IS 480. Advanced Database

Lecture Notes

Spring 2019

C. Sophie Lee Ph.D.

Professor
Department of Information Systems
College of Business Administration
California State University, Long Beach

Part I

SQL

SQL Topic 1. Basic SQL

Note! For students who do not have Oracle background, please read Chapters of Oracle Complete Reference

Data Definition Language (DDL)

Creating a Table

```
create table t (fd_name1 fd_type, fd_name2 fd_type, ...);
        create table customers (ssn char(9),
                                cname char(30),
                                caddress char(30),
                                amount number(10,2));
```

* field type:

```
char(n)
varchar2(n)
date
number(x,n)
```

Deleting a table

```
drop table t;
        drop table customers;

truncate table t;
        truncate table customers;
```

/* delete all records in the table but not the field definitions */

Add A Column

```
alter table t add (fx fd_type, fy fd_type);
        alter table customers add (zip char(5), phone char(10));
```

Modify Column Definition

```
alter table table_name modify (f1 new_type, f2 new_type, ...);
        Alter table customers modify (cname char(50));
```

/* Certain restrictions apply. See p.388 */

Display Table Structure

```
describe t;
        describe customers;
```

Data Manipulation Language (DML)

Insert Records

```
insert into t values (f1value, f2value,...);
insert into t (f1, f2,...) values (f1value, f2value,...);
```

Delete Records

```
delete from t where [conditions];
```

Update Records

```
update t set f1=newvalue1, f2=newvalue2, ... where [conditions];
```

Display Records/Join Tables

```
select * from t;
select * from t where [conditions];
select f1, f2, ...from t;
select f1, f2, ...from t where [conditions];
```

Conditions

Mathematical and Logical operators

=, <>, !=, >, <, >=, <=, and, or, not

ex: *... where price > 10;*

ex: *... where price > 10 and code <> 1234;*

is null, is not null

ex: ... where major is null;
ex: ... where major is not null;
ex: ... where major=null; -- does not work!

Wildcard

like

Compares to similar character string

% represents any group of characters
_ represents one character

ex: *... where name like 'S%';*
... where name like '_llen';

Sorting

... order by

default is ascending.

... order by f1;

add DESC for descending

... order by f1 desc;

Join Mulitple Tables

```
SELECT t1.f1, t2.f2, ...
  FROM t1, t2, ...
 WHERE [conditions];
```

ex. *select sname from student, advisor where student.anum=advisor.anum;*

Transaction Control Statements

Save and Undo records

commit;

rollback; /*"Undo" to the last "commit" status */

Primary Key Commands

When the pk contains a single column

ex:
```
create table t1 (
        f1 char(2) primary key, ....
```

When the pk contains multiple columns

ex:
```
create table t2 (
f1 char(2), f2 char(2), f3 char(2),...,
        primary key (f1, f2));
```

ADD new constraints:
```
alter table t2 add primary key (f1,f2);
```

DROP constraints:
```
alter table t2 drop primary key (f1,f2);
```

Foreign Key Commands

When the fk contains a single column:
```
create table child (
        f1 char(2),
        f2 char(2) constraint c1 references parent(f1),
        ....);
```

When the fk contains multiple columns:
```
create table child (
        f1 char(2), f2 char(2),...,
constraint c2 foreign key (f1,f2) references parent(f1,f2)
        ... );

alter table child add constraint c1
        foreign key (f1,f2) references parent(f1,f2));

alter table child drop constraint c1;
```

Other Commands

dual
```
select 1+2 from dual;
select sysdate from dual;
```

USER
```
insert into Trans_Audit values (user, sysdate, ....);
```

rownum
```
select * from transaction
where rownum < 10
order by amount desc; -- what does this do?

select * from transaction
where rownum = 10
order by amount desc;  -- will this work?
```

in
```
select sname from student where major in ('IS,'FIN','MKT');
select sname from student where major not in ('IS,'FIN','MKT');
```

Sequence

```
create sequence transaction_number start with 10000;

insert into trans values (transaction_number.nextval, ......);
select Transaction_Number.currval from dual;
```

Sample Code

```
SQL> drop table trans;
drop table trans
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> drop table stores;
drop table stores
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> drop table items;
drop table items
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> create table items (
  2     i# varchar2(3) primary key,
  3     iname varchar2(15),
  4     type varchar2(4),
  5     price number(8,2));

Table created.

SQL> create table stores (
  2     s# varchar2(2) primary key,
  3     region varchar2(4));

Table created.

SQL> create table trans (
  2     t# number(8) primary key,
  3     tdate date,
  4     i# varchar2(3) constraint fk_trans_i# references items (i#),
  5     s# varchar2(2) constraint fk_trans_s# references stores (s#),
  6     qty number(8));

Table created.

SQL> select table_name from user_tables;

TABLE_NAME
------------------------------
ITEMS
STORES
TEST
TRANS

SQL> drop table test;

Table dropped.

SQL> select table_name from user_tables;

TABLE_NAME
------------------------------
BIN$zqAEydoGRD2H+QFtm+hxXg==$0
ITEMS
STORES
TRANS

SQL> purge recyclebin;

Recyclebin purged.

SQL> select table_name from user_tables;

TABLE_NAME
------------------------------
```

```
ITEMS
STORES
TRANS

SQL> describe items;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 I#                                        NOT NULL VARCHAR2(3)
 INAME                                              VARCHAR2(15)
 TYPE                                               VARCHAR2(4)
 PRICE                                              NUMBER(8,2)

SQL> describe stores;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 S#                                        NOT NULL VARCHAR2(2)
 REGION                                             VARCHAR2(4)

SQL> describe trans;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 T#                                        NOT NULL NUMBER(8)
 TDATE                                              DATE
 I#                                                 VARCHAR2(3)
 S#                                                 VARCHAR2(2)
 QTY                                                NUMBER(8)

SQL> insert into items values ('101','Batman','DVD',14.99);

1 row created.

SQL> insert into items values ('102','Batman','VHS',8.99);

1 row created.

...


SQL> insert into stores values ('28','LA');

1 row created.

SQL> insert into stores values ('29','LA');

1 row created.

...

SQL> insert into trans values (1001,'19-aug-12','101','28',10);

1 row created.

SQL> insert into trans values (1002,sysdate-10,'101','28',8);

1 row created.

...


SQL> commit;

Commit complete.

SQL> select * from items;

I#  INAME           TYPE      PRICE
--- --------------- ---- ----------
101 Batman          DVD       14.99
102 Batman          VHS        8.99
103 Lord of Ring    DVD       18.99
104 Mask            VHS        5.99

SQL> select * from stores;

S# REGI
-- ----
28 LA
29 LA
30 SD
```

```
31 SF
32 LA
33 SD

6 rows selected.

SQL> select * from trans;

        T# TDATE     I#  S#        QTY
---------- --------- --- -- ----------
      1001 19-AUG-12 101 28         10
      1002 19-AUG-12 101 28          8
      1003 19-AUG-12 102 28          3
      1004 20-AUG-12 102 32          1
      1005 22-AUG-12 101 29         11
      1006 26-AUG-12 102 29         12
      1007 26-AUG-12 104 30         15
      1008 26-AUG-12 103 30         19

8 rows selected.

SQL> update trans
  2     set qty=21,
  3         s#=29
  4     where t#=1008;

1 row updated.

SQL> commit;

Commit complete.

SQL> delete from trans
  2     where tdate between '19-aug-12' and '21-aug-12';

4 rows deleted.

SQL> rollback;

Rollback complete.

SQL> select * from trans where s# is null;

no rows selected

SQL> select * from stores where region like 'S%';

S# REGI
-- ----
30 SD
31 SF
33 SD

SQL> select * from trans
  2     where tdate>'15-aug-12'
  3     order by qty;

        T# TDATE     I#  S#        QTY
---------- --------- --- -- ----------
      1004 20-AUG-12 102 32          1
      1003 19-AUG-12 102 28          3
      1002 19-AUG-12 101 28          8
      1001 19-AUG-12 101 28         10
      1005 22-AUG-12 101 29         11
      1006 26-AUG-12 102 29         12
      1007 26-AUG-12 104 30         15
      1008 26-AUG-12 103 29         21

8 rows selected.

SQL> -- Join Tables, calculated field, column alias
SQL> select t#, s#, items.i#, price, qty, price*qty amount
  2     from trans, items
  3     where trans.i#=items.i#
  4     order by items.i#, qty;

        T# S# I#       PRICE        QTY     AMOUNT
---------- -- --- ---------- ---------- ----------
      1002 28 101      14.99          8     119.92
```

```
      1001 28 101      14.99         10      149.9
      1005 29 101      14.99         11     164.89
      1004 32 102       8.99          1       8.99
      1003 28 102       8.99          3      26.97
      1006 29 102       8.99         12     107.88
      1008 29 103      18.99         21     398.79
      1007 30 104       5.99         15      89.85

8 rows selected.

SQL> -- Table alias
SQL> select t#, s#, i.i#, price, qty, price*qty amount
  2     from trans t, items i
  3     where t.i#=i.i#
  4     order by i.i#, qty;

        T# S# I#       PRICE        QTY     AMOUNT
---------- -- --- ---------- ---------- ----------
      1002 28 101      14.99          8     119.92
      1001 28 101      14.99         10      149.9
      1005 29 101      14.99         11     164.89
      1004 32 102       8.99          1       8.99
      1003 28 102       8.99          3      26.97
      1006 29 102       8.99         12     107.88
      1008 29 103      18.99         21     398.79
      1007 30 104       5.99         15      89.85

8 rows selected.

SQL> -- the DUAL table
SQL> select 1+2 from dual;

       1+2
----------
         3

SQL> select sysdate from dual;

SYSDATE
---------
29-AUG-12

SQL> select * from dual;

D
-
X

SQL> describe dual;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 DUMMY                                              VARCHAR2(1)

SQL> -- User
SQL> select user from dual;

USER
------------------------------
SOPHIE480

SQL> -- rownum
SQL> select * from trans where rownum<4;

        T# TDATE     I#  S#        QTY
---------- --------- --- -- ----------
      1001 19-AUG-12 101 28         10
      1002 19-AUG-12 101 28          8
      1003 19-AUG-12 102 28          3

SQL> select * from trans where rownum=4;

no rows selected
```

```
SQL> -- I want to find the smallest 3 orders, would this code work? .. no..
SQL> select * from trans where rownum<4 order by qty;

        T# TDATE     I#  S#        QTY
---------- --------- --- -- ----------
      1003 19-AUG-12 102 28          3
      1002 19-AUG-12 101 28          8
      1001 19-AUG-12 101 28         10

SQL> -- Try this :D
SQL> select * from
  2     (select * from trans order by qty)
  3     where rownum<4;

        T# TDATE     I#  S#        QTY
---------- --------- --- -- ----------
      1004 20-AUG-12 102 32          1
      1003 19-AUG-12 102 28          3
      1002 19-AUG-12 101 28          8

SQL> -- RowID
SQL> select rowid, trans.* from trans;

ROWID                      T# TDATE     I#  S#        QTY
------------------ ---------- --------- --- -- ----------
AACD9GAAEAABGwNAAA       1001 19-AUG-12 101 28         10
AACD9GAAEAABGwNAAB       1002 19-AUG-12 101 28          8
AACD9GAAEAABGwNAAC       1003 19-AUG-12 102 28          3
AACD9GAAEAABGwNAAD       1004 20-AUG-12 102 32          1
AACD9GAAEAABGwNAAE       1005 22-AUG-12 101 29         11
AACD9GAAEAABGwNAAF       1006 26-AUG-12 102 29         12
AACD9GAAEAABGwNAAG       1007 26-AUG-12 104 30         15
AACD9GAAEAABGwNAAH       1008 26-AUG-12 103 29         21

8 rows selected.

SQL>
```

(Lab) ORACLE surroundings

Step 1. Log in to ORACLE

Open SQL: Start --> Programs --> Oracle .... --> SQL*Plus

Log in: A log in screen appears. Log in to ORACLE by (your username and password will be announced in class)

`username`	_____	press [tab]
`password`	_____	press [tab]
`host`	_____	press [enter]

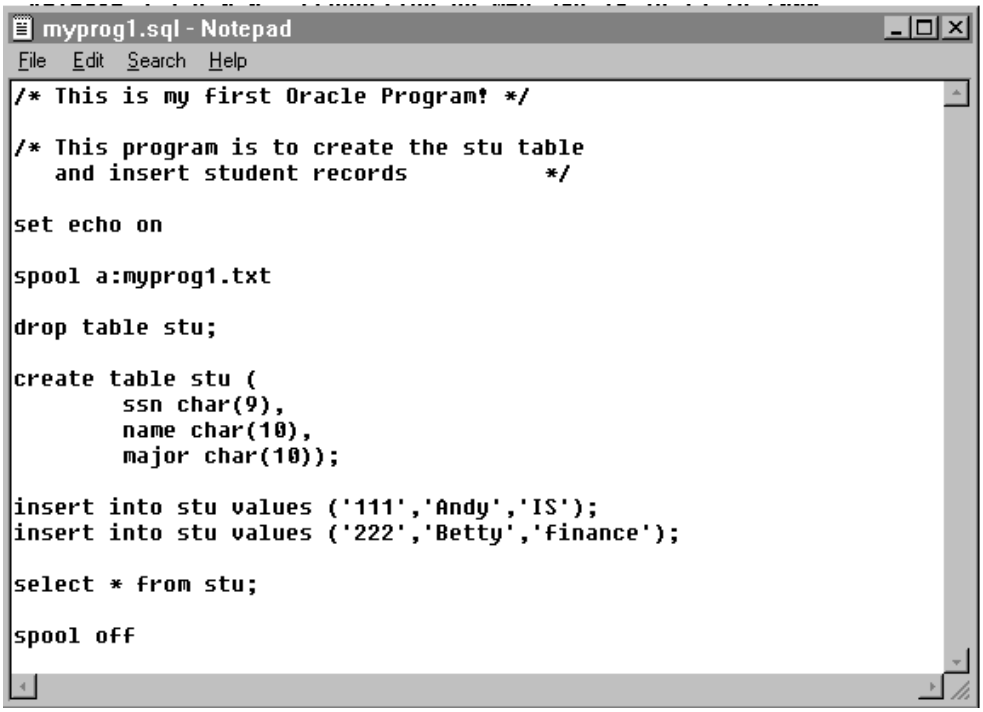
You should see the `SQL>` prompt now.

Log off: To log-off from Oracle, type **`exit`** at the prompt. Another way to exit is to close the window. Make sure you exit Oracle properly.

Step 2. Write SQL commands in a text file

Open Notepad: Choose Start -→ Programs → Accessories → Notepad

At the Notepad file, enter the following commands:



now choose **File - Save.** Specify your filename as **`a:myprog1.SQL`**.

Oracle Commands:

`set echo on` -- this tells Oracle to reprint your commands when running it.

`spool a:myprog1.txt` -- this tells Oracle to record screen output to the file `myprog1.txt` on Drive A. The recording stops when the program comes to the `spool off` command. It is common practice to call your SQL program files `.sql` and output (spool) file `.txt`.

`drop table stu;` -- This is the common practice to delete old, existing tables before your create table... command. Since your account is brand new, this will result in a Table or View Not Exist... error message, which is alright.

Step 3. Run the Program

Go back to the Oracle SQL*Plus window (by clicking anywhere on the window).

To run your program, at the SQL prompt, enter

```
SQL> start a:myprog1.sql
```

press [enter]. The program will be executed.

Your output will look something like this:

```
SQL> spool a:myprog1.txt
SQL>
SQL> drop table stu;
drop table stu
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
SQL> create table stu (
  2          ssn char(9),
  3          name char(10),
  4          major char(10));

Table created.

SQL>
SQL> insert into stu values ('111','Andy','IS');

1 row created.

SQL> insert into stu values ('222','Betty','finance');

1 row created.

SQL>
SQL> select * from stu;

SSN       NAME       MAJOR
--------- ---------- ----------
111       Andy       IS
222       Betty      finance

SQL>
```

Your execution should look similar to this. Scroll the window up or [PageUp] to check your execution. If you do not see any problems, that's great! You can go directly to Step 5. Otherwise, you need to debug your program. Go to Step 4.

Step 4. Debug the Program

If there is any unintended error message (which is quite normal), you can edit your program file and run it again (this process is called "debugging"). Go to the Notepad window and check if there is any typos. Repeat Step 3 to run it again. Do this until there is no error.

Step 5. View and Print your "spool" file

with the **`spool a:myprog1.txt`** command, your screen output is now captured in the file `a:myprog1.txt`.

To view it, go to Notepad and choose File → Open → ..., and open a:myprog1.txt.

You will see the screen output of your a:myprog1.sql program.

If you want to print it, choose **File-Print**

** This is the way that you will turn in most of the homework.

Step 6. Write and Run another program

Let's write another Oracle program which would query the STU table that you just created.

Go to Notepad and choose File → New.

At the Notepad file, enter the following commands:

```
/* Program 2. Query the STU table */

set echo on

spool a:myprog2.txt

select * from stu;
select name from stu where major='IS';
insert into stu values ('333','Cindy','IS');
commit;
select name from stu where major='IS';

spool off
```

now choose **File - Save.** Specify the new file name to be `a:myprog2.sql`.

Go back to Oracle SQL*Plus.

To run your program, at the SQL prompt, enter

```
SQL> start a:myprog2.sql
```

press [enter], and see what happens.

with the **`spool a:myprog2.txt`** command, your screen output is now captured in the file **`a:myprog2.txt`**.

To view it, go to Notepad and choose File → Open → ..., and open a:myprog2.txt.

You will see the screen output of your a:myprog2.sql program.

If you want to print it, choose **File-Print**

Step 7. Write Commands Directly at the SQL> Prompt

You can also write commands directly at the SQL> prompt.

At the SQL> prompt, write

```
SQL> select * from stu;
```

Press [Enter], and see the command being executed.

Note: Usually people write commands in a text file so they can be executed together. The program is sometimes called a "script". If you have several commands that you are likely to rerun them in your future, it is better to write them in a script file so you do not have to type them over and over again. However, if you just want to check something quickly, you can enter the command directly at the SQL> prompt.

Exercise 1.1

Make sure you can do the following without any problem.

1. Write the following SQL codes in a `myprog3.sql` file. Make sure you have `set echo on` in the beginning of the file. Also, spool your output to a file named `myprog3.txt`.

```
select * from stu;
select * from stu where ssn='333';
select ssn, name from stu where ssn='333';
```

2. Run `myprog3.sql` until there is no error.
3. View the content of `myprog3.txt`.

Exercise 1.2

Download **create480tables.sql** from BeachBoard and run it under your account. You should have something similar to this (please check your actual tables and data):

Students

SNUM	SNAME	Standing	Major	GPA	MajorGPA
101	Andy	4	IS	2.8	3.2
102	Betty	2		3.2	
103	Cindy	3	IS	2.5	3.5
104	David	2	FIN	3.3	3.0
105	Ellen	1		2.8	
106	Frank	3	MKT	3.1	2.9

Courses

DEPT	CNUM	CTITLE	CrHr	Standing
IS	300	Intro to MIS	3	2
IS	301	Statistics	3	2
IS	310	Business Comm	3	2
IS	355	Networks	3	3
IS	380	Database	3	3
IS	385	Systems	3	3
IS	480	Adv Database	3	4

SchClasses

CallNum	Year	Semester	DEPT	CNUM	Section	Capacity
10110	2013	Sp	IS	300	1	45
10115	2013	Sp	IS	300	2	35
10120	2013	Sp	IS	380	1	35
10125	2013	Fa	IS	300	1	118
10130	2013	Fa	IS	301	1	33
....		.....	....	....	....	.....

Majors

Major	MDesc
ACC	Accounting
FIN	Finance
IS	Info Sys
MKT	Marketing

Enrollments

SNUM	CallNum	GRADE
101	10110	F
102	10110	A
103	10120	A
101	10125	
102	10130	

PreReq

Dept	Cnum	PDept	PCnum
IS	380	IS	300
IS	380	IS	301
IS	380	IS	310
IS	385	IS	310
IS	355	IS	300
IS	480	IS	380

1. Use SQL commands to see which tables you have.
2. Use SQL commands to display table structure of the tables: Students, Enrollments, SchClasses.
3. Use SQL commands to display data in the following tables: Students, Enrollments, SchClasses.
4. Insert a new student record to the STUDENTS table. The student's Snum is 107, name is George, he has not declared a major and he has no GPA (leave them null).
5. Display the `SName` and `Major` columns of the Students table.
6. Update student 102's GPA to 4.0 and major to 'IS'.
7. Save the previous update (Hint: use commit;)
8. Delete all <u>enrollment</u> records of student 101.
9. "Undo" the previous deletion.
10. Display the <u>name</u> of students who were enrolled in 2009 'Sp'.
11. Display Snum, Sname, and Callnum of enrollments where no grade is assigned (grade is null).
12. Display grade that student 101 received from taking IS 300 during Spring 2009.
13. Display Callnum, Dept, and Cnum of courses that student 102 has received an "A" from.
14. Display students whose name start with 'C'.

SQL Topic 2. Join, Group

Join

```
Select column1, column2, ...
From table1, table2, ...
Where conditions;
```

Exercise 2.1

According to tables in Exercise 1.2, answer the following questions:

1. Display SName and Major of students who are enrolled in IS 300 courses during Spring 2013.

2. Display SName and Major of students who are enrolled in IS 300 courses during Spring 2013, who are not IS major.

3. Display Dept, Cnum, Title of courses student 101 took during Spring 2013.

4. Andy has taken IS 300 several times. Display the year, semester, and grade where he took IS 300.

5. Display Sname of students who received an "A" in IS 300 and who is not an IS major.

6. A student is interested in taking IS 380, and he's like to know what are the prerequisites of IS 380. Write a query to display Dept, CNum, and Title of those prerequisite courses.

Group Functions

```
select group_function(f2) ... from table;

select ... count(field)
select ... count(*)
select ... sum(field)
select ... avg (field)
select ... max(field)
select ... min(field)
select ... distinct(field)
```

Students

Snum	SName	Standing	Major	GPA	Gender
101	Andy	3	IS	2.8	M
102	Betty	4	ACC	3.2	F
103	Cindy	2	IS	2.5	F
104	David	2	FIN	3.3	M
105	Ellen	1	IS	2.8	M
106	Frank	3	MKT	3.1	F

Courses

Dept	CNum	CTitle	CrHr	Standing
BIO	101	Biology Lab	1	2
IS	300	Intro to MIS	3	2
IS	301	Statistics	3	3
IS	310	Business Comm	3	3
ACC	300	Basic Accounting	4	3
ACC	480	Adv Accounting	4	4

SchClasses

CallNum	Year	Semester	Dept	CNum	Section	Capacity
10110	2013	Sp	IS	300	1	45
10115	2013	Sp	IS	300	2	35
10120	2013	Sp	BIO	101	1	35
10125	2013	Fa	ACC	300	1	118
10130	2013	Fa	ACC	300	2	33

Majors

Major	MDesc
ACC	Accounting
FIN	Finance
IS	Info Sys
MKT	Marketing
BIO	Biology

Enrollments

SNum	CallNum	Grade	GdPt
101	10110	A	4
102	10110	B	3
103	10120	A	4
101	10125	C	2
102	10130	F	0

Exercise 2.2

1. In one query, display how many students are 'IS' major.
2. In one query, display how many students are 'Senior' (standing=4).
3. In one query, display the average GPA of all students.
4. In one query, display the average GPA of 'M'ale students.
5. In one query, display how many courses student 101 has taken.
6. In one query, display how many total credit units student 101 has taken.
7. Failed courses (ie, grade is 'F') do not count toward graduation credit units. In one query, display how many total credit units student 101 has accumulated toward graduation.
8. Total grade point is the total of your grade point multiples the credit units of the course. For instance, if a student received an C (2 grade points) of a 3-unit course, and a A (4 grade points) of a 1-unit course, then his/her total grade points is (2x3)+(4x1)=10. write a query to calculate student 101's total grade points.
9. GPA is calculated by total grade points divided by total credit units. For instance, if a student received an C (2 grade points) of a 3-unit course, and a A (4 grade points) of a 1-unit course, his/her GPA is ((2x3)+(4x1))/(3+1)=2.5. Write a query to calculate student 101's GPA.

Exercise 2.3

Amazon.com

Ord#	OrdDate	CustNum	Amount
1001	8/1/2015	101	$400
1002	8/15/2015	102	$1,500
1003	12/3/2015	103	$800
1004	2/6/2016	101	$300
1005	3/7/2016	103	$200
1006	8/24/2016	104	$1,100
1007	1/5/2017	101	$1,400
1008	5/3/2017	101	$50
1009	8/8/2017	103	$89
....	...	...	...

1. In total, how much money has customer 101 spent on amazon.com?
2. On average, how much money does Andy spend on amazon.com **every year**?
   Note: you can use **to_char(OrdDate,'yyyy')** to get the year code
   Note: how many ways can you interpret this question? ^__*
3. On average, how much money does customer 101 spend on amazon.com every month?
4. In 2017, on average, how much money does every customer spend on amazon.com?

Group by and Having

```
select f1, group_function(f2), group_function(f3),...
from table, ...
where...
group by f1
having [group_function related conditions];
```

Students

SNum	SName	Standing	Major	GPA	Gender	Zip	Status
101	Andy	3	IS	2.8	M	91101	Active
102	Betty	4	ACC	3.2	F	91102	Active
103	Cindy	2	IS	1.5	F	91101	Probation
104	David	2	FIN	3.3	M	91104	Active
105	Ellen	1	IS	1.8	M	91102	Probation
106	Frank	3	MKT	3.1	F	91103	Acvitve

Exercise 2.4

1. In one query, display how many students are in each major.
2. In one query, display how many students are male vs. female.
3. In one query, display the average GPA of male vs. female students.
4. In one query, display the average GPA of students in each major.
5. Display majors that have more than 100 students.
6. Display majors where the average GPA is higher than 3.5.
7. Display majors where there are more than 35 students on Probation.
8. ISSA wants to recruit freshman students by passing out flyers in zip codes where there are a lot of freshman students. Display zip codes where there are more than 200 Freshman students (Freshman is standing=1).

Exercise 2.5

Amazon.com

Ord#	OrdDate	CustNum	Amount
1001	8/1/2015	101	$400
1002	8/15/2015	102	$1,500
1003	12/3/2015	103	$800
1004	2/6/2016	101	$300
1005	3/7/2016	103	$200
1006	8/24/2016	104	$1,100
1007	1/5/2017	101	$1,400
1008	5/3/2017	101	$50
1009	8/8/2017	103	$89
....	...	...	...

Customers

CustNum	CustName	Gender	Prime
101	Andy	M	Y
102	Betty	F	Y
103	Cindy	F	N
104	David	M	N
105	Ellen	F	Y
106	Frank	M	Y
107	George	M	Y

1. In one query, display how many male vs. female customers does Amazon have.
2. In one query, display how many prime vs. non-prime members does Amazon have.
3. In one query, display the total spending of male vs. female customers.
4. In one query, display the total spending of prime vs. non-prime members.
5. What is the average spending of male vs. female customers per person?
6. What is the average spending of prime vs. non-prime members per person?

SQL Topic 3. Built in Functions

The `NVL` Function

NVL

```
nvl(price, 0)
nvl(quantity, 0)

nvl(Status, 'N/A')
nvl(MajorCode, 'Undeclared')

select qty * price from transactions where....
select nvl(qty, 0) * nvl(price, 0) from transaction where...

select count(*) from transactions where qty >= 100;
select count(*) from transactions where qty < 100;
select count(*) from transactions;

select count(*) from transactions where nvl(qty,0) >= 100;
```

Exercise 3.1

Students

SNum	SName	Standing	Major	GPA	MajorGPA
101	Andy	4	IS	2.8	3.2
102	Betty	2		3.2	
103	Cindy	3	IS	2.5	3.5
104	David	2	FIN	3.3	3.0
105	Ellen	1		2.8	
106	Frank	3	MKT	3.1	2.9

Note that there are NULL values in the table.

1. Write a SQL statement to display students who are not IS major. (Note: some students have NULL major, and they are "not" IS major).
2. Write a SQL statement to display students whose MajorGPA is lower than 3. (Note: If MajorGPA is NULL, consider it "lower" than 3.)
3. Write a SQL statement to display SNum, SName, and Major. If major is NULL, display 'Undeclared'.
4. Write a SQL statement to display students and their major GPA. If major GPA is NULL, display 0.

The `greatest` and `least` function

```
greatest(3,20,18,4) returns  _______

least(3,20,18,4) returns _______
```

Note: What is the difference between **greatest** and **max**, or **least** and **min**?

Exercise 3.2

Students

SNUM	SNAME	Standing	Major	GPA	MajorGPA
101	Andy	4	IS	2.8	3.2
102	Betty	2		3.2	
103	Cindy	3	IS	2.5	3.5
104	David	2	FIN	3.3	3.0
105	Ellen	1		2.8	
106	Frank	3	MKT	3.1	2.9

1. For each student, display SNum, SName, and the higher of his GPA and MajorGPA.
2. For each student, display SNum, SName, and the lower of his GPA and MajorGPA.

Note: What is the effect of NULL value?

Note: How about "Of all students, display the highest GPA?"

Exercise 3.3

IS380

SNUM	SNAME	Hw1	Hw2	Hw3	Hw4
101	Andy	10	8	3	10
102	Betty	10	7	7	10
103	Cindy	9	9	7	10
104	David	10	9	8	0
105	Ellen	10	4	5	10
106	Frank	10	5	8	10

1. This is the grading sheet of IS 380. There are 4 homework and a student can drop the lowest homework. Write a query to calculate/display students and their total homework points.
2. Write a query to display all students and their highest homework grade.
3. Write a query to display all students and their lowest homework grade.

Exercise 3.4

Customers

CNUM	CNAME	AccountBalance	TotalMileage
101	Andy	85.25	2,152
102	Betty	170.00	808
103	Cindy	-55.13	31
104	David	1,308.02	5,510
105	Ellen	99.77	11,154
106	Frank	-220.48	380

1. Each customer has an AccountBalance. Usually we sent them a BillDue for the account balance. However, if the account balance is less than 0 (meaning, they have a credit with us), then we send them a BillDue of $0 (in other words, BillDue cannot be lower than $0). Write one SQL statement to display the BillDue for each customer.

2. Each customer has an total mileage with us. However, they can only use up to 1,000 miles in each billing cycle. In other words, if they have less than 1,000 miles, they can use whatever they want. If you have over 1,000 miles, they can at most use 1,000 miles. Write a SQL statement to display customer number, name, and their "Usable Mileage" during this billing cycle (ie, usable mileage cannot go over 1,000).

The `decode` Function

decode

```
decode(grade,'A',4,'B',3,'C',2,'D',1,'F',0,0)
```

If grade = 'C', the above statement will return _____
If grade is NULL, the above statement will return _____

```
decode(grade,'A',4,'B',3,'C',2,'D',1,'F',0,'Not a Valid Grade')
```

Does this work??

```
decode(grade, null, 'No Grade', 'Has Grade')
```

If grade is NULL, it returns _____
If grade='A', it returns _____
If grade='B', if returns _____

Exercise 3.5

Students

SNum	SName	Standing	Major	GPA	MajorGPA
101	Andy	4	IS	2.8	3.2
102	Betty	2		3.2	
103	Cindy	3	IS	2.5	3.5
104	David	2	FIN	3.3	3.0
105	Ellen	1		2.8	
106	Frank	3	MKT	3.1	2.9

Write a SQL to display student and their standing; if standing is 4, then display "Senior"; if it is 3 then display "Junior"; 2 then "Sophomore", and 1 then "Freshman". If it is not 1-4, then print "Others".

Exercise 3.6

Semester is coded as **Sp** for spring, **Fa** for Fall, **Su1** for Summer I, **Su3** for Summer III. However, when you sort by year, semester, you will get **Fa** first, then **Sp**, then **Su1** and **Su2**. The users would like to see it displayed in the order of **Sp**, **Su1**, **Su3**, **Fa**, for a given year. Write a SQL command to display Schedule of Classes records where the semester is sorted by the order specified by the users.

Exercise 3.7

Here is the Trans table:

Trans

TransNum	TransDate	AcctNum	TransType	Amount
101	1-1-2016	123-0097	Credit	10
102	1-1-2016	X089-056	Credit	10
103	1-2-2016	123-0097	Debit	8
104	1-2-2016	123-0097	Debit	10
105	1-2-2016	F3377-D	Credit	10
106	1-2-2016	X089-056	Debit	9
107	1-2-2016	X089-056	Debit	9
108	1-3-2016	123-0097	Credit	9

1. "Credit" is to increment (increase) the account value and "debit" is to decrement (decrease) the account value. Write one SQL statement to display the following from the original table. (Note. you cannot create temp tables ☺)

Trans

TransNum	TransDate	AcctNum	Credit	Debit
101	1-1-2016	123-0097	10	0
102	1-1-2016	X089-056	10	0
103	1-2-2016	123-0097	0	8
104	1-2-2016	123-0097	0	10
105	1-2-2016	F3377-D	10	0
106	1-2-2016	X089-056	0	9
107	1-2-2016	X089-056	0	9
108	1-3-2016	123-0097	9	0

2. Write one SQL statement to display the following

Trans

TransNum	TransDate	AcctNum	Amount
101	1-1-2016	123-0097	10
102	1-1-2016	X089-056	10
103	1-2-2016	123-0097	-8
104	1-2-2016	123-0097	-10
105	1-2-2016	F3377-D	10
106	1-2-2016	X089-056	-9
107	1-2-2016	X089-056	-9
108	1-3-2016	123-0097	9

Exercise 3.8

Students

SNUM	SNAME	Standing	Major	GPA	Gender	Zip	Status
101	Andy	3	IS	2.8	M	91101	Active
102	Betty	4	ACC	3.2	F	91102	Active
103	Cindy	2	IS	1.5	F	91101	Probation
104	David	2	FIN	3.3	M	91104	Active
105	Ellen	1	IS	1.8	M	91102	Probation
106	Frank	3	MKT	3.1	F	91103	Acvitve

1. Write a query to display how many students are male vs. female, like this:

```
MALE      FEMALE
----      ------
1381         782
```

2. Write a query to display how many students are in each major, like this:

```
 ACC        FIN         IS        MKT
----        ---        ---        ---
 654        631        316        562
```

3. Write a query to display how many male vs. female students are in each major, like this:

```
            ACC      FIN     IS       MKT
           ----     ---     ---      ---
 Male      530      408     131      312
 Female    124      223     185      250
```

4. Write a query to display how many male vs. female students are in each major, like this:

```
            Male       Female
            ----       ------
 ACC         530          124
 FIN         408          223
 IS          131          185
 MKT         312          250
```

Homework 1. Due Date _____

Complete _____.

Write your SQL statements in a text file. Set ECHO on. Run your programs until there is no error. Turn in a print out of your spool file so I can see your program AND the result of the execution.

Character Functions

```
'A'||'B' or concat('A', 'B')
ex:     select 'Student Name is: '||sname from STUDENT;
```

initcap('HeLLo worLD') returns _____

lower('HeLLo worLD ') returns _____

upper('HeLLo worLD ') returns _____

lpad('Andy', 10,'X') returns _____

rpad('Andy', 10,'X') returns _____

lpad(ProdDesc, 30,' ') returns _____

Ex. The **PRODUCTS** table has the following columns:

```
PRODUCTS(PNum varchar2(8),
         PDesc varchar2(30),
         Price Number(8,2))
```

What is the difference between the next two SQL statements?

```
Select pnum||PDesc||Price from PRODUCTS;
Select rpad(pnum,8)||rpad(PDesc,30)||rpad(Price,8) from PRODUCTS;
```

How about

```
Select pnum, PDesc, Price from PRODUCTS;
```

ltrim('0011-12110') returns _____

ltrim('0011-12110', '0') returns _____

rtrim('0011-12110') returns _____

rtrim('0011-12110', '0') returns _____

substr('0011-12110',5,3) returns _____

substr('0011-12110',5) returns _____

substr('0011-12110',-4) returns _____

instr('0011-12110','-') returns _____

length('0011-12110') returns _____

Numeric Functions

ceil(3.45) returns _____

floor(3.45) returns _____

round(123.45) returns _____

round(123.65) returns _____

round(123.45,1) returns _____

round (123.45,-1) returns _____

trunc(123.45) returns _____

trunc(123.65) returns _____

mod(9,4) returns _____

Date Functions

sysdate returns _____

sysdate - 1 returns _____
sysdate + 1 returns _____
sysdate - My_Date_of_Birth returns _____
round(sysdate) returns _____
trunc(sysdate) returns _____
to_char(sysdate, 'mm/dd/yyyy') returns _____
to_char(sysdate, 'dd/mm/yyyy') returns _____
to_char(sysdate, 'J') returns _____
to_date('10/01/2017','mm/dd/yyyy') returns _____
to_date('10/01/2017','dd/mm/yyyy') returns _____

```
SQL> select sysdate from dual;

SYSDATE
---------
05-FEB-17

SQL> select to_char(sysdate,'mm/dd/yyyy') from dual;

TO_CHAR(SY
----------
02/05/2017

SQL> select to_char(sysdate,'Mon dd, yyyy') today from dual;

TODAY
------------
Feb 05, 2017

SQL> select to_char(sysdate,'J') from dual;

TO_CHAR
-------
2457790
```

Exercise 4.8

The **STUDENTS** table has three columns: **SSN**, **LASTNAME**, and **USERNAME**.

1. Write one SQL statement to display the last character of every student's last name.

2. A student's username starts with the letter 'Z', followed by the first 2 letters of his last name, and the last 4 digits of his SSN. Currently the USERNAME column is null. Write an SQL statement to populate the USERNAME column.

Exercise 4.9

Accounts

AcctNum	Prefix	Suffix
123-0097		
X089-056		
123-0097		
123-0098		
F3377-D		
X089-057		
X089-058		
3-009712		

Each account has an AcctNum. There is always a '-' in the account number, but the position differs by account.

1. Write one SQL statement to display the position of '-' in AcctNum.

2. What does the following code return?

```
select substr(AcctNum, 1, instr(AcctNum,'-')-1)
       from Accounts;
```

3. Write one SQL statement to display the portion of text after '-' .

4. `Prefix` is the portion of the account number before the '-' and `suffix` is the portion after the '-'. Write one SQL command to populate the prefix and suffix columns.

5. Write one SQL command to display account_number without '-'. (In other words, if the account_number is 123-456, then display 123456; if it is 72-8597, then display 728597).

Exercise 4.10

StudentEmails

SNum	SName	LastName	FirstName	Emails	Username	EmailHost
101	Smith, John			jsmith@csulb.edu		
102	Liu, Ali			aliu@chapman.edu		
103	Jones, Bob			bob1@msn.com		
104	Walker, Sky			skywalker@yahoo.com		

Each student has a name. There is a ‘,’ separating the last name and first name. (Note that there is a space after comma).
(1) Write a SQL statement to populate the last name of each student.
(2) Write a SQL statement to populate the first name of each student.

Each student has an email address.
(3) Write a SQL statement to display the portion of text before the ‘@’ sign (ie, the username).
(4) Write a SQL statement to populate/update the portion of text before the ‘@’ sign to the Username column.
(5) Write a SQL statement to display the portion of text after ‘@’ and before ‘.’.
(6) Write a SQL statement to populate/update the portion of text between ‘@’ and ‘.’ to the EmailHost column.

Exercise 4.11

Use **select ... from dual;** for the following three questions.

1. Write a SQL command to display the remainder of 7 divided by 2 (ie, 1).

2. Write a SQL command to display the integer portion of 7 divided by 2 (ie, 3).

3. Write a SQL command to display the rounded result of 7 divided by 2 (ie, 4).

SQL Topic 4. Union, Intersect, Minus

Union, Intersect, Minus

```
select column, column, column, ..... from table
UNION
select column, column, column, .... from table;

select column, column, column, ..... from table
INTERSECT
select column, column, column, .... from table;

select column, column, column, ..... from table
MINUS
select column, column, column, .... from table;
```

```
SQL> -- Find SSN of students who have never enrolled in any class
SQL> select snum from students
  2  minus
  3  select snum from enrollments;

SNU
---
104
105

SQL>
SQL> -- Find NAME of students who have never enrolled in any class
SQL> select sname from students
  2  where snum in
  3         (select snum from students
  4          minus
  5          select snum from enrollments);

SNAME
----------
David
Ellen

SQL>
SQL> -- How many students have never enrolled in any class
SQL> select count(*) from
  2         (select snum from students
  3          minus
  4          select snum from enrollments);

  COUNT(*)
----------
         2
```

Exercise 4.1

1. Find students who have not taken MGT 425.
2. Find ‘IS’ students who have not taken IS 380.
3. Display students who have taken both IS 380 and IS 385.
4. Display students who have taken IS 380 but never took IS 300.
5. Find students who did not take any course in Spring 2013.

SQL Topic 5. Self Join, outer join

Self join

There are times where you need to use the same table multiple times in a query, where you need to use "self join".

```
SQL> -- Display courses, their title, their prerequisites, and the prerequisite course titles

SQL> select p.dept,p.cnum, c1.ctitle,
  2          p.pdept,p.pcnum, c2.ctitle
  3  from courses c1, courses c2, prereq p
  4  where p.dept=c1.dept and p.cnum=c1.cnum
  5  and p.pdept=c2.dept and p.pcnum=c2.cnum;

DEP CNU CTITLE                         PDE PCN CTITLE
--- --- ------------------------------ --- --- ------------------------------
IS  355 Networks                       IS  300 Intro to MIS
IS  380 Database                       IS  310 Statistics
IS  380 Database                       IS  301 Business Communicatons
IS  380 Database                       IS  300 Intro to MIS
IS  385 Systems                        IS  310 Statistics
IS  480 Adv Database                   IS  380 Database

6 rows selected.
```

Exercise 5.1

Each employee has one manager, who is also an employee

Employees

Enum	Ename	MNum
101	Andy	106
102	Betty	106
103	Cindy	106
104	David	105
105	Ellen	101
106	Frank	

1. Display employee number, employee name, and his/her manager's number and name.

Exercise 5.2

Each human has one spouse, who is also a human.

Humans

HNum	HName	SNum
101	Adam	102
102	Eve	101
103	Cindy	104
104	David	105
105	Ellen	
106	Frank	107
107	George	106

1. Display human number, human name, and his/her spouses' number and name.
2. Display humans whose spouse is married to somebody else. For instance, if 101's spouse is 102, then 102's spouse should be 101. However, in this table, 103's spouse is 104, but 104's spouse is 105. Write a query to find all records of such mis-match.

Outer join

When you join tables, if join column has a NULL value, this record will disappear after the join. You may use OUTER JOIN so those records would not disappear.

```
SQL> -- Display snum, sname, major code, and major description.
SQL> -- For student with null major, still display snum and sname.
SQL>
SQL> -- Without outer join, students with null major would disappear
SQL> select s.snum, sname, s.major, mdesc
  2  from students s, majors m
  3  where s.major=m.major
  4  order by snum;

SNU SNAME      MAJ MDESC
--- ---------- --- ------------------------------
101 Andy       IS  Information Systems
103 Cindy      IS  Information Systems
104 David      MKT Marketing

SQL>
SQL> -- With outer join... (+) is placed on the "blank" side
SQL> -- Here, (+) is on the major side. So, display ALL students with possibly blank major
SQL> select s.snum, sname, s.major, mdesc
  2  from students s, majors m
  3  where s.major = m.major (+)
  4  order by snum;


SNU SNAME      MAJ MDESC
--- ---------- --- ------------------------------
101 Andy       IS  Information Systems
102 Betty
103 Cindy      IS  Information Systems
104 David      MKT Marketing
105 Ellen

SQL>
SQL> -- Here, (+) is on the Student side. So, display ALL majors with possibly blank student.
SQL> select s.snum, sname, s.major, m.major, mdesc
  2  from students s, majors m
  3  where s.major(+) = m.major
  4  order by snum;

SNU SNAME      MAJ MAJ MDESC
--- ---------- --- --- ------------------------------
101 Andy       IS  IS  Information Systems
103 Cindy      IS  IS  Information Systems
104 David      MKT MKT Marketing
                   FIN Finance
                   ACC Accounting
```

Exercise 5.3

Based on the table of **Exercise 5.1**, display employee number, employee name, and his/her manager's number and name. For employees who do not have a manager, display blank under manager's number and name.

SQL Topic 6. Subquery

Use subquery in the where... condition

```
select....
update...
delete...
        where snum = (select snum from ....);
        where snum in (select snum from ...);
```

Exercise 6.1

Customers

CNum	CName	Status	Member
101	Andy	Active	Regular
102	Betty	Active	Regular
103	Cindy	Inactive	Premium
104	David	Active	Gold
105	Ellen	Inactive	Permium
...	...	...	

Products

PNum	PName	Statis
P1	Pencil	Active
P2	Pen	Active
P3	Paper	Discontinue
P4	Box	Active
...	...	...

Orders

O#	Status	CNum	PNum	Amount
1001	C	101	P1	$1,500
1002	X	101	P5	$2,000
1003	O	103	P1	$500
1004	O	105	P1	$700
1005	O	101	P3	$1,000
1006	O	102	P1	$200

1. Display the order (O#, CNum, PNum) with the largest order amount.
2. Display the customer (CNum, CName) who placed the order of the largest amount.
3. Among all orders that are still 'O'pen, display the order (O#, CNum, PNum) with the largest order amount.
4. Display orders where the Amount is higher than average amount.
5. Display orders where the Amount is lower than average amount.

6. Write an SQL statement to delete orders if the product is "Discontinue"d.
7. Write an SQL statement to cancel orders (Status 'X') for customers that are Inactive.
8. Write an SQL statement to update the member to "Gold" if he/she has spent over $10,000.
9. Write an SQL statement to discontinue a product(s) if we have received more than 10 cancellations (Status 'X').

Use subquery to create a new table

```
create table newtable as
select * from oldtable;

create table newtable as
select column1, column2, ...
from oldtable
where.....;

SQL> create table students2 as
select * from students;

SQL> create table students2 as
select ssn, sname, gpa from students;

SQL> create table students2 as
        Select ssn, sname, gpa
        From students
        Where major='IS';
```

Use subquery to insert records

```
insert into to_table
select * from from_table;

insert into to_table (column1, column2, ...)
select column1, column2, ... from from_table;

SQL> Insert into StudentBackup
```

```
select * from students;

SQL> Insert into StudentBackup (ssn, sname, gpa)
select ssn, sname, gpa from students;

SQL> insert into studentsBackup (ssn, sname, gpa)
       Select ssn, sname, gpa
       From students
       Where status='Graduated';
```

Use subquery as a temp table for join

```
Select temp1.snum, .....
From (select snum, ... from students where......) temp1,
      enrollments
Where temp1.snum=enrollments.snum
   and ..... ;
```

```
SQL> -- Use Subquery as a temp table
SQL> -- Display courses, their capacity, and current enrollment
SQL> select e.callnum, s.capacity, e.scount
  2  from
  3  (select callnum, count(snum) scount
  4         from enrollments
  5         group by callnum) temp,
  6  schclasses s
  7  where temp.callnum=s.callnum;

   CALLNUM   CAPACITY     SCOUNT
---------- ---------- ----------
     10110         45          2
     10120         35          1
     10125        118          1
     10130         33          1

SQL> -- Display courses that current enrollment is less than capacity
SQL> select e.callnum, s.capacity, e.scount
  2  from
  3  (select callnum, count(snum) scount
  4         from enrollments
  5         group by callnum) temp,
  6  schclasses s
  7  where temp.callnum=s.callnum
  8    and temp.scount<s.capacity;

   CALLNUM   CAPACITY     SCOUNT
---------- ---------- ----------
     10110         45          2
     10120         35          1
     10125        118          1
     10130         33          1
```

Exercise 6.2

1. In the **SchClasses** table, the **NumStu** shows the number of students currently enrolled in the class. Write an SQL statement to display courses where the **NumStu** figure is not consistent with data from the **Enrollments** table.

2. In the **Students** table, the GPA shows the student's average grade (assuming all courses are 3-credit hour). Write an SQL statement to display students where the GPA is not consistent with data from the **Enrollment** table.

Exercise 6.3

Items

I#	IName	UnitPrice
101	Cheese Burger	$3.99
102	Double Cheese Buger	$4.99
103	French Fries	$1.19
104	Medium Coke	$1.39
105	Large Coke	$1.89
106	Combo Meal 1	$6.99
107	Combo Meal 2	$8.99

ItemDetails

I#	IncludeI#
106	101
106	103
106	104
107	102
107	103
107	105

1. Write an SQL statement to display items where buying the combo is cheaper than buying individual items.
2. Write an SQL statement to display items where the combo and the sum of individual items are the same price.
3. Write an SQL statement to find items where the combo price is more expensive than buying individual items.

Exercise 6.4

According to tables in **create480tables.sql**, write the following queries:

1. Write an SQL command to display, for the 10110 class, how many students received A, how many students received B, how many students received C, D, and F respectively.

1a. (continue from question 1) In addition, if say, nobody receives D, then D will not be displayed. Write the query in such a way that it will display D with a student count of 0, like this:

```
GRADE        HEAD_COUNT
----- -----------------
A                     2
B                     1
C                     0
D                     0
F                     0
```

Note: You need to create a table GRADES that include all letter grades.
Note: this question is VERY interesting (and challenging) ☺ I hope somebody asks you this in job interviews ☺

2. For students in the 10110 class, display the student count by standing, ie, how many students have the standing of 4, 3, 2, or 1.
3. Display number of enrollments by Callnum (ie, how many students are enrolled in each Callnum).

3a. Similar to 1a, if nobody is enrolled in this particular CallNum, it will not be displayed at all. Write your query in such a way that all CallNum are displayed; if it does not have any enrollments, it shows 0.

4. Display number of enrollment by year/semester (ie, how many students are there in each Year/Semester).
5. Display number of enrollment by year, semester, course, like

```
SEMESTER               COURSE                Number of Enrollment
----------------      --------------         --------------------
Spring 2013            IS 380                105
Spring 2013            IS 480                33
Spring 2013            IS 485                68
....
Fall 2013              IS 380                117
Fall 2013              IS 480                28
.....   .....   .....
```

6. Display how many courses student 101 has enrolled in Fall 2013.
7. Display how many times Andy has enrolled in IS 300.
8. Display the highest grade that Andy has ever received on IS 300.
9. Display courses that Andy has taken for 2 or more times.
10. Display courses where there is any NULL grade.
11. Displays students who are in both IS 300 and IS 301 in Fall 2013.
12. Display courses that have 2 or more students enrolled.
13. Display courses where no student has enrolled.
14. Display student who are currently enrolled in multiple sections of the same course

15. Find classes (callnum) where all grades are assigned; in other words, every student in that class has received a letter grade.
16. Find classes (callnum) where no grades is assigned; in other words, in this particular class, all grades are null.
17. Find classes (callnum) where partial grades are assigned; in other words, some students have received letter grade while others have null grades.
18. In one SQL command, find that, for class 10110, how many students received a NULL grade and how many students received a letter grade.

Homework 2. Advanced SQL

Complete _____. Write your SQL statements in a text file. Set echo on. Run your programs until there is no error. Turn in a print out of your spool file so I can see your program AND the result of the execution.

Part II

PL/Sql

PL/Sql

- SQL: Structured Query Language, or Fourth Generation Language (4GL)
  - "script" language
  - Data Definition Language: creating table structures
  - Data Manipulation Language (DML): `insert`, `update`, `delete` data
  - Query: `select...`
  - Transaction Control: `commit`, `rollback`
- Third Generation Language (3GL), or procedural program, such as Pascal, COBOL, Ada, Basic
  - Step by step procedures
  - Control structure, such as `IF...THEN` and `LOOP...`
- PL/SQL: An Oracle product, stands for Procedural Language/SQL; a 3GL. Can work seamlessly with SQL.
  Note: Use pl/sql as keyword to search dice.com!

- The simplest pl/sql program is this:

```
begin
    null;
end;
/
```

- However, most PL/SQL contains three blocks: Declaration, Executable, and Exception, like this:

```
declare
    declaration section
begin
    -- executable section
    Blah;
    Blah;
    Blah;
exception
    exception section
end;
/
```

Note: Only the executable section is required.

- PL/SQL commands are NOT case sensitive. Note: For character strings, an 'A' is different from an 'a'.
- Flow of Control
  - A block is run as a whole

Write your first PL/SQL program!

Open the text editor, and write the following program and save it as `a:HelloWorld.sql`.

```
set serveroutput on
begin
    dbms_output.put_line ('Hello World!');
end;
/
```

Run it in SQL*Plus, and you will see the following feedback:

```
SQL> start a:helloworld
Hello World!

PL/SQL procedure successfully completed.
```

- **What is `set serveroutput on`?**

put vs. put_line

```
set serveroutput on
begin
    dbms_output.put ('Hello');
    dbms_output.put ('World');
    dbms_output.put_line ('Hello World');
    dbms_output.put_line ('Hello!' || vSname || 'How are you?');
end;
/
```

Topic 1. Write a Stored Procedure

Anonymous Block vs. Stored Procedure

Anonymous Block: a program without a name; stored on floppy disk or hard disk. Not permanent; cannot be accessed by other users; cannot be accessed by other programs.

Stored Procedure/Function: programs with a name; stored in Oracle server; it is an Oracle object, like a table.

Creating a Stored Procedure

Creating a stored procedure requires two steps:

(1) Write a text file containing the procedure creation code.
(2) Run this program (use **START**) so the procedure is actually created and stored in server.
(3) To "call" (ie, "use") the procedure you have created, use the **EXECUTE** command.

Example:
(1) create the following notepad file and save it as **proc1.sql** on A drive.

```
create or replace procedure SayHi as
begin
    dbms_output.put_line ('Hi!');
end;
  /
```

(2) at SQL>, run this program

```
SQL> start a:proc1.sql
```

Now this procedure is created.

(3) To call this procedure, use the command **`execute`**

```
SQL> execute SayHi
Hi
```

Make sure your `serveroutput` is set to on.

Note: If you need to make changes to a procedure, since your procedure is stored in the server, making changes on your floppy disk file is not enough! You need to run the procedure again to update its code (that's why we put `create or replace...` in the beginning of the program).

```
create or replace procedure SayHi as
begin
    dbms_output.put_line ('Hi! ^__^ ');
end;
/
```

Note: How do you know that your procedure was created successfully? Oracle provides are few data dictionary tables for you to view your objects.

The **USER_OBJECTS** table: to view your objects

```
SQL> select object_name, object_type
  2  from user_objects;

OBJECT_NAME                                    OBJECT_TYPE
---------------------------------------------- ---------------
ADDME                                          PROCEDURE
ADDSTUDENT                                     FUNCTION
ADVISORS                                       TABLE
BABY                                           PROCEDURE
…


SQL> select object_name, object_type
  2  from user_objects
  3  where object_type='TABLE'
  4  order by object_name;

OBJECT_NAME                                    OBJECT_TYPE
---------------------------------------------- ---------------
ADVISORS                                       TABLE
CALLERS_LOG                                    TABLE
COURSES                                        TABLE
CUSTOMER                                       TABLE
….

SQL> select object_name, object_type
  2  from user_objects
  3  where object_type=upper('procedure')
  4  order by object_name;

OBJECT_NAME                                    OBJECT_TYPE
---------------------------------------------- ---------------
ADDME                                          PROCEDURE
DOUBLEENROLLMENT                               PROCEDURE
DROPSTUDENT                                    PROCEDURE
ENROLLSTUDENT                                  PROCEDURE
GETGRADE                                       PROCEDURE
….
```

The **USER_SOURCE** table: to view your program code

```
SQL> select line, text from user_source where name='ADDME';

     LINE TEXT
--------- ------------------------------------------------------------
        1 procedure AddMe (
        2       pSNum student.snum%type,
        3       pSName student.sname%type)
        4 as
        5 begin
        6       insert into student (snum, sname) values (pSnum, pSname);
        7 end;

7 rows selected.
```

Procedure with Parameters

(1) Parameter is one way to send input to the procedure. Check out the following program

```
create or replace procedure SayWhat (
    p_what varchar2  ) as

begin
    dbms_output.put_line('I said ' || p_what || '! ' );
end;
/
```

```
SQL> exec SayWhat('hola');
```

(2) You can have multiple parameters in a procedure

```
create or replace procedure greeting(
    p_who varchar2,
    p_what varchar2  ) is
    begin
        dbms_output.put_line ( p_who||', '||'I said '||p_what||'!');
    end;
```

```
SQL> exec SayWhat('John','hi');
SQL> exec SayWhat('Mary','bye');
```

The input value is sent in according to the position of the parameter. That is, the first value 'John' goes into the first parameter **p_who**, the second value 'How are you' goes into the second parameter **p_what**, etc.

Note 1: Procedure parameters are not allowed to have any precision; that is, it is ok to decalre **number**, but **number(6)** would bring you an error message.

Note 2: The syntax and sections of a procedures is

```
CREATE OR REPLACE PROCEDURE Procedure_Name (
    parameter1 type,
    parameter2 type, ....) AS

    variable1 type;
    variable2 type;
    ....
BEGIN
    statement1;
    statement2;
    ...
END;
/
```

Debugging

Any bug in a procedure code will result in the following seemingly harmless little warning:

```
Warning: Procedure created with compilation errors.
```

This means that there are bugs in your program! To view the error messages, use

```
SQL> show errors
Errors for PROCEDURE ARCHIVEGRADUATION:

LINE/COL ERROR
-------- -----------------------------------------------------------------
2/23     PLS-00103: Encountered the symbol "(" when expecting one of the
         following:
         := . ) , @ % default
         The symbol ":=" was substituted for "(" to continue.
```

Note: The **show errors** or **show err** command will display errors associated by the most recently run procedure. To view errors of other function/procedures, you can query the data dictionary table **user_errors**.

```
SQL> describe user_errors;
 Name                            Null?    Type
 ------------------------------- -------- ----
 NAME                            NOT NULL VARCHAR2(30)
 TYPE                                     VARCHAR2(12)
 SEQUENCE                        NOT NULL NUMBER
 LINE                            NOT NULL NUMBER
 POSITION                        NOT NULL NUMBER
```

```
 TEXT                            NOT NULL VARCHAR2(2000)

SQL> SELECT LINE, POSITION, TEXT FROM USER_ERRORS
  2   WHERE NAME='ADDNEWSTUDENT';

no rows selected
```

Exercise 1.1

1. Create a procedure **SayHello** which will print Hello when you execute it. Make sure you can create the procedure, run it, and call it from SQL>.
2. Edit your procedure **SayHello** so that when you call it, it prints **'Hello Again!'** .
3. Use DBA commands to confirm that the procedure is in your account.
4. Use DBA commands to display the code of the procedure.

Exercise 1.2

1. Create the **SayWhat** procedure with one input parameter. Test the procedure by giving it different parameter values, like ‘Hi’, ‘Hello’, ‘Bye’, etc.

2. Create the **Greeting** procedure with two input parameters. Test the procedure by giving it different parameter values, like ‘John’’Hi’, ‘Mary’’Bye’,’Sophie’’Hello’, etc.

3. Query the USER_OBJECTS table to see that your procedures are actually there.

4. Query the USER_SOURCE table to see the program code of the procedures.

Topic 2. Simple Program

1. The Declaration Section

· You "declare" variables to be used in the program in this section.
· Variable names are NOT case sensitive. The maximum length for a variable name is 30 characters. You can use letters, dollar signs, underscores, and number signs in a variable name. A variable name must start with a letter.
· A variable must be "declared" at first. If it is an anonymous block, use the **declare** key word to declare variables. If it is in a procedure or function, declare variables after the "as" keyword and there is no need for the "declare" keyword.

· Variable Name: Name something meaningful. Starting with a v (my habit, at least)

v_StudentName
v_CustomerAmount
v_LoopIndex1

· The syntax of declaring is

variable_name type;
variable_name type :=value;
vabiable_name type default value;
variable_name constant type :=value;

Example:

```
DECLARE
    v_EmpLastname char(30);
    v_EmpSalary number(7,2);
    v_BaseSalary number(7,2) := 12000;
    v_BaseSalary number(7,2) default 12000;
    v_Boss char(30) := 'Sophie';
    v_Pi constant number := 3.14159;
```

Variable Type

number
number(n)
number(n,d)
n: precision (total length)
d: scale (number of decimal digits)
eg., 999.99 is number(5,2)
varchar2(n)
- **`varchar2`** without any length is not allowed
char
char(n)
- **`char`** without any length means char(1)
Date

Boolean

```
declare
  vCreditCheck boolean := true;
begin
  IF vCreditCheck = false
          THEN ....
```

2. The Executable Section

Assignment vs. Expressions

:= To assign the value of the right-hand-side to (the variable) on the left-hand-side.
For example, **x:=5;**

= equality (expression); to compare the two values
For example, **if x=5 then...**

```
/* Program ex1 */
declare
    x1 number;
    x2 char(20);
    x3 number;
begin
    x1 := 5;
    x2 := 'Hello World!';
    x3 := x1 + 3;
    x1 := x1 + 1;

    dbms_output.put_line ('x1 is: ' || x1);
    dbms_output.put_line ('x2 is: ' || x2);
    dbms_output.put_line ('x3 is: ' || x3);
end;
/
```

```
SQL> start a:ex1.sql
x1 is: 6
x2 is: Hello World!
x3 is: 8

PL/SQL procedure successfully completed.
```

Exercise 2.1

Write a pl/sql procedure with 2 input parameters: **x1** and **x2**. Declare 4 local variables: **y1**, **y2**, **y3** and **y4**. Assign **y1** to be **x1** plus **x2**, **y2** to be **x1** minus **x2**, **y3** to be **x1** multiple **x2**, and **y4** to be **x1** divided by **x2**.

Also write a code to print **y1**, **y2**, **y3**, and **y4**.

Execute your program with different input values for **x1** and **x2** and examine your result.

Exercise 2.2

Write a pl/sql procedure with 2 input parameters: **x1** and **x2**. Declare 3 local variables: **y1**, **y2**, and **y3**. Assign **y1** to be the remainder of **x1** divided by **x2**, assign **y2** to be the integer portion of **x1** divided by **x2**, assign **y3** to be the rounded result of **x1** divided by **x2**.

For instance, if x1 is 7 and x2 is 2, then y1 would be 1, y2 would be 3, y3 would be 4.

Print the result of **y1**, **y2**, and **y3**.

Exercise 2.3

Ralphs is having a "buy 3 get 1 free", which means for every 3 units, you get the 4th unit for free. In other words, if you buy 4 units, you only pay for 3. Write a procedure with one input parameter **p_buy** which is the number of units you buy. Declare a variable **v_pay**, which is the number of units you have to pay for. Write pl/sql program to calculate **v_pay** according to the input value of **p_buy**. Print **v_pay**.

Exercise 2.4

Ralphs is having a "buy 4 get 1 free", which means for every 4 units, you get the 5th unit for free. Write a procedure with one input parameter **p_buy** which is the number of units you buy. Declare a variable **v_pay**, which is the number of units you have to pay. Write pl/sql program to calculate **v_pay** according to the input value of **p_buy**. Print v_pay.

Exercise 2.5

Ralphs is having a "buy x get 1 free". Write a procedure with two input parameter **p_buy**, which is the number of units you buy, and **p_x**, which is the units you buy. Declare a variable **v_pay**, which is the number of units you have to pay. Write pl/sql program to calculate v_pay according to the input value of p_buy and p_x. Print v_pay.

Exercise 2.6

Ralphs is having a "buy x get y free". Write a procedure with three input parameter **p_buy**, which is the number of units you buy, and **p_x**, **p_y**, for “buy x get y free”.. Declare a variable **v_pay**, which is the number of units you have to pay. Write pl/sql program to calculate v_pay according to the input value of p_buy and p_x. Print v_pay.

Exercise 2.7

Write a procedure with two input parameters **p_text**, which is a text string, and **p_char**, which is a character. Declare two local variables **v_before** and **v_after**. Assign v_before to be the portion p_text that is before p_char, and v_after to be the portion of p_text that is after p_char. Print v_before and v_after.

Exercise 2.8

Write a PL/SQL procedure **MyRemoveOne(p_text, p_char_1)** that will remove the first p_char_1 from p_text. For instance, MyRemoveOne ('123-456789','-') will remove the first '-' from '123-456789'; in other words, your program will print 123456789. Assume that p_char_1 has only one character.

Exercise 2.9

Improve the procedure **MyRemoveOne(p_text, p_char_1)** so you allow p_char_1 to be multiple characters.

Exercise 2.10

Write a PL/SQL procedure **MyReplaceOne(p_text, p_char_1, p_char_2)** that will replace the first p_char_1 with p_char_2 in p_text. For instance, MyReplaceOne ('123-456789','-','#') will replace the first '-' with '#' in '123-456789'; in other words, your program will print 123#456789. Assume that p_char_1 has only one character and p_char_2 has only one character.
Note: Oracle has a function called Replace. Please don't use this function in your code! ^__^

Exercise 2.11

Improve the procedure **MyReplaceOne(p_text, p_char_1, p_char_2)** so you allow p_char_1 to be multiple characters and p_char_2 to be multiple characters.

Note: A handy script to run testing of your program

It is likely that you will revise your program and test many times! You can write a script like the following to do all together

a:MyCode_TestScript.sql

```
-- set echo and spool
Set echo on
spool a:MyCode_TestScript.txt

-- create your procedure
Create or replace procedure MyCode (
    p1 number,
    p2 number) as
Begin
    .....;
    .....;
End;
/

-- show compiling errors and pause
Show err
Pause

-- test it with different test data
Exec mycode(5,2);
Exec mycode(10,3);

Spool off
```

The IF Statement

```
    IF condition THEN
        statement1;
        statement2;
        statement3;
        ...
    END IF;
```

```
    IF condition THEN
        <statements>
    ELSE
        <statements>
    END IF;
```

```
    IF condition THEN
        statement;
        statement;
        ...
    ELSIF conditions THEN
        <statements>;...
        ...<as many ELSIF as you wish>
    ELSE
        statements;
    END IF;
```

Note:

· **ELSIF**

```
/* Program 1.1  Simple IF-THEN-ELSE  */
-- This program will tell you whether a number is an even or an odd number.

create or replace PROCEDURE TellMeEvenOrOdd (p_Number number) as
    v_number  number := p_Number;
```

```
begin
    IF mod(v_number  ,2) = 0 THEN
        dbms_output.put_line ('It is an even number.');
    ELSE
        dbms_output.put_line ('It is an odd number.');
    END IF;
end;
/
```

Nested IF Statement

```
/* Program 1.1 Nested IF Statement */
-- This program will examine if the number entered is negative or non-integer.

create or replace PROCEDURE TellMeEvenOrOdd (p_Number Number) as
    v_number number := p_number;
begin
    IF v_number < 0 THEN
        dbms_output.put_line ('x1 must be a positive number! Please try again!');
    ELSE
        IF mod(v_number,2) = 0 THEN
            dbms_output.put_line ('x1 is an even number.');
        ELSIF mod(v_number, 2) = 1 THEN
            dbms_output.put_line ('x1 is an odd number.');
        ELSE
            dbms_output.put_line ('x1 must be an integer number! Please try again!');
        END IF;
    END IF;
end;
/
```

Exercise 2.11

Write a PL/SQL procedure **GradePoint(p_LetterGrade)**. The input parameter **p_LetterGrade** is a letter grade, such as A, B, C, D, or F. The procedure displays the grade point (that is, 4, 3, 2, 1, or 0). If the input is any other letters (such as H), then the program will display the message: **H is not a valid grade**.

Think point: The knowledge that 'A' is 4, 'B' is 3, ... is now coded in your program. Can you store the knowledge in tables? What are the pros and cons? How would that change your programs?

Exercise 2.12

For the following PL/SQL Program:

```
BEGIN
    IF ConditionA THEN
        actionA1;
        actionA2;
    ELSE
        action X;
        IF ConditionB THEN
            actionB1;
        ELSE
            actionB2;
        END IF;
        actionY;
    END IF;
END;
/
```

Answer the following questions:
a) If condition A is true and condition B is true, which action(s) would be executed?
b) If condition A is not true and condition B is true, which action(s) would be executed?
c) If condition A is not true and condition B is not true, which action(s) would be executed?

Exercise 2.13

a) What is the value of x at the end of the program?

```
x:=5;
IF x>= 5 then
    x:=x*2;
ELSIF x>=10 then
    x:=x*3;
ELSE
    x:=x*4;
END IF;
```

b) What is the value of x at the end of the program?

```
x:=5;
IF x>=5 then
    x:=x*2;
If x>=10 then
        x:=x*3;
    End If;
ELSE
    x:=x*4;
END IF;
```

Exercise 2.14

Write a PL/SQL procedure **`StudentStatus(p_CreditHour)`** to do the following:

If `p_CreditHour` is between 0 and 30 (including), the system prints "This student is a Freshmen" on the screen; if between 31 and 60 credits, print "This student is a Sophomore", between 61-90 credits, print "This student is a Junior"; for more than 91 credits, print "This student is a Senior." For negative numbers, print "This is an invalid input. Number of credits must be a positive number."

Exercise 2.15

Does the following program reflect the logic of the previous problem?

```
IF credit <= 30 THEN
        dbms_output.putline ('This student is a Freshmen'.);
END IF;
IF credit <= 60 THEN
        dbms_output.putline ('This student is a
Sophomore.');
END IF;
IF credit <= 90 THEN
        dbms_output.putline ('This student is a Junior.');
END IF;
IF credit > 90 THEN
        dbms_output.putline ('This student is a Senior.');
END IF;
```

Exercise 2.16

The following is the return/refund policy of Thing'n Things.

If an item is returned with a receipt and with the tag on, then give full-price refund. If the customer paid cash originally, then refund cash; if it was a credit card transaction, then credit to the credit card. If the customer does not have the original credit card, then issue a full-price store credit. If the item is returned without receipt but with the

tag on, then issue a sales-price store credit (that is, the lowest sales price within the last 30 days). For any item returned without a tag, call supervisor.

Write **IF.. THEN** statements to describe the above logic. (You don't need to write it in perfect PL/SQL statements. Just use structured English to code it; such as

```
IF receipt is yes and tag is yes
        THEN give full-price refund;
        ....
END IF;
```

Exercise 2.17

Customers are paying at the cash register**. p_AmountDue** is the amount of the merchandise, and **p_Pay** is how much cash the customer is giving to the cashier. This exercise requires you to write a program to tell the cashier how many twenty-dollar-bill, ten-dollar-bill, five-dollar-bill, and one-dollar-bill the cashier should give back. Call this procedure **GetChange** with 2 input parameters, **p_AmountDue** and **p_Pay**, and print your results.

Note 1. Assume that all the amounts are integer (there is no coins). Also, there are only 20, 10, 5, and 1-dollar bills.

Note 2. If the **p_Pay** is less than the amount due, the program should print "You need to give me more money!". If the paid is exactly the amount, the program should print "You just gave me exact change! Thank you!".

```
SQL> exec GetChange(12,200);

9  Twenty Dollar Bill
1  Five Dollar Bill
3  One Dollar Bill

PL/SQL procedure successfully completed.
```

Homework 3. PL/Sql Programming

1. Complete Exercise _____. Test your program until it is correct.
2. Turn spool on. Execute your procedure several times with different amount.
3. Turn in a copy of your procedure code.
4. Turn in a copy of your spool file (so I can see that the program is running correctly).

The LOOP and EXIT Statement

The syntax is

```
LOOP
    statement1;
    statement2;
    ...
END LOOP;
```

Make sure there is an exit point of your loop.

```
LOOP
    statements;
    IF <condition> THEN
        EXIT;
    END IF;
    statements;
```

```
END LOOP;
```

or

```
LOOP
    statements;
    EXIT WHEN <conditions>;
    statements;
END LOOP;
```

Note: EXIT will take you out of the "current" loop.

```
declare
    n integer := 1;
begin
    loop
        dbms_output.put_line (n);
        n := n+1;
        if n >= 10 then
            exit;
        end if;
    end loop;
end;
/
```

The WHILE LOOP Statement

The syntax is

```
WHILE <condition> LOOP
    statements;
END LOOP;
```

The program will first examine the condition. If it is true, then the program executes the statements; otherwise, it jumps out of the loop.
Note: any condition may have three possible states: true, false, or null. The WHILE loop will execute only when the while condition is true. i.e., make sure your condition will not become NULL.

```
declare
    n integer := 1;
begin
    while n < 10 loop
        dbms_output.put_line (n);
        n := n+1;
    end loop;
end;
/
```

Notice that the conditions in `LOOP-EXIT` and `WHILE-LOOP` are reversed (one is when n<10 and the other one is when n<=10).

The FOR LOOP Statement

In the FOR LOOP statement, you declare the loop index, a starting number, and an ending number. The loop index increments 1 each time until the loop is completed.

```
-- no need to declare n
begin
    FOR n IN 1..10 LOOP
        dbms_output.put_line (n);
    END LOOP;
end;
/
```

Exercise 2.18

What will the following program print?

a)

```
j := 0;
For i in 1..10 LOOP
    If i >= 3 then
        exit;
        dbms_output.put_line (j);
    end if;
    j := i;
end Loop;
```

b)

```
For i in 1..10 LOOP
    j := 0;
    If i >= 3 then
        dbms_output.put_line (j);
        exit;
    end if;
    j := i+j;
end loop;
```

Exercise 2.19

Write a procedure `PrintTable(p_BaseNumber)` that will produce a multiplication table according to the base number. For instance,

```
SQL> exec PrintTable(2)
```

will print the following:

```
2x1=2
2x2=4
2x3=6
2x4=8
2x5=10
2x6=12
2x7=14
2x8=16
2x9=18
```

```
SQL> exec PrintTable(3)
```

will print the following:

```
3x1=3
3x2=6
3x3=9
3x4=12
3x5=15
3x6=18
3x7=21
3x8=24
```

```
3x9=27
```

Exercise 2.20 The Multiplication Table

Write a procedure **`PrintWholeTable`** to print the entire multiplication table:

```
2x1=2
2x2=4
2x3=6
2x4=8
2x5=10
2x6=12
2x7=14
2x8=16
2x9=18
3x1=3
3x2=6
3x3=9
....
...
9x7=63
9x8=72
9x9=81
```

Exercise 2.21 The Love Wizard

Write a procedure **LoveWizard(p_MagicNumber)** that will produce the following output

```
    SQL> exec LoveWizard(7)

    Welcome to the Love Wizard!
    your magic number is 7 ......

    He loves you...
    He loves you not...
    He loves you...
    He loves you not...
    He loves you...
    He loves you not...
    He loves you...

    ===> He loves you!!!
```

```
    SQL> exec LoveWizard(8)

    Welcome to the Love Wizard!
    your magic number is 8 ......

    He loves you...
    He loves you not...
    He loves you...
    He loves you not...
    He loves you...
    He loves you not...
    He loves you...
    He loves you not...

===> He loves you not :-(
```

Exercise 2.22 The MyFill Procedure

(1) Write a procedure **MyFill(p_start, p_step, p_times)** to imitate the "fill" function in spreadsheet:

```
    SQL> exec myfill (1, 2, 5)
```

```
1
3
5
7
9
```

```
SQL> exec myfill (2, 2, 5)

2
4
6
8
10
```

```
SQL> exec myfill (101, 1, 8)

101
102
103
104
105
106
107
108
```

```
SQL> exec myfill (1000, 10, 6)

1000
1010
1020
1030
1040
1050
1060
```

(2) Write the MyFill procedure in LOOP..EXIT, WHILE LOOP, and FOR LOOP. 

Exercise 2.23

Write a PL/SQL procedure **MyRemoveAll(p_text, p_char_1)** that will remove ALL p_char_1 from p_text. For instance, MyRemoveOne ('123-45-67-89','-') will remove the all '-'s from '123-45-67-89; in other words, your program will print 123456789. Assume the following: p_char_1 has only one character.

-- Improve your code by writing a MyRemoveAll(p_text, p_char_1) to handle situations when p_char_1 contains multiple characters.

Exercise 2.24

Write a PL/SQL procedure **MyReplaceAll(p_text, p_char_1, p_char_2)** that will replace ALL p_char_1 with p_char_2 in p_text. For instance, MyReplaceOne ('123-45-67-89','-','#') will replace all '-' with '#' in '123-45-67-89; in other words, your program will print 123#45#67#89. Assume the following: p_char_1 has only one character and p_char_2 has only one character.
Note: Oracle has a function called Replace. Please don't use this function in your code!

-- Improve your code by writing a **MyReplaceAll(p_text, p_char_1, p_char_2)** to handle situations when p_char_1 and p_char_2 each contain multiple characters.

Homework 4. PL/Sql Programming

1. Complete Exercise _____. Test your program until it is correct.
2. Turn spool on. Execute your procedure several times with different test data.
3. Turn in a copy of your procedure code.
4. Turn in a copy of your spool file (so I can see that the program is running correctly).

Topic 3. SQL in PL/SQL

You can write DMLs (**INSERT**, **UPDATE**, **DELETE**), query (**SELECT**), and transaction control language (**COMMIT**, **ROLLBACK**) in PL/sql blocks. Data Definition Language (**CREATE**, **ALTER**, **DROP**, etc.) are not allowed. Of the SQL allowed in PL/SQL blocks, some work the same way while others are slightly different.

The INSERT Statement

· PL/SQL **INSERT** works the same way as SQL.
· The **students** table has five fields: **snum**, **sname**, **major**, **standing**, and **balance**. All three of the following `INSERT` statements work in the PL/SQL block.

```
begin
    insert into students values ('102','Betty',null,null,null);
    insert into students (snum, sname) values ('102','Betty');
    insert into students select * from students;
end;
/
```

The UPDATE Statement

· The **UPDATE** statement works the same way as in SQL.

```
declare
    vNewStanding varchar2(30);
    vBalanceIncrease number;
begin
    vNewStanding := 'Sophomore';
    update student
       set standing = vNewStanding
     where snum='102';

    vBalanceIncrease:=20;
    update student
       set balance = balance + vBalanceIncrease
     where snum='101';
end;
/
```

The DELETE Statement

The **DELETE** statement works the same as in the SQL.

```
create or replace PROCEDURE DeleteStudent (
    p_major IN OUT varchar2) IS
begin
    p_Major := 'IS';
    delete from student where major=p_Major;
end;
/
```

However, watch out for the following case!!

```
create or replace PROCEDURE DeleteStudent (
    major IN OUT varchar2 ) IS
begin
    major := 'IS';
    delete from student where major=major;
end;
/
```

What will it do? Since in the delete statement, the condition is where major=major (which is always true!), ALL students will be deleted, not just the IS student!

The SELECT..INTO Statement

In PL/SQL, the function of **SELECT...** is to copy the data to a local, PL/SQL variable so you can further use it in your PL/SQL code. The syntax goes like this:

```
SELECT    columns or     INTO    PL/SQL          FROM tables WHERE conditions;
          expressions            variables
```

Notice that the PL/SQL variables need to be declared in the **declare...** first.

```
declare
    vSName students.sname%type;
    vMajor students.major%type;
    vGPA student.gpa%type;

    vTotalCrhr number(3);

    rStudent   student%rowtype;  -- a row type

begin
   -- get one value
   select sname into vSName from students where snum='101';

   -- get several things in one trip
   select sname, major, gpa
     into vSName, vMajor, vGPA
     from students where snum='101';

   -- get a calculated result
   select sum(crhr) into vTotalCrhr from students where snum='101';

   -- select the entire row in one shot
   select * into rStudent from students where snum='101';


   IF rStudent.major = 'IS' THEN
      Blah blah blah;
```

Note:
1. **%type**
2. **%RowType**

Important Note!
• **SELECT... INTO...** should return only one row. If it returns no row, there will be an **NO_DATA_FOUND** error message and **the program will exit**. If it returns more than one row, there will be a **TOO_MANY_ROWS** error message and **the program will exit**. When either situation occurs, it should be handled with the Exception section, which we will go over in the future.
• If you need to select multiple rows into your PL/SQL variables, you need to use **cursor** (we will talk about this in future topic!)
• Note that for insert, update, delete, it does not matter how many rows are affected by the action.

SQL%...

pl/sql has built-in **SQL%...** functions that will return values.

SQL%FOUND

Returns TRUE if the last SQL statement (SELECT...INTO, INSERT, UPDATE, or DELETE) has found any data. ie., for a select..., it has selected data; for an update, it has updated data, and so forth. It returns FALSE if otherwise.

SQL%NOTFOUND

Returns FALSE if the last SQL statement (SELECT...INTO, INSERT, UPDATE, or DELETE) has found any data. ie., for a select..., it has selected data; for an update, it has updated data, and so forth. It returns TRUE if otherwise.

SQL%ROWCOUNT

Return the number of records processed by the latest SQL statement.

Note:

You can use **SQL%NOTFOUND** with **Select...into**, but it is not useful. Since selecting nothing is treated as an error in Oracle, the program would exit or jump to the EXCEPTION section right away. Therefore, the **SQL%NOTFOUND** part will never be executed.

```
SQL> select * from student;

SNUM      SNAME           MAJOR      S   BALANCE
-------- --------------- ---------- - ---------
222       Betty           IS         5       100
111       Andy            IS         4       160
333       Cindy           MKT        4       100
```

```
/* Program p33. SQL%... */
set serveroutput on
begin
    update student set major='MIS' where major='IS';
    IF sql%found then
        dbms_output.put_line ('Update completed.');
    END IF;
    IF sql%notfound then
        dbms_output.put_line ('There is no record to be updated.');
    END IF;
    dbms_output.put_line ('There are '||sql%rowcount||' Records Updated.');
end;
/
```

```
SQL> start a:p33.sql
Update completed.
There are 2 Records Updated.

PL/SQL procedure successfully completed.
```

The COMMIT and ROLLBACK Statements

COMMIT and ROLLBACK work in PL/SQL blocks the same way as they work in SQL.

```
begin
    insert into student (snum, sname) values ('101','Andy');
    insert into student (snum, sname) values ('102','Betty');
    insert into student (snum, sname) values ('103','Cindy');
    commit;
end;
/
```

```
begin
    insert into student (snum, sname) values ('101','Andy');
    insert into student (snum, sname) values ('102','Betty');
    insert into student (snum, sname) values ('103','Cindy');
    rollback;
    commit;
end;
/
```

```
begin
    insert into student (snum, sname) values ('101','Andy');
    insert into student (snum, sname) values ('102','Betty');
    insert into student (snum, sname) values ('103','Cindy');
    commit;
    rollback;
end;
/
```

For Exercise 3.1 - 3.4, assume the following tables are in place:

Students

SNUM	SNAME	STANDING	Major	GPA	Major_GPA
101	Andy	4	IS	2.8	3.2
102	Betty	2		3.2	
103	Cindy	3	IS	2.5	3.5
....	....	....	...	...	...

Courses

DEPT	CNUM	CTITLE	CRHR	STANDING
IS	300	Intro to MIS	3	2
IS	310	Statistics	3	3
IS	380	Database	3	3
....	....	.....	...	....

SchClasses

CallNum	Year	Semester	DEPT	CNUM	Section	Day	Time	Room	Instructor	Capacity
10110	2013	Sp	IS	300	1	MW	800-930	222	Smith	45
10115	2013	Sp	IS	300	2	MW	900-1015	235	Lee	35
10120	2013	Sp	IS	380	1	TTh	900-1015	112	Jones	35
10125	2013	Fa	IS	300	1	MW	1330-1445	111	TBA	118
10130	2013	Fa	IS	310	1	MWF	1000-1100	121	TBA	33
....		.....	....	....	....	.....	.....	....	.....	.....

Enrollments

SNUM	CallNum	GRADE
101	10110	F
102	10110	A
103	10120	A
101	10125	
102	10130	
....	.....	....

Exercise 3.1

Write a procedure **AddMe (p_snum, p_CallNum)** to add/enroll a student to the course.

Exercise 3.2

Certain classes require academic standing, such as Junior or Senior. The coding for standing is 1 for freshman, 2 for sophomore, 3 for junior, and 4 for senior. A student can enroll in a class only when his standing is equal or higher than the standing required by the course. Refine the **AddMe** procedure so that the system will add a student only when the standing requirement is met.

Exercise 3.3

Within the same semester, a student can only enroll for 15 or less credit hours. Write a PL/SQL procedure **AddMe (p_snum, p_CallNum)** that will enroll a student only when after enrolling, his total semester credit hours of the current semester is equal or less to 15.

Exercise 3.4

For a given student, if his total credit hours is between 0-30, update his standing to "1"; if his total credit hours is between 31-60, update his standing to "2"; if his total credit hours is between 61-90, update his standing to "3"; if his total credit hours is greater than 91, then update his standing to "4". Write a procedure **Update_Standing (p_snum)** to do that.

Exercise 3.5

If a student receives a D or a F of a course, its credit hours will not count toward his standing or his graduating credits. Refine the **Update_Standing(p_snum)** procedure so that courses with D or F grades are not considered toward his standing requirements.

Exercise 3.6

Each course has a capacity limitation. Write a procedure **AddMe (p_snum, p_CallNum)** that will enroll a student only if after his enrollment, the course is still kept within the capacity limitation.

Exercise 3.7

Write a procedure **Update_GPA(p_snum)** to update a student's GPA. A GPA is calculated in the following fashion. Assume a student receives an A on a 3-credit-hour course and a D on a 2-credit-hour course. His grade is (4x3 + 1x2) / (3+2) = 2.8.

Exercise 3.8

Write a procedure **Validate_Student (p_snum)** that prints 'The student number is valid.' if the student is a valid student (i.e., his/her record exists in the Student table). This procedure prints "The student number is invalid." if the student is not a valid student.

Exercise 3.9

Write a procedure **Validate_Callnum(p_callnum)** to examine whether the call number is valid.

Exercise 3.10

Write a procedure **DropMe(p_snum, p_callnum)** to drop the student from a course. This procedure marks a "W" in the student's GRADE column of the ENROLL table. If this student is not enrolled in this class, print an error message.

Exercise 3.11

Write a procedure **Conflict(p_start_1, p_end_1, p_start_2, p_end_2)**. All 4 parameters are NUMBER. **p_start_1** and **p_end_1** is the starting and ending time of course 1; **p_start_2** and **p_end_2** is the starting and ending time of course 2. Write the procedure to display "There is a time conflict" if there is a time conflict between two courses; it displays "There is no time conflict" if there is no time conflict between two courses.

Homework 5. SQL in pl/sql

Due Date: _____

1. Complete Exercise _____. Test your program until it is correct.
2. Turn spool on. Execute your procedure several times with different amount.
3. Turn in a copy of your procedure code.
4. Turn in a copy of your spool file (so I can see that the program is running correctly).

Topic 4. Procedures and Functions

Procedure vs. Function

* Both are "stored subprograms"
* Both can pass parameters
* Procedure can be stand alone; functions need to be embedded in a command
* You can call a procedure/function from a procedure/function
* Modulization and reusable programming

Procedure

· Calling a procedure from a procedure

```
    create or replace procedure Print_Receipt (
        p_snum students.snum%type,
        p_callnum enrollments.callnum%type) as
    begin
        dbms_output.put_line (p_snum ||'is enrolled in '||p_callnum);
    end;
    /
```

```
    create or replace procedure AddMe (
        p_snum students.snum%type,
        p_callnum enrollments.callnum%type) as
    begin
        insert into Enrollments values (p_snum, p_callnum, null);
        commit;
        Print_Receipt (p_snum, p_callnum);
    end;
    /
```

· Parameters can be IN, OUT, or IN OUT; default is IN

```
create or replace procedure Validate_Student (
    p_snum         IN         students.snum%type,
    p_Error_Text       OUT    varchar2) as

    v_count number(3);

begin
    SELECT count(*) INTO v_count
    FROM students
    WHERE snum=p_snum;

    IF v_count = 0 THEN
        p_Error_Text := 'Student Number '||p_snum||' Invalid. ';
    END IF;
end;
/
```

```
create or replace procedure AddMe (
    p_snum students.snum%type,
    p_callnum enrollments.callnum%type) as

    v_Error_Text varchar2(200);

begin
    Validate_Student (
        p_snum,
        v_Error_Text);
    IF v_Error_Text is null THEN
        insert into Enrollments values (
            p_snum, p_callnum, null);
    ELSE
        dbms_output.put_line (v_Error_Text);
    END IF;
end;
/
```

Putting it together...

```
create or replace procedure Validate_Callnum (
    p_CallNum          IN             Enrollments.CallNum%type,
    p_Error_Text       OUT            varchar2) as

    v_count number(3);

begin
    SELECT count(*) INTO v_count
    FROM SchClasses
    WHERE CallNum=p_CallNum;

    IF v_count = 0 THEN
        p_Error_Text := Call Number '||p_CallNum||' Invalid. ';
    END IF;
end;
/
```

```
create or replace procedure AddMe (
    p_snum students.snum%type,
    p_callnum enrollments.callnum%type) as

    v_Error_Text varchar2(200);
    v_Error_Msg varchar2(200);

begin
    Validate_Student (
        p_snum,
        v_Error_Text);
    v_Error_Msg := v_Error_Text;
    Validate_Callnum (
        p_CallNum,
        v_Error_Text);
    v_Error_Msg := v_Error_Msg||v_Error_Text;
    IF v_Error_Msg is null THEN
        insert into Enrollments values (
            p_snum, p_callnum, null);
    ELSE
        dbms_output.put_line (v_Error_Msg);
    END IF;
end;
/
```

Exercise 4.1

Write a procedure **Validate_Credit_Limit (p_snum)** that will return an error text 'Student exceeds 15-credit-hour limit after enrollment." if after the enrollment, this student will be enrolled in more than 15 credit hours..

Incorporate this procedure **Validate_Credit_Limit (p_snum)** into the **AddMe** procedure. That is, a student can be enrolled to a course only when after his/her enrollment, his/her total semester credit hours is equal or less than 15.

Exercise 4.2

Write a procedure **Validate_Standing(p_snum, p_callnum)** that will return an error text "Course Standing not met." if the student does not have meet the standing requirement of the course.

Incorporate this procedure **Validate_Standing (p_snum, p_CallNum)** into the **AddMe** procedure. That is, a student can be enrolled to a course only he/she meets the standing requirement.

Functions

• A function will "return"(or "becomes") a value.
• Function calls have to be embedded in a command.
• You have used built-in functions, such as max, min, greatest, least, rtrim, substr, etc.

One Example of Functions:

```
create or replace FUNCTION Student_is_Valid (
    p_snum  students.snum%type)

    Return Boolean is

    v_count number(3);

begin
    SELECT count(*) INTO v_count
    FROM students
    WHERE snum=p_snum;

    IF v_count = 0 THEN
        return FALSE;
    ELSE
        return TRUE;
    END IF;
end;
/
```

```
create or replace procedure AddMe (
    p_snum students.snum%type,
    p_callnum enrollments.callnum%type) as

    v_Error_Text varchar2(200);

begin

    IF Student_is_Valid (p_snum) is true THEN
        insert into Enrollments values (p_snum, p_callnum, null);
    ELSE
        dbms_output.put_line ('Student Number '||p_snum||' Invalid. ');
    END IF;
end;
/
```

or

```
create or replace procedure AddMe (
    p_snum students.snum%type,
    p_callnum enrollments.callnum%type) as

    v_Error_Text varchar2(200);

begin
```

```
    IF Student_is_Valid (p_snum) THEN
        insert into Enrollments values (p_snum, p_callnum, null);
    ELSE
        dbms_output.put_line ('Student Number '||p_snum||' Invalid. ');
    END IF;
end;
/
```

Or this way, too...

```
create or replace FUNCTION Validate_Student (
    p_snum  students.snum%type)

    Return varchar2 is

    v_count number(3);
    v_Error_Text varchar2(200);

begin
    SELECT count(*) INTO v_count
    FROM students
    WHERE snum=p_snum;

    IF v_count = 0 THEN
        v_Error_Text := 'Student Number '||p_snum||' Invalid. ';
    ELSE
        v_Error_Text := null;
    END IF;

    RETURN v_Error_Text;

end;
/
```

```
create or replace procedure AddMe (
    p_snum students.snum%type,
    p_callnum enrollments.callnum%type) as

    v_Error_Text varchar2(200);

begin
    v_Error_Text := Validate_Student(p_snum);

    IF v_Error_Text is null THEN
        insert into Enrollments values (p_snum, p_callnum, null);
    ELSE
        dbms_output.put_line (v_Error_Text);
    END IF;
end;
/
```

Or even better…

```
    …
    IF func_Validate_Student(p_snum) is null THEN
        insert into Enrollments values (p_snum, p_callnum, null);
    …
```

Note1. Can functions have **IN** and **OUT** parameters just like procedures?? Yes! However, it is considered bad practice to use OUT parameters in functions. The rule of thumb is, a function is used when you have only ONE thing to return. Otherwise, use a procedure.

Exercise 4.3

Write a function **GradePoint(p_LetterGrade)** that will return 4 if the letter grade is 'A', 3 if letter grade is 'B', etc.

Exercise 4.4

Write a function **Standing(p_TotalCreditHour)** that will return 1 if total credit hour is between 0-30, 2 if between 31-60, etc.

Exercise 4.5

Write a function **StandingOK(snum, cnum)** that returns True if the student has appropriate standing to take the course, false otherwise.

Exercise 4.6

Write a function **ClassCapacityOK(cnum)** that will return true if the class still has room for one more student, and false otherwise.

Exercise 4.7

Incorporate the functions **StandingOK** and **ClassCapacityOK** into **AddMe**. A student is enrolled only if he/she meets the standing requirement and the class is not full.

How to <u>Test a Procedure</u> with OUT Parameter

```
create or replace procedure proc_Validate_Snum  (
    p_snum students.snum%type,
    p_Error_Text OUT Varchar2) as
    v_count number;
begin
    select count(*) into v_count
    from students
    where snum=p_snum;

    IF v_count=0 then
        p_Error_Text := 'Student Invalid';
    End If;
end;
```

```
-- this doesn't work
SQL> exec proc_Validate_Snum (101);
BEGIN proc_Validate_Snum (101); END;

      *
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00306: wrong number or types of arguments in call to 'PROC_VALIDATE_SNUM'
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored

-- this doesn't work either
SQL> exec proc_Validate_Snum (101, v_Error_Text);
BEGIN proc_Validate_Snum (101, v_Error_Text); END;

                          *
ERROR at line 1:
ORA-06550: line 1, column 27:
PLS-00201: identifier 'V_ERROR_TEXT' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored

--You need to write a script
```

```
declare
    v_Error_Text varchar2(1000);
begin
    proc_Validate_Snum (101, v_Error_Text);
    dbms_output.put_line (v_Error_Text);
end;
/
```

```
SQL> declare
  2   v_Error_Text varchar2(1000);
  3  begin
  4   proc_Validate_Snum (101, v_Error_Text);
  5   dbms_output.put_line (v_Error_Text);
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL> declare
  2   v_Error_Text varchar2(1000);
  3  begin
  4   proc_Validate_Snum (999, v_Error_Text);
  5   dbms_output.put_line (v_Error_Text);
  6  end;
  7  /
Student Invalid

PL/SQL procedure successfully completed.
```

How to <u>Test a Function</u> with a RETURN value

```
create or replace function func_Validate_Snum  (
    p_snum students.snum%type)
    return varchar2 as

    v_count number;

begin
    select count(*) into v_count
    from students
    where snum=p_snum;

    IF v_count=0 then
        Return 'Student Invalid';
    Else
        Return 'Student Valid';
    End If;
end;
/
```

```
--You need to write a script
```

```
declare
    v_Error_Text varchar2(1000);
begin
    v_Error_Text := func_Validate_Snum (999);
    dbms_output.put_line (v_Error_Text);
end;
/
```

```
SQL> declare
  2   v_Error_Text varchar2(1000);
  3  begin
  4   v_Error_Text := func_Validate_Snum (101);
  5   dbms_output.put_line (v_Error_Text);
  6  end;
  7  /
Student Valid

PL/SQL procedure successfully completed.

SQL> declare
  2   v_Error_Text varchar2(1000);
  3  begin
  4   v_Error_Text := func_Validate_Snum (999);
  5   dbms_output.put_line (v_Error_Text);
  6  end;
  7  /
Student Invalid

PL/SQL procedure successfully completed.
```

Or this way, too:

```
SQL> select func_validate_snum(101) from dual;

FUNC_VALIDATE_SNUM(101)
--------------------------------------------------
Student Valid

SQL> select func_validate_snum(999) from dual;

FUNC_VALIDATE_SNUM(999)
--------------------------------------------------
Student Invalid
```

Return

The word **RETURN** in procedures or functions is to stop processing and return to the calling program. In functions, **RETURN** is to return a value. In procedures, `RETURN` cannot be used with a value.

Question: What does **Exit** do??

Object Dependencies

Note1. There are dependencies among objects, including procedures, functions, and tables.
Note2. A change of a procedure/function/table will cause objects that call it to become INVALID.
Note3. An Invalid object will be re-compiled at run time. Make sure all objects are VALID when your system goes live.
Note4. **Create or replace...** will automatically recompile the procedure/function. Or you can use the following command to explicitly recompile a procedure/function.

```
Alter procedure MyProcedure1 compile;
Alter Function MyFunction1 compile;
```

-- In the following illustration, the **AggregateOne** function calls the **Enroll** table; the **AggregateAll** procedure calls the **AggregateOne** function.

```
SQL> select object_name, object_type, created, timestamp, status
  2  from user_objects;

OBJECT_NAME                 OBJECT_TYPE          CREATED   TIMESTAMP           STATUS
------------------------- ------------------ --------- ------------------- -------
ADVISOR                      TABLE               20-MAR-11 2011-03-20:14:01:16 VALID
AGGREGATEALL                PROCEDURE           19-MAR-11 2011-03-19:17:11:41 VALID
AGGREGATEONE                FUNCTION            19-MAR-11 2011-03-19:17:11:38 VALID
ENROLL                       TABLE               19-MAR-11 2011-03-19:16:16:46 VALID
...

8 rows selected.

SQL> alter table ENROLL modify (StudentName char(35));

Table altered.

SQL> select object_name, object_type, created, timestamp, status
  2  from user_objects;

OBJECT_NAME               OBJECT_TYPE        CREATED   TIMESTAMP           STATUS
------------------------- ------------------ --------- ------------------- -------
ADVISOR                   TABLE              20-MAR-11 2011-03-20:14:01:16 VALID
AGGREGATEALL              PROCEDURE          19-MAR-11 2011-03-19:17:11:41 INVALID
AGGREGATEONE              FUNCTION           19-MAR-11 2011-03-19:17:11:38 INVALID
ENROLL                       TABLE              19-MAR-11 2011-03-20:16:55:34 VALID

8 rows selected.

SQL> alter function AggregateOne compile;

Function altered.

SQL> select object_name, object_type, created, timestamp, status
  2  from user_objects;

OBJECT_NAME               OBJECT_TYPE        CREATED   TIMESTAMP           STATUS
------------------------- ------------------ --------- ------------------- -------
ADVISOR                   TABLE              20-MAR-11 2011-03-20:14:01:16 VALID
AGGREGATEALL              PROCEDURE          19-MAR-11 2011-03-19:17:11:41 INVALID
AGGREGATEONE              FUNCTION           19-MAR-11 2011-03-20:16:56:17 VALID
ENROLL                    TABLE              19-MAR-11 2011-03-20:16:55:34 VALID

8 rows selected.

SQL> alter table enroll modify (StudentName char(35));

Table altered.

SQL> select object_name, object_type, created, timestamp, status
  2  from user_objects;

OBJECT_NAME               OBJECT_TYPE        CREATED   TIMESTAMP           STATUS
------------------------- ------------------ --------- ------------------- -------
ADVISOR                   TABLE              20-MAR-11 2011-03-20:14:01:16 VALID
AGGREGATEALL              PROCEDURE          19-MAR-11 2011-03-19:17:11:41 INVALID
AGGREGATEONE              FUNCTION           19-MAR-11 2011-03-20:16:56:17 INVALID
ENROLL                    TABLE              19-MAR-11 2011-03-20:16:56:47 VALID

8 rows selected.

SQL> Alter procedure AggregateAll compile;

Procedure altered.

SQL> select object_name, object_type, created, timestamp, status
  2  from user_objects;

OBJECT_NAME               OBJECT_TYPE        CREATED   TIMESTAMP           STATUS
------------------------- ------------------ --------- ------------------- -------
ADVISOR                   TABLE              20-MAR-11 2011-03-20:14:01:16 VALID
AGGREGATEALL              PROCEDURE          19-MAR-11 2011-03-20:16:56:58 VALID
AGGREGATEONE              FUNCTION           19-MAR-11 2011-03-20:16:56:58 VALID
ENROLL                    TABLE              19-MAR-11 2011-03-20:16:56:47 VALID
```

Homework 6. Functions and Procedures

Write a procedure **AddMe(p_snum, p_CallNum)** that will enroll the student to a course. The program should check for the following things:

1. The student must be a valid student.

2. The Call Num must be a valid call num.

3. There is still room in the class.

4. After enrolling, the total credit hours of this student do not exceed 15 credit hours.

If the student number <u>or</u> the call number is invalid, your program will not check for 3 or 4. If both are valid, then your program proceeds to check the 3 and 4. (Hint: You need to build a IF... structure to achieve this.)

• Your program will print an error message to explain <u>all</u> reasons why the enrollment has failed.

• You need to call at least 1 function and 1 procedure in the **AddMe** procedure.

Topic 5. Package

Package is a collection of PL/SQL modules (such as functions and procedures).

A package consists of a Package Specification and a Package Body. A package specification includes only the names of modules. Actual programs are written in the package body.

```
Create or Replace Package ENROLL is

Function CheckValidStudent (
    p_snum     Students.snum%type)
    Return Boolean;

Function CheckValidCourse (
    p_callnum  SchClasses.callnum%type)
    Return Boolean;

Procedure AddMe (
    p_snum     IN     Students.snum%type,
    p_callnum  IN     SchClasses.callnum%type,
    p_ErrorText    OUT        char);

End ENROLL;
/
```

```
Create or Replace Package Body ENROLL is

Function CheckValidStudent (p_snum Students.snum%type) Return Boolean is
    Begin
        <actual program>
    End;

Function CheckValidCourse (p_callnum SchClasses.callnum%type) Return Boolean is
    Begin
        <actual program>
    End;

Procedure CapacityOK (
    p_callnum      IN         SchClasses.callnum%type,
    p_ErrorText        OUT    char) is
    Begin
        If CheckValidCourse(p_callnum) THEN
            blah blah blah...
    End;

Procedure AddMe (
    p_snum     IN     Students.snum%type,
    p_callnum  IN     SchClasses.callnum%type,
    p_ErrorText    OUT        char) is
    Begin
        CapacityOK (p_snum, v_ErrorText);
        IF v_ErrorText is null THEN
            blah blah blah...
    End;

End ENROLL;
/
```

Notes:
1. Syntax of Package Spec and Package Body.
2. Package Spec must be created before the Package Body.
3. Function/Procedure name and parameters must be IDENTICAL between spec and body, otherwise you would get an error message. For instance, if in the header you write

```
Function CheckValidStudent (p_snum Students.snum%type) Return Boolean;
```

but in the body you write

```
Function CheckValidStudent (p_snum char) Return Boolean;
```

Even if student.snum is char, you will still get an error message!!

4. Modules declared in the spec must also be declared in the body. However, modules in the body do not need to be in the spec.

5. Only modules declared in the spec can be called by other packages/procedure/functions (in other words, "public"). Modules declared in the body but not in the spec are not "visible" by other objects (in other words, "private").

6. Modules in one package can be called directly.

7. To call a module in another package, use the following syntax

```
package.module
```

For instance, if I were to run the AddMe module in the Enroll package, I will write

```
begin
    Enroll.AddMe('101','10110');
end;
/
```

In another example, suppose you are writing a Withdraw package and in the the DropMe procedure you want to verify whether the student is valid or not. Since you have written the CheckValidStudent function in the ENROLL package and also declared it in the spec, you can reuse this code like this:

```
Package Body WITHDRAW is

.....

Procedure DropMe (......) is

    Begin
        If Enroll.CheckValidStudent THEN
            ....
```

Overloading

Modules in the same package can have the same name! This is the famous "overloading". For instance,

```
Package Body ENROLL is
...
Procedure AddMe (
    p_snum Students.snum%type,
    p_callnum SchClasses.callnum%type) is
    <actual code>

Procedure AddMe (
    p_snum student.snum%type,
    p_courseNum SchClasses.courseNum%type,
    p_SectionNum SchClasses.SectionNum%Type) is
    <actual code>
```

Although you have two procedures named AddMe, Oracle is able to distinguish one from the other by the way you call it: If you call it with 2 parameters, Oracle will run the one with 2 parameters; if you call it with 3 parameters, Oracle will run the one with 3 parameters. For instance,

```
Begin
    Enroll.AddMe ('101','10110');
End;
```

and

```
Begin
    Enroll.AddMe ('101','IS380','1');
End;
```

will trigger Oracle to run different AddMe module.

Restrictions of Overloading: Overloading works when the modules of the same name (1) have different number of parameters, or (2) have the same number of parameters, but at least one parameter differ in datatype "family". What is a datatype family? CHAR and VARCHAR2 are in the same family; NUMBER and NUMBER(3,1) are the same family. Therefore, the following overloading will NOT work:

```
Package Body ENROLL is
...
Procedure AddMe (
    p_snum char,
    p_callnum number) is
    <actual code>

Procedure AddMe (
    p_snum varchar2,
    p_courseNum integer) is
    <actual code>
```

Homework 7. Package

Due Date: _____

Put modules you wrote for Homework ____ in a Package. Create the package spec and package body. Compile both and make sure it runs. Run the Enroll module in the package.

Turn in a copy of the Package Spec and Package Body creation code.
Turn in a copy of execution, which shows that you call the module in the package and things work.

Topic 6. Cursors

"Cursor" defines of an “area” (a select... statement) to be processed. This “area” may contain multiple records. Each record can be passed to a PL/SQL program and be processed.

This program will move records of students who are cleared for graduation from the `STUDENT` table to the **StudentArchive** table. In the **StudentArchive** table, we also record the graduating year as '2010’.

Simple Cursor: Open, Fetch, Close

```
set serveroutput on

declare
    vSnum student.snum%type;
    vSname student.sname%type;
    vMajor student.major%type;
    CURSOR cStudent IS
        select snum, sname, major from student
        where standing='5';
    -- standing '5' means 'cleared for graduation'

begin
    open cStudent;
    LOOP
        fetch cStudent into vSnum, vSname, vMajor;
        EXIT when cStudent%NOTFOUND;
        insert into StudentArchive values (vSnum, vSname, vMajor, '2010');
        delete from student where snum=vSnum;
    END LOOP;
    close cStudent;
end;
/
```

There are 4 steps in a cursor processing:

1. Declaring a cursor: define what the cursor is.
   CURSOR *CursorName* **IS select....**

<u>Note 1.</u> The **SELECT...** statement has no `INTO` clause.
<u>Note 2.</u> You can use other PL/SQL variables in the **where...** clause. These variables need to be declared *before* the **CURSOR...** statement; such as

```
declare
    vMajor student.major%type;
    CURSOR cStudent is
        select snum from student where major=vMajor;
```

The following will produce an error message:

```
declare
    CURSOR cStudent is
        select snum from student where major=vMajor;
    vMajor student.major%type;
```

2. Open a Cursor
   OPEN *CursorName*;

<u>Note 1.</u> When the `open...` statement is executed, all variables in the cursor are "binded", i.e., their value will not change through out the cursor execution.

3. Fetch value
   FETCH *CursorName* **INTO** *PL/SQLVariables*;

<u>Note 1.</u> Each "fetch" return one record of value to the PL/SQL variable.

4. Close a Cursor
   CLOSE *CursorName*;

The For Loop Cursor

```
declare
    CURSOR cStudent is
        Select SSN, FirstName, LastName, Major from student;
begin
    FOR EachStudent IN cStudent LOOP
        Insert into TestTable values (EachStudent.SSN, EachStudent.FirstName, ...);
        < ..... >
    END LOOP;
End;
```

or

```
declare
    -- do not need to declare cursor
begin
    FOR EachStudent IN (
        Select SSN, FirstName, LastName, Major
        from student ) LOOP

        Insert into TestTable values (EachStudent.SSN, EachStudent.FirstName, ...);
        < ..... >

    END LOOP;
End;
```

Note 1. You do not need to declare **EachStudent**.

For Update and For Delete Cursor

Consider the following program:

```
    declare
        Cursor cStudent is
            Select SSN from Student where major = 'IS';
    begin
        For EachStudent in cStudent Loop
            Update Student set GPA=GPA+1 where ssn = EachStudent.SSN;
        End Loop;
    End;
```

You can use For Update (or For Delete) Cursor to be more efficient.

```
    declare
        Cursor cStudent is
            select SSN from Student where major = 'IS' For Update;
    begin
        For EachStudent in cStudent Loop
            Update Student set GPA=GPA+1 where CURRENT OF cStudent;
        End Loop;
    End;
```

Note: For Update/Delete cursor results in locking of records in cursor definition until a commit is issue.

Cursors with Parameters

```
    Craete
        Cursor cStudent (p_Major Student.Major%type) is
            select SSN from Student where Major = p_Major;
    begin
        Open cStudent ('IS');
            ....
        Close cStudent;
    end;
```

or

```
    declare
        Cursor cStudent (p_Major Student.Major%type) is
            select SSN from Student where Major = p_Major;
    begin
        For EachRecord in cStudent('IS') LOOP
            ....
    end;
```

Cursor Attributes

There are 4 pre-defined cursor attributes that is handy to use with exit or other conditions.

`CursorName%FOUND`
: Returns TRUE when the fetch found data; FALSE when the fetch did not find any data.

`CursorName%NOTFOUND`
: Returns TRUE when the fetch did not find data; FALSE when the fetch did find data.

`CursorName%ISOPEN`
: Returns TRUE if the cursor is open; FALSE otherwise.

`CursorName%ROWCOUNT`
: Returns the number of records that have been fetched.

Detailed Processing in Cursor

What happened when the fetch does not return any record?

```
declare
    vSnum student.snum%type;
    vSname student.sname%type;
    vMajor student.major%type;
    CURSOR cStudent IS
        select snum, sname, major from student
            where standing='5';
    -- standing: 1=Freshman, 2=Sophomore, 3=Junior, 4=Senior, 5=Cleared for Graduation

begin
    open cStudent;
    LOOP
        fetch cStudent into vSnum, vSname, vMajor;
        insert into StudentArchive values (vSnum, vSname, vMajor, '2010');
        delete from student where snum=vSnum;
        EXIT when cStudent%NOTFOUND;
    END LOOP;
    close cStudent;
end;
/
```

Note 1. When **fetch…** does not return more rows, it does not overwrite the variable with NULL either. The variable will still contain the value form the *previous* fetch!
Note 2. the difference between **fetch...** and **select...into**: If there is nothing to fetch, **fetch** will not return an error message. However, if **select...into** does not find any record, it will return an error message and exit entirely from the program.

Exercise 6.1

Write a PL/SQL program that will update every student's standing according to his/her current credit hours.

Exercise 6.2

Write a PL/SQL procedure to calculate and update every student's GPA. Note: please reference Ex 3.7 for the logic of GPA calculation.

Exercise 6.3

The administration wants to add a new column to the SchClasses table: CurrEnroll, which records the current enrollment figure of each class. Write a PL/SQL procedure to update the current enrollment number of each class.

Exercise 6.4

When a course is full and a student tries to add, the system would put the student on the waiting list. Students often want to know "where am I on the waiting list"? Write a PL/SQL program **GetWaiting**(p_CallNum) to display students who are on the waiting list for class p_CallNum by the order of their requested time. The student who called in first will be displayed first, with a ranking number of 1; the student who called in second will be displayed second, with a ranking number of 2, so on and so forth, like the following:

```
SQL> Exec GetWaiting(10110);

        Ranking SID     SName   RequestedTime
        ------- ---     ------- --------------
        1       107     George  1/1/2010 10am
        2       103     Cindy   1/1/2010 11am
        3       104     David   1/1/2010 2 pm
        4       106     Frank   1/1/2010 4 pm
        …..
```

Exercise 6.5

Program 5.4 may be unrealistic if there are hundreds of students on the waiting list, and the list gets really long. Write another program **GetWaitingRank** (p_Snum, p_Callnum) and let the student his/her ranking on the list, for instance,

```
SQL> Exec GetWaitingRank(10110, 103);
You are number 2 on the waiting list

SQL> Exec GetWaitingRank(10110, 106);
You are number 4 on the waiting list

SQL> Exec GetWaitingRank(10110, 199);
Sorry! You are not on the waiting list
```

Exercise 6.6

Write a PL/SQL program **RankGPA** to display all students by their GPA. The student with the highest GPA will be displayed first, with a ranking number of 1; the second highest student will be displayed with a ranking number of 2, so on and so forth, like the following:

```
        Ranking SID     SName   GPA     Major
        ------- ---     ------- ---     ------
        1       102     Betty   4.0     FIN
```

```
2       107     George  3.9     ACC
3       103     Cindy   3.8     ACC
4       111     Kyle    3.8     IS
…..
```

Exercise 6.7

Write a PL/SQL program **RankGPAbyMajor** to display students by their major and GPA. For each major, the student with the highest GPA will be displayed first, with a ranking number of 1; the second highest student will be displayed with a ranking number of 2, so on and so forth, like the following:

```
Ranking SID     SName   GPA     Major
------- ---     ------- ---     ------
1       107     George  3.9     ACC
2       103     Cindy   3.8     ACC
3       104     David   3.7     ACC
4       106     Frank   3.0     ACC
5       101     Andy    2.4     ACC
1       102     Betty   4.0     FIN
2       109     Irene   3.5     FIN
3       105     Ellen   2.8     FIN
1       111     Kyle    3.8     IS
2       108     Harry   3.5     IS
3       110     Jack    3.4     IS
4       115     Oreo    3.1     IS
```

Exercise 6.8

Write a PL/SQL procedure **TopGPAPerMajor** to display the top student (in terms of GPA) per major, like the following:

```
Ranking SID     SName   GPA     Major
------- ---     ------- ---     ------
1       107     George  3.9     ACC
1       102     Betty   4.0     FIN
1       111     Kyle    3.8     IS
```

Exercise 6.9

There are many majors in the College. Write a procedure **Top3EachMajor** that displays the top 3 student (in terms of GPA) per major. That is, display the top 3 students of IS, the top 3 students of Finance, the top 3 students of Marketing, and so on and so forth.

Exercise 6.10

There is a huge table – millions of rows -- that you need to duplicate to another table. Of course you may do **insert into table2 select * from table1**, but this command can take a long time and slow down the database. Write a PL/SQL program to copy all records from table1 to table2, and **commit** every one thousand records.

Exercise 6.11

Use the **SCHEDULES** table created in Homework 4. Assume the course CallNum1 meets once per week (and thus corresponds to only one record in **SCHEDULES**), and the course CallNum2 meets multiple times per week (and thus corresponds to many records in **SCHEDULES**). Write a function **TimeOK** (p_CallNum1, p_CallNum2) that returns TRUE if there is no time conflict between the two courses, returns FLASE if there is a time conflict.

Exercise 6.12

Same as Exercise 5.11 but this time assume both courses meet multiple times per week. Rewrite **TimeOK** (p_CallNum1, p_CallNum2) that returns TRUE if there is no time conflict between the two courses, returns FLASE if there is a time conflict.

Exercise 6.13

Continued from Exercise 5.11. Each student is enrolled in multiple courses and this student wants to add a new course. If this new course has time conflict with ANY of the courses that he is currently enrolled, he cannot add

this new course. Write a procedure **Validate_Time (p_snum, p_CallNum)** that returns TRUE is this new course has no time conflict with any of his current enrollment, and FALSE otherwise.

Exercise 6.14

The OrderDetails table consists of OrderNumber, LineNumber, ProductCode, Quantity, Warehouse, and Status. Status 'V' indicates Valid and 'X' indicates Cancelled. Write a procedure **WarehouseSplit**(p_OrderNumber) that does the following: Create a new order for details of the same warehouse and re-number the Line Number. Detail status becomes 'X' after it is "moved" to the new order, like the following:

OrderDetails (Before)

OrdNum	Line	Pcode	Qty	Warehouse	Status
…	…	…	…	…	…
1001	1	P1	10	SJ	V
1001	2	P2	80	LA	V
1001	3	P3	70	LA	V
1001	4	P4	30	SD	V
1001	5	P5	40	SD	V
1001	6	P6	20	SJ	V
1001	7	P7	50	SJ	V
1001	8	P8	10	LA	V
…	…	…	…	…	…

OrderDetails (After)

OrdNum	Line	Pcode	Qty	Warehouse	Status
…	…	…	…	…	…
1001	1	P1	10	SJ	X
1001	2	P2	80	LA	X
1001	3	P3	70	LA	X
1001	4	P4	30	SD	X
1001	5	P5	40	SD	X
1001	6	P6	20	SJ	X
1001	7	P7	50	SJ	X
1001	8	P8	10	LA	X
…	……	…	…	…	…
1010	1	P2	80	LA	V
1010	2	P3	70	LA	V
1010	3	P8	10	LA	V
1011	1	P4	30	SD	V
1011	2	P5	40	SD	V
1012	1	P1	10	SJ	V
1012	2	P6	20	SJ	V
1012	3	P7	50	SJ	V
…	…	…	…	…	…

Exercise 6.15

Continued from Exercise 5.14. The management wants to "save" the original order number. Please modify **WarehouseSplit** (p_OrderNumber) to accomplish that.

OrderDetail (Before)

OrdNum	Line	Pcode	Qty	Warehouse	Status
…	…	…	…	…	…
1001	1	P1	10	SJ	V
1001	2	P2	80	LA	V
1001	3	P3	70	LA	V
1001	4	P4	30	SD	V
1001	5	P5	40	SD	V
1001	6	P6	20	SJ	V
1001	7	P7	50	SJ	V
1001	8	P8	10	LA	V
…	…	…	…	…	…

OrderDetail (After)

OrdNum	Line	Pcode	Qty	Warehouse	Status
…	…	…	…	…	…
1001	1	P1	10	SJ	X
1001	2	P2	80	LA	V
1001	3	P3	70	LA	V
1001	4	P4	30	SD	X
1001	5	P5	40	SD	X
1001	6	P6	20	SJ	X
1001	7	P7	50	SJ	X
1001	8	P8	10	LA	V
…	……	…	…	…	…
1010	1	P4	30	SD	V
1010	2	P5	40	SD	V
1011	1	P1	10	SJ	V
1011	2	P6	20	SJ	V
1011	3	P7	50	SJ	V
…	…	…	…	…	…

Exercise 6.16

Continued from Exercise 5.14. Instead of processing one order, write a procedure **WarehouseSplit** that processes all records in the table. (that is, split by warehouse for EACH order).

Exercise 6.17

Some of our customers do not like orders with more than 1000 line details. Write a procedure **LargeOrderSplit**(p_OrderNumber) that splits an order into orders with equal or less than 1000 detail lines. That is, if the order has 3500 details, this program will maintain the original 1000 details, and create 3 more orders with 1000, 1000, and 500 details respectively. Line Number and status needs to be updated as well.

Exercise 6.18

Sunshine apartment is a 8-unit apartment building. The owner raises rents occasionally. The owner has kept a log of tenants' rent like the following:

Unit	StartDate	Rent
1	1/1/2015	$1,100
1	3/1/2017	$1,200
1	10/1/2017	$1,250
2	2/15/2013	$950
2	4/1/2016	$1,050
3	1/1/1993	$350
3	8/1/2003	$1,350
3	5/1/2015	$1,100
4	3/1/2017	$1,200
...		

IRS is requesting a report of rental income as of 1/1/2016.

1) Find the rent of Unit 1 on the date of 1/1/2016.
2) What is the total rental income of all 8 units as of 1/1/2016.

Homework 8. Cursor

Code of Exercise 6.14

```
Create or Replace Procedure WarehouseSplit (
    p_OrdNum OrderDetail.OrdNum%type) is

v_Previous_Warehouse   OrderDetail.Warehouse%type;
v_Current_OrdNum               OrderDetail.ORdNum%type;
v_count                number(8);
vr_New                 OrderDetail%RowType;
vr_OrderDetail         OrderDetail%RowType;

CURSOR Cur_OrderDetail IS
    SELECT * FROM OrderDetail
   WHERE OrdNum = p_OrdNum
    Order by Warehouse;

BEGIN
    Open Cur_OrderDetail;
    v_Previous_Warehouse := 'XX';
    LOOP
        Fetch Cur_OrderDetail into vr_OrderDetail;
      Exit when Cur_OrderDetail%NotFound;

        /*
        || IF a different warehouse code, then set a new Order Number and
        || Reset Line number to 1.
        */
        IF vr_OrderDetail.Warehouse != v_Previous_Warehouse THEN
            v_count := 1;
            vr_New := vr_OrderDetail;
            Select OrdNumSeq.NextVal into vr_New.OrdNum from Dual;
            vr_New.Line := v_count;
            v_Current_OrdNum := vr_New.OrdNum;
            v_Previous_Warehouse := vr_New.Warehouse;
        /*
        || If same warehouse code, then use the current order number and
    || increment Line number.
        */
        ELSE
            v_count := v_count + 1;
            vr_New := vr_OrderDetail;
            vr_New.OrdNum := v_Current_OrdNum;
            vr_New.Line := v_count;
        END IF;

        --
            ADD_OrderDetail(vr_New);
        --

        -- Update the status of the original order detail to 'X'
        Update OrderDetail
            Set Status = 'X'
         Where OrdNum = vr_OrderDetail.OrdNum
            and Line = vr_OrderDetail.Line;

    END LOOP;
    Commit;
    Close Cur_OrderDetail;

Exception
    When others then
        dbms_output.put_line (SqlErrM);
End;
/
```

```
create or replace procedure ADD_OrderDetail (
    vr_OrderDetail OrderDetail%RowType) IS

BEGIN
    Insert into OrderDetail (
        OrdNum,                        Line,                              ProdCode,
        Qty,                           Warehouse,                         Status)
    Values (
        vr_OrderDetail.OrdNum, vr_OrderDetail.Line, vr_OrderDetail.ProdCode,
        vr_OrderDetail.Qty, vr_OrderDetail.Warehouse, vr_OrderDetail.Status);
Exception
    When Others then
        raise;
End;
/
```

```
SQL> select * from orderdetail;

   ORDNUM      LINE PRODC       QTY WAREH S
--------- --------- ----- --------- ----- -
     1001         1 P1           10 SJ    V
     1001         2 P2           80 LA    V
     1001         3 P3           70 LA    V
     1001         4 P4           30 SD    V
     1001         5 P5           40 SD    V
     1001         6 P6           20 SJ    V
     1001         7 P7           50 SJ    V
     1001         8 P8           10 LA    V

8 rows selected.
```

```
SQL> exec WarehouseSplit(1001);

PL/SQL procedure successfully completed.

SQL> select * from orderdetail;

   ORDNUM      LINE PRODC       QTY WAREH S
--------- --------- ----- --------- ----- -
     1001         1 P1           10 SJ    X
     1001         2 P2           80 LA    X
     1001         3 P3           70 LA    X
     1001         4 P4           30 SD    X
     1001         5 P5           40 SD    X
     1001         6 P6           20 SJ    X
     1001         7 P7           50 SJ    X
     1001         8 P8           10 LA    X
     1030         1 P2           80 LA    V
     1030         2 P3           70 LA    V
     1030         3 P8           10 LA    V
     1031         1 P4           30 SD    V
     1031         2 P5           40 SD    V
     1032         1 P1           10 SJ    V
     1032         2 P6           20 SJ    V
     1032         3 P7           50 SJ    V

16 rows selected.

SQL> spool off
```

Topic 7. The Exception Section

The Exception Section is an error handling section.

Benefits of using Exception Section:

1. Handle Error situations in one place.
2. Handle un-expected errors.
3. Prevent the program from totally stop.

Flow of Control: Once the control goes to the Exception section, it does not return back to the Execution Section.

Syntax of Exception

```
Begin
    ....
Exception
    WHEN <Error1> THEN
        do thing 1;
        do thing 2;
        ...
    WHEN <Error2> THEN
        do thing a;
        do thing b;
        ...
    WHEN...
        ...
End;
```

Two types of Oracle exceptions: Predefined and User-defined.

Pre-defined Exceptions

There are several exceptions defined by Oracle. They all have an exception name that you can use.

DUP_VAL_ON_INDEX	When unique constraints are violated
NO_DATA_FOUND	No data found during a select...into phrase
TOO_MANY_ROWS	When a select...into returns more than one row
INVALID_NUMBER	When a conversion to a number field failed
ZERO_DIVIDE	Division by zero

Example:
Give this table, what would the following program do?

Students

SNUM	SNAME	STANDING	Major	GPA	Major_GPA
101	Andy	4	IS	2.8	3.2
102	Betty	2		3.2	
103	Cindy	3	IS	2.5	3.5
				…	…

Enrollments

SNUM	CallNum	GRADE
101	10110	F
102	10110	A
103	10120	A
101	10125	
102	10130	

```
/* Program 38  Pre-defined Exceptions */

declare
    vMajor number;
begin
    insert into Students (snum) values ('101');
    select snum into vSnum from Students where snum='999';
    select snum into vSnum from Enrollments where snum='101';
    select Major into vMajor from Students where snum='101';

exception
    when dup_val_on_index then
        dbms_output.put_line ('ERROR1: Insert failed -- duplicate student record
exists! ');
    when no_data_found then
        dbms_output.put_line ('ERROR2: This student does not exist!');
    when too_many_rows then
        dbms_output.put_line ('ERROR3: Single value record expected when returning
multiple records.');
    when invalid_number then
        dbms_output.put_line ('ERROR4: Mismatched data types.');
    when others then
        dbms_output.put_line ('ERROR: I don''t know what it is but there is something
wrong about your program! ');

end;
/
```

```
SQL> start a:38.sql
ERROR1: Insert failed -- duplicate student record exists!

PL/SQL procedure successfully completed.
```

User-Defined Exceptions

You can declare your own exceptions in PL/SQL.

```
declare
    eErrorSituation1 exception;
    ....
begin
    IF ....
    THEN raise eErrorSituation1;
    ...
exception
    when eErrorSituation1 then
         ...;
    when eErrorSituation2 then
         ...;
    when eErrorSituation3 or eErrorSituation4 then
         ... ;
end;
/
```

```
/* Program 39  User-Defined Exceptions */

declare
    eNoEnrollRecord exception;
begin
    update e set grade='A' where snum='222' and cnum='240';
    IF sql%notfound THEN
        raise eNoEnrollRecord;
    END IF;
exception
    when eNoEnrollRecord then
        dbms_output.put_line ('There is no such enrollment record.');
end;
/
```

Behavior of Exceptions

1. An "un-handled" error results in the program exits unsuccessfully.

```
declare
    x number;
begin
    x:= to_number('A');
end;
/
```

```
SQL> start a:Test
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error
ORA-06512: at line 4
```

2. A "handled" error results in successful execution of the program

```
declare
    x number;
begin
    x:= to_number('A');
exception
    when others then
        null;
end;
/
```

```
SQL> start a:Test

PL/SQL procedure successfully completed.
```

3. Use of SqlErrM

```
declare
    x number;
begin
    x:= to_number('A');
exception
    when others then
        dbms_output.put_line (SqlErrM);
end;
/
```

```
SQL> start a:test
ORA-06502: PL/SQL: numeric or value error: character to number conversion error

PL/SQL procedure successfully completed.
```

4. Exceptions in Nested Blocks

```
DECLARE
    x number
BEGIN
    -- Blah blah blah
    Begin
        x:= to_number('A');
    Exception
        when others then
            dbms_output.put_line ('Exception Inner Block');
    End;
EXCEPTION
    When Others then
        dbms_output.put_line ('Exception Outer Block');
END;
/
```

```
SQL> start a:test
Exception Inner Block

PL/SQL procedure successfully completed.
```

```
DECLARE
    x number
BEGIN
    -- blah blah blah
    begin
        x:= to_number('A');
    end;
EXCEPTION
    When Others then
        dbms_output.put_line ('Exception Outer Block');
END;
/
```

```
SQL> start a:test
Exception Outer Block

PL/SQL procedure successfully completed.
```

5. Compare the following three scenarios:

```
BEGIN
    open cursor;
    Loop
        fetch next record;
        exit when cursor%notfound;
        -- Errors may happen in any of the three processes
        process this;
        process that;
        process more;
    End Loop;
EXCEPTION
    When Others then
        do something;
End;
/
```

```
BEGIN
    Open Cursor;
    Loop
        Fetch Next Record;
        Exit when cursor%notfound;
        begin
            -- Errors may happen in any of the three processes
            process this;
            process that;
            process more;
        exception
            when others then
                do something;
        end;
    End Loop;
    Close Cursor;
END;
/
```

```
BEGIN
    Open Cursor;
    Loop
    Fetch Next Record;
    Exit when cursor%notfound;
        begin
            -- Errors may happen in any of the three processes
            process this;
            process that;
            process more;
            commit;
        exception
            when others then
                do something;
                rollback;
        end;
    end Loop;
    Close Cursor;
END;
/
```

6. Error Handling Behavior involves Procedure/Function Calls

```
create or replace procedure Proc1 is
        x number;
Begin
        x:= to_number('A');
End;
/
```

```
create or replace procedure Main is
Begin
    proc1;
    dbms_output.put_line ('Process this');
    dbms_output.put_line ('Process that');
Exception
    When Others Then
        dbms_output.put_line ('Exception in Main');
End;
/
```

```
SQL> exec main
Exception in Main

PL/SQL procedure successfully completed.
```

Homework 9. Exception

Due Date : _____

Refine the enroll procedure so that if the student number or call number is invalid, the program prints an error message and exits. This homework requires you to use Exception section in two places: the main Enroll procedure and at least one place in one of the sub-program.

Note – if you don't want to mess with your Enroll procedure, you can create a small MAIN and SUB programs for this homework. Just put in some code with error to trigger the exception.

Run two scenarios: (1) Some error happens so the Exception in the SUB program fires, but not the Main program. (2) Some error happens so the exception in the MAIN fires.

You can move your errors around to trigger the exception you want.

Turn in a print out of the execution of both scenarios.

Topic 8. Triggers

- Trigger: A PL/SQL program that will be "fired" (executed) when a DML (**insert**, **update**, or **delete**) is performed on a table.

- A trigger is "attached" to a table; just like a foreign key is "attached" to a table.

```
CREATE OR REPLACE TRIGGER BeforeDeleteStudent
    before delete on STUDENTS
begin

    dbms_output.put_line ('A Student Record is about to be deleted! ');
end;
/
```

where **BeforeDeleteStudent** is the name of the trigger; `STUDENT` is the table that the trigger is "attached" to.

After this program is executed, the trigger is saved in your database. In the future, whenever there is a deletion performed on the **STUDENTS** table, the trigger would "fire."

There are 12 types of triggers:

before **after**	×	**insert** **update** **delete**	×	**for each statement** **for each row**	*← this is the default* *← can use :old :new*

for each statement: the trigger is fired for each statement. The **:old**, **:new** option does NOT work in **for each statement** triggers. <u>This is the default</u> if you omit the **for each ...** clause.

for each row: the trigger is fired each time <u>a row</u>(record) is being inserted/modified/deleted.

```
:old.column
:new.column
```

`insert`: The value to be inserted is denoted by **`:new`**, such as

`IF :new.gpa < 1.5 then...`

`:old` is not defined.

`update`: The value to be replaced is denoted by **`:old`**, and the new value is denoted by **`:new`**.

`delete`: The value to be deleted is denoted as **`:old`**. **`:new`** is not defined.

```
/* Program 22. A Sample Trigger */

create or replace trigger trig_AUStudentBalance
    after update on students
    for each row
begin
    IF :new.balance - :old.balance >= 50 THEN
        insert into StudentsAudit (snum, balance) values (:old.snum, :old.balance);
    END IF;
end;
/
```

After Program 22 is run and saved in my database, any update to the student table will cause the trigger to fire.

```
SQL> select * from students;

SNUM      SNAME           MAJOR      S   BALANCE
--------- --------------- ---------- - ---------
101       Andy            MKT        5       100
102       Betty           MKT        5       100
103       Cindy           MKT        4       100

SQL> select * from StudentsAudit;

no rows selected

SQL> update students set balance=160 where snum='101';

1 row updated.

SQL> select * from students;

SNUM      SNAME           MAJOR      S   BALANCE
--------- --------------- ---------- - ---------
101       Andy            MKT        5       160
102       Betty           MKT        5       100
103       Cindy           MKT        4       100

SQL> select * from StudentAudit;

SNUM      SNAME           MAJOR      S   BALANCE
--------- --------------- ---------- ----------
101                                          100
```

An Example on different triggers

```
SQL> -- I created 4 triggers on STUDNETS table:
SQL> -- For Each ROW: before insert, after insert
SQL> -- For Each STATEMENT: before insert, after insert
SQL>
SQL> create or replace trigger trig_BIStudents
  2  BEFORE Insert on Students
  3  for each row
  4  begin
  5     dbms_output.put_line (:new.snum||' is about to be added');
  6  end;
  7  /

Trigger created.

SQL>
SQL> create or replace trigger trig_AIStudents
  2  AFTER Insert on Students
  3  for each row
  4  begin
  5     dbms_output.put_line (:new.snum||' has been added');
  6  end;
  7  /

Trigger created.

SQL>
SQL> create or replace trigger trig_BIStudents_stmt
  2  BEFORE Insert on Students
  3  begin
  4     dbms_output.put_line ('BEFORE insert statement: insert statement about to begin');
  5  end;
  6  /

Trigger created.
```

```
SQL>
SQL> create or replace trigger trig_AIStudents_stmt
  2  AFTER Insert on Students
  3  begin
  4     dbms_output.put_line ('AFTER insert statement: insert statement completed');
  5  end;
  6  /

Trigger created.

SQL>
SQL> ----------------------------
SQL> -- SNUM is the primary key, which I will use as a way to fail insert
SQL> desc students;
 Name                                                  Null?    Type
 ----------------------------------------------------- -------- --------------------------------
 SNUM                                                  NOT NULL VARCHAR2(3)
 SNAME                                                          VARCHAR2(10)
 STANDING                                                       NUMBER(1)
 MAJOR                                                          VARCHAR2(3)
 GPA                                                            NUMBER(2,1)
 MAJOR_GPA                                                      NUMBER(2,1)

SQL> select * from students;

SNU SNAME        STANDING MAJ        GPA  MAJOR_GPA
--- ---------- ---------- --- ---------- ----------
101 Andy                3 IS         2.8        3.2
102 Betty               2            3.2
103 Cindy               3 IS         2.5        3.5
104 David               2 FIN        3.3          3
105 Ellen               1            2.8
106 Frank               3 MKT        3.1        2.9

6 rows selected.

SQL> --
SQL> -- Insert ONE record
SQL> -- separate insert statements, some work some don't
SQL> insert into students (snum, sname) values ('999','whoever');
BEFORE insert statement: insert statement about to begin
999 is about to be added
999 has been added
AFTER insert statement: insert statement completed

1 row created.

SQL> insert into students (snum, sname) values ('101','repeat');
BEFORE insert statement: insert statement about to begin
101 is about to be added
insert into students (snum, sname) values ('101','repeat')
*
ERROR at line 1:
ORA-00001: unique constraint (SOPHIE.SYS_C0015970) violated


SQL> insert into students (snum, sname) values ('888','anyhow');
BEFORE insert statement: insert statement about to begin
888 is about to be added
888 has been added
AFTER insert statement: insert statement completed

1 row created.

SQL>
SQL> -- 999 and 888 are in
SQL> select * from students;

SNU SNAME        STANDING MAJ        GPA  MAJOR_GPA
--- ---------- ---------- --- ---------- ----------
101 Andy                3 IS         2.8        3.2
102 Betty               2            3.2
103 Cindy               3 IS         2.5        3.5
104 David               2 FIN        3.3          3
105 Ellen               1            2.8
106 Frank               3 MKT        3.1        2.9
999 whoever
888 anyhow

8 rows selected.
```

```
SQL> --
SQL> -- Insert MULTIPLE records in one insert statement
SQL> -- Created a test table STUDNETS2
SQL> select * from students2;

SNU SNAME        STANDING MAJ        GPA  MAJOR_GPA
--- ---------- ---------- --- ---------- ----------
107 New guy
108 New gal
101 repeat


SQL> -- INSERT Students2 table to Students table as an insert "statement"
SQL> -- 107, 108 work, 101 fails, the statement fails

SQL> insert into students select * from students2;

BEFORE insert statement: insert statement about to begin
107 is about to be added
107 has been added
108 is about to be added
108 has been added
101 is about to be added
insert into students select * from students2
*
ERROR at line 1:
ORA-00001: unique constraint (SOPHIE.SYS_C0015970) violated


SQL> -- Insert "statement" fails, nobody is inserted
SQL> select * from students;

SNU SNAME        STANDING MAJ        GPA  MAJOR_GPA
--- ---------- ---------- --- ---------- ----------
101 Andy                3 IS         2.8        3.2
102 Betty               2            3.2
103 Cindy               3 IS         2.5        3.5
104 David               2 FIN        3.3          3
105 Ellen               1            2.8
106 Frank               3 MKT        3.1        2.9
999 whoever
888 anyhow

8 rows selected.
```

Notes of using Trigger:

- Trigger is like a gate keeper to the table.
- Programming tasks can be centralized in triggers. For instance, if it is required that a deleted record be copied to an audit table, and there are many places in your system that a deletion may happen. Instead of writing the copying code each time after each delete statement, you can put a `after delete` trigger on the table, and thus centralize your programming tasks.
- Triggers are sometimes overlooked by developers! Most people concentrate on reading package/procedure code, but forget to examine triggers that are fired quietly.

Exercise 7.1

An IS 380 students broke into the Brodman Hall and updated all IS majors' GPA to 4. Write a database trigger to capture the log in name, system time, Student Number, before_GPA, and after_GPA of all records that were affected, and save the information in a Student_Audit table. Note: You need to create the **Student_Audit** table first, with all necessary columns.

Exercise 7.2

Write a trigger **`AfterUpdateGPA`** to print a warning message if a student's GPA is being updated, and the new GPA is lower than the previous GPA by 1 point or more.

Exercise 7.3

Assume you have a table Course_Audit, which the following columns: Course Number, Course Title, Credit Hour, Time, and Operator. Write an AfterInsertCourse trigger that will fire after a new record is inserted to the Course

table. The trigger copies the new record and store it to the Course_Audit table. The time of insert and the operator of insert is also recorded.

Exercise 7.4

This is an Transactions table. AmountDue should always equal to UnitPrice x Qty.

Tr#	PNum	UnitPrice	Qty	AmountDue
1001	P1	$10	10	$100
1002	P3	$5	10	$50
1003	P2	$2	5	$10
1004	P3	$8	10	$80
1005	P1	$10	8	$80
1006	P2	$10	10	$100
...	...	...	...	...

Note that an IS 380 student can easily log in to the system and write an UPDATE statement to update either UnitPrice, Qty, or AmountDue, then the data become inconsistent.

Write a database trigger to prevent this from happening. If UnitPrice or Qty is changed, your trigger should update AmountDue accordingly automatically. If any of them is NULL, update it to 0 and use 0 for your calculation. If the user tries to update AmountDue, the value should be unchanged, like the following:

```
SQL> select * from transactions;

      TNUM PNUM        UNITPRICE        QTY  AMOUNTDUE
---------- ---------- ---------- ---------- ----------
      1001 p1                  3         10         30
      1002 p2                  3          5         15

SQL> update transactions set unitprice=10;

2 rows updated.

SQL> select * from transactions;

      TNUM PNUM        UNITPRICE        QTY  AMOUNTDUE
---------- ---------- ---------- ---------- ----------
      1001 p1                 10         10        100
      1002 p2                 10          5         50

SQL> update transactions set amountdue=0;

2 rows updated.

SQL> select * from transactions;

      TNUM PNUM        UNITPRICE        QTY  AMOUNTDUE
---------- ---------- ---------- ---------- ----------
      1001 p1                 10         10        100
      1002 p2                 10          5         50

SQL> spool off
```

Exercise 7.5

When a letter grade is assigned, the student's GPA should be updated right away. Write a trigger to update a student's GPA when a new grade is assigned.

Homework 10. Triggers

Due Date: _____

Topic 10. UTL_FILE

utl_file is an Oracle built-in package, just like dbms_output. **utl_file** enables you to read and write to operating systems files (such as on your c: drive and a: drive). You can interact with operating files both on client and on the server.

In order to have **utl_file** work, you have to set up your computer the following way:

(1) Find your init.ora file and open it with Notepad. It should be under your Oracle directory\Admin\orcl\pfile\...

(2) Add this line at the end of the file:

If you want to be able to read and write from all directories, add this line:

```
utl_file_dir = *
```

If you only want to be able to read and write to a particular drive/subdirectory, add lines such as:

```
utl_file_dir = a:
```

for your a: drive.

(3) Shutdown Oracle. Restart Oracle (so the init.ora takes effect).

Note: Later Oracle does not require changes to init.ora. Please see later part of the lecture notes.

To write to a file, use the **utl_file.put_line** command:

```
declare
    v_file_handle utl_file.file_type;
begin
    v_file_handle := utl_file.fopen('a:','test.txt','w');
    utl_file.put_line (v_file_handle, 'Hi');
    utl_file.fclose (v_file_handle);
end;
/
```

To read from a file, use the **utl_file.get_line** command:

```
declare
    v_file_handle utl_file.file_type;
    v_input varchar2(100);
begin
    v_file_handle := utl_file.fopen('a:','test.txt','r');
    utl_file.get_line (v_file_handle, v_input);
    insert into MyTable Values (v_input....);
    utl_file.fclose (v_file_handle);
end;
/
```

To read/write multiple lines, use a loop:

```
declare
    v_file_handle utl_file.file_type;
    v_text char(2000);
begin
    v_file_handle := utl_file.fopen('a:','test1.txt','r');
    Loop
        begin
            utl_file.get_line (v_file_handle, v_text);
            dbms_output.put_line(ltrim(rtrim(v_text)));
        exception
```

```
            when no_data_Found then
                exit;
        end;
    end loop;

    utl_file.fclose(v_file_handle);

end;
/
```

Exercise 10.1

(a) Use **utl_file** to write '`Hello World!`' to a file named outfile1.txt on your A: Drive.
(b) Use Notepad to create a file infile1.txt on your A: Drive. This file contains one line

```
Hello World to you, too!
```

Use **utl_file** to read this line into your program, and print it on your screen.

Exercise 10.2

Assume you have the following table:

SNUM	SNAME	STANDING	GPA
111	Andy	4	2.5
222	Betty	2	3.2
333	Cindy	3	3.5

Use **utl_file** to write to your outfile2.txt file on your A: drive so your outfile2.txt file contains the following text:

```
111  Andy     4     2.5
222  Betty     2     3.2
333  Cindy     3     3.5
```

Exercise 10.3

Use Notepad to create the following text file on your a:infile1.txt:

```
444  David     4     3.1
555  Ellen      1     4.0
```

Use **utl_file** to read these lines and insert to your Student table.

Exercise 10.4

EDI (Electronic Data Interchange) is a popular way to transmit large amount of data between companies who conduct business regularly. Both companies agree on the file format (mapping), and text files are transmitted between their servers. Each of the partners has backend programs to decode these lines and process them in their own database.
In order to ensure complete transmission of their files, each file has a Header line and a Trailer line. Between the Header and the Trailer are the sales/transactions data, such as the follows:

```
FILE2842520HEADER020410WALMART0005
V1190000
18237566384750019
18239984567360002
18240938475660098
19023847566320007
FILE2842520TRAILER020415WALMART0005
```

This is FILE number 2842520 from Walmart, contains 5 lines, and it was sent on 2002 April 10th. V1190000 is the store number, and the 13-digit code starting with 18 is the product's UPC number, and the last 4 digits is order quantity.

Suppose you have a **File_Transmissions** table like the following:

FileNumber	SentDate	ReceiveDate	Status

Write the file data into the **File_Transmissions** table. If the file (1) has both a Header and a Trailer, and (2) Header and Trailer's data matches, and (3) Number of records read matches the Header and Trailer, then update the status to be 'OK'. Otherwise, the status is 'ERR'.

Exercise 10.5

Suppose you have a **Transactions** table like the following:

TransNumber	TransDate	StoreNumber	Product	Quantity

Use **utl_file** to write file data into the **Transmissions** table.

Exercise 10.6

Populate the **Transactions** table with the following EDI data, where

```
FILE2842520HEADER020415WALMART0013
V1190000
18237566384750019
18239984567360002
18240938475660098
19023847566320007
V1190023
18237566444750010
18239984567360002
18230938475660031
V1190087
18237000084750008
18239984567360002
18240938475660006
FILE2842520TRAILER020415WALMART0013
```

Exercise 10.7

Populate the **Transactions** table with the following EDI data.

```
FILE2842520HEADER020415WALMART0007
V1190000
18237566384750019     18239984567360002     18240938475660098
19023847566320007
V1190023
18237566444750010     18239984567360002     18230938475660031
V1190087
18237000084750008     18239984567360002     18240938475660006
FILE2842520TRAILER020415WALMART0007
```

Use of DIRECTORY

Later Oracle uses **DIRECTORY** to handle all I/O directories. In the following sample code, I first authorize a directory in my local c:\temp folder. Then I use Oracle commands to "write to" a file in this directory. You can also use Oracle commands to "read from" this file. In this example, I read the data then simply print it. In your application, you can insert this data into your Oracle tables and your life will be beautiful ☺

```
SQL> -- drop directory
SQL> drop directory test_dir;

Directory dropped.

SQL>
SQL> -- create directory
SQL> create directory test_dir as 'c:\temp\';

Directory created.

SQL>
SQL> -- query directory
SQL> select * from all_directories;

OWNER      DIRECTORY_NAME       DIRECTORY_PATH
---------- -------------------- ------------------------------------------------------------
SYS        DM_PMML_DIR          D:\oracle\product\10.1.0\Db_1\dm\admin
SYS        LOG_FILE_DIR         D:\oracle\product\10.1.0\Db_1\demo\schema\log\
SYS        DATA_FILE_DIR        D:\oracle\product\10.1.0\Db_1\demo\schema\sales_history\
SYS        MEDIA_DIR            D:\oracle\product\10.1.0\Db_1\demo\schema\product_media\
SYS        TEST_DIR             c:\temp\

SQL>
SQL> -- query privileges
SQL> select * from all_tab_privs
  2  where table_name = 'TEST_DIR';

GRANTOR    GRANTEE    TABLE_SCHE TABLE_NAME PRIVILEGE  GRANTABLE  HIE
---------- ---------- ---------- ---------- ---------- ---------- ---
SYS        SOPHIE     SYS        TEST_DIR   READ       YES        NO
SYS        SOPHIE     SYS        TEST_DIR   WRITE      YES        NO

SQL>
SQL> -- write to a directory, use the put_line command
SQL> declare
  2      v_file_handle utl_file.file_type;
  3      v_input varchar2(100);
  4   begin
  5      v_file_handle := utl_file.fopen ('TEST_DIR','code1.txt','w');
  6      utl_file.put_line (v_file_handle, 'super testing');
  7      utl_file.fclose (v_file_handle);
  8   end;
  9   /

PL/SQL procedure successfully completed.

SQL> -- If you check your c:\temp folder now, you should see a new file called code1.txt. If you
open this file, it should contain one line that says super testing.
SQL>
SQL>
SQL> -- read from a directory, use the get_line command
SQL>  declare
  2      v_file_handle utl_file.file_type;
  3      v_input varchar2(100);
  4   begin
  5      v_file_handle := utl_file.fopen ('TEST_DIR','code1.txt','r');
  6      utl_file.get_line (v_file_handle, v_input);
  7      dbms_output.put_line ('**'||v_input);
  8      utl_file.fclose (v_file_handle);
  9   end;
 10   /
**super testing

PL/SQL procedure successfully completed.
```

Note: You can grant directory privilege to certain users:

```
GRANT create any directory to USER1;
GRANT drop any directory to USER2;
```