References
- The textbook – *for concepts of DB programming*
- Sunderraman's book – for PL/SQL and other oracle features
- The PL/SQL manuel: http://cis.gvsu.edu/facilities/eos#oracle

Agenda

1. Basics
2. Substitution and bind variables
3. Embedded SQL and EXCEPTIONS
4. COMMIT and ROLLBACK
5. CURSORS
6. Procedures and functions
7. Stored procedures and functions
8. Triggers
9. Using SYSDATE in Oracle

1. BASICS

The Relationship among SQL*Plus, PL/SQL, and SQL

- SQL*Plus is an environment for running SQL and/or PL/SQL
- SQL*Plus enters PL/SQL mode when it encounters a `DECLARE` or `BEGIN` statements.

Data types

- Scalar types: all SQL types + `Boolean` + a few others
- Anchored types: e.g.
  - `amount  Employees.salary%TYPE;  /*a column type*/`
  - `oneEmployee  Employees%ROWTYPE;  /*a row type*/`
- Composite types
  - Record data type … similar to C `struct`; can be:
    - Table-based record
    - Cursor-based record
    - Programmer-defined record
  - Table `data` type … a table that consists of one column

Conditional statements

- `if/then`
- `if/then/else`
- `if/then/else/elsif` **<<< notice how `elsif` is spelled!!**

Exceptions

- When an exception is raised, control is transferred from the enclosing PL/SQL block to the exception handler.
- After handling an exception, control returns to the statement immediately after the block which raised the exception.
- Exception are of two types:
  - System-defined
  - User-defined

Loops

- `Loop` … needs an `EXIT` statements inside it
- `for loop`
- `while loop`

PL/SQL Blocks

- A PL/SQL Program is one or more (possibly nested) blocks.

- A typical block structure consists of three sections

```
DECLARE                                    /* optional */
    < variables, types, and local subprograms>
BEGIN                                   /* required */
    < Program logic + SQL>
EXCEPTION                                  /* optional */
    < Error handling statements>
```

To run a block, end it with a slash, /, on a new line.

- There are two types of blocks:

  - **Anonymous** block (any unnamed block)
    - It can be executed within the SQL*Plus environment just like a SQL statements.

  - **Subprogram** (a named **procedure** or **Function**)

2. SUBSTITUTION and BIND VARIABLE

Bind variables

- Offer a way to communicate between SQL*Plus and PL/SQL subprograms.
- Declared in SQL*Plus with the `VARIABLE` command
- Can be referenced in PL/SQL subprograms or SQL (using the '**:**' prefix).
- Can be printed, in SQL*Plus, using `PRINT`.
- Can only be modified in a PL/SQL block.

Substitution variables

- Just a place holder (to parameterize statements)
- Exercise: Create the sailors DB and run the following:
  - Notice how **&&c** `DEFINE`s a substitution variable while **&c** doesn't.

`SELECT &c FROM &t;`
`DEFINE`
	`SELECT &&c FROM &t1;`
	`DEFINE`
	`UNDEFINE c`

Example PL010

Demonstrates

- Some "scope" issues among SQL*Plus, SQL, and PL/SQL.
- Bind and substitution variables.
- Using the standard package DBMS_OUTPUT for output.

Input

- Run it for 3, 7, …

```
-- PL010
-- author: JRA
-- -------------------------------------------------------------
-- display result of procedure DBMS_OTPUT.PUT_LINE
SET SERVEROUTPUT ON
-- Don't display the before/after caused by substitution variable, s
SET VERIFY OFF

-- declaring a bind variable, b, in SQL*plus
VARIABLE b NUMBER;

-- Entering a PL/SQL block - notice how we reference s and b.
BEGIN
   :b := &&s + 1;
   DBMS_OUTPUT.PUT_LINE('+++++ here it is .. '||:b||' and '|| &s);
END;
/
-- Now printing, in SQL*Plus, the bind variable b.
PRINT b;
UNDEFINE s   //just to clean up the environment
```

3. EMBEDDED SQL and EXCEPTIONS

Example: PL025

Demonstrates
- An embedded SQL query that returns one row only.
- Using more than one system-defined EXCEPTION.
- The use of **ACCEPT** to customize the prompt and DEFINE substitution variables.

Specs
- Prompt the user for the sid.
- Check for valid input.

Input… *(Note: Start with a fresh copy of the database.)*
- 58
- 27
- xyz ***… notice the difference in handling this case in isqlplus vs. sqlplus***

```
-- PL025
-- author: JRA
-- ---------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- ---------------------------------
ACCEPT sailorID NUMBER PROMPT 'Please enter a sailor ID: '
DECLARE
   p_sid   sailors.sid%TYPE;
   p_sname sailors.sname%TYPE;
   p_age   sailors.age%TYPE;
BEGIN
   SELECT  sid, sname, age
   INTO    p_sid, p_sname, p_age
   FROM    sailors
   WHERE   sid = &sailorID;
   DBMS_OUTPUT.PUT_LINE('+++++ '||p_sid||'  '||p_sname||p_age);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('++++ Error ... '||&sailorID
                            ||' is not a valid ID');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('+++++ '||SQLCODE||'...'||SQLERRM);
END;
/
UNDEFINE sailorID
```

Example PL030

Demonstrates

– Updating the database.
– Using a bind variable, savedRating, to communicate between a PL/SQL block and a non-embedded SQL query.

Specs

– Prompt the user for an sid, and an increment for his/her rating
– Increase the rating of that sailor.
– Using a SQL query, print the rows of all the sailors who have the new rating.

Input (*Note: Start with a fresh copy of the database for each run.)*

– 22 and 3
– 22 and 9
– 27, and 9

```
-- PL030
-- author: JRA
-- ----------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
----------------------------------
VARIABLE savedRating NUMBER;
ACCEPT sailorID NUMBER PROMPT 'Please enter a sailor ID: '
ACCEPT ratingInc NUMBER PROMPT 'Please enter an increment for the rating: '

BEGIN
-- Retrieve and save the original rating.
-- Notice the bind variable savedRating
   SELECT  rating
   INTO    :savedRating
   FROM    sailors
   WHERE   sid = &sailorID;

-- Increase the rating for our sailor
   UPDATE sailors SET rating = rating + &ratingInc
   WHERE   sid = &sailorID;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('++++ Error ... '||&sailorID||
                                        ' is not a valid ID');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('+++++ '||SQLCODE||'...'||SQLERRM);
END;
/
-- Notice the bind variable savedRating
SELECT * FROM sailors
WHERE  sailors.rating = :savedRating + &ratingInc;
--
UNDEFINE sailorID
UNDEFINE ratingInc
```

4. COMMIT and ROLLBACK

Example: PL040

Demonstrates
- Using COMMIT and ROLLBACK

Specs
- Prompt the user for a boat name.
- Prompt the user for the desired increment in the ratings for all sailors who reserved that boat.
- Prompt the user for a max allowed final rating for the above sailors.
- Reject all, or accept all, the increments based upon whether any updated sailor's new rating exceeds the max allowed rating or not.
- Print the rows for the sailors whose ratings were incremented.

Input *(Note: Start with a fresh copy of the database for each run.)*

- Clipper, 4, 12
- Clipper, 7, 12 <<<<<<< ***Notice why this update is rejected***
- Clipper, 4, 14
- Interlake, 4, 12 <<<<<<< ***Notice the run-time error you get here***
-

```
-- PL040
-- author: JRA
-- --------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- --------------------------------
ACCEPT boatName CHAR PROMPT 'Enter a boat name: '
ACCEPT ratingInc NUMBER PROMPT 'Enter an increment for ratings: '
ACCEPT allowedMaxRating NUMBER PROMPT 'Enter the max allowed rating: '
DECLARE
   newMaxRating sailors.rating%TYPE;
   boatID       boats.bid%TYPE;
BEGIN
-- If boat name is invalid or boat has no reservations, raise an exception.
   SELECT B.bid INTO boatID FROM   Boats B WHERE  B.bname='&boatName' AND
                  EXISTS(SELECT * FROM Reservations R WHERE R.bid=B.bid);
-- Increment the ratings.
   UPDATE sailors
   SET rating = rating + &ratingInc
   WHERE sailors.sid IN
       (SELECT S.sid
       FROM   sailors S, reservations R, boats B
       WHERE  S.sid = R.sid AND R.bid = B.bid   AND B.bname = '&boatName');
-- Get the new max rating
   SELECT MAX(S.rating) INTO newMaxRating
   FROM   sailors S, reservations R, boats B
   WHERE  S.sid = R.sid AND
          R.bid = B.bid   AND
          B.bname = '&boatName';
-- Check if any new rating is above the allowed max rating
   IF   newMaxRating <= &allowedMaxRating
   THEN COMMIT;
        DBMS_OUTPUT.PUT_LINE('+++++ DB has been updated');
   ELSE ROLLBACK;
      DBMS_OUTPUT.PUT_LINE
      ('+++++ Updates rolled back - newMaxRating would have been: '
                                  || newMaxRating);
   END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
    ('++++ '||'&boatName'||' is not a valid boat or has no reservations');
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('+++++'||SQLCODE||'...'||SQLERRM);
END;
/
-- Let's see what happened to the database
SELECT S.sid, S.sname, S.rating, S.age
FROM   sailors S, reservations R, boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.bname = '&boatName';
--
UNDEFINE boatName
```

5. CURSORS

- Needed when the result of a query can be more than one row.
- A cursor acts as a pointer to the results.
- Cursors are declared in the DECLARE section, as

> **DECLARE CURSOR** *cursorName* [INSENSITIIVE] [SCROLL] **IS**
> *someQuery …*
> [**ORDER BY** *orderItemList*]
> [**FOR READ ONLY | FOR UPDATE**]

  - FOR UPDATE or FOR READ ONLY
    - Declares whether result can or can't be updated.
    - The default is FOR UPDATE; but …
    - The default is FOR READ ONLY if the cursor is scrollable or insensitive.
  - SCROLL - more flexible variants of FETCH can be used.
  - INSENSITIIVE – result of cursor (after its OPEN) are not affected by updates due to other concurrent transactions.

- Cursors are controlled by three commands:
  - OPEN – executes the query and points to 'just before' the first row.
  - FETCH – points to the *next* row and retrieves it. (FETCH can have more complex positioning parameters.)
  - CLOSE – releases the cursor. A cursor may be reopened after closing it.

- Cursor properties:
  - %ISOPEN – returns true if the cursor is already open.
  - %FOUND – returns TRUE if the last FETCH returned a row; otherwise, returns FALSE. We can also use %NOTFOUND (the logical opposite of %FOUND).
  - %ROWCOUNT – returns the number of rows fetched.

Example PL050

Demonstrates
- Cursors.
- The %ROWTYPE data type
- User-defined exception.

Specs
- Essentially does what PL040 did (except for printing the before/after rows)
- Prompt the user for a boat name.
- Prompt the user for the desired increment in the ratings of all sailors who reserved that boat.
- Prompt the user for a max allowed final rating for the above sailors.
- Reject all, or accept all, the increments based upon whether any sailor's new rating exceeds the max allowed rating or not.
- Print the row of each one of the above sailors before and after the update (if any).

Input *(Note: Start with a fresh copy of the database for each run.)*
- Clipper, 2, 10
- Clipper, 7, 16
- Clipper, 3, 14
-

```
-- PL050
-- author: JRA
-- ---------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- ---------------------------------
ACCEPT boatName CHAR PROMPT 'Enter a boat name: '
ACCEPT ratingInc NUMBER PROMPT 'Enter an increment for ratings: '
ACCEPT allowedMaxRating NUMBER PROMPT 'Enter the max allowed rating: '

DECLARE
   sr     sailors%ROWTYPE;

   CURSOR sCursor IS
          SELECT S.sid, S.sname, S.rating, S.age, S.trainee
          FROM   sailors S, reservations R, boats B
          WHERE  S.sid = R.sid  AND
                 R.bid = B.bid    AND
                 B.bname = '&boatName';

   aboveAllowedMax EXCEPTION;

-- Continued on next page
```

```
-- continued from previous page

BEGIN
   OPEN sCursor;
   LOOP
      -- Fetch the qualifying rows one by one
      FETCH sCursor INTO sr;
            -- Try this: comment out next line and run for Clipper, 1, 14
      EXIT WHEN sCursor%NOTFOUND;
      -- Print the sailor' old record
      DBMS_OUTPUT.PUT_LINE ('+++++ old row: '||sr.sid||'  '
             ||sr.sname||sr.rating||'  '||sr.age||'  '||sr.trainee);

-- Raise the user-defind exception aboveAllowedMax, if necessary
      sr.rating := sr.rating + &ratingInc;
      IF   sr.rating > &allowedMaxRating
      THEN RAISE aboveAllowedMax;
      ELSE UPDATE sailors
           SET rating = sr.rating
           WHERE sailors.sid = sr.sid;
      -- Print the sailor' new record
      DBMS_OUTPUT.PUT_LINE ('+++++ new row: '||sr.sid||'  '
               ||sr.sname||sr.rating||'  '||sr.age||'  '||sr.trainee);
      END IF;
   END LOOP;
   COMMIT;
   DBMS_OUTPUT.PUT_LINE('+++++ DB has been updated');
   CLOSE sCursor;

EXCEPTION
-- aboveAllowedMax is a user-defined exception.
WHEN aboveAllowedMax THEN
     ROLLBACK;
     DBMS_OUTPUT.PUT_LINE ('+++++ All updates rolled back: ' ||
               'A new rating would have been: '|| sr.rating);
WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('+++++'||SQLCODE||'...'||SQLERRM);
END;
/

-- Let's see what happened to the database
SELECT S.sid, S.rating
FROM   sailors S, reservations R, boats B
WHERE  S.sid = R.sid  AND
       R.bid = B.bid  AND
       B.bname = '&boatName';
--
UNDEFINE boatName
UNDEFINE ratingInc
UNDEFINE allowedMaxRating
```

Example PL060

Demonstrates
- Cursors, nested blocks, and the control flow associated with exceptions.

Specs
- Prompt the user for a boat name.
- Prompt the user for an increment in the ratings of all sailors who reserved that boat.
- Prompt the user for a max allowed final rating for above sailors.
- Reject the increment if the sailors' new rating exceeds the max allowed rating. Otherwise, accept it. **(THIS IS DIFFERENT FROM PL050)**
- Print the row of each one of the above sailors before and after the update (if any).

Input *(Note: Start with a fresh copy of the database for each run.)*
- Clipper, 2, 10
- Clipper, 7, 16
- Clipper, 3, 14

```
-- PL060
-- author: JRA
-- ---------------------------------------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- ---------------------------------
ACCEPT boatName CHAR PROMPT 'Enter a boat name: '
ACCEPT ratingInc NUMBER PROMPT 'Enter an increment for ratings: '
ACCEPT allowedMaxRating NUMBER PROMPT 'Enter the max allowed rating: '
DECLARE
   sr     sailors%ROWTYPE;
   CURSOR sCursor IS
          SELECT S.sid, S.sname, S.rating, S.age, S.trainee
          FROM   sailors S, reservations R, boats B
          WHERE  S.sid = R.sid  AND
                 R.bid = B.bid    AND
                 B.bname = '&boatName';

-- Continued on next page
```

```
-- continued from previous page

BEGIN
   OPEN sCursor;
   LOOP
      -- Fetch the qualifying rows one by one
      FETCH sCursor INTO sr;
      EXIT WHEN sCursor%NOTFOUND;
      -- Print the sailor' old record
      DBMS_OUTPUT.PUT_LINE ('+++++ old row: '||sr.sid||' '
             ||sr.sname||sr.rating||'  '||sr.age||'  '||sr.trainee);
      sr.rating := sr.rating + &ratingInc;

      -- A nested block
      DECLARE
         aboveAllowedMax EXCEPTION;
      BEGIN
         IF   sr.rating > &allowedMaxRating
         THEN RAISE aboveAllowedMax;
         ELSE UPDATE sailors
              SET rating = sr.rating
              WHERE sailors.sid = sr.sid;
              -- Print the sailor' new record
             DBMS_OUTPUT.PUT_LINE ('+++++ new row: '||sr.sid||' '
                  ||sr.sname||sr.rating||'  '||sr.age||'  '||sr.trainee);
         END IF;

      EXCEPTION
        WHEN aboveAllowedMax THEN
           DBMS_OUTPUT.PUT_LINE('+++++ Update rejected: '||
                'The new rating would have been: '|| sr.rating);
        WHEN OTHERS THEN
           DBMS_OUTPUT.PUT_LINE('+++++ update rejected: ' ||
                                SQLCODE||'...'||SQLERRM);
      END;
      -- end of the nested block
END LOOP;

   COMMIT;
   CLOSE sCursor;
END;
/

-- Let's see what happened to the database
SELECT S.sid, S.rating
FROM   sailors S, reservations R, boats B
WHERE  S.sid = R.sid  AND
       R.bid = B.bid  AND
       B.bname = '&boatName';
--
UNDEFINE boatName
UNDEFINE ratingInc
UNDEFINE allowedMaxRating
```

6. PROCEDURES AND FUNCTIONS

- They are **named**, rather than **anonymous** blocks.
- They offer:
  - **Modularity** – break complex logic into functional units.
  - **Reusability** and maintainability – test once and reuse always.
  - **Abstraction** – shows the 'what' and hides the 'how'.

Example PL070

Demonstrates
- Procedures and Functions.

Specs
- Similar to **PL040**; but uses a procedure and a function
- Prompt the user for a boat name.
- Prompt the user for the increment in the ratings of all sailors who reserved that boat.
- Prompt the user for a max allowed final rating for above sailors.
- **Call a function** to update the database; and return the status of the update.
- **Call a procedure** to compute the new max rating
- **Reject all, or accept all**, the increments based upon whether any updated sailor's new rating exceeds the max allowed rating or not.
- Using a SQL query, print the rows of all the sailors who have the new rating.

Input *(Note: Start with a fresh copy of the database for each run.)*
- Clipper, 4, 12
- Clipper, 7, 12
- Clipper, 4, 14

```
-- PL070
-- author: JRA
-- ---------------------------------------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- --------------------------------
ACCEPT boatName CHAR PROMPT 'Enter a boat name: '
ACCEPT ratingInc NUMBER PROMPT 'Enter an increment for ratings: '
ACCEPT allowedMaxRating NUMBER PROMPT 'Enter the max allowed rating: '

DECLARE
   updateStatus CHAR(5);
   newMaxRating sailors.rating%TYPE;
   -- ------------------------------------------------------
   FUNCTION updateDB (theBoat IN boats.bname%TYPE,
                      theInc  IN sailors.rating%TYPE)
                      RETURN  VARCHAR2 IS

   status    CHAR(5)  := 'notOK';
   isItHere  boats.bname%TYPE;
   BEGIN
      -- If boat name is invalid, raise an exception
      SELECT bname INTO isItHere FROM boats WHERE bname = theBoat;
      -- If boat name is valid, increment the ratings.
      UPDATE sailors
      SET rating = rating + theInc
      WHERE sailors.sid IN
                    (SELECT S.sid
                     FROM   sailors S, reservations R, boats B
                     WHERE  S.sid = R.sid AND R.bid = B.bid   AND
                            B.bname = theBoat);
      status := 'ok';
      RETURN (status);
      EXCEPTION
      WHEN NO_DATA_FOUND THEN
             DBMS_OUTPUT.PUT_LINE ('++++ '||theBoat||
                ' is either not a boat, or has no reservations');
             RETURN (status);
      WHEN OTHERS THEN
             DBMS_OUTPUT.PUT_LINE('+++++'||SQLCODE||'...'||SQLERRM);
             RETURN (status);
   END updateDB;
   -- ------------------------------------------------------
   PROCEDURE getMax(someBoat IN  boats.bname%TYPE,
                    itsMax   OUT sailors.rating%TYPE) IS
   BEGIN
      SELECT MAX(S.rating) INTO itsMax
      FROM   sailors S, reservations R, boats B
      WHERE  S.sid = R.sid AND
             R.bid = B.bid   AND
             B.bname = someBoat;
   END getMax;
   -- ------------------------------------------------------
-- The main program follows …
```

```
-- Main Program
BEGIN
   -- call the function to update the DB and report the status of update
   updateStatus := updateDB('&boatName', &ratingInc);

   -- if the updates are ok, call the procedure to get the new max rating
   IF   updateStatus = 'ok'
   THEN getMax('&boatName', newMaxRating);
        -- Check if any new rating is above the allowed max rating
        IF   newMaxRating <= &allowedMaxRating
        THEN COMMIT;
             DBMS_OUTPUT.PUT_LINE ('+++++ DB has been updated');
        ELSE ROLLBACK;
             DBMS_OUTPUT.PUT_LINE ('+++++ Updates rolled back: ' ||
                   'A newMaxRating would have been: '|| newMaxRating);
        END IF;
   END IF;
END;
/
-- Let's see what happened to the database
SELECT S.sid, S.sname, S.rating, S.age
FROM   sailors S, reservations R, boats B
WHERE  S.sid = R.sid AND
       R.bid = B.bid   AND
       B.bname = '&boatName';
--
UNDEFINE boatName
```

7. STORED PROCEDURES AND FUNCTIONS

Introduction

- SQL defines a standard: **Persistent, Stored Modules** (**SQL/PSM**)
- Oracle implements SQL/PSM through PL/SQL Stored Procedures and Functions.
- Our discussion will be Oracle-oriented.
- Non-PSM procedures and functions (discussed earlier) are `DECLARE`d inside, and invoked from within, anonymous PL/SQL blocks.
- In contrast, PSM are stored as schema objects and can be invoked from other environments (e.g. SQL, other PSM, triggers, etc.)
- To see the PSM's in the schema, run: **`SELECT object_name FROM user_procedures;`**
- To drop a procedure XYZ: **`DROP PROCEDURE`** ***XYZ*****;**
- To drop a function XYZ: **`DROP FUNCTION`** ***XYZ*****;**
- PSM can be written, not only in PL/SQL, but also in C, Java, etc.

Advantages of PSM

- Efficiency
  - They run in the process space of the DBMS.
- Reuse and easy maintainability
  - Once created, they can be used by other application logic
- Encapsulation
  - Can hide schema details from application programmers if data access is encapsulated into stored procedures.

Example PL080 using a stored function and a stored procedure (files **PL080a** and **PL080b** below).

Demonstrates

- Stored Procedures and Functions.

Specs

- Does exactly what PL070 does, except that it uses a stored procedure and a stored function. Notice that, except for the function and procedure declarations, the code is the same.
- Prompt the user for a boat name.
- Prompt the user for the increment in the ratings of all sailors who reserved that boat.
- Prompt the user for a max allowed final rating for above sailors.
- Call a stored function to update the DB if possible.
- Call a stored procedure to compute the new max rating
- Reject all, or accept all, the increments based upon whether any updated sailor's new rating exceeds the max allowed rating or not.
- Using a SQL query, print the rows of all the sailors who have the new rating.

Input … *(Note: Start with a fresh copy of the database for each run.)*

- Clipper, 4, 12
- Clipper, 7, 12
- Clipper, 4, 14

```
-- File: PL080a
-- Author: JRA
-- A STORED FUNCTION ----------------------
CREATE OR REPLACE FUNCTION updateDB (theBoat IN boats.bname%TYPE,
                                     theInc  IN sailors.rating%TYPE)
                                     RETURN  VARCHAR2 IS
    status    CHAR(5)  := 'notOK';
    isItHere  boats.bname%TYPE;
BEGIN
   -- If boat name is invalid, raise an exception
   SELECT bname INTO isItHere FROM boats WHERE bname = theBoat;

   -- Increment the ratings.
   UPDATE sailors
   SET rating = rating + theInc
   WHERE sailors.sid IN
                 (SELECT S.sid
                  FROM   sailors S, reservations R, boats B
                  WHERE  S.sid = R.sid AND
                         R.bid = B.bid   AND
                         B.bname = theBoat);
   status := 'ok';
   RETURN (status);
   EXCEPTION
   WHEN NO_DATA_FOUND THEN
       DBMS_OUTPUT.PUT_LINE ('++++ '||theBoat||
               ' is either not a boat, or has no reservations');
       RETURN (status);
  WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE('+++++'||SQLCODE||'...'||SQLERRM);
       RETURN (status);
END updateDB;
/
SHOW ERROR
SELECT OBJECT_NAME FROM USER_PROCEDURES;
```

```
-- File: PL080b
-- Author: JRA
-- A STORED PROCEDURE -------------------
CREATE OR REPLACE PROCEDURE getMax(someBoat IN  boats.bname%TYPE,
                           itsMax   OUT sailors.rating%TYPE) IS
BEGIN
   SELECT MAX(S.rating) INTO itsMax
   FROM   sailors S, reservations R, boats B
   WHERE  S.sid = R.sid AND
          R.bid = B.bid   AND
          B.bname = someBoat;
END getMax;
/
SHOW ERROR
SELECT OBJECT_NAME FROM USER_PROCEDURES;
```

```
-- The above two PSM's can be compiled/stored separately
--  and then used by the following PL/SQL block PL080
--
```

```
-- A PL/SQL BLOCK THAT USES THE ABOVE PSM's-------------
-- PL080
-- author: JRA
-----------------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-----------------------
ACCEPT boatName CHAR PROMPT 'Enter a boat name: '
ACCEPT ratingInc NUMBER PROMPT 'Enter an increment for ratings: '
ACCEPT allowedMaxRating NUMBER PROMPT 'Enter the max allowed rating:'
--
DECLARE
   updateStatus CHAR(5);
   newMaxRating sailors.rating%TYPE;

BEGIN
   -- update the DB and report the update status
   updateStatus := updateDB('&boatName', &ratingInc);

   -- if updates are OK, get the new max rating
   IF   updateStatus = 'ok'
   THEN getMax('&boatName', newMaxRating);
        -- Check if any new rating is above the allowed max rating
        IF   newMaxRating <= &allowedMaxRating
        THEN COMMIT;
             DBMS_OUTPUT.PUT_LINE('+++++ DB has been updated');
        ELSE ROLLBACK;
             DBMS_OUTPUT.PUT_LINE ('+++++ Updates rolled back: '||
               'A newMaxRating would have been: '|| newMaxRating);
        END IF;
   END IF;
END;
/
-- Let's see what happened to the database
SELECT S.sid, S.sname, S.rating, S.age
FROM   sailors S, reservations R, boats B
WHERE  S.sid = R.sid AND
       R.bid = B.bid   AND
       B.bname = '&boatName';
--
UNDEFINE boatName
```

```
-- File: PL083
-- Author: JRA
-- A STORED PROCEDURE that can be used to implement a computed attribute
CREATE OR REPLACE FUNCTION numReservations(id IN  Reservations.sid%TYPE)
          RETURN INTEGER IS
--
num INTEGER;
   BEGIN
     SELECT COUNT(*) INTO num
     FROM   reservations R
     WHERE  R.sid = id;
   RETURN (num);
END numreservations;
/
SHOW ERROR
SELECT OBJECT_NAME FROM USER_PROCEDURES;
--
-- Now test it:
   SELECT numReservations(22)
   FROM DUAL;
--
-- Now use it to implement a computed attribute for the Sailor entity.
   SELECT S.sid, S.sname, numReservations(S.sid)
   FROM   Sailors S;
```

8. TRIGGERS

- A trigger is a PL/SQL program that is:
  - Stored as a database object.
  - Called automatically - user program can't call the trigger explicitly.
  - To see what triggers are in the schema: **`SELECT TRIGGER_NAME FROM USER_TRIGGERS`**;
  - To drop a trigger XYZ: **`DROP TRIGGER`** ***XYZ*;**

- A trigger is composed of three major components: **event**, **condition** (*optional*), and **action**.

- General syntax:

```
CREATE [OR REPLACE] TRIGGER triggerName
BEFORE | AFTER | INSTEAD OF
INSERT | UPDATE | DELETE  ON tableName          <<< event
[WHEN condition]                                <<< condition (optional)
[FOR EACH ROW]
A DECLARE/BEGIN/EXCEPTION/END PL/SQL block      << < action
```

Key Options in the Specification of Triggers

- The event on the database may be an `INSERT`, `DELETE`, or `UPDATE`
- An optional condition can be specified by a `WHEN` clause.
  - The condition may be a logical condition or a query.
  - The trigger fire if the condition evaluates to `TRUE` or if the query result is non-empty.
- The action can be executed either `BEFORE` or `AFTER` the triggering event.
- The action can refer the `OLD` and/or `NEW` values of the tuples modified in the event.
- The action can be performed once `FOR EACH ROW` affected by the triggering event, or just once for all the rows that are affected; the two cases are called, respectively, **row-level** and **statement-level** triggers.

Caution!!!

- An event/condition may require the firing of more than one trigger. The DBMS may process them in some arbitrary order.
  - Therefore, we should not count on any specify order – it is often implementation-dependent.
- The action of a trigger may cause the firing of another (even same) trigger. Therefore.
  - It may become difficult to comprehend the effects of such chains.
  - Infinite loops my form.

Some Uses of Triggers

- Maintaining database consistency; but weigh:
  - The case for using ICs (e.g. foreign keys and CHECK) instead of triggers:
    - They are declarative and, thus, easier to understand.
    - They are enforced whenever any statement modifies the database.
    - The DBMS optimizes their enforcement.
  - The case for triggers instead of ICs:
    - The declarative specification of complex constraints is not supported by most databases.

- Automatically entering data (e.g. item price when manually entering an order).

- Collecting statistics or logs of events.

- Initiating events (e.g. student drops course; trigger checks whether status is still full-time).

Example PL090

Demonstrates
– A **statement-level** trigger
Specs
– If a new boat is inserted in the database, notify all sailors whose rating is above 8.

```
-- PL090
-- author: JRA
-- ---------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- ---------------------------------
CREATE OR REPLACE TRIGGER notify_1

AFTER INSERT ON boats                          /*Event*/
DECLARE                                        /*Action*/
     sr     sailors%ROWTYPE;
     CURSOR sCursor IS
            SELECT *
            FROM   sailors S
            WHERE  S.rating > 8;
BEGIN
   OPEN sCursor;
   LOOP
      -- Fetch the qualifying rows one by one
      FETCH sCursor INTO sr;
      EXIT WHEN sCursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE ('+++++ Trigger notify_1: '
                            ||sr.sid||'  '||sr.sname);
   END LOOP;
   CLOSE sCursor;
EXCEPTION
WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('+++++'||SQLCODE||'...'||SQLERRM);
END;
/
SHOW ERROR
SELECT TRIGGER_NAME FROM USER_TRIGGERS;
```

Testing … *(Start with a fresh copy of the database for each run.)*
– Create the above trigger.
– Run the following update to the database, see what happens, then restore the database.

```
INSERT INTO boats VALUES (120, 'titanic', 'white', 350, 32, 31);
DELETE FROM boats WHERE bid = 120;
```

– Drop the trigger: **DROP TRIGGER notify_1;**

Example PL092

Demonstrates

– A **row-level trigger** that fires upon a condition.

Specs

– If a new boat is inserted whose length is over 30 ft, notify all sailors whose rating is above 8.

```
-- ------------------------------------------------------
-- PL092
-- author: JRA
-- ---------------------------------
SET SERVEROUTPUT ON
SET VERIFY OFF
-- ---------------------------------
CREATE OR REPLACE TRIGGER notify_2
AFTER INSERT ON boats                          /*Event*/
FOR EACH ROW
WHEN (NEW.length >30)                          /*Condition*/
DECLARE                                        /*Action*/
     sr     sailors%ROWTYPE;
     CURSOR sCursor IS
            SELECT * FROM sailors S WHERE  S.rating > 8;
BEGIN
   OPEN sCursor;
   LOOP
      FETCH sCursor INTO sr;
      EXIT WHEN sCursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE ('+++++ Trigger notify_2: '
                                  ||sr.sid||'  '||sr.sname);
   END LOOP;
   CLOSE sCursor;
EXCEPTION
WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('+++++'||SQLCODE||'...'||SQLERRM);
END;
/
SHOW ERROR
```

Testing … *(Start with a fresh copy of the database for each run.)*

– Create the above trigger.

– Run the following update to the database, see what happens, then restore the database.

```
INSERT INTO boats VALUES (120, 'titanic', 'white', 350, 32, 31);
DELETE FROM boats WHERE bid = 120;
```

– Run the following update to the database, see what happens, then restore the database.

```
INSERT INTO boats VALUES (120, 'titanic', 'white', 350, 29, 31);
DELETE FROM boats WHERE bid = 120;
```

– Drop the trigger: **`DROP TRIGGER notify_2;`**

Example PL100 ***(see PL110 too)***

Specs

– This trigger implements part of the inclusion dependency bIC5 (see the Sailors schema).
– It verifies, upon INSERT or UPDATE on **Boats**, that the logKeeper is a trainee.

– Note: In order to fully implement the inclusion dependency, another trigger (see PL100) is needed.

```
-- ------------------------------------------------------
-- PL100
-- author: JRA
--
CREATE OR REPLACE TRIGGER bIC5_TA
BEFORE INSERT OR UPDATE OF logKeeper ON Boats
FOR EACH ROW
DECLARE
     numFound  INTEGER;
BEGIN
   SELECT COUNT(*)
   INTO   numFound
   FROM   Sailors S   WHERE  S.trainee = :NEW.logKeeper;

   IF  numFound < 1
   THEN
     RAISE_APPLICATION_ERROR(-20001, '+++++INSERT or UPDATE rejected. '||
                    'logKeeper '||:NEW.logkeeper|| ' is not a trainee');
   END IF;
END;
/
SHOW ERROR
```

```
--
SET ECHO ON
-- Testing the trigger for INSERT, and then restoring the database
SELECT * FROM BOATS;
INSERT INTO boats VALUES (120, 'titanic', 'white', 350, 32, 47);
--
INSERT INTO boats VALUES (120, 'titanic', 'white', 350, 32, 32);
-- Inspect then restore the Boats table
SELECT * FROM BOATS;
DELETE FROM Boats WHERE bid=120;
SELECT * FROM BOATS;
--
-- Testing the trigger for UPDATE, and then restoring the database
UPDATE Boats SET logkeeper=55 WHERE bid=101;
--
UPDATE Boats SET logkeeper=29 WHERE bid=101;
-- Inspect then restore the Boats table
SELECT * FROM BOATS;
UPDATE Boats SET logkeeper=95 WHERE bid=101;
SELECT * FROM BOATS;
--
DROP TRIGGER bIC5_TA;
SET ECHO OFF
```

Example PL110 ***(see PL100 too)***

- This trigger implements part of the inclusion dependency bIC5 (see our Sailors schema).
- It verifies, upon `DELETE,` or `UPDATE` on the **Sailors** table, that `boats.logKeeper` is in `Sailors.trainee`
- Notice the use of **`PRAGMA AUTONOMOUS_TRANSACTION`** (needed to avoid 'mutating table' issue.)

- Note: In order to fully implement the inclusion dependency, another trigger (see PL100) is needed.

```
-------------------------------------------------------
-- PL110
-- author: JRA
--
CREATE OR REPLACE TRIGGER bIC5_TB
BEFORE DELETE OR UPDATE OF trainee ON Sailors
FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
    numTrainers   INTEGER;
    difference    INTEGER;
--
BEGIN
--
--
-- You are encouraged to complete this as an exercise & test it as below
--
--
END;
/
SHOW ERROR
-------------------------------------------------------------------------
-- test script
-- SET ECHO ON
-- Test the trigger for DELETE, and UPDATE of Sailors
DELETE FROM Sailors where trainee=22;
UPDATE Sailors SET trainee=71 where trainee=22;
DELETE FROM Sailors where sid=58;
-- Restore the database
INSERT INTO Sailors VALUES (58, 'Jim', 10, 35.0, 32);
-- Inspect the database
SELECT * FROM Sailors order by sid;
-- Drop the trigger
Drop Trigger bIC5_TB;
SET ECHO OFF
```

Example … showing how to compare with SYSDATE

– This trigger shows how to compare an attribute of type DATE with the current date (called **SYSDATE**).

– Suppose you have a table:

```
CREATE TABLE Student (
…
…
DOB             DATE;
dateEnrolled    DATE
…
…
-- In Oracle you can say:
CONSTRAINT IC1 CHECK (dateEnrolled > DOB);
--
-- But you can't say:
CONSTRAINT IC2 CHECK (SYSDATE > DOB);
);
```

Here is how to implement IC2:

```
CREATE OR REPLACE TRIGGER IC2

BEFORE INSERT OR UPDATE  ON Student
FOR EACH ROW
BEGIN
   IF( :NEW.DOB <= SYSDATE )
   THEN
      RAISE_APPLICATION_ERROR(-20001,'+++++INSERT or UPDATE rejected …');
   END IF;
END;
/
SHOW ERROR
```