MONASH University

MONASH
INFORMATION
TECHNOLOGY

Week 10 – PL-SQL

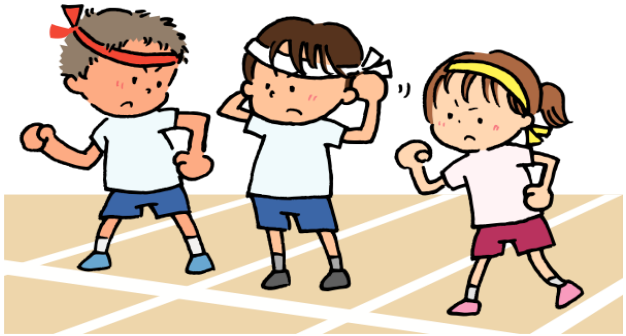
FIT3171 Databases
Semester 1 2022

Malaysia Campus

Preparation for the Forum - ready, set ……

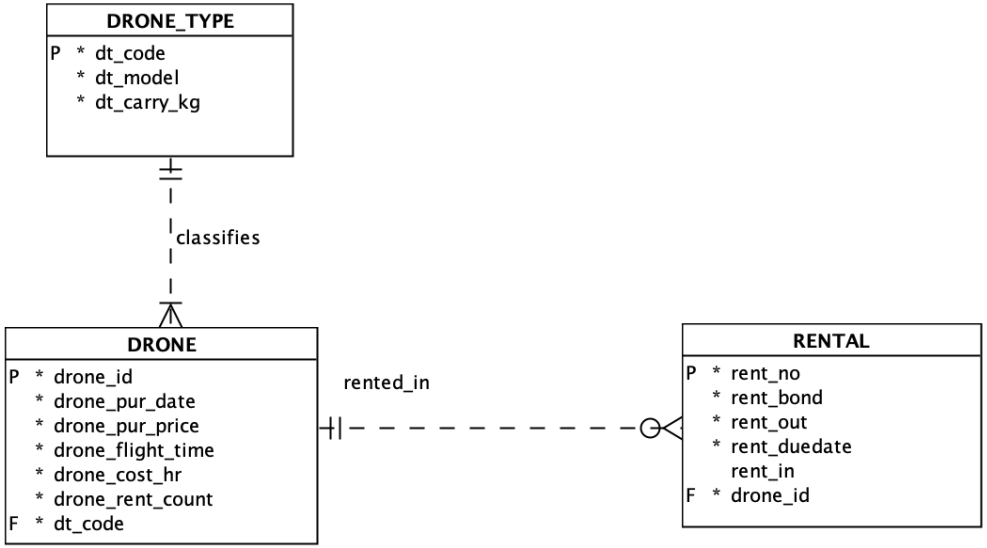
Please

- connect to Flux - flux.qa and be ready to answer questions
  - flux.qa/QBGYRS
- login to the Oracle database via **SQL Developer** (you will need to run the CISCO or Global VPN first if you are off campus)



PL/SQL - Triggers

Simplified Drone Model



Download and run drone_rental_schema.sql to create these three tables under your account

Oracle Triggers

- A trigger is PL/SQL code associated with a table, which performs an action when a row in a table is inserted, updated, or deleted.
- Triggers are used to implement some types of data integrity constraints that cannot be enforced at the DBMS design and implementation levels
- A trigger is a stored procedure/code block associated with a table
- Triggers specify a condition and an action to be taken whenever that condition occurs
- The DBMS automatically executes the trigger when the condition is met ("fires")
- A Trigger can be ENABLE'd or DISABLE'd via the ALTER command
  - ALTER TRIGGER *trigger_name* ENABLE;

Oracle Triggers - general form

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
        {BEFORE | AFTER | INSTEAD OF }
        {UPDATE | INSERT | DELETE}
            [OF <attribute_name>] ON <table_name>
        [FOR EACH ROW]
        [WHEN]
DECLARE
BEGIN
                    …. trigger body goes here …..
END;
```

ROW BASED TRIGGER

THIS IS THE CONDITION PART

DECLARE ATTRIBUTES

Triggering Statement

BEFORE|AFTER INSERT|UPDATE [of colname]|DELETE ON Table

- The triggering statement specifies:
  - the type of SQL statement that fires the trigger body.
  - the possible options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
  - the table associated with the trigger.
- Column List for UPDATE
  - if a triggering statement specifies UPDATE, *an optional list of columns can be included in the triggering statement*.
  - if you include a column list, the trigger is fired on an UPDATE statement only when one of the specified columns is updated.
  - if you omit a column list, the trigger is fired when any column of the associated table is updated

we can have more than one col

Trigger Body

```
BEGIN
    .....
END;
```

- is a PL/SQL block that can include SQL and PL/SQL statements. These statements are executed if the triggering statement is issued and the trigger restriction (if included) evaluates to TRUE.
- Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the **old** and **new** column values of the current row affected by the triggering statement.
- Two correlation names exist for every column of the table being modified: **one for the old column value** and **one for the new column value**.

Correlation Names

- Oracle uses two correlation names in conjunction with every column value of the current row being affected by the triggering statement. These are denoted by:

  OLD.ColumnName & NEW.ColumnName

  - For DELETE, only OLD.ColumnName is meaningful
  - For INSERT, only NEW.ColumnName is meaningful
  - For UPDATE, both are meaningful
- A colon must precede the OLD and NEW qualifiers when they are used in a trigger's body, but a colon is not allowed when using the qualifiers in the WHEN clause.
- Old and new values are available in both BEFORE and AFTER **row triggers**.

FOR EACH ROW Option

- The FOR EACH ROW option determines whether the trigger is a row trigger or a statement trigger. If you specify FOR EACH ROW, the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option means that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

```
CREATE OR REPLACE TRIGGER display_high_flighttime AFTER
    UPDATE OF drone_flight_time ON drone
    FOR EACH ROW
    WHEN ( new.drone_flight_time > 1000 )
BEGIN
    dbms_output.put_line('Drone:  ' || :new.drone_id
                || ' old flight time:  ' || :old.drone_flight_time
                || ' and new flight time:  ' || :new.drone_flight_time);
END;
```

Statement Level Trigger

- Executed once for the whole table but will have to check all rows in the table.
- In many cases, it will be inefficient.
- **No access to the correlation values :new and :old.**

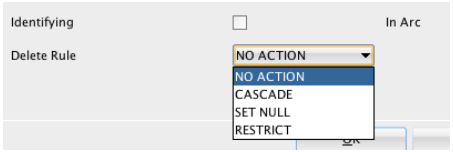
```
CREATE OR REPLACE TRIGGER check_rental_time BEFORE
    INSERT OR DELETE OR UPDATE ON rental
DECLARE
    current_time number;
BEGIN
    current_time := to_number(to_char(sysdate,'hh24'));
    if (current_time >=18 or current_time < 9) then
        raise_application_error(-20001,'The operation must be done within the
business hours (ie. 9AM-6PM)');
    end if;
END;
```

variable (attributes)

current time

Oracle Data FK Integrity

- Oracle offers the options:
  - –UPDATE
    - •no action (the default - not specified)
  - –DELETE
    - •no action (the default - not specified)
    - •cascade
    - •set null



- Subtle difference between "no action" and "restrict"
  - –RESTRICT - will not allow action if child records exist, checks first
  - –NO ACTION - allows action and any associated triggers, *then* checks integrity
- Databases implementations vary, for example:
  - –Oracle no RESTRICT
  - –IBM DB2, SQLite implement both as above

Common use of triggers

- In the Simplified Drone Model, DRONE_TYPE is the PARENT (PK end) and DRONE is the CHILD (FK end)
- What should the database do to maintain integrity if the user:
  - attempts to UPDATE the dt_code of the drone type (parent)
  - attempts to DELETE a dt_code which is referred in the drone table (child)
- Oracle, by default, takes the safe approach
  - UPDATE NO ACTION (no update of PK permitted if child records)
  - DELETE NO ACTION (no delete permitted if child records)
  - what if you as the developer want UPDATE CASCADE?

Update Cascade using Oracle Trigger

```
CREATE OR REPLACE TRIGGER dronetype_update_cascade
BEFORE UPDATE OF dt_code ON drone_type
FOR EACH ROW
BEGIN
   UPDATE drone
   SET          dt_code = :new.dt_code
   WHERE  dt_code = :old.dt_code;
   DBMS_OUTPUT.PUT_LINE ('Corresponding drone type code in the
   DRONE table has also been updated');
END;
```

Implement UPDATE CASCADE rule

DRONE_TYPE 1 ---- classifies --- M DRONE

:new.dt_code – value of dt_code after update

:old.dt_code – value of dt_code before update

Common use of triggers - data integrity

- A trigger can be used to enforce user-defined integrity by triggering on a preset condition, carrying out some kind of test and then if the test fails, the trigger can raise an error (and stop the action) via a call to **`raise_application_error`**

  The syntax for this call is:

```
raise_application_error(-20000, 'Error message to display');
```

  the -20000 is the error number which is reported to the user, the error message is the error message the user will see. The error number can be any number less than or equal to -20000.

Common use of triggers - data integrity - example

Create a trigger which will ensure any drone_type added (ie. inserted) to the DRONE_TYPE table has a drone type code which starts with 'D'.

```
CREATE OR REPLACE TRIGGER check_drone_type_code BEFORE
    INSERT ON drone_type
    FOR EACH ROW
BEGIN
    IF  :new.dt_code NOT LIKE 'D%' THEN
        raise_application_error(-20001, 'Drone type code must start with D');
    END IF;
END;
```

Mutating Table

- A table that is currently being modified through an INSERT, DELETE or UPDATE statement SHOULD NOT be read from or written to because it is in a transition state between two stable states (before and after) where data integrity can be guaranteed.
  - Such a table is called **mutating table**.

```
CREATE OR REPLACE TRIGGER dronetype_update_cascade BEFORE
    UPDATE OF dt_code ON drone_type
    FOR EACH ROW
DECLARE
    dt_code_number NUMBER;
BEGIN
    SELECT COUNT(*) INTO dt_code_number
    FROM drone_type
    WHERE dt_code = :new.dt_code;

    IF dt_code_number = 1 THEN
        UPDATE drone
        SET dt_code = :new.dt_code
        WHERE dt_code = :old.dt_code;
        dbms_output.put_line('Corresponding drone type code in the DRONE table has also been updated');
    END IF;
END;
```

```
update drone_type
set dt_code = 'INS2'
where dt_code = 'DIN2'
Error report -
ORA-04091: table DARAY1.DRONE_TYPE is mutating, trigger/function may not see it
ORA-06512: at "DARAY1.DRONETYPE_UPDATE_CASCADE", line 4
ORA-04088: error during execution of trigger 'DARAY1.DRONETYPE_UPDATE_CASCADE'
```

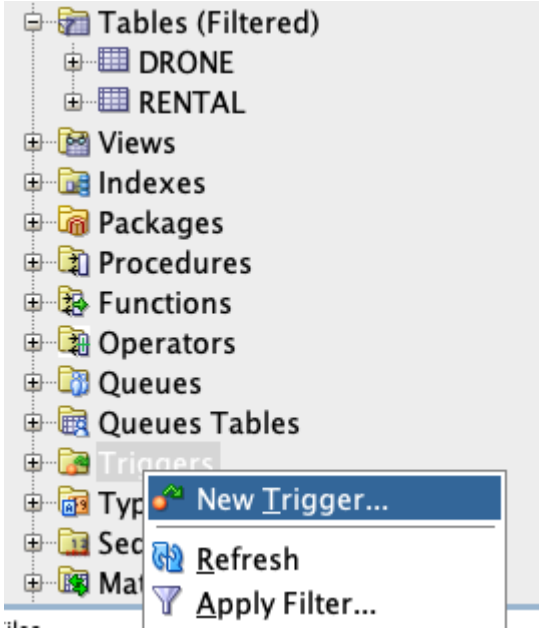
Q1. Create a trigger, which automatically maintains the drone_rent_count values, ie.:

- **insert a rental increases the count for drone_rent_count**
- **delete a rental decreases the count for drone_rent_count**

Write a test harness to demonstrate the successful operation of your trigger/stored procedure

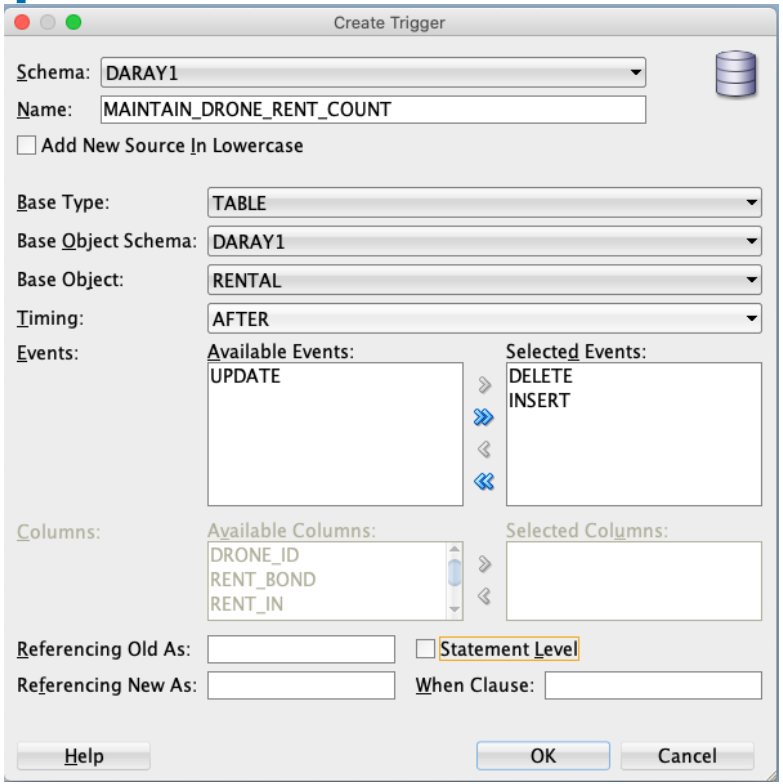
Create Trigger using GUI - step 1

Create a new trigger using the SQL Developer GUI to build the trigger framework:



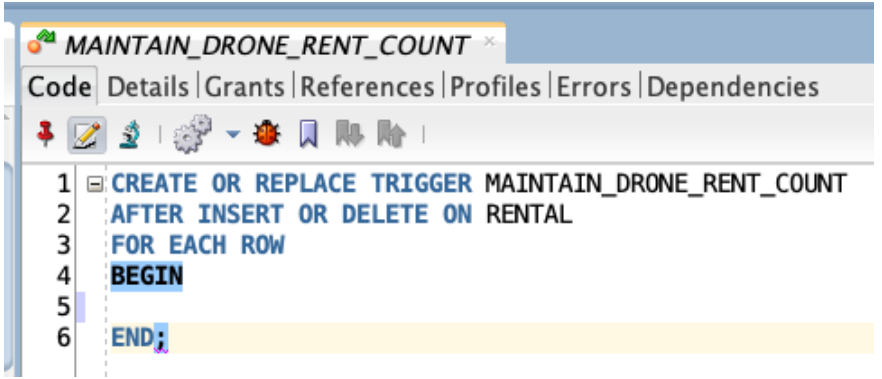
Create Trigger using GUI - step 2

Create a new trigger using the SQL Developer GUI to build the trigger framework:



Create Trigger using GUI - step 3

Select OK and you will be transferred to the PL/SQL Editor, where you can then enter the trigger BODY (the part between the begin and end). First remove the placeholder null;



Q1. What would be the logic to update the drone_rent_count attribute in the DRONE table when a new row is inserted to RENTAL?

A.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :old.drone_id;
```

B.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :new.drone_id;
```

C.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1;
```

D.
```
UPDATE drone
SET drone_rent_count = drone_rent_count - 1
WHERE drone_id = :old.drone_id;
```

Q2. What would be the logic to update the drone_rent_count attribute in the DRONE table when a row is deleted to RENTAL?

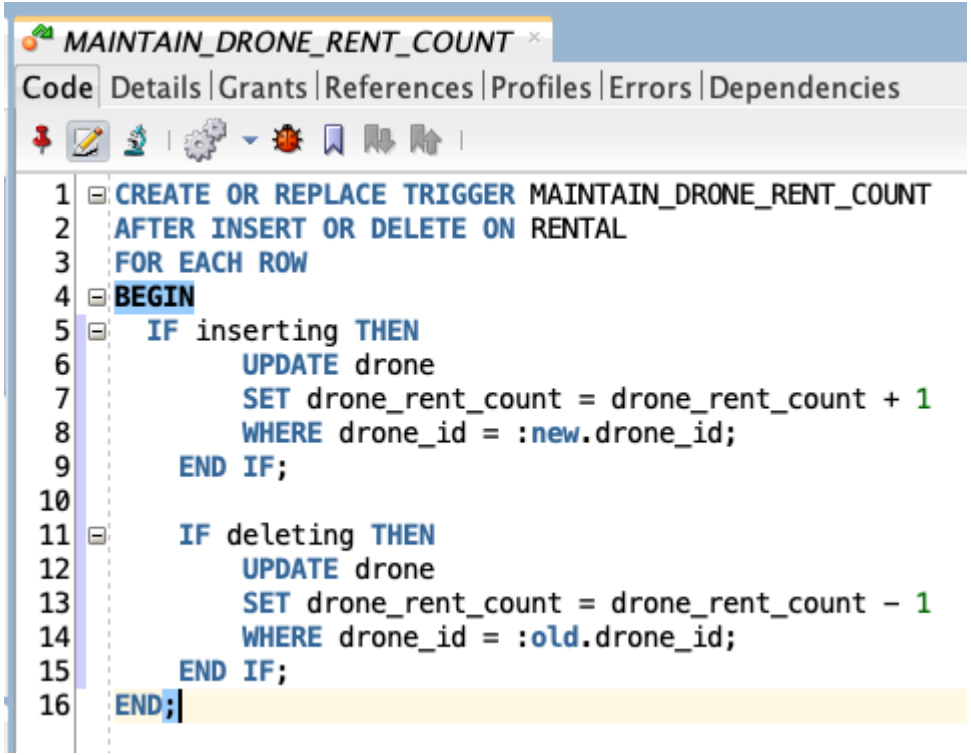
A.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :old.drone_id;
```

B.
```
UPDATE drone
SET drone_rent_count = drone_rent_count + 1
WHERE drone_id = :new.drone_id;
```

C.
```
UPDATE drone
SET drone_rent_count = drone_rent_count - 1;
```

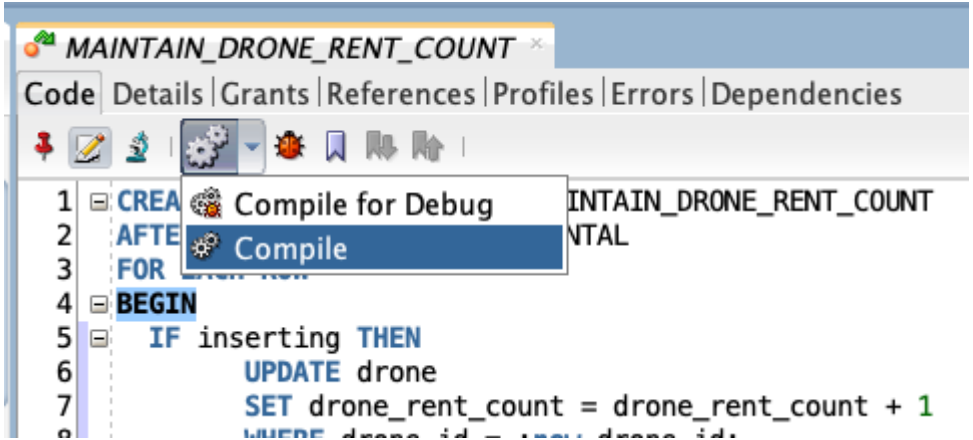
D.
```
UPDATE drone
SET drone_rent_count = drone_rent_count - 1
WHERE drone_id = :old.drone_id;
```

Create Trigger using GUI - step 4



Create Trigger using GUI - step 4

Finally compile the trigger, correcting any errors which appear:



Create Trigger using Script - Assignment 2

An alternative way of adding a trigger is via a script or entering the code directly into the SQL worksheet window.

Note when using this approach after the end of the trigger (line 16, you must include a ***/*** **on a line by itself in column 1** (line 17) **followed by a blank line** (line 18).

SQL Worksheet History

Worksheet Query Builder

```
1  CREATE OR REPLACE TRIGGER MAINTAIN_DRONE_RENT_COUNT
2  AFTER INSERT OR DELETE ON RENTAL
3  FOR EACH ROW
4  BEGIN
5    IF inserting THEN
6         UPDATE drone
7         SET drone_rent_count = drone_rent_count + 1
8         WHERE drone_id = :new.drone_id;
9     END IF;
10
11    IF deleting THEN
12        UPDATE drone
13        SET drone_rent_count = drone_rent_count - 1
14        WHERE drone_id = :old.drone_id;
15    END IF;
16 END;
17 /
18
```

Trigger Screen Output

- If you wish your trigger to provide output to the screen, you can include a message which will be written to the screen using DBMS_OUTPUT.PUT_LINE. You should note that **such messages are for your use as a developer**, in a production environment normal users will not see such messages. *We will also use these messages for you to demonstrate a trigger works for assessment purposes*.
- To obtain DBMS_OUTPUT in SQL Developer you need to run the command

```
SET SERVEROUTPUT ON
```

- A typical developer output message in our current example might be:

```
DBMS_OUTPUT.PUT_LINE('New rental has been added to drone ' || :new.drone_id);
```

  then when a new employee is inserted a message such as:

```
New rental has been added to drone 10
```

  would be output by DBMS Output.

Complete Trigger Command

SQL Worksheet History

Worksheet Query Builder

```
1  CREATE OR REPLACE TRIGGER MAINTAIN_DRONE_RENT_COUNT
2  AFTER INSERT OR DELETE ON RENTAL
3  FOR EACH ROW
4  BEGIN
5    IF inserting THEN
6          UPDATE drone
7          SET drone_rent_count = drone_rent_count + 1
8          WHERE drone_id = :new.drone_id;
9
10         DBMS_OUTPUT.PUT_LINE('New rental has been added for drone '
11         || :new.drone_id);
12
13     END IF;
14
15     IF deleting THEN
16         UPDATE drone
17         SET drone_rent_count = drone_rent_count - 1
18         WHERE drone_id = :old.drone_id;
19
20         DBMS_OUTPUT.PUT_LINE('A rental has been deleted for drone '
21         || :old.drone_id);
22
23     END IF;
24 END;
25 /
26
```

Testing Harness

- It is not sufficient to code a trigger only, a suitable test harness must be developed at the same time and used to ensure the trigger is working correctly.
- Now test the trigger that you have just created:
  - Insert a drone with a drone rent count of zero.
  - Insert an rental for that drone and observe what happens with the drone.drone_rent_count attribute.
  - Delete the rental data, again check the effect on the drone.drone_rent_count attribute.

Test the Trigger

```
--Testing Harness
--Initial data
insert into drone values (10,to_date('11/JAN/2020','dd/MON/yyyy'),1399,0,15,0);

--before value
select * from drone;
select * from rental;

--test for insert
insert into rental values (1, 100, to_date('30/JAN/2020','dd/MON/yyyy'), null, 10 );
--after value
select * from drone;
select * from rental;

--test for delete
delete from rental where rent_no = 1;
--after value
select * from drone;
select * from rental;

--closes transaction
rollback;
--End of Testing Harness
```

Oracle Triggers

- Use triggers where:
  - a specific operation is performed, to ensure related actions are also performed
  - to enforce integrity where data has been denormalised
  - to maintain an audit trail
  - global operations should be performed, regardless of who performs the operation
  - they do NOT duplicate the functionality built into the DBMS
  - their size is reasonably small (< 50 - 60 lines of code)
- Do not create triggers where:
  - they are recursive
  - they modify or retrieve information from triggering tables

PL/SQL - Stored Procedure

Oracle Stored Procedures

- A procedure is a named collection of procedural and SQL statements.
- Stored procedures is used to encapsulate and represent business logics
- SQL statements are encapsulated within a single block of code and executed as a single transaction
- Advantages:
  - –reduce network traffic and increase performance
  - –reduce code duplication
  - –reduce SQL injection risk in web based applications

Oracle Stored Procedures - general form

```
CREATE OR REPLACE PROCEDURE procedure_name [(argument [IN/OUT]  data-type, ... )]
        [IS/AS]
        [variable_namedata type[:=initial_value] ]
BEGIN
        PL/SQL or SQL statements;
        …
END;
```

- argument specifies the parameters that are passed to the stored procedure. A stored procedure could have zero or more arguments or parameters.
- IN/OUT indicates whether the parameter is for input, output, or both.
- data-type is one of the procedural SQL data types used in the RDBMS. The data types normally match those used in the RDBMS table creation statement.
- Variables can be declared between the keywords IS and BEGIN. You must specify the variable name, its data type, and (optionally) an initial value.

Stored procedure - no argument

Create a procedure to increase the drone cost per hr by 10%.

```
CREATE OR REPLACE PROCEDURE prc_drone_cost_increase10
AS BEGIN
    UPDATE drone
    SET drone_cost_hr = drone_cost_hr * 1.1;
    dbms_output.put_line('Drone cost per hour has been increased by 10%');
END;
/
--execute the procedure
exec prc_drone_cost_increase10();
```

Stored procedure - with input argument

Create a procedure to input a rental. Use a sequence to generate the rental number and the current system date/time to record the time the drone is taken out. The due date is set as 7 days after the drone is taken out and the bond is 10% of the drone purchase price.

```
CREATE SEQUENCE rental_seq start with 1;
CREATE OR REPLACE PROCEDURE prc_insert_rental ( p_drone_id IN NUMBER) AS
    p_drone_bond NUMBER;
BEGIN
    SELECT drone_pur_price * 10 / 100 INTO p_drone_bond
    FROM drone
    WHERE drone_id = p_drone_id;

    INSERT INTO rental VALUES (rental_seq.NEXTVAL,p_drone_bond,(SELECT sysdate FROM dual),
        (SELECT sysdate+7 FROM dual), NULL, p_drone_id);
    dbms_output.put_line('A new rental for drone ' || p_drone_id || ' has been inserted');
END;
/
exec prc_insert_rental(100);
```

What if the drone_id does not exist?

Stored procedure - with input and output arguments

```
CREATE OR REPLACE PROCEDURE prc_insert_rental_output (p_drone_id IN NUMBER,p_output OUT VARCHAR2) AS
    p_drone_found NUMBER;
    p_drone_bond NUMBER;
BEGIN
    SELECT count(*) into p_drone_found FROM drone WHERE drone_id = p_drone_id;

    IF (p_drone_found = 0) THEN
        p_output := 'Invalid drone id, rental cannot be added';
    ELSE
        SELECT drone_pur_price * 10 / 100 INTO p_drone_bond FROM drone WHERE drone_id = p_drone_id;

        INSERT INTO rental VALUES (rental_seq.NEXTVAL, p_drone_bond,(SELECT sysdate FROM dual),
            (SELECT sysdate+7 FROM dual), NULL, p_drone_id
        );
        p_output := 'A new rental for drone ' || p_drone_id || ' has been inserted';
    END IF;
END;
/
```

```
--execute the procedure
DECLARE
    output VARCHAR2(200);
BEGIN
    --call the procedure - success
    prc_insert_rental_output(100,output);
    dbms_output.put_line(output);
END;
/
DECLARE
    output VARCHAR2(200);
BEGIN
    --call the procedure - error
    prc_insert_rental_output(101,output);
    dbms_output.put_line(output);
END;
/
```

Q4. Create a procedure to handle the return of a rental. The procedure must take two input parameters: rental number, and the number hours the drone has been flown. The output from this procedure, returned by an OUT parameter, must be a status string indicating successful return of a rental or the issue that prevented the return.

Solution will be posted on Moodle Sunday 2 PM