q7;
FETCH lab_5_q7 INTO q7_cursor;
WHILE lab_5_q7 %FOUND LOOP
    IF q7_cursor.salary < 20000 THEN RAISE_APPLICATION_ERROR
    (-123456, 'You have employees with salaries < €20,000, you should
pay them more than that!');
    END IF;
IF q7_cursor.salary > 90000 THEN RAISE q7_exception;
END IF;
FETCH lab_5_q7 INTO q7_cursor;

END LOOP;
CLOSE lab_5_q7;
END; -- sub-block ends

EXCEPTION
    WHEN q7_exception THEN
BEGIN
    q7_salary_over_ninety_k;
RAISE; -- reraise the current exception
END;

/
```

Figure 7.1 – Creating Procedure



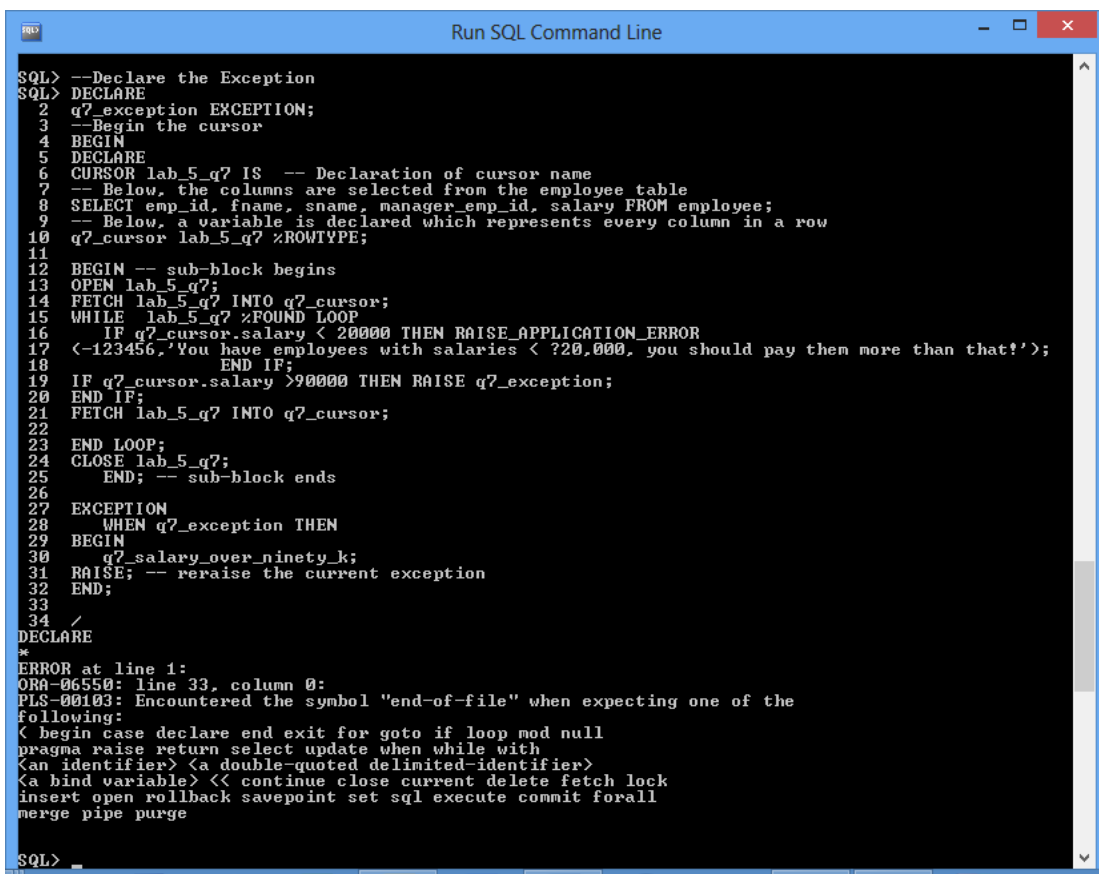
```

SQL*Plus: Release 11.2.0.2.0 Production on Tue Apr 23 13:2
Copyright (c) 1982, 2010, Oracle. All rights reserved.

SQL> connect ad Doyle;
Enter password:
Connected.
SQL> set serveroutput on
SQL> CREATE OR REPLACE PROCEDURE q7_salary_over_ninety_k
2 IS
3 BEGIN
4     UPDATE employee SET salary = '90000';
5 END q7_salary_over_ninety_k;
6 /

Procedure created.
  
```

Figure 7.2 – Implementation of Exception and Cursor



```

SQL> --Declare the Exception
SQL> DECLARE
2 q7_exception EXCEPTION;
3 --Begin the cursor
4 BEGIN
5 DECLARE
6 CURSOR lab_5_q7 IS -- Declaration of cursor name
7 -- Below, the columns are selected from the employee table
8 SELECT emp_id, fname, sname, manager_emp_id, salary FROM employee;
9 -- Below, a variable is declared which represents every column in a row
10 q7_cursor lab_5_q7 %ROWTYPE;
11
12 BEGIN -- sub-block begins
13 OPEN lab_5_q7;
14 FETCH lab_5_q7 INTO q7_cursor;
15 WHILE lab_5_q7 %FOUND LOOP
16     IF q7_cursor.salary < 20000 THEN RAISE_APPLICATION_ERROR
17     (-123456, 'You have employees with salaries < ?20,000, you should pay them more than that!');
18     END IF;
19     IF q7_cursor.salary > 90000 THEN RAISE q7_exception;
20     END IF;
21     FETCH lab_5_q7 INTO q7_cursor;
22
23 END LOOP;
24 CLOSE lab_5_q7;
25 END; -- sub-block ends
26
27 EXCEPTION
28 WHEN q7_exception THEN
29 BEGIN
30     q7_salary_over_ninety_k;
31 RAISE; -- reraise the current exception
32 END;
33 /
DECLARE
*
ERROR at line 1:
ORA-06550: line 33, column 0:
PLS-00103: Encountered the symbol "end-of-file" when expecting one of the
following:
< begin case declare end exit for goto if loop mod null
pragma raise return select update when while with
<an identifier> <a double-quoted delimited-identifier>
<a bind variable> << continue close current delete fetch lock
insert open rollback savepoint set sql execute commit forall
merge pipe purge
SQL>
  
```

[Read here for more information on %ROWTYPE](#)

Explanatory Notes

A procedure is created to specify what happens (update to 90000) when a salary is found to be over 90000. An exception is then declared. A cursor is then created and all columns are selected from the table. %ROWTYPE is used when creating the variable **q7_cursor**; this represents every column in a row.

Then, the cursor is opened and fetched into the **q7_cursor**. A loop is carried out and if a salary is found to be less than 20,000, an application error is raised together with a custom output message. Another if-statement tests if any salary fields are over 90,000, and if so, the exception **q7_exception** is raised.

In the exception declaration, the procedure is called to update the salary fields and is re-raised.

QUESTION 8

SQL USED

DECLARE

```
firstname VARCHAR2(50);
```

BEGIN

```
SELECT fname INTO firstname FROM employee WHERE  
emp_id = 1200;
```

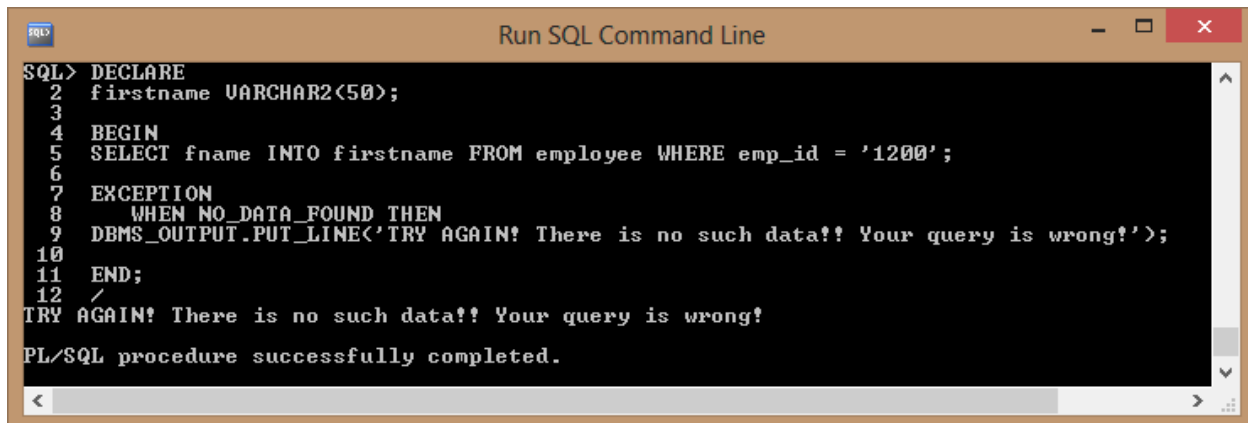
EXCEPTION

```
    WHEN NO_DATA_FOUND THEN  
DBMS_OUTPUT.PUT_LINE('TRY AGAIN! There is no such  
data!! Your query is wrong!');
```

```
END;
```

```
/
```

Figure 8.1 – Implementation of Exception Catch



```
SQL> DECLARE  
2  firstname VARCHAR2(50);  
3  
4  BEGIN  
5  SELECT fname INTO firstname FROM employee WHERE emp_id = '1200';  
6  
7  EXCEPTION  
8  WHEN NO_DATA_FOUND THEN  
9  DBMS_OUTPUT.PUT_LINE('TRY AGAIN! There is no such data!! Your query is wrong!');  
10  
11  END;  
12  /  
TRY AGAIN! There is no such data!! Your query is wrong!  
PL/SQL procedure successfully completed.
```

Explanatory Notes

A variable is declared, **firstname**. A **SELECT INTO** statement takes the **fname** attribute from the employee table where the employee id is 1200. Such an employee ID does not exist.

An exception is created to the **NO_DATA_FOUND** error message, so that when no data is found, the custom error message is displayed in lieu of the pre-defined error; as shown in Figure 8.1.