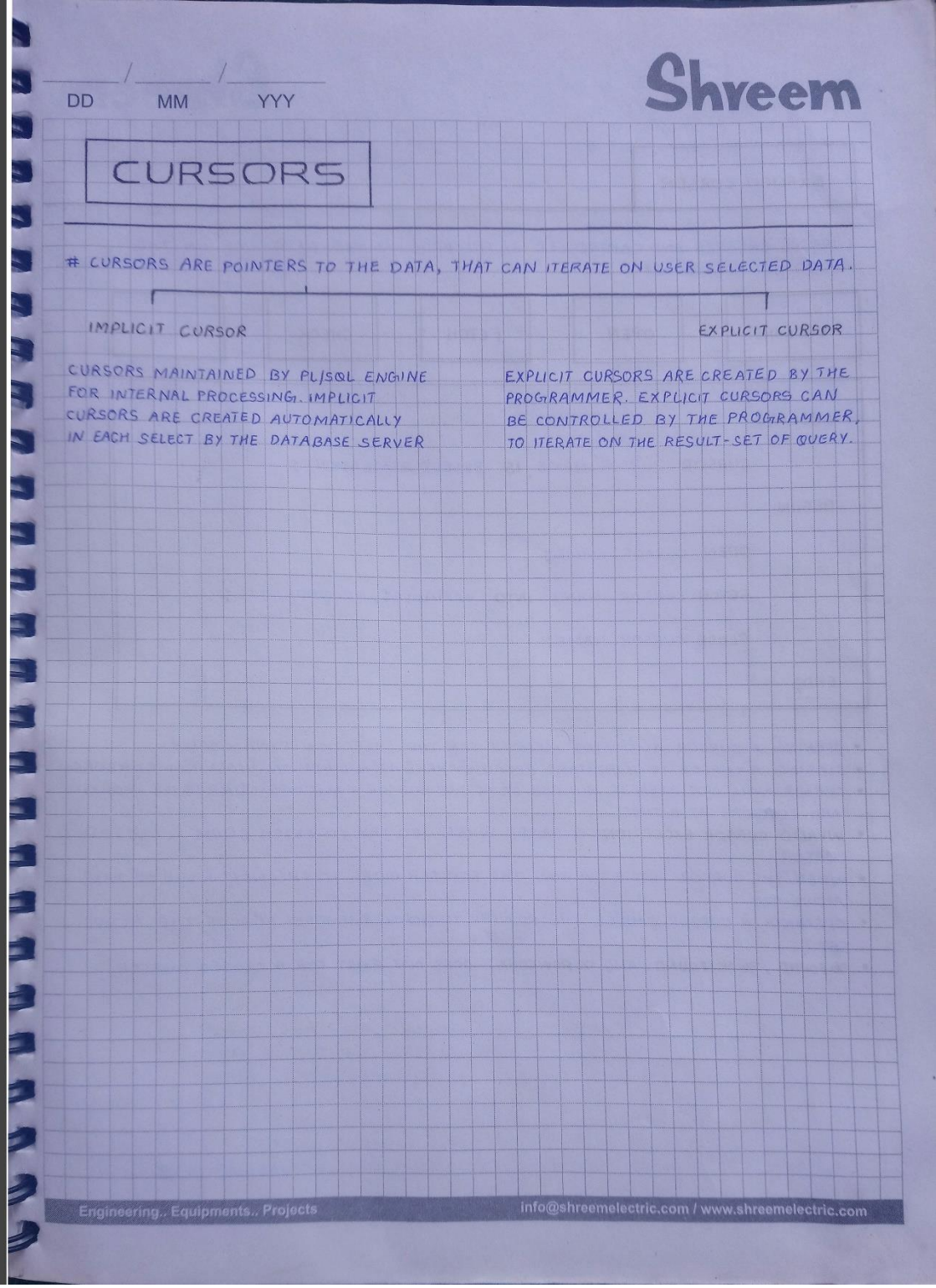
PL/SQL

Revise Oracle PLSQL in 42 pages



Shreem

EXPLICIT CURSOR

CURSORS CREATED AND CONTROLLED BY THE PROGRAMMER TO ITERATE ON THE RESULT-SET OF A QUERY.

DECLARE | OPEN | FETCH | CHECK | CLOSE

```
DECLARE
        CURSOR CURSOR_NAME IS SELECT STATEMENT;
BEGIN
        OPEN CURSOR_NAME;
        FETCH CURSOR_NAME INTO VARIABLES, RECORDS, ETC;
        CLOSE CURSOR_NAME;
END;
```

- OPENING A CURSOR ALLOCATES MEMORY FOR THE SELECTION DYNAMICALLY.
- CLOSING A CURSOR IS IMPORTANT BECAUSE THERE IS A LIMIT ON WORKING CURSORS.
- CURSOR WILL NOT OPEN IF NUMBER OF ACTIVE CURSOR = ALLOWED NUMBER OF WORKING CURSORS AT A TIME.
- INVALID CURSOR EXCEPTION IS RAISED WHEN YOU TRY TO FETCH FROM A CLOSED CURSOR.
- WHEN THE CURSOR FETCHES A ROW, THE POINTER GOES TO THE NEXT ROW OF THE ACTIVE-SET.
- OPENING A CURSOR CAUSES THE POINTER TO GO TO THE FIRST ROW OF THE ACTIVE SET.
- %FOUND, %NOTFOUND, AND %ROWCOUNT DOES NOT EXIST FOR A CLOSED CURSOR.

Shreem

#USING CURSORS WITH VARIABLES AND RECORDS

```
DECLARE
        CURSOR CURSOR_NAME IS
        SELECT COLUMN1, COLUMN2 FROM TABLE_NAME;
        C1_BOX TABLE_NAME.COLUMN1%TYPE;
        C2_BOX TABLE_NAME.COLUMN2%TYPE;
BEGIN
        OPEN CURSOR_NAME;
        FETCH CURSOR_NAME INTO C1_BOX, C2_BOX;
        DBMS_OUTPUT.PUT_LINE(C1_BOX || ' ' || C2_BOX);
        CLOSE CURSOR_NAME;
END;
```

```
DECLARE
        CURSOR CURSOR_NAME IS
        SELECT * FROM TABLE_NAME;
        MY_ROW TABLE_NAME%ROWTYPE;
BEGIN
        OPEN CURSOR_NAME;
        FETCH CURSOR_NAME INTO MY_ROW;
        DBMS_OUTPUT.PUT_LINE(MYROW.COLUMN1);
        DBMS_OUTPUT.PUT_LINE(MY_ROW.COLUMN2);
        CLOSE CURSOR_NAME;
END;
```

```
DECLARE
        CURSOR CURSOR_NAME IS
        SELECT COLUMN1, COLUMN2 FROM TABLE_NAME;
        MY_ROW CURSOR_NAME%ROWTYPE;
BEGIN
        OPEN CURSOR_NAME;
        FETCH CURSOR_NAME INTO MY_ROW;
        DBMS_OUTPUT.PUT_LINE(MY_ROW.COLUMN1);
        DBMS_OUTPUT.PUT_LINE(MY_ROW.COLUMN2);
        CLOSE CURSOR_NAME;
END;
```

- VARIABLES FOR ALL COLUMNS SHOULD BE USED WHEN USING CURSORS WITH VARIABLES.

TRAVERSING THROUGH CURSORS

POINTER	COLUMN1	COLUMN2	ATTRIBUTE VALUES	ACTION
	1	APPLE	CURSOR_NAME%ISOPEN : FALSE	DECLARE
	2	BALL	CURSOR_NAME%FOUND : X	
	3	CAT	CURSOR_NAME%NOTFOUND : X	
	4	DOG	CURSOR_NAME%ROWCOUNT : X	
→	1	APPLE	CURSOR_NAME%ISOPEN : TRUE	OPEN
	2	BALL	CURSOR_NAME%FOUND : NULL	
	3	CAT	CURSOR_NAME%NOTFOUND : NULL	
	4	DOG	CURSOR_NAME%ROWCOUNT : 0	
→	1	APPLE	CURSOR_NAME%ISOPEN : TRUE	FETCH
	2	BALL	CURSOR_NAME%FOUND : TRUE	
	3	CAT	CURSOR_NAME%NOTFOUND : FALSE	
	4	DOG	CURSOR_NAME%ROWCOUNT : 1	
	1	APPLE	CURSOR_NAME%ISOPEN : TRUE	FETCH
→	2	BALL	CURSOR_NAME%FOUND : TRUE	
	3	CAT	CURSOR_NAME%NOTFOUND : FALSE	
	4	DOG	CURSOR_NAME%ROWCOUNT : 2	
	1	APPLE	CURSOR_NAME%ISOPEN : TRUE	FETCH
	2	BALL	CURSOR_NAME%FOUND : TRUE	
→	3	CAT	CURSOR_NAME%NOTFOUND : FALSE	
	4	DOG	CURSOR_NAME%ROWCOUNT : 3	
	1	APPLE	CURSOR_NAME%ISOPEN : TRUE	FETCH
	2	BALL	CURSOR_NAME%FOUND : TRUE	
	3	CAT	CURSOR_NAME%NOTFOUND : FALSE	
→	4	DOG	CURSOR_NAME%ROWCOUNT : 4	
	1	APPLE	CURSOR_NAME%ISOPEN : TRUE	FETCH
	2	BALL	CURSOR_NAME%FOUND : FALSE	
	3	CAT	CURSOR_NAME%NOTFOUND : TRUE	
→	4	DOG	CURSOR_NAME%ROWCOUNT : 4	
	1	APPLE	CURSOR_NAME%ISOPEN : FALSE	CLOSE
	2	BALL	CURSOR_NAME%FOUND : X	
	3	CAT	CURSOR_NAME%NOTFOUND : X	
	4	DOG	CURSOR_NAME%ROWCOUNT : X	


```
DECLARE
        CURSOR  CURSOR_NAME IS SELECT * FROM TABLE_NAME;
        BOX     TABLE_NAME%ROWTYPE;
BEGIN
        OPEN    CURSOR_NAME;
        FETCH   CURSOR_NAME INTO BOX; DBMS_OUTPUT.PUT_LINE(BOX.C1);
        FETCH   CURSOR_NAME INTO BOX; DBMS_OUTPUT.PUT_LINE(BOX.C1);
        FETCH   CURSOR_NAME INTO BOX; DBMS_OUTPUT.PUT_LINE(BOX.C1);
        FETCH   CURSOR_NAME INTO BOX; DBMS_OUTPUT.PUT_LINE(BOX.C1);
        FETCH   CURSOR_NAME INTO BOX; DBMS_OUTPUT.PUT_LINE(BOX.C1);
        CLOSE   CURSOR_NAME;
END;
```

```
DECLARE
        CURSOR  CURSOR_NAME IS SELECT * FROM TABLE_NAME;
        BOX     TABLE_NAME%ROWTYPE;
BEGIN
        OPEN    CURSOR_NAME;
        LOOP
                FETCH CURSOR_NAME INTO BOX;
                EXIT WHEN CURSOR_NAME%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(BOX.C1);
        END LOOP;
        CLOSE   CURSOR_NAME;
END;
```

```
DECLARE
        CURSOR  CURSOR_NAME IS SELECT * FROM TABLE_NAME;
        BOX     TABLE_NAME%ROWTYPE;
BEGIN
        OPEN    CURSOR_NAME;
        FETCH   CURSOR_NAME INTO BOX;
        WHILE   CURSOR_NAME%FOUND
        LOOP
                DBMS_OUTPUT.PUT_LINE(BOX.C1);
                FETCH   CURSOR_NAME INTO BOX;
        END LOOP;
        FETCH   CURSOR_NAME INTO BOX;
        DBMS_OUTPUT.PUT_LINE(BOX.C1);
        CLOSE   CURSOR_NAME;
END;
```

```
DECLARE
        CURSOR  CURSOR_NAME IS SELECT * FROM TABLE_NAME;
        BOX     TABLE_NAME%ROWTYPE;
BEGIN
        OPEN    CURSOR_NAME;
        FOR i IN 1..4
        LOOP
                FETCH CURSOR_NAME INTO BOX;
                DBMS_OUTPUT.PUT_LINE(BOX.C1);
        END LOOP;
        FETCH   CURSOR_NAME INTO BOX;
        DBMS_OUTPUT.PUT_LINE(BOX.C1);
        CLOSE   CURSOR_NAME;
END;
```

```
DECLARE
        CURSOR CURSOR_NAME IS SELECT * FROM TABLE_NAME;
BEGIN
        FOR i IN CURSOR_NAME LOOP
                DBMS_OUTPUT.PUT_LINE(i.C1);
        END LOOP;
END;
```

```
BEGIN
        FOR i IN (SELECT * FROM TABLE_NAME)
        LOOP
                DBMS_OUTPUT.PUT_LINE(i.C1);
        END LOOP;
END;
```


CURSOR WITH PARAMETER(S)

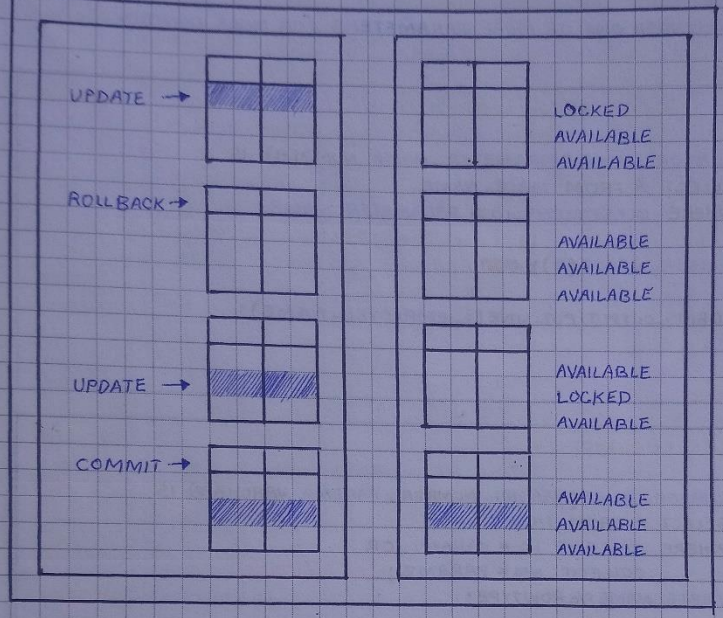
CURSORS CAN BE DECLARED WITH ONE OR MORE PARAMETERS FOR MORE CONTROL.

```
DECLARE
        CURSOR  CURSOR_NAME (PARAMETER_NAME NUMBER) IS
                SELECT * FROM TABLE_NAME
                WHERE DEPARTMENT_ID = PARAMETER_NAME;
BEGIN
        FOR i IN CURSOR_NAME(7)
        LOOP
                DBMS_OUTPUT.PUT_LINE(i.EMPLOYEE_NAME);
        END LOOP;
END;
```

```
DECLARE
        CURSOR  CURSOR_NAME(PARAM1 NUMBER, PARAM2 VARCHAR2) IS
                SELECT * FROM TABLE_NAME
                WHERE COLLEGE_ID = PARAM1 OR
                      COLLEGE_NM = PARAM2;
        BOX     TABLE_NAME%ROWTYPE;
BEGIN
        OPEN  CURSOR_NAME(3,'VINCENT PALLOTTI');
        LOOP
                FETCH CURSOR_NAME INTO BOX;
                EXIT WHEN CURSOR_NAME%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(CURSOR_NAME.EMP_NM);
        END LOOP;
        CLOSE CURSOR_NAME;
END;
```

- DO NOT DEFINE DATATYPE LENGTH IN PARAMETER DECLARATION.
- VARIABLES AND BIND VARIABLES CAN BE USED AS PARAMETER ARGUMENTS.
- SCOPE OF PARAMETERS IS BOUND INSIDE SELECT STATEMENT OF A CURSOR.
- AVOID USING COLUMN NAMES AS PARAMETER NAMES.

FOR UPDATE CLAUSE



A ROW UPDATED BY A USER AUTOMATICALLY GETS LOCKED FOR OTHER USERS UNTIL ROLLBACK OR COMMIT.

A USER CAN LOCK ALL THE ROWS FOR OTHER USERS BY UPDATING COLUMNS WITH SAME VALUES.
BUT,
UPDATE OPERATION CAUSES HEAVY UTILIZATION OF RESOURCES

FOR UPDATE CLAUSE IS USED WITH CURSOR TO LOCK ROWS IN SINGLE STEP.

FOR UPDATE CLAUSE LOCKS ALL THE ROWS OF THE SELECT QUERY OF A CURSOR IN ONE STEP AS SOON AS THE CURSOR IS OPENED. ALL LOCKS ARE ROW LEVEL LOCKS.

- IF A ROW OF THE SELECT STATEMENT OF A CURSOR IS ALREADY LOCKED, THEN THE CURSOR WAITS UNTIL THE OTHER USER ROLLBACK OR COMMIT.
- NOWAIT OPTION IS USED TO AVOID WAITING FOR A LOCKED ROW BY AVOIDING / TERMINATING COMMAND EXECUTION.
- DEFAULT OPTION IS WAIT.
- WHEN MULTIPLE TABLES ARE JOINED IN SELECT QUERY OF A CURSOR WITH FOR UPDATE CLAUSE, ALL THE SELECTED ROWS OF ALL TABLES ARE LOCKED. THEREFORE, OF OPTION IS USED WITH FOR UPDATE CLAUSE TO DEFINE COLUMN-NAME OF TABLES WHOSE ROWS ARE TO BE LOCKED.

```
DECLARE
        CURSOR  CURSOR_NAME ( PARAMETER_NAME  DATATYPE, ... )
                IS  SELECT_STATEMENT
                FOR UPDATE [OF COL_NM(S)][ NOWAIT | WAIT SECONDS ] ;
```


WHERE CURRENT OF CLAUSE

WHERE CURRENT OF CLAUSE GET ROWID OF TABLE IN SELECT QUERY OF CURSOR FOR DIRECT SELECTION OF TABLE ROW USING CURSOR.

DML OPERATIONS INSIDE CURSOR FOR LOOP PERFORMS FASTER WITH WHERE CURRENT OF CLAUSE.

```
DECLARE
        CURSOR  CURSOR_NAME IS SELECT * FROM TABLE_NAME
                               WHERE COLUMN_NAME = 'VOWEL'.
                               FOR UPDATE ;
BEGIN
      FOR i IN CURSOR_NAME
      LOOP
            UPDATE TABLE_NAME
            SET    COLUMN_NAME = 'V'
            WHERE CURRENT OF CURSOR_NAME ;
      END LOOP;
END;
```

- FOR UPDATE CLAUSE IS MANDATORY FOR USING WHERE CURRENT OF CLAUSE.
- WHEN A CURSOR IS CREATED USING A TABLE, PLSQL CAN DETERMINE TABLE ROWID USING CURSOR FOR LOOP.
- WHERE CURRENT OF CLAUSE WILL NOT GET TABLE ROWID IF TABLES ARE JOINED IN CURSOR SELECT QUERY.

- THE FOLLOWING PROGRAM WILL PERBORM SAME ACTION, BUT IS SLOWER.

```
DECLARE
        CURSOR  CURSOR_NAME IS SELECT * FROM TABLE_NAME
                               WHERE COLUMN_NAME = 'VOWEL'
                               FOR UPDATE ;
BEGIN
      FOR i IN CURSOR_NAME
      LOOP
            UPDATE TABLE_NAME
            SET    COLUMN_NAME = 'V'
            WHERE  ID = i.ID ;
      END LOOP;
END;
```

REFERENCE CURSORS

REFERENCE CURSORS ARE POINTERS TO ACTUAL CURSORS.
REFERENCE CURSORS HELP US TO USE ACTUAL CURSORS WITH MULTIPLE QUERIES.

- REFERENCE CURSORS CANNOT BE ASSIGNED NULL VALUES.
- REFERENCE CURSORS CANNOT BE USED IN TABLE-VIEW CREATE CODES.
- REFERENCE CURSORS CANNOT BE STORED IN COLLECTIONS.
- REFERENCE CURSORS CANNOT BE COMPARED.

STRONG CURSORS RESTRICTIVE CURSORS •(RETURN TYPE IS DEFINED)•	WEAK CURSORS NON-RESTRICTIVE CURSORS •(RETURN TYPE IS NOT DEFINED)•
• CURSOR_NAME%ROWTYPE IS VALID. • DO NOT SUPPORT DYNAMIC SQL OPEN STATEMENT • LESS ERROR PRONE	• CURSOR_NAME%ROWTYPE IS INVALID. • DO SUPPORT DYNAMIC SQL OPEN STATEMENT • MORE ERROR PRONE

SYS_REFCURSOR

SYS_REFCURSORS CAN BE USED WITHOUT DECLARING, AND ARE BETTER THAN SCHEMA-LEVEL WEAK REF CURSOR TYPE.

BUILT-IN WEAK REFERENCE CURSOR, THAT CAN BE USED WITH ANY RESULT-SET.

```
DECLARE
        REF_CUR_VAR   SYS_REFCURSOR;
        BOX           TABLE_NAME%ROWTYPE;
BEGIN
        OPEN REF_CUR_VAR FOR SELECT * FROM TABLE_NAME;
        LOOP
                FETCH REF_CUR_VAR INTO BOX;
                EXIT WHEN REF_CUR_VAR%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE(BOX.COLUMN1);
        END LOOP;
        CLOSE REF_CUR_VAR;
END;
```

- ACTUAL CURSORS ARE ASSOCIATED WITH FIXED SELECT QUERY.
- REFERENCE CURSORS CAN BE SENT BETWEEN DIFFERENT PROGRAMS OR BLOCKS.
- FOR IN LOOPS CANNOT BE USED WITH REFERENCE CURSORS.

STRONG CURSOR

- LINE 05 CAN BE REPLACED BY BOX REF_CUR_VAR%ROWTYPE;
- IF A VARCHAR(300), SUB_Q IS 'SELECT * FROM TABLE_NAME; THEN SUB_Q CANNOT BE USED WITH FOR IN LINE 07.

```
00 DECLARE
01         TYPE REF_TYPE IS REF CURSOR
02              RETURN TABLE_NAME%ROWTYPE;
03         REF_CUR_VAR  REF_TYPE;
04         BOX          TABLE_NAME%ROWTYPE;
05 BEGIN
06         OPEN REF_CUR_VAR FOR SELECT * FROM TABLE_NAME;
07         LOOP
08                 FETCH REF_CUR_VAR INTO BOX;
09                 EXIT WHEN REF_CUR_VAR%NOTFOUND;
10                 DBMS_OUTPUT.PUT_LINE(BOX.COLUMN1);
11         END LOOP;
12         CLOSE REF_CUR_VAR;
13 END;
14
15
```

WEAK CURSOR

- LINE 04 CANNOT BE REPLACED BY BOX REF_CUR_VAR%ROWTYPE;
- IF A VARCHAR(300), SUB_Q IS 'SELECT * FROM TABLE_NAME; THEN SUB_Q CAN BE USED WITH FOR IN LINE 06.

```
00 DECLARE
01         TYPE REF_TYPE IS REF CURSOR;
02         REF_CUR_VAR  REF_TYPE;
03         BOX   TABLE_NAME%ROWTYPE;
04 BEGIN
05         OPEN REF_CUR_VAR FOR SELECT * FROM TABLE_NAME;
06         LOOP
07                 FETCH REF_CUR_VAR INTO BOX;
08                 EXIT WHEN REF_CUR_VAR%NOTFOUND;
09                 DBMS_OUTPUT.PUT_LINE(BOX.COLUMN1);
10         END LOOP; CLOSE REF_CUR_VAR;
11 END;
12
13
```

EXCEPTION HANDLING

- COMPILER ERRORS ARE DETECTED AND THROWN BEFORE RUNNING THE PROGRAM.
- RUNTIME ERRORS CANNOT BE DETECTED WITHOUT RUNNING THE PROGRAM.
- EXCEPTIONS ARE USED TO HANDLE RUNTIME ERRORS.
- WHEN A RUNTIME ERROR OCCURS, PLSQL ENGINE THROWS AN EXCEPTION AND TERMINATE THE EXECUTION. THIS PROCESS OF THROWING AN EXCEPTION IS CALLED RAISING AN EXCEPTION.

EXCEPTIONS HAVE TWO ATTRIBUTES
- ERROR CODE (ORA-XXXXX)(SQLCODE)
- ERROR MESSAGE (SQLERRM)

AN EXCEPTION CAN BE RAISED
- IMPLICITLY BY PLSQL ENGINE
- EXPLICITLY BY PROGRAMMER

EXCEPTIONS CAN BE HANDLED IN THREE WAYS
- TRAP
- PROPAGATE
- TRAP AND PROPAGATE

EXCEPTION TYPES

PREDEFINED (20) ORACLE SERVER ERRORS
- TRAP THE EXCEPTION IN EXCEPTION SECTION.

NON-PREDEFINED ORACLE SERVER ERRORS
- DECLARE EXCEPTION IN THE DECLARATION BLOCK
- ASSIGN ERROR-CODE TO THAT EXCEPTION

USER-DEFINED ERRORS
- DECLARE EXCEPTION IN THE DECLARATION BLOCK.
- RAISE EXCEPTION MANUALLY

EXCEPTIONS CANNOT BE TRAPPED WITH THEIR EXCEPTION CODE.
EACH BEGIN-END BLOCK CAN HAVE ITS OWN EXCEPTION SECTION. (EXCEPTION HANDLED IN TO OUT)

```
DECLARE
        BOX  VARCHAR2(1);
BEGIN
        SELECT ALPHABET INTO BOX
        FROM TABLE_NAME WHERE ID = 'APPLE';
EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE ('MISTAKE');
END;
```

ID	ALPHABET	EXPANSION
1	A	APPLE
2	B	BALL
3	C	CAT
4	D	DOG
5	E	ELEPHANT
6	F	FISH
7	G	GOAT
8	H	HEN

ORACLE PREDEFINED EXCEPTIONS

```
DECLARE
        BOX  VARCHAR2(30);
BEGIN
        SELECT NAME INTO BOX FROM TABLE_NAME
                        WHERE VC_TYPE = 'C';
EXCEPTION
        WHEN TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE('MISTAKE');
END;
```

ID	NAME	VC_TYPE
1	APPLE	V
2	BALL	C
3	CAT	C
4	DOG	C
5	ELEPHANT	V
6	FISH	C
7	GOAT	C
8	HEN	C

LIST OF PREDEFINED EXCEPTIONS

01	ACCESS_INTO_NULL
02	CASE_NOT_FOUND
03	COLLECTION_IS_NULL
04	CURSOR_ALREADY_OPEN
05	DUP_VAL_ON_INDEX
06	INVALID_CURSOR
07	INVALID_NUMBER
08	LOGIN_DENIED
09	NO_DATA_FOUND
10	NO_DATA_NEEDED
11	NOT_LOGGED_ON
12	PROGRAM_ERROR
13	ROWTYPE_MISMATCH
14	SELF_IS_NULL
15	STORAGE_ERROR
16	SUB_SCRIPT_BEYOND_COUNT
17	SUB_SCRIPT_OUTSIDE_LIMIT
18	SYS_INVALID_ROWID
19	TIMEOUT_ON_RESOURCE
20	TOO_MANY_ROWS
21	VALUE_ERROR
22	ZERO_DIVIDE

NON-PREDEFINED EXCEPTIONS

```
DECLARE
        EXCEPTION_NAME  EXCEPTION;
        PRAGMA EXCEPTION_INIT(EXCEPTION_NAME, -01407);
BEGIN
        UPDATE EMPLOYEES_COPY SET EMAIL = NULL
        WHERE EMPLOYEE_ID = 100;
EXCEPTION
        WHEN  EXCEPTION_NAME THEN
        DBMS_OUTPUT.PUT_LINE('MISTAKE');
END;
```

- PREDEFINED AND NON-PREDEFINED EXCEPTIONS CAN BE CALLED IMPLICITLY AS WELL AS EXPLICITLY.
- EXCEPTION_INIT IS ALWAYS FEED WITH ORACLE SERVER LISTED ERROR CODE.

USER-DEFINED EXCEPTIONS

```
DECLARE
        EXCEPTION_NAME  EXCEPTION;
BEGIN
        NULL;
        RAISE EXCEPTION_NAME;
EXCEPTION
        WHEN EXCEPTION_NAME THEN
                DBMS_OUTPUT.PUT_LINE('MISTAKE');
END;
```

- USER-DEFINED EXCEPTIONS CAN BE CALLED EXPLICITLY ONLY.
- RAISE CLAUSE CAN BE USED TO RAISE PREDEFINED, NON-PREDEFINED AND USER-DEFINED EXCEPTION.
- USER-DEFINED EXCEPTIONS ARE KNOWN INSIDE PL/SQL BLOCK ONLY.

RAISE_APPLICATION_ERROR PROCEDURE

USER-DEFINED EXCEPTIONS ARE NOT KNOWN OUTSIDE THE PL/SQL BLOCKS, WHEREAS RAISE_APPLICATION_ERROR PROCEDURE IS KNOWN OUTSIDE THE PL/SQL BLOCKS FOR COMMUNICATING WITH SUBPROGRAMS OR APPLICATIONS.

```
RAISE_APPLICATION_ERROR(ERROR_NUMBER, ERROR_MESSAGE, ERROR_STACK MODE);
```

- ERROR_NUMBER : CUSTOM ERROR CODE RANGING FROM -20000 TO -20999.
- ERROR_MESSAGE : CUSTOM ERROR MESSAGE UPTO 2048 BYTES
- ERROR_STACK : ERROR STACK CAN BE ENABLED OR DISABLED BY TRUE OR FALSE.

WHEN AN EXCEPTION CAUSES MULTIPLE EXCEPTIONS TO RAISE FROM INNER TO OUTER BLOCKS, ALL ERROR MESSAGES GETS STACKED INTO ERROR STACK.

```
DECLARE
        BOX VARCHAR2(30);
BEGIN
        NULL;
        RAISE_APPLICATION_ERROR(-20003,'MISTAKE');
END;
```

SUBPROGRAMS

SUBPROGRAMS ARE USED FOR ENCAPSULATION OF REPETITIVE TASK OR CALCULATION CODE FOR FLEXIBLE AND EFFICIENT USAGE OR REUSABILITY.

- SUBPROGRAMS ARE STORED IN DATABASE WITH NAMES.
- SUBPROGRAMS ARE COMPILED ONCE AND CALLED MANY TIMES.
- SUBPROGRAMS CAN BE CALLED BY DIFFERENT BLOCKS.
- SUBPROGRAMS CAN TAKE AND UPDATE PARAMETERS.
- AN UNHANDLED SUBPROGRAM EXCEPTION IS RAISED TO ITS CALLER BLOCK.

SUBPROGRAM PARAMETER MODES

- IN : READS VALUE PASSED IN SUBPROGRAM CALL STATEMENT
- OUT : WRITE TO VARIABLE PASSED IN SUBPROGRAM CALL STATEMENT.
- IN OUT : READ AND WRITE VARIABLE PASSED IN SUBPROGRAM CALL STATEMENT.

SUBPROGRAM PARAMETER NOTATION

- POSITION SEQUENCED NOTATION
- NAMED NOTATION
- MIXED NOTATION

PLSQL FEATURED 2 SUBPROGRAMS

FUNCTIONS		PROCEDURES	
STORED FUNCTIONS FUNCTIONS KNOWN TO ALL BLOCKS WITH PRIVILEGES	LOCAL FUNCTIONS FUNCTIONS KNOWN TO DEFINITION BLOCKS	STORED PROCEDURE PROCEDURES KNOWN TO ALL BLOCKS WITH PRIVILEGES.	LOCAL PROCEDURE PROCEDURES KNOWN TO DEFINITION BLOCK

- DEFAULT VALUES CAN BE SET FOR PARAMETERS IN FUNCTION DEFINITION.
- LOCAL SUBPROGRAMS MUST BE DECLARED LAST IN DECLARATION SECTION.
- LOCAL SUBPROGRAMS CAN BE NESTED.
- PARAMETERS HAVING DEFAULT VALUE SET IN SUBPROGRAM DEFINITION AND HAVING IN MODE CAN BE SKIPPED DURING FUNCTION CALL.
- RETURN STATEMENT EXITS FROM ALL ENCLOSING BLOCKS.
- DO NOT FORGET MENTIONING RETURN STATEMENT WHILE HANDLING FUNCTION EXCEPTIONS.

PROCEDURES

GENERALLY USED WHEN A PROCEDURAL TASK IS TO BE PERFORMED MANY TIMES.
PROCEDURES CAN BE OVERLOADED.
PROCEDURES CANNOT BE DIRECTLY USED INSIDE A QUERY

```
CREATE  PROCEDURE  PROCEDURE_NAME1 IS
        BOX        VARCHAR2(30) := '1 LITTLE';
BEGIN   DBMS_OUTPUT.PUT_LINE(BOX);   END;
```

```
CREATE  PROCEDURE  PROCEDURE_NAME2 ( PARAM NUMBER) AS
BEGIN   DBMS_OUTPUT.PUT_LINE(PARAM||'LITTLE');  END;
```

```
CREATE  PROCEDURE  PROCEDURE_NAME3 ( PARAM NUMBER DEFAULT 3) IS
BEGIN   DBMS_OUTPUT.PUT_LINE(PARAM||'LITTLE');  END;
```

```
CREATE  PROCEDURE  PROCEDURE_NAME4 ( PARAM IN NUMBER) AS
BEGIN   DBMS_OUTPUT.PUT_LINE(PARAM||'LITTLE');  END;
```

```
CREATE  PROCEDURE  PROCEDURE_NAME5 ( P1 NUMBER, P2 IN VARCHAR2) IS
BEGIN   DBMS_OUTPUT.PUT_LINE(P1||' '||P2);   END;
```

```
CREATE  PROCEDURE  PROCEDURE_NAME6 (PARAM OUT VARCHAR2) AS
BEGIN   PARAM := '6 LITTLE';  END;
```

```
CREATE  PROCEDURE  PROCEDURE_NAME7 (PARAM IN OUT VARCHAR2) IS
BEGIN   PARAM := PARAM||'LITTLE';  END;
```

- ALL THE ABOVE PROCEDURES ARE STORED PROCEDURES / STANDALONE PROCEDURES.
- WHEN PROCEDURES ARE DEFINED INSIDE DECLARATION SECTION OF AN ANONYMOUS BLOCK, IT IS CALLED AS LOCAL PROCEDURE AND ARE KNOWN INSIDE BLOCK ONLY.


```
DECLARE
        TICKET6  VARCHAR2(30);
        TICKET7  VARCHAR2(30) := 7;
BEGIN
        PROCEDURE_NAME1;
        PROCEDURE_NAME2(2);
        PROCEDURE_NAME3;
        PROCEDURE_NAME4(4);
        PROCEDURE_NAME5(5,'LITTLE');
        PROCEDURE_NAME6(TICKET6);  DBMS_OUTPUT.PUT_LINE(TICKET6);
        PROCEDURE_NAME7(TICKET7);  DBMS_OUTPUT.PUT_LINE(TICKET7);
        PROCEDURE_NAME4(PARAM => 8);
        PROCEDURE_NAME5(9, P2 => 'LITTLE');
        PROCEDURE_NAME5(P2 => 'LITTLE', P1 => 10);
END;
```

```
DECLARE
        TKT6   VARCHAR2(30);
        TKT7   VARCHAR2(30) := 7;
        PROCEDURE  PN1 IS
                BOX  VARCHAR2(30) := 'LITTLE';
        BEGIN DBMS_OUTPUT.PUT_LINE(BOX);                    END;
        PROCEDURE  PN2(PARAM NUMBER) IS
        BEGIN DBMS_OUTPUT.PUT_LINE(PARAM||'LITTLE');        END;
        PROCEDURE  PN3(PARAM NUMBER DEFAULT 3) IS
        BEGIN DBMS_OUTPUT.PUT_LINE(PARAM||'LITTLE');        END;
        PROCEDURE  PN4(PARAM IN NUMBER) IS
        BEGIN DBMS_OUTPUT.PUT_LINE(PARAM||'LITTLE');        END;
        PROCEDURE  PN5(P1 NUMBER, P2 IN VARCHAR2) IS
        BEGIN DBMS_OUTPUT.PUT_LINE(P1||' '||P2);            END;
        PROCEDURE  PN6(PARAM OUT VARCHAR2) AS
        BEGIN PARAM := '6 LITTLE';                          END;
        PROCEDURE  PN7(PARAM IN OUT VARCHAR2) AS
        BEGIN      PARAM := PARAM||'LITTLE';                END;
BEGIN
        PN1;
        PN2(2);
        PN3;
        PN4(4);
        PN5(5,'LITTLE');
        PN6(TKT6);          DBMS_OUTPUT.PUT_LINE(TKT6);
        PN7(TKT7);          DBMS_OUTPUT.PUT_LINE(TKT7);
END;
```

Shreem

FUNCTIONS

GENERALLY USED WHEN A BLOCK OF CODE IS TO BE SUBSTITUTED BY AN OBJECT OR A CALCULATED VALUE. A FUNCTION ALWAYS RETURNS A VALUE.
FUNCTIONS CAN BE OVERLOADED.
FUNCTIONS CAN BE USED IN QUERIES SUBJECT TO FUNCTION QUERY RULES/RESTRICTIONS.

RESTRICTIONS OF USING FUNCTIONS IN SQL EXPRESSIONS
- FUNCTION MUST BE COMPILED AND STORED IN DATABASE
- FUNCTION MUST NOT HAVE AN OUT MODE PARAMETER.
- FUNCTION MUST RETURN A VALID SQL DATATYPE.
- FUNCTION CANNOT BE USED IN CHECK CLAUSE OF CREATE TABLE OR ALTER TABLE.
- FUNCTION MUST NOT CONTAIN DML STATEMENT.
- FUNCTION MUST NOT CONTAIN COMMIT, ROLLBACK OR DDL STATEMENT.
- CORRESPONDING PRIVILEGES MUST BE AVAILABLE.

```
DECLARE

        FUNCTION FNNM RETURN VARCHAR2 IS
                 BOX  VARCHAR2(50) := 'O LITTLE FROM F';
        BEGIN    RETURN BOX;  END;

        FUNCTION FNNM (P1 NUMBER) RETURN VARCHAR2 AS
        BEGIN    RETURN  P1||' LITTLE FROM F(N)';   END;

        FUNCTION FNNM(P2 VARCHAR2) RETURN VARCHAR2 IS
        BEGIN    RETURN  P2||' LITTLE FROM F(V)';   END;

        FUNCTION FNNM(P3A NUMBER, P3B VARCHAR2) RETURN VARCHAR2 AS
        BEGIN    RETURN  P3A||P3B||'LITTLE FROM F(N,V)';  END;

        FUNCTION FNNM(P4A NUMBER DEFAULT 0, P4B NUMBER) RETURN VARCHAR2 IS
        BEGIN    RETURN  P4A||P4B||'LITTLE FROM F(N,N)';   END;

BEGIN
        DBMS_OUTPUT.PUT_LINE(FNNM);
        DBMS_OUTPUT.PUT_LINE(FNNM(1));
        DBMS_OUTPUT.PUT_LINE(FNNM('2'));
        DBMS_OUTPUT.PUT_LINE(FNNM(0,'3'));
        DBMS_OUTPUT.PUT_LINE(FNNM(0,4));
        DBMS_OUTPUT.PUT_LINE(FNNM(P4B=>5));
        DBMS_OUTPUT.PUT_LINE(FNNM(P3B=>6, P3A=>0));
END;
```


Shreem

PIPELINED TABLE FUNCTIONS

FUNCTIONS RETURNING COLLECTION OF ROWS THAT CAN BE QUERIED LIKE A REGULAR DATABASE TABLE IS CALLED AS TABLE FUNCTION

- A TABLE FUNCTION EITHER RETURNS A NESTED TABLE OR A VARRAY.
- A TABLE FUNCTION RETURNS AFTER COMPLETING THE WHOLE DATA (ALL ROWS).
- PGA PROBLEMS MAY ARISE WHEN A TABLE FUNCTION HANDLES LARGE DATA.
- BIGGER DATA CAUSE TABLE FUNCTIONS TO SLOW DOWN PERFORMANCE.

PIPELINED TABLE FUNCTIONS RETURN EACH ROW ONE-BY-ONE.
- PIPELINED TABLE FUNCTIONS OCCUPY LESS MEMORY.
- PIPELINED TABLE FUNCTIONS ARE FASTER.

```
CREATE TYPE REC_TYPE IS OBJECT (SEQ NUMBER, MERIDINA VARCHAR2(30));
/
CREATE TYPE NTB_TYPE IS TABLE OF REC_TYPE;
```

```
CREATE FUNCTION TFN (MERIDINA_COUNT NUMBER) RETURN NTB_TYPE IS
        SOLID  NTB_TYPE := NTB_TYPE();
BEGIN
        FOR i IN 1.. MERIDINA_COUNT
        LOOP
                SOLID.EXTEND;
                SOLID(i) := REC_TYPE(i, i||'LITTLE');
        END LOOP;
        RETURN SOLID;
END;
```

▼ PIPELINED TABLE FUNCTION

```
CREATE FUNCTION PTFN (MERIDINA_COUNT NUMBER) RETURN NTB_TYPE PIPELINED IS
BEGIN
        FOR i IN 1.. MERIDINA_COUNT
        LOOP
                PIPE ROW (REC_TYPE (i, i ||' LITTLE'));
        END LOOP;
        RETURN;
END;
```

DD MM YYY

Shreem

PACKAGES

- WHEN DATABASE OBJECTS OR SUBPROGRAMS ARE CALLED BY A USER, IT IS LOADED FROM DATABASE STORAGE TO DATABASE MEMORY IN CORRESPONDING USER PGA. THIS MAY CAUSE PGA TO RUN OUT OF MEMORY DURING HEAVEY TRANSACTIONS.
- WHEN A PACKAGE IS CALLED BY A USER, IT IS LOADED FROM DATABASE STORAGE TO DATABASE MEMORY IN SGA (SYSTEM GLOBAL AREA).
- SGA IS AVAILABLE TO ALL PGAs OF ALL THE USERS AND ARE LOADED ONLY ONCE.

PACKAGES ARE USED TO LOGICALLY GROUP OBJECTS, SUBPROGRAMS AND VARIABLES TO GAIN PERFORMANCE AND IMPROVE CODE READABILITY.

PACKAGE

- PACKAGE SPECIFICATION
- PACKAGE BODY

- A PACKAGE CAN EXIST WITHOUT A BODY.
- A PACKAGE CANNOT BE RENAMED.
- PACKAGES CAN BE DROPPED.
- DROP PACKAGE PACKAGE_NAME WILL DELETE PACKAGE SPECIFICATION AS WELL AS PACKAGE BODY.
- DROP PACKAGE BODY PACKAGE_NAME WILL DELETE PACKAGE BODY ONLY.
- PACKAGE SPECIFICATION MUST BE COMPILED FIRST.
- A PACKAGE CAN HAVE : FUNCTIONS, PROCEDURES, TYPES, VARIABLES, CONSTANTS, EXCEPTIONS, CURSORS, PRAGMAs.

PACKAGE SPECIFICATION
- DECLARE OBJECTS
- ASSIGN VALUES TO VARIABLES.
- EVERYTHING DECLARED IN PACKAGE SPECIFICATION IS PUBLIC. VARIABLES CAN BE DECLARED BEFORE OR AFTER OTHER OBJECTS/SUBPROGRAMS.

PACKAGE BODY
- IMPLEMENTATION OF SUBPROGRAMS.
- OBJECTS, SUBPROGRAMS AND VARIABLES DECLARED ONLY IN PACKAGE-BODY ARE PRIVATE.
- VARIABLES MUST BE DECLARED BEFORE ANY OTHER OBJECT/SUBPROGRAMS.

OBJECTS DECLARED IN PACKAGE SPECIFICATION → PUBLIC OBJECTS

PRIVATE OBJECTS

LOCAL OBJECTS

PRIVATE SUBPROGRAMS IN PACKAGE BODY CANNOT BE CALLED BEFORE DECLARING. FORWARD REFERENCE IS USED TO DECLARE CALLED SUBPROGRAM DEFINED AFTER CALLER.

DD MM YYY

Shreem

```
CREATE PACKAGE PKG_NM AS
        VAR_NM1  VARCHAR2(30);
        VAR_NM2  NUMBER := 1035;
        VAR_NM3  CONSTANT NUMBER := 1035;
        VAR_NM4  EXCEPTION;
        REC_NM   TABLE_NAME%ROWTYPE;
        TYPE     TYPE_NM IS TABLE OF VARCHAR2(30);
        CURSOR   CUR_NAME IS SELECT * FROM TABLE_NAME;
        PROCEDURE LITTLEx(TKT NUMBER);
END;
```

```
CREATE PACKAGE BODY PKG_NM AS
        FUNCTION MERIDINA(TKT NUMBER) RETURN VARCHAR2;
        VAR_NM5 VARCHAR2(300);
        VAR_NM6 TYPE_NM := TYPE_NM();
        PROCEDURE LITTLEx(TKT NUMBER) IS
        BEGIN
            FOR I IN 1..TKT LOOP DBMS_OUTPUT.PUT_LINE(I||' LITTLE'); END LOOP;
            DBMS_OUTPUT.PUT_LINE(MERIDINA(TKT));
        END;
        FUNCTION MERIDINA(TKT NUMBER) RETURN VARCHAR2 IS
        BEGIN
            FOR I IN 1..TKT LOOP VAR_NM5 := 'MERIDINA'||VAR_NM5; END LOOP;

        END;
BEGIN
        VAR_NM1 := 1000;
END;
```

```
DECLARE
        MY_COLLECTION PACKAGE_NM.TYPE_NM := PACKAGE_NM('A','B','C','D','E');
BEGIN
        DBMS_OUTPUT.PUT_LINE(PKG_NM.VAR_NM1);
        DBMS_OUTPUT.PUT_LINE(PKG_NM.VAR_NM2); RAISE PKG_NM.VAR_NM4;
EXCEPTION
        WHEN PKG_NM.VAR_NM4 THEN
        DBMS_OUTPUT.PUT_LINE('EXCEPTION CAPTURED');
        DBMS_OUTPUT.PUT_LINE(MY_COLLECTION(1)); PKG_NM.LITTLEx(6);
        OPEN PKG_NM.CUR_NAME;
        FETCH PKG_NM.CUR_NAME INTO PKG_NM.REC_NM;
        DBMS_OUTPUT.PUT_LINE(PKG_NM.REC_NM.NAME);
END;
```

PACKAGE INITIALIZATION

PACKAGE BODY OF A PACKAGE CAN HAVE INITIALIZATION BLOCK THAT IS INVOKED ON THE FIRST CALL OF A PACKAGE DURING A SESSION.

- VARIABLE ASSIGNMENT IN INITIALIZATION BLOCK OVERRIDES VALUES OF PACKAGE VARIABLE.

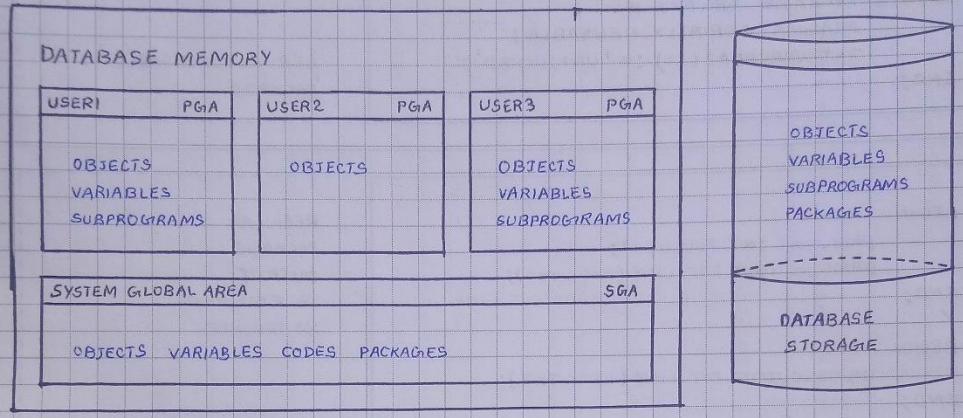
```
CREATE PACKAGE PACKAGE_NAME AS
        PUBLIC_VARIABLE1 NUMBER;
        PUBLIC_VARIABLE2 VARCHAR2(30);
        PROCEDURE AxB(A NUMBER, B VARCHAR2);
END;
```

```
CREATE PACKAGE BODY PACKAGE_NAME IS
        PROCEDURE AxB(A NUMBER, B VARCHAR2) IS
        BEGIN
                FOR i IN 1..A
                LOOP
                        DBMS_OUTPUT.PUT_LINE(i||B);
                END LOOP;
        END;
BEGIN
        PUBLIC_VARIABLE1 := 5
        PUBLIC_VARIABLE2 := 'LITTLE';
END;
```

```
EXEC PACKAGE_NAME.AxB(PACKAGE_NAME.PUBLIC_VARIABLE1,
                      PACKAGE_NAME.PUBLIC_VARIABLE2);
/
EXEC PACKAGE_NAME.PUBLIC_VARIABLE1 := 30;
/
EXEC PACKAGE_NAME.AxB(PACKAGE_NAME.PUBLIC_VARIABLE1,
                      PACKAGE_NAME.PUBLIC_VARIABLE2);
```


PERSISTENT STATE OF PACKAGES

- DATABASE OBJECTS ARE STORED IN DATABASE STORAGE.
- DATABASE OBJECTS ARE FETCHED FROM DATABASE STORAGE TO RESPECTIVE USER PGA IN DATABASE MEMORY, DURING EXECUTION.
- EACH USER HAS ITS OWN PGA IN DATABASE MEMORY, AND CAN ACCESS ITS OWN PGA ONLY. NO USER CAN ACCESS ANOTHER USER'S PGA.
- USER PGAs ARE SESSION SPECIFIC.
- DATABASE MEMORY ALSO HAS SGA. SGA ARE ACCESSIBLE TO ALL USERS.



A PACKAGE IS LOADED INTO THE SGA AND PGA PARTIALLY WHEN CALLED FOR THE FIRST TIME BY ANY USER.

- PUBLIC VARIABLES AND OBJECTS ARE STORED SEPARATELY IN ALL USER PGA.
- PRIVATE VARIABLES, OBJECTS AND SUBPROGRAMS ARE STORED INTO SGA ONCE.
- WHEN A PREVIOUSLY CALLED PACKAGE IN A SESSION IS RECOMPILED, IT IS RELOADED IN THE MEMORY ON FIRST CALL AFTER COMPILATION. THEREFORE ALL OBJECTS IN SGA AND PGA GETS REFRESHED.
- PACKAGE SUBPROGRAMS ARE STORED IN SGA SINCE IT CANNOT BE CHANGED.

THE VARIABLES AND OTHER OBJECTS CAN BE FORCE STORED IN SGA BY USING PRAGMA SERIALLY_REUSABLE

- SERIALLY REUSABLE PACKAGES CANNOT BE ACCESSED FROM TRIGGERS.
- SERIALLY REUSABLE PACKAGES CANNOT BE ACCESSED FROM SUBPROGRAMS CALLED FROM SQL STATEMENTS.
- CHANGES TO SERIALLY-REUSABLE-PACKAGE-VARIABLES ARE TEMPORARY AND PERSIST TILL EXECUTION OF BLOCK IN WHICH CHANGES HAVE BEEN MADE. HOWEVER, THESE TEMPORARY CHANGES WILL NEVER BE EXPERIENCED BY OTHER USER EVEN IF THE BLOCK IS UNDER EXECUTION.

PRAGMA_SERIALLY_REUSABE_EXAMPLE

```
CREATE PACKAGE PKG IS
       TKT VARCHAR2(30) := 'UNTOUCHED';
END;
```

REGULAR PACKAGE

```
CREATE PACKAGE SR_PKG IS
       PRAGMA SERIALLY_REUSABLE;
       TKT VARCHAR2(30) := 'UNTOUCHED';
END;
```

SERIALLY_REUSABLE PACKAGE

```
BEGIN
       PKG.TKT := 'TOUCHED';
       DBMS_OUTPUT.PUT_LINE(PKG.TKT);
END;
/
BEGIN
       DBMS_OUTPUT.PUT_LINE(PKG.TKT);
END;
```

REGULAR PACKAGE OUTPUT
>>> TOUCHED
>>> TOUCHED

```
BEGIN
       SR_PKG.TKT := 'TOUCHED';
       DBMS_OUTPUT.PUT_LINE(SR_PKG.TKT);
END;
/
BEGIN
       DBMS_OUTPUT.PUT_LINE(SR_PKG.TKT);
END;
```

SERIALLY REUSABLE PACKAGE OUTPUT
>>> TOUCHED
>>> UNTOUCHED

TRIGGERS

NAMED PLSQL BLOCKS STORED IN DATABASE THAT EXECUTES BEFORE OR AFTER OR INSTEAD OF SPECIFIC EVENT.

TRIGGERS

- TABLE
  - COMPOUND TRIGGER — DML: INSERT, UPDATE, DELETE
  - BEFORE [S] [R] — DML: INSERT, UPDATE, DELETE
  - AFTER [S] [R] — DML: INSERT, UPDATE, DELETE
- VIEW
  - INSTEAD OF [R] — DML: INSERT, UPDATE, DELETE
  - COMPOUND TRIGGER — DML: INSERT, UPDATE, DELETE
- SCHEMA
  - BEFORE — DDL: DROP, CREATE, ALTER, RENAME, TRUNCATE, COMMENT, GRANT, REVOKE; OTHER: LOGOFF
  - AFTER — DDL: DROP, CREATE, ALTER, RENAME, TRUNCATE, COMMENT, GRANT, REVOKE; OTHER: LOGON
- DATABASE
  - BEFORE — DDL: DROP, CREATE, ALTER, RENAME, TRUNCATE, COMMENT, GRANT, REVOKE; OTHER: LOGOFF, SHUTDOWN
  - AFTER — DDL: DROP, CREATE, ALTER, RENAME, TRUNCATE, COMMENT, GRANT, REVOKE; OTHER: LOGON, STARTUP, SERVER ERROR

TRIGGER TIMING EVENT BASE_OBJECT QUALIFIERS LEVEL ORDERING STATE RESTRICTION DECLARATION

```
CREATE [OR REPLACE] TRIGGER TRIGGER_NAME
BEFORE | AFTER | INSTEAD OF
INSERT | UPDATE [OF COLUMN_NAME] | DELETE
ON OBJECT_NAME
[REFERENCING OLD AS OLD_NAME NEW AS NEW_NAME]
[FOR EACH ROW]
[FOLLOWS OTHER_TRIGGER_NM | PRECEDES OTHER_TRIGGER_NM]
[ENABLE | DISABLE]
[WHEN CONDITION-STATEMENT]
[DECLARE DECLARATION_STATEMENT ;]
BEGIN
        TRIGGER_BODY;
[EXCEPTION EXCEPTION_HANDLING;]
END;
```

- ONE OR MORE COLUMNS CAN BE DEFINED INSIDE UPDATE OF EVENT IN TRIGGER DEFINITION.
- IF QUALIFIER ALIASES ARE DEFINED IN TRIGGER DEFINITION, THEN ONLY DEFINED ALIASES MUST BE USED INSIDE TRIGGER BODY.
- TRIGGER BODY (BEGIN-END) CAN BE REPLACED BY PROCEDURE AS CALL PROCEDURE_NM;
- TRIGGER IS CALLED STATEMENT LEVEL TRIGGER IF FOR EACH ROW IS NOT USED IN TRIGGER DEFINITION.
- TRIGGER IS CALLED ROW LEVEL TRIGGER IF FOR EACH ROW IS USED IN TRIGGER DEFINITION.
- EXECUTION ORDER OF TRIGGERS HAVING SAME TIMING CAN BE DEFINED WITH THE HELP OF FOLLOWS OR PRECEDES.
- STATEMENT LEVEL TRIGGERS ARE FIRED BEFORE AND AFTER COMPLETION OF AN EVENT. THEREFORE, QUALIFIERS ARE NOT AVAILABLE FOR STATEMENT LEVEL TRIGGERS.
- ROW-LEVEL TRIGGERS ARE FIRED ON RETRIEVAL OF EACH ROW, THEREFORE, ROW LEVEL TRIGGERS HAVE :OLD AND :NEW QUALIFIERS AVAILABLE.
- A COLUMN CAN BE READ OR WRITTEN WITH QUALIFIERS, IN TRIGGERS.
- A COLUMN CANNOT BE READ OR WRITTEN WITHOUT QUALIFIERS.
- A COLUMN CANNOT BE MODIFIED IN AFTER TRIGGERS.
- COLON PREFIX SHOULD NOT BE USED WITH QUALIFIER NAMES IN WHEN CONDITION OF TRIGGER DEFINITION.
- WHEN WHEN CONDITION OF TRIGGER IS FALSE, THE TRIGGER IS FIRED BUT NOT EXECUTED.
- STATEMENT LEVEL TRIGGER FIRE ON DML STATEMENTS IRRESPECTIVE OF NUMBER OF ROWS MODIFIED.
- ROW-LEVEL TRIGGER FIRE ON DML STATEMENTS RESPECTIVE OF NUMBER OF ROWS MODIFIED.

CONDITIONAL PREDICATES

```
CREATE TRIGGER TRIGGER_NAME1
BEFORE INSERT OR UPDATE OR DELETE ON TABLE_NAME1
FOR EACH ROW
BEGIN
    IF    INSERTING THEN DBMS_OUTPUT.PUT_LINE('INSERT [' || :OLD.NAME || '-> ' || :NEW.NAME || ']');
    ELSIF UPDATING THEN DBMS_OUTPUT.PUT_LINE('UPDATE [' || :OLD.NAME || '-> ' || :NEW.NAME || ']');
    ELSIF DELETING THEN DBMS_OUTPUT.PUT_LINE('DELETE [' || :OLD.NAME || '-> ' || :NEW.NAME || ']');
    END IF;
END;
```

```
INSERT INTO TABLE_NAME1 VALUES (1,'ERWIN');
UPDATE TABLE_NAME1 SET NAME = 'SMITH';
DELETE FROM TABLE_NAME1;
```

OUTPUT

```
>>> INSERT [      -> ERWIN ]
>>> UPDATE [ERWIN -> SMITH ]
>>> DELETE [SMITH -> ]
```

```
CREATE TRIGGER TRIGGER_NAME2
BEFORE UPDATE OF ID,NAME ON TABLE_NAME2
FOR EACH ROW
BEGIN
    IF    UPDATING('ID')   THEN DBMS_OUTPUT.PUT_LINE('ID MODIFIED TO' || :NEW.ID);
    ELSIF UPDATING('NAME') THEN DBMS_OUTPUT.PUT_LINE('NAME MODIFIED TO' || :NEW.NAME); END IF;
END;
```

ID	NAME
1	A
2	B

```
UPDATE TABLE_NAME2 SET ID = 9 WHERE ID = 2;
UPDATE TABLE_NAME2 SET NAME = 1 WHERE ID = 3;
```

OUTPUT

```
>>> ID MODIFIED TO 9
>>> NAME MODIFIED TO 1
```

TRIGGER TIMINGS

ID	NAME
1	WARFARE1
2	WARFARE2
3	WARFARE3

```
CREATE TABLE TABLE_NAME (ID NUMBER, NAME
INSERT INTO TABLE_NAME VALUES (1,'WARFARE1');
INSERT INTO TABLE_NAME VALUES (2,'WARFARE2');
INSERT INTO TABLE_NAME VALUES (3,'WARFARE3');
```

```
CREATE TRIGGER BST
BEFORE INSERT OR UPDATE OR DELETE ON TABLE_NAME
BEGIN
        DBMS_OUTPUT.PUT_LINE('BEFORE STATEMENT TRIGGER');
END;
```

```
CREATE TRIGGER BRT
BEFORE INSERT OR UPDATE OR DELETE ON TABLE_NAME
FOR EACH ROW
BEGIN
        DBMS_OUTPUT.PUT_LINE(:OLD.ID||' '||:NEW.NAME||'<-'||:OLD.NAME);
END;
```

```
CREATE TRIGGER ART
AFTER INSERT OR UPDATE OR DELETE ON TABLE_NAME
REFERENCING OLD AS O NEW AS N
FOR EACH ROW
BEGIN
        DBMS_OUTPUT.PUT_LINE(:N.ID||' '||:N.NAME||'<-'||:O.NAME);
END;
```

```
CREATE TRIGGER AST
AFTER INSERT OR UPDATE OR DELETE ON TABLE_NAME
BEGIN
        DBMS_OUTPUT.PUT_LINE('AFTER STATEMENT TRIGGER');
END;
```



```
UPDATE TABLE_NAME SET NAME = 'UNIMOD1' WHERE ID = 1;
```

ID	NAME
1	UNIMOD1
2	WARFARE2
3	WARFARE3

```
OUTPUT
>>> BEFORE STATEMENT TRIGGER
>>> 1 UNIMOD1 <- WARFARE1
>>> 1 UNIMOD1 <- WARFARE1
>>> AFTER STATEMENT TRIGGER
```

```
UPDATE TABLE_NAME SET NAME = 'UNIMOD2' WHERE ID = 2;
```

ID	NAME
1	UNIMOD1
2	UNIMOD2
3	WARFARE3

```
OUTPUT
>>> BEFORE STATEMENT TRIGGER
>>> 2 UNIMOD2 <- WARFARE2
>>> 2 UNIMOD2 <- WARFARE2
>>> AFTER STATEMENT TRIGGER
```

```
UPDATE TABLE_NAME SET NAME = 'UNIMOD9' WHERE ID = 9;
```

ID	NAME
1	UNIMOD1
2	UNIMOD2
3	WARFARE3

```
OUTPUT
>>> BEFORE STATEMENT TRIGGER
>>> AFTER STATEMENT TRIGGER
```

```
UPDATE TABLE_NAME SET NAME = 'MULTIMOD';
```

ID	NAME
1	MULTIMOD
2	MULTIMOD
3	MULTIMOD

```
OUTPUT
>>> BEFORE STATEMENT TRIGGER
>>> 1 MULTIMOD <- UNIMOD1
>>> 1 MULTIMOD <- UNIMOD1
>>> 2 MULTIMOD <- UNIMOD2
>>> 2 MULTIMOD <- UNIMOD2
>>> 3 MULTIMOD <- WARFARE3
>>> 3 MULTIMOD <- WARFARE3
>>> AFTER STATEMENT TRIGGER
```

INSTEAD OF TRIGGER

INSTEAD OF TRIGGER CAN ONLY BE CREATED ON VIEWS.
QUALIFIERS WORK NORMALLY FOR INSTEAD OF TRIGGER.
THE DML OPERATION FOR WHICH INSTEAD OF TRIGGER IS FIRED, DOES NOT EXECUTE.
INSTEAD OF TRIGGERS ARE GENERALLY USED FOR UPDATING COMPLEX VIEWS.

ID	NAME	VCTYPE
1	A	V
2	E	V
3	I	V
4	O	V
5	U	V

```
CREATE VIEW VIEW_NAME AS
SELECT * FROM TABLE_NAME
```

IN ORDER TO MODIFY A COMPLEX VIEW, INSTEAD OF TRIGGER CAN BE USED TO UPDATE TABLES THAT DEFINE COMPLEX VIEW.

```
CREATE OR REPLACE TRIGGER TRIGGER_NAME
INSTEAD OF INSERT OR UPDATE OR DELETE ON VIEW_NAME
BEGIN
        DBMS_OUTPUT.PUT_LINE('INSTEAD OF DML TRIGGER FIRED');
END;
```

```
UPDATE VIEW_NAME SET VCTYPE = 'UNKNOWN' WHERE VCTYPE = 'V';
```

ID	NAME	VCTYPE
1	A	V
2	E	V
3	I	V
4	O	V
5	U	V

OUTPUT
>>> INSTEAD OF DML TRIGGER FIRED
>>> INSTEAD OF DML TRIGGER FIRED
>>> INSTEAD OF DML TRIGGER FIRED
>>> INSTEAD OF DML TRIGGER FIRED
>>> INSTEAD OF DML TRIGGER FIRED

- WHETHER FOR EACH ROW IS MENTIONED OR NOT, AN INSTEAD OF TRIGGER IS ALWAYS A ROW-LEVEL-TRIGGER.

COMPOUND TRIGGERS

- COMPOUND TRIGGERS CAN BE CREATED ON TABLES.
- COMPOUND TRIGGERS CAN BE CREATED ON VIEWS.
- COMPOUND TRIGGERS ARE GENERALLY USED FOR HANDLING MUTATING TABLE ERROR.

```
CREATE [OR REPLACE] TRIGGER TRIGGER_NAME
FOR INSERT | UPDATE [OF COLUMN_NAME] | DELETE
ON OBJECT_NAME
[REFERENCING OLD AS
[FOLLOWS OTHER_TRIGGER | PRECEDES OTHER_TRIGGER
[ENABLE | DISABLE]
[WHEN CONDITION]
COMPOUND TRIGGER

        [DECLARATION_SECTION]

        BEFORE STATEMENT IS
        BEGIN
                EXECUTION_SECTION
        END BEFORE STATEMENT;

        BEFORE EACH ROW IS
        BEGIN
                EXECUTION_SECTION
        END BEFORE EACH ROW;

        AFTER EACH ROW IS
        BEGIN
                EXECUTION_SECTION
        END AFTER EACH ROW;

        AFTER STATEMENT IS
        BEGIN
                EXECUTION_SECTION
        END AFTER STATEMENT;

END;
```

```
INSTEAD OF EACH ROW IS
BEGIN
        EXECUTION_SECTION
END INSTEAD OF EACH ROW;
```

- COMPOUND TRIGGER CAN HAVE MINIMUM 1 AND MAXIMUM 4 SECTIONS.
- ORDER OF SECTIONS CAN BE SHUFFLED.
- EACH SECTION CAN HAVE ITS OWN EXCEPTION SECTION.

Shreem

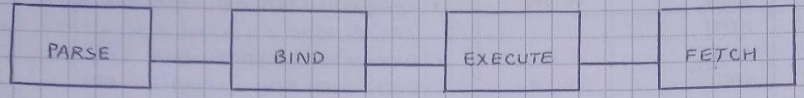
MUTATING TABLE ERROR

A TABLE BEING MODIFIED IS CALLED AS A MUTATING ~~ERROR~~ TABLE.

- MUTATING TABLE OCCUR WHEN ROW-LEVEL DML TRIGGER CONTAINS QUERY ON ITS BASE-OBJECT (BASE-TABLE)
- MUTATING TABLE OCCUR WHEN STATEMENT LEVEL TRIGGER IS FIRED AS A RESULT OF ON DELETE CASCADE.
- MUTATING TABLE ERROR DOES NOT OCCUR IF A TRIGGER CONTAINS QUERY ON ITS BASE-TABLE BECAUSE STATEMENT LEVEL TRIGGER RUNS BEFORE AND AFTER COMPLETE TABLE TRANSACTIONS.
- MUTATING TABLE ERROR OCCURS IF A TRIGGER CONTAINS QUERY ON ITS BASE TABLE BECAUSE ROW-LEVEL TRIGGERS RUN BEFORE AND AFTER INCOMPLETE TABLE TRANSACTIONS.
- VIEWS BEING MODIFIED BY INSTEAD OF TRIGGER ARE NOT CONSIDERED MUTATING BECAUSE TRIGGERING EVENTS ARE NOT EXECUTED WITH INSTEAD OF TRIGGERS.
- COMPOUND TRIGGERS ARE USED TO HANDLE MUTATING TABLE ERRORS BY PERFORMING QUERIES ON BASE OBJECT (TABLE) WITH STATEMENT LEVEL TRIGGER SECTIONS AND PASSING OBJECTS TO ROW-LEVEL TRIGGER SECTIONS.
- IF A TRIGGER CONTAINS DML OPERATION SAME AS ITS FIRING EVENT, IT MAY CAUSE RECURSIVE INFINITE LOOPS.

Shreem

DYNAMIC SQL AND PLSQL



PARSE	BIND	EXECUTE	FETCH
VALIDATION OF SYNTAX AND PRIVILEGES	BINDING VALUES IF QUERY CONTAINS BIND VARIABLES	STATEMENT EXECUTION	ROW RETRIEVAL FROM DISKS TO USER (SELECT)

DYNAMIC SQL / PLSQL

- NATIVE DYNAMIC SQL
  - EXECUTE IMMEDIATE STATIC
  - EXECUTE IMMEDIATE DYNAM USING BIND_ARG
  - EXECUTE IMMEDIATE SEL_S BULK COLLECT INTO VAR
  - EXECUTE IMMEDIATE SEL_S INTO VAR
  - EXECUTE IMMEDIATE SEL_S INTO VAR USING BIND_ARG
  - EXECUTE IMMEDIATE SEL_S BULK COLLECT INTO VAR USING BIND_ARG
  - EXECUTE IMMEDIATE I_U_D RETURNING ~~INTO VAR~~
  - EXECUTE IMMEDIATE I_U_D USING BIND_ARG RETURNING INTO VAR
  - EXECUTE IMMEDIATE I_U_D RETURNING BULK COLLECT INTO VAR
  - EXECUTE IMMEDIATE I_U_D USING BIND_ARG RETURNING BULK COLLECT INTO VAR
  - OPEN-FOR, FETCH AND CLOSE STATEMENT
- NATIVE DYNAMIC PLSQL
  - EXECUTE IMMEDIATE STATIC;
  - EXECUTE IMMEDIATE DYNAMIC; USING BIND_ARG
- DBMS_SQL PACKAGE

- STATIC QUERIES ARE PARSED AT COMPILE TIME.
- DYNAMIC STATEMENTS ARE CONSTRUCTED AS STRINGS AND EXECUTED DYNAMICALLY AT RUNTIME.
- BIND VARIABLES CAN BE USED IN DYNAMIC STATEMENTS
- NATIVE DYNAMIC SQL HANDLES IN BIND_ARGUMENTS SUPPLIED IN POSITIONED MANNER.
- NATIVE DYNAMIC PLSQL HANDLES IN, IN OUT, OUT BIND_ARGUMENTS SUPPLIED IN POSITIONED MANNER.

DYNAMIC SQL EXAMPLES

ID	CHARACTERS	NAME	VCTYPE
01	A	APPLE	V
02	B	BALL	C
03	C	CAT	C
04	D	DOG	C
05	E	ELEPHANT	V
06	F	FISH	C
07	G	GRAPES	C
08	H	HORSE	C
09	I	ICE-CREAM	V
10	J	JOKER	C
11	K	KITE	C
12	L	LION	C
13	M	MANGO	C
14	N	NEST	C
15	O	ORANGE	V
16	P	PARROT	C
17	Q	QUEEN	C
18	R	RABBIT	C
19	S	SUN	C
20	T	TIGER	C
21	U	UMBRELLA	V
22	V	VAN	C
23	W	WATER	C
24	X	XEROX	C
25	Y	YAK	C
26	Z	ZEBRA	C

- CONCATENATED DYNAMIC STRINGS ARE PARSED ON EACH RUN.
- UNCONCATENATED DYNAMIC STRINGS EVEN WITH BIND VARIABLES ARE PARSED ONCE DURING COMPILE.
- BIND VARIABLES CANNOT BE PASSED FOR OBJECT NAMES IN DYNAMIC STRING.
- DYNAMIC SQL STRING MUST NOT CONTAIN SEMICOLON AT END.
- DYNAMIC SQL STRING SUPPORTS IN MODE BIND VARIABLES ONLY.
- VARIABLES CAN BE OBTAINED BY RETURNING CLAUSE AS SHOWN IN EXAMPLES.
- BIND VARIABLES CANNOT BE NULL.
- CURSOR CANNOT BE CLOSED ONCE CODE EXECUTION TERMINATES.

INTO CLAUSE IS USED FOR SINGLE ROW QUERIES. BULK COLLECT INTO CLAUSE IS USED FOR MULTIPLE ROW QUERIES.

USING CLAUSE IS USED TO ASSIGN BIND VARIABLES TO DYNAMIC STRINGS IN POSITIONED ORDER, IRRESPECTIVE OF BIND_VARIABLE_NAMES.

```
DECLARE
        TYPE   CUR_TYPE IS REF CURSOR;
        SOLID  CUR_TYPE;
        BOX    TABLE_NAME%ROWTYPE;
BEGIN
        OPEN SOLID FOR 'SELECT * FROM TABLE_NAME WHERE VCTYPE = :BV_NM' USING 'V';
        FETCH SOLID INTO BOX;   DBMS_OUTPUT.PUT_LINE(BOX.NAME);
        FETCH SOLID INTO BOX;   DBMS_OUTPUT.PUT_LINE(BOX.NAME);
        FETCH SOLID INTO BOX;   DBMS_OUTPUT.PUT_LINE(BOX.NAME);
        FETCH SOLID INTO BOX;   DBMS_OUTPUT.PUT_LINE(BOX.NAME);
        FETCH SOLID INTO BOX;   DBMS_OUTPUT.PUT_LINE(BOX.NAME);
        CLOSE SOLID;
END;
```


DYNAMIC SQL EXAMPLES

```
BEGIN
        EXECUTE IMMEDIATE 'UPDATE TABLE_NAME
                           SET NAME = :BV_NM1
                           WHERE CHARACTERS = :BV_NM2'
        USING 'ERWIN','E';
END;
```

```
SQL%ISOPEN   : F
SQL%FOUND    : T
SQL%NOTFOUND : F
SQL%ROWCOUNT : 1
```

```
DECLARE
        TYPE NTB_TYP IS TABLE OF VARCHAR2(30);
        BOX  NTB_TYP
BEGIN
        EXECUTE IMMEDIATE 'SELECT NAME
                           FROM TABLE_NAME
                           WHERE VCTYPE = :BV_NM'
        BULK COLLECT INTO BOX
        USING 'V';
        FOR i IN 1..BOX.LAST
        LOOP
                DBMS_OUTPUT.PUT_LINE(BOX(i));
        END LOOP;
END;
```

```
OUTPUT
>>> APPLE
>>> BAEPHANT
>>> ICE-CREAM
>>> ORANGE
>>> UMBRELLA

SQL%ISOPEN   F
SQL%FOUND    T
SQL%NOTFOUND F
SQL%ROWCOUNT 5
```

```
DECLARE
        TYPE NTB_TYPE IS TABLE OF VARCHAR2(30);
        BOX  NTB_TYPE;
BEGIN
        EXECUTE IMMEDIATE 'UPDATE TABLE_NAME
                           SET NAME = "V_"||NAME
                           WHERE VCTYPE = :BV_NM1
                           RETURNING NAME INTO :BV_NM2'
        USING 'V'
        RETURNING BULK COLLECT INTO BOX;
        FOR i IN 1..BOX.COUNT
        LOOP
                DBMS_OUTPUT.PUT_LINE(BOX(i));
        END LOOP;
END;
```

```
OUTPUT
>>> V_APPLE
>>> V_ELEPHANT
>>> V_ICE-CREAM
>>> V_ORANGE
>>> V_UMBRELLA

SQL%ISOPEN   F
SQL%FOUND    T
SQL%NOTFOUND F
SQL%ROWCOUNT 5
```

Shreem

DYNAMIC PLSQL EXAMPLES

```
DECLARE
        BOX VARCHAR2(30);
BEGIN
        EXECUTE IMMEDIATE q'[ DECLARE
                                BOX1 VARCHAR2(30) := 'HELLO';
                                BOX2 VARCHAR2(30) := 'FROM DYNAMIC
                                                     PLSQL';
                              BEGIN
                                DBMS_OUTPUT.PUT_LINE(BOX1||' '||BOX2);
                              END;]';
END;
```

OUTPUT >>> HELLO FROM DYNAMIC PLSQL.

```
DECLARE
        BOX VARCHAR2(30);
BEGIN
        EXECUTE IMMEDIATE q'[ DECLARE
                                BOX1 VARCHAR2(30) := :BV1;
                                BOX2 VARCHAR2(30) := :BV2;
                              BEGIN
                                :BV3 := BOX1 || BOX2;
                              END;]'
        USING 'HELLO','FROM DYNAMIC PLSQL', OUT BOX;
        DBMS_OUTPUT.PUT_LINE(BOX);
END;
```

OUTPUT >>> HELLO FROM DYNAMIC PLSQL.

Shreem

DBMS_SQL PACKAGE

DBMS_SQL PACKAGE IS USED WHEN EXECUTING THE SAME SQL STATEMENT MULTIPLE TIMES WITH DIFFERENT BIND VALUES, OR WHEN THE DYNAMIC STATEMENT HAS AN UNKNOWN NUMBER OF BIND VARIABLES OR COLUMNS UNTIL RUNTIME.

OPEN *	F	OPEN_CURSOR( ); OPENS A NEW CURSOR AND RETURNS THE ROW_ID
PARSE *	P	PARSE(CURSOR_ID, STATEMENT, EDITION) PARSES THE DYNAMIC SQL/PLSQL STATEMENT
EXECUTE *	F	EXECUTE(CURSOR_ID) EXECUTES THE STATEMENTS AND RETURN THE NUMBER OF ROWS PROCESSED.
FETCH	F	FETCH_ROWS(CURSOR_ID) RETRIEVES THE NEXT ROW.
CLOSE	P	CLOSE_CURSOR(CURSOR_ID) CLOSES THE GIVEN CURSOR.

P	DEFINE_COLUMN(CURSOR_ID, POSITION, COLUMN)	Adds columns to column list
P	BIND_VARIABLE(CURSOR_ID, NAME, VALUE)	Binds value to the query.
P	COLUMN_VALUE(CURSOR_ID, POSITION, VALUE)	Retrieves the selected column value.

```
DECLARE
        CURSOR_ID   NUMBER;
        D_RESULT    NUMBER;
BEGIN
        CURSOR_ID := DBMS_SQL.OPEN_CURSOR;
        DBMS_SQL.PARSE(CURSOR_ID,'SELECT * FROM TABLE_NAME',DBMS_SQL.NATIVE);
        D_RESULT := DBMS_SQL.EXECUTE(CURSOR_ID);
        DBMS_SQL.CLOSE_CURSOR(CURSOR_ID);
END;
```

Shreem

COMPOSITE DATATYPES

- **RECORDS**
  - SINGLE ROW MULTIPLE VARIABLE VALUES, WITH DIFFERENT DATATYPES.
- **COLLECTIONS**
  - MULTIPLE ROW VALUES OF SAME DATATYPE
    - **NESTED TABLES**
      - UNBOUNDED LIMIT
      - SEQUENCED INDEX
    - **VARRAYS**
      - BOUNDED LIMIT
      - SEQUENCED INDEX
    - **ASSOCIATIVE ARRAYS**
      - UNBOUNDED LIMIT
      - NON-SEQUENCED INDEX

RECORDS

	DATATYPE1	DATATYPE2	DATATYPE3	DATATYPE2	DATATYPE7
RECORD_NAME	VALUE1	VALUE2	VALUE3	VALUE4	VALUE9

- A RECORD CAN HAVE ANOTHER RECORD INSIDE.
- A RECORD CAN BE CREATED FROM A TABLE STRUCTURE USING %ROWTYPE.
- A RECORD TYPE CAN BE CREATED AS FOLLOWS AND IS USED TO DEFINE A RECORD OBJECT.

```
TYPE TYPE_NAME IS RECORD ( VARIABLE_NAME DATATYPE , ..... );
```

```
REC_NM TYPE_NAME;
```

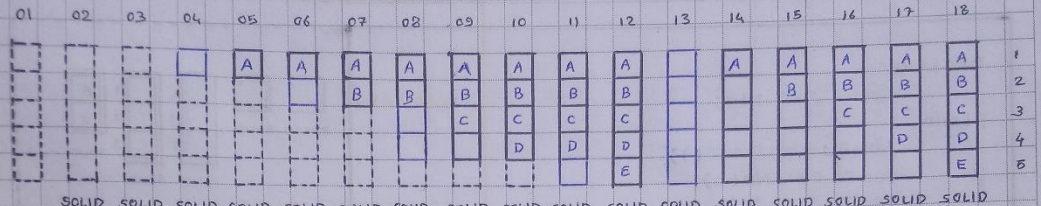
- RECORD VALUES CAN BE ACCESSED AS: RECORD_NAME.FIELDNAME
- OBJECTS ARE SIMILAR TO RECORD, BUT USED AT SCHEMA LEVEL.
- RECORDS CAN BE USED IN DML OPERATION TO PASS ALL VALUES IN ONE GO.

Shreem

VARRAYS

SINGLE DIMENTIONAL ARRAY OF A PARTICULAR DATATYPE SEQUENTIALLY INDEXED FROM 1 TO A DEFINED UPPER LIMIT.

- VARRAY ARE NULL BY DEFAULT.
- MAXIMUM SIZE OF VARRAY IS 2GB.
- TYPE ASSIGNMENT TO A VARRAY VARIABLE IS CALLED VARRY INITIALIZATION.
- INDEX CANNOT BE DELETED.




```
CREATE TYPE VARR_TYP IS VARRAY(5) OF VARCHAR2(3);
```

```
DECLARE
        SOLID VARR_TYP := VARR_TYP();                    03
BEGIN
        SOLID.EXTEND;                                    04
        SOLID(1) := 'A';                                 05
        SOLID.EXTEND;                                    06
        SOLID(2) := 'B';                                 07
        SOLID.EXTEND(2);                                 08
        SOLID(3) := 'C';                                 09
        SOLID(4) := 'D';                                 10
        SOLID.EXTEND;                                    11
        SOLID(5) := 'E';                                 12
END;
```

```
DECLARE
        SOLID VARR_TYP;                                  02
BEGIN
        SOLID := VARR_TYP();                             03
        SOLID.EXTEND(5);                                 13
        SOLID(1) := 'A';                                 14
        SOLID(2) := 'B';                                 15
        SOLID(3) := 'C';                                 16
        SOLID(4) := 'D';                                 17
        SOLID(5) := 'E';                                 18
END;
```

```
DECLARE
        SOLID VARR_TYP := VARR_TYP('A','B','C','D','E');  18
BEGIN
        NULL;
END;
```

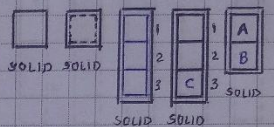
NESTED TABLES

NESTED TABLES ARE SIMILAR TO VARRAYS EXCEPT FOR THE FACT THAT.

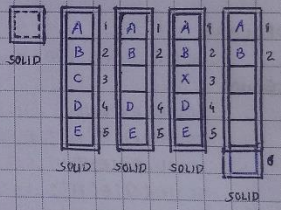
- NESTED TABLES ARE UNBOUNDED
- INDEX CAN BE DELETED.

```
CREATE TYPE NTB_TYP IS TABLE OF VARCHAR2(3);
```

```
DECLARE
        SOLID NTB_TYP;
BEGIN
        SOLID := NTB_TYP();
        SOLID.EXTEND(3);
        SOLID(3) := 'C';
        SOLID := NTB_TYP('A','B');
END;
```



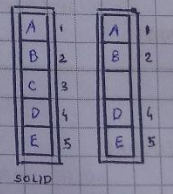

```
DECLARE
        SOLID NTB_TYP := NTB_TYP();
BEGIN
        SOLID := NTB_TYP('A','B','C','D','E');
        SOLID.DELETE(3);
        SOLID(3) := 'X';
        SOLID.DELETE(3,5);
        SOLID.EXTEND;
END;
```




```
DECLARE
        SOLID NTB_TYP := NTB_TYP('A','B','C','D','E');
BEGIN
        SOLID.DELETE(3);
        DBMS_OUTPUT.PUT_LINE(SOLID.NEXT(2));
        DBMS_OUTPUT.PUT_LINE(SOLID.PRIOR(4));
END;
```

OUTPUT

>>> 4
>>> 2



- TYPE ASSIGNMENT TO A NESTED TABLE VARIABLE IS CALLED INITIALIZATION.