An Introduction to Oracle

Victor Matos

**SQL * Plus**

SQL*Plus is a user-friendly-interface imitating the Windows notepad. Its primary use is supporting the creation of server-side schema objects.

- Normally used to easily interact with the Oracle server
- Facilitates the prototyping of SQL commands
- Quick and dirty way of developing/testing PL-SQL and other Oracle objects
ORACLE Objects

VIEWS

• A view is a tailored presentation of the data contained in one or more tables or other views.
• A view takes the output of a query and treats it as a table.
• A view can be thought of as a stored query or a virtual table.
• You can use views in most places where a table can be used.

For example

the employees table has several columns and numerous rows of information. If you want users to see only some of these columns or only specific rows, then you can create a view of that table for other users to access.

CREATE VIEW employees_view AS
SELECT SSN, Fname || ' ' || Lname as Name, Salary, Dno, Dname
FROM Employee JOIN Department
ON (Employee.Dno = Department.Dnumber)
WHERE (Salary > 30000);

SELECT * FROM employees_view;

<table>
<thead>
<tr>
<th>SSN</th>
<th>NAME</th>
<th>SALARY</th>
<th>DNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>333445555</td>
<td>Franklin Wong</td>
<td>40000</td>
<td>5</td>
<td>Research</td>
</tr>
<tr>
<td>987654321</td>
<td>Jennifer Wallace</td>
<td>43000</td>
<td>4</td>
<td>Administration</td>
</tr>
<tr>
<td>666884444</td>
<td>Ramesh Narayan</td>
<td>38000</td>
<td>5</td>
<td>Research</td>
</tr>
<tr>
<td>888665555</td>
<td>James Borg</td>
<td>55000</td>
<td>1</td>
<td>Headquarters</td>
</tr>
</tbody>
</table>
**ORACLE Objects**

**VIEWS** are often used to:

- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table.
- Hide data complexity
  - For example, a single view can be defined with a join, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables.
- Simplify statements for the user
  - For example, views allow users to select information from multiple tables without actually knowing how to perform a join.
- Present the data in a different perspective from that of the base table
  - For example, the columns of a view can be renamed without affecting the tables on which the view is based.
- Isolate applications from changes in definitions of base tables
  - For example, if a view’s defining query references three columns of a four column table, and a fifth column is added to the table, then the view’s definition is not affected, and all applications using the view are not affected.
- Express a query that cannot be expressed without using a view
  - For example, a view can be defined that joins a GROUP BY view with a table, or a view can be defined that joins a UNION view with a table.
- Save complex queries
  - For example, a query can perform extensive calculations with table information. By saving this query as a view, you can perform the calculations each time the view is queried.

---

**Using SQL *Plus* to create Objects**

**Create User Command**

Creates a database user, or an account through which you can log in to the database. You can optionally assign the following properties to the user:
- default tablespace
- temporary tablespace
- quotas for allocating space in tablespaces
- profile containing resource limits

**Example:** You can create the user MARIA by issuing the following statement:

```
CREATE USER Maria
IDENTIFIED BY Macarena
DEFAULT TABLESPACE payroll_ts
QUOTA 10M ON payroll_ts
QUOTA 5M ON temp_ts
QUOTA 5M ON system
PROFILE accountant;
Grant Connect, Resource to Maria;
```
Using SQL*Plus to create Objects

**TABLES**

- Tables are the basic unit of data storage in an Oracle database.
- Data is stored in rows and columns.
- Typical datatypes are: CHAR, VARCHAR2, NUMBER, DATE

A row is a collection of column information corresponding to a single record.

You can optionally specify rules for each column of a table. These rules are called integrity constraints. Example: NOT NULL

Once you create a table, you insert rows of data using SQL statements.

Table data can then be queried, deleted, or updated using SQL.

---

Create Table Command

To define the EMP2 table you could issue the following statement.

```sql
CREATE TABLE emp2 (  
  empno   NUMBER CONSTRAINT pk_emp PRIMARY KEY,  
  ename   VARCHAR2(10) CONSTRAINT nn_ename NOT NULL CONSTRAINT upper_ename CHECK (ename = UPPER(ename)),  
  job     VARCHAR2(9),  
  mgr     NUMBER CONSTRAINT fk_mgr REFERENCES emp2(empno),  
  hiredate DATE DEFAULT SYSDATE,  
  sal     NUMBER(10,2) CONSTRAINT ck_sal CHECK (sal > 500),  
  comm    NUMBER(9,0) DEFAULT NULL,  
  deptno  NUMBER(2) DEFAULT 20 CONSTRAINT nn_deptno NOT NULL CONSTRAINT fk_deptno REFERENCES dept(deptno) )
  PCTFREE 5
  PCTUSED 75 ;
```
Using **SQL * Plus** to create Objects

### Create Table Command

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7300</td>
<td>1985-04-01</td>
<td>800.00</td>
<td>300.00</td>
<td>20</td>
</tr>
<tr>
<td>7442</td>
<td>ADAMS</td>
<td>SALESMAN</td>
<td>7782</td>
<td>1985-04-02</td>
<td>1200.00</td>
<td>30.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JAMES</td>
<td>MANAGER</td>
<td>7782</td>
<td>1985-04-02</td>
<td>2850.00</td>
<td>200.00</td>
<td>30</td>
</tr>
</tbody>
</table>

### Notes

1. This table contains 8 columns. Ex: EMPNO column is of datatype NUMBER and has an associated integrity constraint named PK_EMP.
2. HIREDATE column is of datatype DATE and has a default value of SYSDATE.
3. PCTFREE of 5 indicates that 5% of space is reserved for future updates
4. PCTUSED of 75, says that a minimum of 75% of the table should be used on each data block
5. The constraint on the mgr field indicates that such a value (mgr’s id) must also be found in the emp2 table (a recursive reference)

---

Using **SQL * Plus** to create Objects

### Create Table Command

#### Typical data types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>Alpha-numeric, Fixed data length</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>Alphanumeric, variable-length</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Numeric data: Integers an float</td>
</tr>
<tr>
<td>DATE</td>
<td>Date data such as: ‘01-APR-97’</td>
</tr>
</tbody>
</table>
Using SQL * Plus to create Objects

DEFAULT Column Values

Table DEPT
DEPTNO | DEPTNO | MANAGER | LOC
       | DEPTNO | DEPTNO | LOC
         | DEPTNO | DEPTNO | LOC

Table EMP
EMPNO | ENAME | JOB | HIREDATE | SAL | COMM | DEPTNO

PL/SQL

PROGRAMMING
ORACLE: PL/SQL

What is PL/SQL?

• PL/SQL stands for *Procedural extensions to SQL*. It adds programming capabilities to SQL.

• It is a *block-structured* language with a syntax similar to C.

• Normally used on the *server-side*, however some *client-side products* (ORA-FORMS) could also have it as a built-in part.

• It is not a stand-alone language.

---

PL/SQL Architecture

[Diagram of PL/SQL Architecture]

Client side:
- ORACLE Forms
  - SQL Generator
  - Local PL/SQL Program
  - PL/SQL Runtime Engine

Server side:
- ORACLE Server
  - SQL Statements Processor
  - PL/SQL V2.x Runtime Engine

Third-Party Products:
- Call to Stored Procedures

Interaction:
- SQL Statements are processed by the SQL Statement Processor.
- Local PL/SQL programs can be called from the client or server.
- PL/SQL Runtime Engine executes local PL/SQL programs.
- Stored Program Units are called by the server and can be accessed by client applications.

---

13

14
PL/SQL Architecture

CLIENT SIDE CODE
- VisualBasic
- Java
- C, C++
- COBOL
- MS-Access
- Delphi
- PowerSoft
- People’s Soft
- SAP
- HTML + scripts
  (JavaScript, Perl, VBS)
- ...

ORACLE SERVER CODE
- PL-SQL (Oracle 7, 8, 9i, 10g…)
- Java (Oracle8i, …)

PL/SQL Programming

- PL/SQL = SQL + standard procedural programming
- Support for procedures, functions, flow-control statements, declared variables, user-defined types.
- A PL/SQL program is either a procedure, a function, a package, a trigger or an anonymous block
- Anonymous blocks are nameless fragments of code normally executed as script files
PL/SQL Programming

Block-Structure

- Declaration Section
- Execution Section
- Exception Section

PL/SQL: Example-1

create or replace
Procedure HELLO
is
Begin
  --my first PL/SQL program
  Dbms_Output.Put_Line ('Hello world');
End;
PL/SQL: Example-1

TESTING THE PROGRAM

... 
CSUPERSON SQL> set serverOutput on 
CSUPERSON SQL> exec hello 
Hello world

PL/SQL procedure successfully completed. 
CSUPERSON SQL>

---

PL/SQL Programming

<table>
<thead>
<tr>
<th>PL/SQL Syntax</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>x := x + 1;</td>
</tr>
<tr>
<td>Null statm.</td>
<td>Null;</td>
</tr>
<tr>
<td>Blocks</td>
<td>BEGIN ... END;</td>
</tr>
<tr>
<td>Condition Testing</td>
<td>if (condition) then</td>
</tr>
<tr>
<td></td>
<td>[elsif (condition) then]</td>
</tr>
<tr>
<td></td>
<td>[else]</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
</tr>
<tr>
<td>Loops</td>
<td>For v in -1..10 Loop ... End Loop;</td>
</tr>
<tr>
<td></td>
<td>While (cond) Loop ... End Loop</td>
</tr>
<tr>
<td></td>
<td>Loop ... End Loop</td>
</tr>
<tr>
<td>Conditions</td>
<td>&lt;, &gt;, &lt;=, =&gt;, =</td>
</tr>
<tr>
<td></td>
<td>and, or, not</td>
</tr>
</tbody>
</table>
**PL/SQL Programming**

**Defining/Initializing Variables**

```plsql
Salary NUMBER(10, 2);
DeptNumber NUMBER (3) := 17;
LastName VARCHAR2(20);
Sex CHAR(1) := ‘F’;
JobType CHAR(15);
BirthDate DATE := ‘15-APR-96’;
```

Most common data types are:
NUMBER, VARCHAR2, DATE, CHAR
A function to retrieve the employee's full name

```sql
create or replace
FUNCTION getName ( theSSN IN NUMBER ) RETURN VARCHAR2
IS
theTotal NUMBER;
theName VARCHAR2(40);
BEGIN
    select count(*) into theTotal from employee where SSN = theSSN;
    if (theTotal = 0) then
        RETURN('');
    else
        select (FName || ' ' || Lname) into theName from employee
        where SSN = theSSN;
        RETURN (theName);
    end if;
EXCEPTION
    when others then
        return ('***ERROR***');
END;
```

1- Testing the function

```sql
CSUPERSON SQL> variable xxx varchar2(40);
CSUPERSON SQL> exec :xxx := getName(123456789);

PL/SQL procedure successfully completed.

CSUPERSON SQL> print xxx

XXX

----------------------------------------
John Smith

CSUPERSON SQL>
```
2- Testing the function

SQL> select getname(123456789) from DUAL;

GETNAME(123456789)
--------------------------------------
John Smith

PL/SQL: Example-2

PL/SQL: Example-3

-- ToCELCIUS (Lowest temp, Highest Temp) Converts temperatures from Fahrenheit to Celcius.
-- Results are saved into TEMP, a predefined table to hold (text, text, text)
CREATE OR REPLACE PROCEDURE ToCelcius (Low IN NUMBER, High IN NUMBER)
IS
    Celcius number(12);
    Fahren number(12);
    Msg varchar2(40);
BEGIN
    DELETE FROM temp;
    FOR Fahren IN Low..High LOOP
        Celcius:= ROUND ( (Fahren -32)*5/9 );
        IF (Fahren < 30 ) THEN
            Msg := 'Too cold';
        ELSIF (Fahren < 70 ) THEN
            Msg := 'Cold';
        ELSIF (Fahren <80) THEN
            Msg := 'Very nice';
        ELSE
            Msg := 'Hot';
        END IF;
        INSERT INTO Temp VALUES (Fahren, Celcius, Msg);
        Dbms_Output.Put_Line (to_char(Fahren) || ' ' || to_char(Celcius) || ' ' || Msg);
    END LOOP;
END;
/
show errors /
## PL/SQL: Example-3

**Testing the procedure**

CSUPERSON SQL> `exec` toCelsius(60,77);
60 16 Cold
... 76 24 Very nice
77 25 Very nice

CSUPERSON SQL> `select * from temp;`

<table>
<thead>
<tr>
<th>F1</th>
<th>F2</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>16</td>
<td>Cold</td>
</tr>
<tr>
<td>61</td>
<td>16</td>
<td>Cold</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>25</td>
<td>Very nice</td>
</tr>
</tbody>
</table>

-- Values generated with DBMS_OUTPUT.PUT_LINE...

## PL/SQL: Example-4

**-- ASSIGN or REMOVE employees from/to projects**

```sql
create or replace
PROCEDURE AdjustWorkLoad ( p_action VARCHAR2,
p_ssn VARCHAR2,
p_pno NUMBER,
p_hours NUMBER:= 40 )
IS
BEGIN
  IF (p_action = 'ASSIGN' ) THEN
    INSERT INTO works_on VALUES (p_ssn, p_pno, p_hours);
  ELSIF (p_action = 'REMOVE') THEN
    DELETE FROM works_on
    WHERE (essn = p_ssn) AND (pno = p_pno);
  ELSE
    dbms_output.put_line ('Invalid Action ' || p_action);
  END IF;
EXCEPTION
  when others then
    dbms_output.put_line ('Problems: ');
END;
```

```plaintext```
SHOW ERROR
```
```
Testing the procedure

```sql
CSUPERSON SQL> exec AdjustWorkLoad('ASSIGN',1,1,22);
CSUPERSON SQL> exec AdjustWorkLoad('ASSIGN',1,2);

CSUPERSON SQL> select * from works_on;
ESN          PNO      HOURS
---------- ---------- ----------
123456789    1        32.5
...
1           1        22
1           2        40
CSUPERSON SQL> exec AdjustWorkLoad('REMOVE',1,2);
```

---

**PL/SQL: Example-4**

### PL/SQL Programming

#### Defining Variables

- `%TYPE` declares a variable with respect to a `Table.Column`

```sql
Bonus EMP.Sal%TYPE;
```

<table>
<thead>
<tr>
<th>EMP</th>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP1</td>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80</td>
<td>800</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>EMP2</td>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81</td>
<td>1600</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>EMP3</td>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81</td>
<td>1250</td>
<td>500</td>
<td>30</td>
</tr>
</tbody>
</table>
```

蝦皮網路: 例 4

Testing the procedure

```sql
CSUPERSON SQL> exec AdjustWorkLoad('ASSIGN',1,1,22);
CSUPERSON SQL> exec AdjustWorkLoad('ASSIGN',1,2);

CSUPERSON SQL> select * from works_on;
ESN          PNO      HOURS
---------- ---------- ----------
123456789    1        32.5
...
1           1        22
1           2        40
CSUPERSON SQL> exec AdjustWorkLoad('REMOVE',1,2);
```

---

**PL/SQL Programming**

#### Defining Variables

- `%TYPE` declares a variable with respect to a `Table.Column`

```sql
Bonus EMP.Sal%TYPE;
```

<table>
<thead>
<tr>
<th>EMP</th>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP1</td>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80</td>
<td>800</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>EMP2</td>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81</td>
<td>1600</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>EMP3</td>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81</td>
<td>1250</td>
<td>500</td>
<td>30</td>
</tr>
</tbody>
</table>
**PL/SQL Programming**

**Defining Variables**

%ROWTYPE Defines a *record* with respect to an existing table definition

TempEmp EMP%ROWTYPE;

Refer to fields using: TempEmp.Salary

---

### EMP

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80</td>
<td>800</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81</td>
<td>1600</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81</td>
<td>1250</td>
<td>500</td>
<td>30</td>
</tr>
</tbody>
</table>

---

**PL/SQL Example 5**

Using %Type and %RowType (and Cursor)

```plsql
create or replace
PROCEDURE Example5
IS
  vSSN Employee.SSN%Type;
  vEmpRec Employee%RowType;
BEGIN
  for vEmpRec in (select * from employee) loop
    vSSN := vEmpRec.SSN;
    dbms_output.put_line(to_Char(vSSN) || ' ' || vEmpRec.Fname);
  end loop;
  dbms_output.new_line;
  dbms_output.put_line('Adios');
END;
```


PL/SQL Example 5

Testing Example5
CSUPERSON SQL> exec example5

123456789 John
333445555 Franklin
999887777 Alicia
907654321 Jennifer
666884444 Ramesh
453453453 Joyce
907987987 Ahmad
888665555 James
Adios

PL/SQL Programming

• SUPPLIED PACKAGES
PL/SQL Programming

Supplied packages

- **DBMS_OUTPUT** routes output from routines
  - `Put_Line(...)` writes line + CrLf
  - `Put (...)` writes token
  - `New_Line` writes CrLf

- **DBMS_RANDOM** generates random numbers/string
  - `String(...)` returns STRING
  - `Seed(...)` starting point
  - `Value( )` returns NUMBER [0, 1)

- **UTL_FILE** programs can read and write operating system text files
  - `Fclose` `Fopen` `Fflush` `Is_Open`
  - `Put_Line` `Put` `New_line`
  - `Get_Line` ...

PL/SQL Example 6

Using the DBMS_OUTPUT package

create or replace
PROCEDURE Example6
IS

BEGIN
  for i in 1 .. 6 loop
    if MOD(i,2) = 0 then
      dbms_output.put ('Blip ');
    else
      dbms_output.put ('Blop ');
    end if;
  end loop;

  dbms_output.New_Line;
  dbms_output.put_line ('Adios');
  dbms_output.put_line ('amigos');

END;
PL/SQL Example 6

Testing Example 6
CSUPERSON SQL> exec example6
Blop Blip Blop Blip Blop Blip
Adios
amigos
PL/SQL procedure successfully completed.
CSUPERSON SQL>  

PL/SQL Example-7

Using the DBMS_RANDOM package

PROCEDURE Example7 IS
    myRandomNumber1 NUMBER(10);
    myRandomNumber2 NUMBER(10);
    myRandomString VARCHAR2(20);
BEGIN
    for i in 0..9 loop
        DBMS_OUTPUT.New_Line;
        myRandomNumber1 := DBMS_RANDOM.Value(50, 101);
        myRandomNumber2 := 101 * DBMS_RANDOM.Value;
        myRandomString := DBMS_RANDOM.String('X', 10);
        DBMS_OUTPUT.Put ( to_char(i) || ' ' );
        DBMS_OUTPUT.Put ( myRandomString || ' ' );
        DBMS_OUTPUT.Put ( to_char(myRandomNumber1) || ' ' );
        DBMS_OUTPUT.Put ( to_char(myRandomNumber2) || ' ' );
        DBMS_OUTPUT.New_Line;
    end loop;
END;
PL/SQL Example-7

Testing DBMS_RANDOM
CSUPERSON SQL> Exec Example7

0 HCOGHALDIO 54 84
1 BM6S2DKG6D 65 8
2 J6XGTPR3T5 54 21
3 NFR07KFZZU 71 25
4 YLX9C620JM 83 45
5 0CG6QBW2X 96 33
6 Y0G57QINRV 73 26
7 VT4CSCKN8S 87 17
8 BRV1S4PWLE 99 39
9 WQJAHHV5L 72 56

PL/SQL procedure successfully completed

PL/SQL Example-8

Using the UTL_FILE package

Preparation
1. Log as DBA using SYSTEM/****
2. Install the UTL_FILE package by executing (@) the script
   C:\oracle\product\10.1.0\db_1\RDBMS\ADMIN\utlFile.sql
3. Create a working directory alias issuing the following commands
   SQL> CREATE DIRECTORY WIN_DIR AS 'C:/Temp';
   SQL> GRANT READ ON DIRECTORY WIN_DIR TO PUBLIC;
   SQL> GRANT WRITE ON DIRECTORY WIN_DIR TO PUBLIC;
PL/SQL Example-8

Log as CSUPERSON/Euclid. Enter the following anonymous procedure

DECLARE
  myfile  UTL_FILE.File_Type;
  myBuffer varchar2(80);
BEGIN
  myfile := UTL_FILE.Fopen('WIN_DIR','XYZ.TXT','W');
  UTL_FILE.Put_Line(myFile, 'Uno');
  UTL_FILE.Put_Line(myFile, 'Dos');
  UTL_FILE.Put_Line(myFile, 'Tres');
  UTL_FILE.Fclose(myFile);
BEGIN
  myfile := UTL_FILE.Fopen('WIN_DIR','XYZ.TXT','R');
  Loop
    UTL_FILE.Get_Line (myfile, myBuffer);
    DBMS_OUTPUT.Put_Line(myBuffer);
  End Loop;
EXCEPTION
  when NO_DATA_FOUND then
    UTL_FILE.Fclose(myFile);
    DBMS_OUTPUT.Put_Line ('Adios ...');
END;
DBMS_OUTPUT.Put_Line ('Amigos');
END;

PL/SQL procedure successfully completed.

Note: File c:/Temp/XYZ.TXT contains

Uno
Dos
Tres
PL/SQL Programming

CURSORS

PL/SQL Cursors

• A cursor is a type of pointer built into PL/SQL to provide sequential (one row at a time) access to the rows produced by a SQL statement.

• Depending on the cursor’s life cycle and management code there are two types
  – Implicit cursors (controlled by PL/SQL on execution of DML-maintenance statements)
  – Explicit cursors (created with a CURSOR … IS select … statement)
PL/SQL Implicit Cursors

- Implicit cursors are opened/closed automatically by Oracle. No need to say OPEN /CLOSE/FETCH, etc.

- They could be created in association to the execution of
  - DML SQL code (insert, delete, update)
  - A CURSOR-FOR-LOOP
    (for (select…) loop … end loop;)

-- Implicit cursor in SQL code

```sql
PROCEDURE EXAMPLE12 AS
BEGIN
  update Project set Pname = Pname where Pnumber = -1;
  dbms_output.put_line('Records affected: ' || to_char(sql%RowCount) );
  if sql%NotFound then
    dbms_output.put_line ('No project found with given Pnumber');
  end if;

  update Project set Pname = Pname;
  dbms_output.put_line('Records affected: ' || to_char(sql%RowCount) );
END;
```

Records affected: 0
No project found with given Pnumber
Records affected: 6
PL/SQL Implicit Cursors

-- Cursor_For_Loop

PROCEDURE EXAMPLE13 AS
BEGIN
  for guy in (select fname, lname, salary
              from employee
              where sex = 'M') loop
    dbms_output.put ('Emp: ' || guy.Fname || ' ' || guy.Lname);
    dbms_output.put ('makes $' || to_char(guy.salary) );
    dbms_output.New_Line;
  end loop;
END;

EXEC example13;

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Found</td>
<td>True when last FETCH operation finds a record in the result set made by the cursor.</td>
</tr>
<tr>
<td>%NotFound</td>
<td>True when last FETCH operation does not find a record.</td>
</tr>
<tr>
<td>%RowCount</td>
<td>Returns the count of records that have been fetched from the cursor at the current point.</td>
</tr>
<tr>
<td>%IsOpen</td>
<td>True when the cursor has been opened.</td>
</tr>
</tbody>
</table>
PL/SQL Cursors Attributes

<table>
<thead>
<tr>
<th></th>
<th>%FOUND</th>
<th>%ISOPEN</th>
<th>%NOTFOUND</th>
<th>%ROWCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>before</td>
<td>exception</td>
<td>FALSE</td>
<td>exception</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td>First FETCH</td>
<td>before</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Next FETCH(es)</td>
<td>before</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Last FETCH</td>
<td>before</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>CLOSE</td>
<td>before</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>exception</td>
<td>FALSE</td>
<td>exception</td>
</tr>
</tbody>
</table>

Notes:
1. Referencing %FOUND, %NOTFOUND, or %ROWCOUNT before a cursor is opened or after it is closed raises INVALID_CURSOR.
2. After the first FETCH, if the result set was empty, %FOUND yields FALSE, %NOTFOUND yields TRUE, and %ROWCOUNT yields 0.

PL/SQL Explicit Cursors

Used with select statements returning more than one row.
Explicit Cursors are declared with the construct

```
CURSOR cursorName [(parameter(s)]
[RETURN returnType ]
IS
    someSqlStatement
```

The life-cycle of the cursor involves
- Open the cursor
- Fetch a row from the cursor
- Close the cursor
PL/SQL Explicit Cursors

Find the top three projects using the most employees of a given gender.

```sql
PROCEDURE Proj_Using_Most_People_ByGender (TheGender Employee.Sex%Type)
IS
    ProjIdWorks_On.Pno%TYPE;
    ProjCnt number;

    CURSOR Get_People_Involved (Gender IN Employee.Sex%Type)
    IS
        select w.Pno, count(*) as TotalPeople
        from Works_On w, Employee e
        where e.Sex = Gender
        and e.Ssn = w.Essn
        group by w.Pno
        order by TotalPeople desc;

    BEGIN
        open Get_People_Involved(TheGender);
        for i in 1..3 loop
            fetch Get_People_Involved into ProjId, ProjCnt;
            exit when Get_People_Involved%NOTFOUND;
        dbms_output.put_line (' Project:  ' || to_char(ProjId) ||
            ' Personnel: ' || to_char(ProjCnt) );
        end loop;
        close Get_People_Involved;
    EXCEPTION
        when others then
            dbms_output.PUT_LINE ( 'Problems' );
    END;
```

TheGender = {M, F}

Find the projects employing the most female/male workers in the company

PL/SQL Explicit Cursors

Find the three projects using the most FEMALE employees.

CSUPERSON SQL> exec Proj_Using_Most_People_ByGender ('F');
Project: 30 Personnel: 2
Project: 1 Personnel: 1
Project: 2 Personnel: 1

<table>
<thead>
<tr>
<th>PNO</th>
<th>TotalPeople</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>
PL/SQL Explicit Cursors

Find top three salaries in the company

```plsql
PROCEDURE EXAMPLE14 AS
    cursor ThreePeople is
        Select * from employee
            where rowNum <= 3
                order by salary desc;
    OneEmp employee%rowType;
    BEGIN
        open ThreePeople;
        fetch ThreePeople into OneEmp;
        loop
            if ThreePeople%FOUND then
                dbms_output.put_line('Looking at rec. no. ' ||
                    ThreePeople%RowCount ||
                    ' Name: ' || OneEmp.Lname ||
                    ' Salary: ' || OneEmp.Salary);
                fetch ThreePeople into OneEmp;
            elsif ThreePeople%NOTFOUND then
                exit; --finish the loop
            end if;
        end loop;
        if ThreePeople%ISOPEN then
            close ThreePeople;
        end if;
    END;
```

Looking at rec. no. 1 Name: Wong Salary: 40000
Looking at rec. no. 2 Name: Smith Salary: 30000
Looking at rec. no. 3 Name: Zelaya Salary: 25000

PL/SQL Ref Cursors

- Implicit and Explicit cursors are **static**. They are defined at compile time. Ref Cursors are **dynamic** in the sense they are created and processed at run-time.

- A result set could be produced by a Function or Procedure and returned using a **SYS_REFCURSOR** variable.

- A cursor variable is more **flexible** because it is not tied to a specific query.

- You can pass a cursor variable as a **parameter** to local and stored subprograms.

- Opening the cursor variable in one subprogram, and processing it in a different subprogram, helps to centralize data retrieval.

- This technique is also useful for **multi-language applications**, where a PL/SQL subprogram might return a result set to a subprogram written in a different language.
PL/SQL Ref Cursors

Get the name of each department and its corresponding manager

PROCEDURE EXAMPLE15 (Managers IN OUT SYS_REFCURSOR) AS
BEGIN
    open Managers for
    select (e.Fname || ' ' || e.Lname)  FullName, d.Dname
    from Department d, Employee e
    where (d.MgrSsn = e.SSN) ;
END;

CSUPERSON SQL> var x refCursor;
CSUPERSON SQL> exec example15(:x);
PL/SQL procedure successfully completed.
CSUPERSON SQL> print x;

<table>
<thead>
<tr>
<th>FULLNAME</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Franklin Wong</td>
<td>Research</td>
</tr>
<tr>
<td>James Borg</td>
<td>Headquarters</td>
</tr>
<tr>
<td>Jennifer Wallace</td>
<td>Administration</td>
</tr>
</tbody>
</table>

PL/SQL Ref Cursors

Return a Database table using a RefCursor variable

PROCEDURE EXAMPLE16 (theTable IN varchar2, myResult IN OUT SYS_REFCURSOR) AS
BEGIN
    if theTable = 'E' then
        open myResult for
        select * from employee;
    elsif theTable = 'D' then
        open myResult for
        select * from department;
    elsif theTable = 'P' then
        open myResult for
        select * from project;
    else
        open myResult for
        select * from works_on;
    end if;
END;

CSUPERSON SQL> var x RefCursor;
CSUPERSON SQL> exec example16('D', :x);

<table>
<thead>
<tr>
<th>DNAME</th>
<th>DNUMBER</th>
<th>MGRSSN</th>
<th>MGRSTARTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research</td>
<td>5</td>
<td>333445555</td>
<td>22-MAY-78</td>
</tr>
<tr>
<td>Administration</td>
<td>4</td>
<td>3976545321</td>
<td>01-JAN-85</td>
</tr>
<tr>
<td>Headquarters</td>
<td>1</td>
<td>888665555</td>
<td>19-JUN-71</td>
</tr>
</tbody>
</table>

CSUPERSON SQL> exec Example16('P', :x);

<table>
<thead>
<tr>
<th>PNAME</th>
<th>PNUMBER</th>
<th>PLOCATION</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductX 1 Bellaire</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ProductY 2 Sugarland</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ProductZ 3 Houston</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td>Houston</td>
<td>1</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td>Stafford</td>
<td>4</td>
</tr>
</tbody>
</table>
PL/SQL Ref Cursors

Passing Cursor Variables As Parameters

- You can declare cursor variables as the **formal parameters** of functions and procedures.

- The following example defines a REF CURSOR type, then declares a cursor variable of that type as a formal parameter:

```sql
DECLARE
    TYPE EmpCurTyp IS REF CURSOR RETURN Employee%ROWTYPE;
    emp  EmpCurTyp;
BEGIN
    -- Once we have a result set, we can process all the rows
    -- inside a single procedure rather than calling a procedure
    -- for each row.
    PROCEDURE process_emp_cv (emp_cv IN EmpCurTyp) IS
        person Employee%ROWTYPE;
    BEGIN
        dbms_output.put_line('-----');
        dbms_output.put_line('Here are the names from the result set:');
        LOOP
            FETCH emp_cv INTO person;
            EXIT WHEN emp_cv%NOTFOUND;
            dbms_output.put_line('Name = ' || person.fname ||' ' || person.lname);
        END LOOP;
    END;
END;
```

```sql
BEGIN
    -- First find 10 arbitrary employees.
    OPEN emp FOR SELECT * FROM employee WHERE ROWNUM < 11;
    process_emp_cv(emp);
    CLOSE emp;
    -- Then find employees matching a condition.
    OPEN emp FOR SELECT * FROM employee WHERE lname LIKE 'W%';
    process_emp_cv(emp);
    CLOSE emp;
END;
/```

Here are the names from the result set:
- Name = John Smith
- Name = Franklin Wong
- Name = Alicia Zelaya
- Name = Jennifer Wallace
- Name = Ramesh Narayan
- Name = Joyce English
- Name = Ahmad Jabbar
- Name = James Borg

-----

Here are the names from the result set:
- Name = Franklin Wong
- Name = Jennifer Wallace
PL/SQL Nested Cursors

A cursor could be opened inside the scope of another. This nesting of cursors provides a powerful programming tool for more complex queries.

Example:
A department number is given. Find all the employees working for this department and their corresponding assignments. Provide only one row for each employee (regardless of the total number of projects in which they work).

For instance

```
Emp SSN: 333445555 Project(s): {2, 3, 10, 20}
```

PL/SQL Example 16
Nesting loops. Employees and their SET of projects

```
PROCEDURE Example16 (DeptNumber number) IS
  EmpRec Employee%ROWTYPE;
  WorkRec Works_On%ROWTYPE;
  Message1 VARCHAR2(40);
  Message2 VARCHAR2(40);
  Comma      VARCHAR2(3) := ' ,';

  CURSOR Guys_in_Dept ( DeptNumber number ) IS
    select * from employee
    where Dno = DeptNumber
    order by ssn;

  CURSOR Working_On ( EmpSsn varchar ) IS
    select * from works_on
    where Essn = EmpSsn
    order by pno;
```
PL/SQL Example 16 cont.

-- Nesting loops. Employees and their SET of projects

```plsql
BEGIN
  FOR EmpRec IN Guys_in_Dept(DeptNumber) LOOP
    Message1 := ' Emp SSN: ' || EmpRec.Ssn;
    Message2 := ' Project(s): {';
    Comma := ' ';
    FOR WorkRec IN Working_On(EmpRec.SSN) LOOP
      Message2 := Message2 || Comma || WorkRec.Pno;
      Comma := ', ';
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(Message1 || Message2 || ' }');
  END LOOP;
EXCEPTION
  WHEN Others THEN
    DBMS_OUTPUT.PUT_LINE('Problems, ssn ' || EmpRec.Ssn);
END;
```

SQL> exec Example16(5)
Emp SSN: 123456789 Project(s): {1, 2}
Emp SSN: 333445555 Project(s): {2, 3, 10, 20}
Emp SSN: 453453453 Project(s): {1, 2}
Emp SSN: 666884444 Project(s): {3}
PL/SQL Exceptions

• PL/SQL implements error detection and processing with *Exceptions* and *Exception Handlers*.

• Errors fall into two categories
  – *Compile-Time* errors (PL/SQL compiler is responsible)
  – *Run-Time* errors (programmer is on charge)

• Undetected run-time errors produce abnormal program termination – and usually total loss of the job.

---

PL/SQL Exceptions

The *exception handler* is similar to a CASE statement in which the programmer could detect what type of error has been risen

**Syntax is**

```sql
EXCEPTION
[WHEN exceptionExpression1 THEN
  statement(s);
] [WHEN exceptionExpression2 THEN
  statement(s);
][ . . .
]
```

*exceptionExpression could be*

– An Oracle pre-defined exception
– An user-define condition
– A PRAGMA EXCEPTION_INIT exception.
PL/SQL Exceptions

- An *internal exception* is raised automatically if your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit.

- PL/SQL predefines some common Oracle errors as exceptions.
  - For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.

### PL/SQL Exceptions

PL/SQL declares predefined exceptions globally in package STANDARD (You need not declare them yourself)

<table>
<thead>
<tr>
<th>Oracle Error</th>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-0001</td>
<td>DUP_VAL_ON_INDEX</td>
<td>Unique constraint key violated</td>
</tr>
<tr>
<td>ORA-0051</td>
<td>TIMEOUT_ONRESOURCE</td>
<td>Timeout occurred while waiting for a resource</td>
</tr>
<tr>
<td>ORA-0904</td>
<td>INVALID_COLUMN_NAME</td>
<td>Illegal name</td>
</tr>
<tr>
<td>ORA-1001</td>
<td>INVALID_CURSOR</td>
<td>Illegal cursor operation</td>
</tr>
<tr>
<td>ORA_1403</td>
<td>NO_DATA_FOUND</td>
<td>No data found</td>
</tr>
<tr>
<td>ORA_1422</td>
<td>TOO_MANY_ROWS</td>
<td>A SELECT INTO statement matches more than one row</td>
</tr>
<tr>
<td>ORA_1722</td>
<td>INVALID_NUMBER</td>
<td>Conversion to a number failed.</td>
</tr>
</tbody>
</table>

PL/SQL Exceptions

User-defined-Exceptions. Ordering a cup of coffee.

PROCEDURE EXAMPLE18 (LiquidTemperature NUMBER) AS
  too_hot EXCEPTION;
  too_cold EXCEPTION;
BEGIN
  -- order a cup of coffee
  IF LiquidTemperature < 90 THEN
    RAISE too_cold;
  ELSIF LiquidTemperature > 104 THEN
    RAISE too_hot;
  ELSE
    dbms_output.put_line('Uhhh, good coffee!!!');
  END IF;
EXCEPTION
  WHEN too_cold THEN
    dbms_output.put_line('Problem - call manager and complaint');
  WHEN too_hot THEN
    dbms_output.put_line('Problem - call a good lawyer and sue');
  WHEN OTHERS THEN
    dbms_output.put_line('Problem - just complaint');
END;

PL/SQL Exceptions

User-Defined-Cond. Raise exception if video (book, bill, …) is overdue.

PROCEDURE EXAMPLE17 AS
  past_due EXCEPTION;
  due_date DATE := SYSDATE - 1;
  todays_date DATE := SYSDATE;
  acct_num NUMBER;
BEGIN
  IF due_date < todays_date THEN
    RAISE past_due;
  END IF;
EXCEPTION
  WHEN past_due THEN
    dbms_output.put_line('Problem - Handling PAST_DUE exception.');
  WHEN OTHERS THEN
    dbms_output.put_line('Problem - something bad happened');
END;
PL/SQL Exceptions

PRAGMA EXCEPTION_INIT

- **Pragmas** are pseudo-instructions telling the compiler to do something.

- **PRAGMA EXCEPTION_INIT** is a mechanism the developer could use to trick Oracle into accepting that one of the developer’s exceptions is (temporarily) replacing a pre-defined Oracle condition.

- You can use the pragma **EXCEPTION_INIT** to associate exception names with other Oracle error codes that you can anticipate.

```sql
PROCEDURE EXAMPLE19 (tableName varchar2, updColumnName varchar2, updColumnValue varchar2, keyName varchar2, keyValue varchar2)
    as
    Invalid_Column EXCEPTION;
    PRAGMA EXCEPTION_INIT (invalid_Column, -904);
    cmd_Update_Text varchar2(200);
    begin
    cmd_Update_Text :=
    'update ' || tableName ||
    '   set ' || updColumnName || ' = ''' || updColumnValue || '''' ||
    ' where ' || keyName || ' = ' || keyValue;
    execute immediate cmd_Update_Text;
    exception
    when Invalid_Column then
    dbms_output.put_line('**MY ERROR** invalid column name in command ');
    dbms_output.put_line(cmd_Update_Text);
    dbms_output.put_line('SqlCode: ' || sqlcode);
    dbms_output.put_line('SqlErrm: ' || sqlErrm);
    end;
    exec Example19('project','Pnameeee','XYZ','Pnumber', '10');
```

**MY ERROR** invalid column name in command
```
update project   set Pnameeee = 'XYZ' where Pnumber = 10
SqlCode: -904
SqlErrm: ORA-00904: "PNAMEEEE": invalid identifier
```
PL/SQL Example19 cont.

Observations

- **Dynamic PL/SQL.** Some programs must build and process SQL statements where some information is not known in advance.

- Dynamic SQL statements are made as strings at *run time*. The strings contain the *text of a SQL statement* or PL/SQL block.

- To process most *dynamic* SQL statements, you use the `EXECUTE IMMEDIATE` statement.

---

PL/SQL Example19 cont.

Apply the following changes on Example19

1. Change “Pnameeee” to “Pname” on line
   ```plsql
   exec Example19('project','Pnameeee','XYZ','Pnumber', '10');
   ```

2. Change error code “904” to “7777” on line
   ```plsql
   PRAGMA EXCEPTION_INIT (invalid_Column, -904);
   ```
PL/SQL Programming

PACKAGES

PL/SQL Packages

• A package is a schema object that groups logically related PL/SQL objects

• Packages usually have two parts, a specification and a body (sometimes the body is unnecessary).

• The specification is the interface to the package. It declares the
  – types,
  – variables,
  – constants,
  – exceptions,
  – cursors, and the
  – signature of the subprograms that can be referenced from outside the package.

• The body defines the queries for the cursors and the code for the subprograms.
PL/SQL Packages

Syntax

Package Specification

PACKAGE package_name
IS
[ declaration of public variables and user-defined-types ]
[ specification of constants, exceptions ]
[ specification of cursors ]
[ signatures of procedures and functions ]
END [package_name ];

Package body definition

PACKAGE BODY package_name
IS
[ private variables, constants, exceptions and types ]
[ CURSOR specifications and SELECT stmts ]
[ Specification of Procedures and Functions ]

[ BEGIN ]
[ Executable statements ]
[ EXCEPTION ]
[ Exception handlers ]
END [package_name ]
PL/SQL Packages

Advantages of PL/SQL Packages

- **Modularity**
  - Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module.

- **Easier Application Design**
  - When designing an application, all you need initially is the interface information in the package specs.

- **Information Hiding**
  - With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible).

- **Added Functionality**
  - Packaged public variables and cursors persist for the duration of a session.

- **Better Performance**
  - When you call a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O.

Example of a PL/SQL Package

- The example below shows the *package way* of how to support operations for the company database.

- The package contains
  - A constant, a global variable, a cursor, an exception, and subs.

- The procedures **hire_employee** and **fire_employee**
  - Check the existence of the person, verifies business rules and either executes or aborts execution by raising an exception.
PL/SQL Packages

The Package Specification (Interface)
CREATE OR REPLACE
PACKAGE emp_actions AS
  BAD_OPERATION EXCEPTION;
  myPI NUMBER CONSTANT := 3.141592;
  myStr VARCHAR2(120);
  TYPE EmpRecTyp IS RECORD (empSSN VARCHAR2(9), salary NUMBER(10,2));
  CURSOR desc_salary RETURN EmpRecTyp;

PROCEDURE hire_employee (pFNAME VARCHAR2, pMINIT VARCHAR2, pLNAME VARCHAR2, pSSN VARCHAR2, pBDATE DATE, pADDRESS VARCHAR2, pSEX VARCHAR2, pSALARY NUMBER, pSUPERSSN VARCHAR2, pDNO NUMBER);

PROCEDURE fire_Employee (pSSN NUMBER);
END
emp_actions;
/

The Package Implementation
CREATE OR REPLACE
PACKAGE BODY emp_actions AS
  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT SSN, salary
    FROM employee
    ORDER BY salary DESC;

PROCEDURE fire_Employee (pSSN NUMBER) IS
  myCount NUMBER;
BEGIN
  -- here you add logic to make sure the employee could be
  -- safely removed from the table (without violating any
  -- business rules and/or referential integrity constraints)
  select count(*) into myCount from employee where ssn = pSSN;
  if (myCount = 0) then
    -- employee not found
    RAISE emp_actions.BAD_OPERATION;
  end if;
  DELETE FROM employee WHERE SSN = pSSN;
END fire_Employee;
PL/SQL Packages

PROCEDURE hire_employee (  
pFNAME VARCHAR2  
, pMINIT VARCHAR2  
, pLNAME VARCHAR2  
, pSSN VARCHAR2  
, pBDATE DATE  
, pADDRESS VARCHAR2  
, pSEX VARCHAR2  
, pSALARY NUMBER  
, pSUPERSSN VARCHAR2  
, pDNO NUMBER  
) IS  
BEGIN  
  -- Here you add logic to validate the data and make sure  
  -- no business rules are violated  
  INSERT INTO employee VALUES (  
    pFNAME , pMINIT , pLNAME , pSSN , pBDATE , pADDRESS  
    , pSEX , pSALARY , pSUPERSSN , pDNO  
  );  
EXCEPTION  
  WHEN OTHERS THEN  
    raise emp_actions.BAD_OPERATION;  
END hire_employee;  
END emp_actions;
PL/SQL Programming

TRIGGERS

• Triggers are SERVER SIDE stored blocks of code that execute -or fire- automatically when some database events occur.

• Triggers are similar to parameter-less procedures, however they can not be called directly.

• Triggers can NOT (call code or) include the COMMIT / ROLLBACK statements.

• Traditionally triggers are associated to maintenance of database tables and are invoked by the run-time Oracle processes on DELETE, UPDATE, INSERT operations.

• User Event Triggers could fire on non-maintenance events such as: startup, shutdown, create, alter, drop, grant, revoke …

• In general, triggers appear to execute quietly without the user even knowing of their existence.
PL/SQL Triggers. Example

Insert into works_on(Essn, Pno, Hours) values (123456789, 10, 40);

<table>
<thead>
<tr>
<th>ESSN</th>
<th>PNO</th>
<th>HOURS</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456789</td>
<td>1</td>
<td>32.5</td>
</tr>
<tr>
<td>666884444</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>453453453</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>453453453</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>333445555</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>333445555</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>999887777</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>987654321</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>888665555</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

TRIGGER T008B
BEFORE INSERT OR UPDATE OR DELETE ON WORKS_ON
FOR EACH ROW
. . .
TRIGGER T008A
BEFORE INSERT OR UPDATE OR DELETE ON WORKS_ON
. . .
-- Employees can not work more than 60 hours/week
. . .
END;

ERROR at line 1:
ORA-20008: T008A - Excessive work load violation (PK_WORKS_ON)
Employee 123456789, Project 10, Hours 4. Operation rejected.
ORA-04088: error during execution of trigger 'CSUPERSON.T008A'

PL/SQL Triggers

Uses of triggers

- Provide sophisticated auditing / logging
- Prevent invalid transactions
- Enforce referential integrity
- Enforce data integrity
- Enforce complex business rules
- Enforce complex security authorizations
PL/SQL Triggers

Consulting Database Dictionary

desc user_triggers;
select trigger_name
  from user_triggers
  where table_name='WORKS_ON';

Enable / Disable

Alter trigger T001 enable;
Alter trigger T001 disable;
ALTER TABLE Employee ENABLE ALL TRIGGERS;
ALTER TABLE Employee DISABLE ALL TRIGGERS;

Abbreviated Trigger syntax

Create or replace
trigger triggerName
{ before | after | instead of } eventClause
on [ schema | database ]
[ referencing new ... old ]
[ for each row ]
[ when condition ]
PL/SQL block | call_procedure_statement
PL/SQL Triggers

Example:
--Business Rule 1: Job assignments to project 10 must be between 4 & 6 hours.

TRIGGER CSUPERSON.T001
BEFORE INSERT OR UPDATE ON CSUPERSON.WORKS_ON
REFERENCING OLD AS OLD_VALUE NEW AS NEW_VALUE
FOR EACH ROW
WHEN (new_value.pno = 10)
BEGIN
  if :new_value.hours >= 6 or
    :new_value.hours <= 4 then
    raise_application_error
    (-20001,
     'Proj. 10 load must be between 4-6 hours/week');
  end if;
END;

PL/SQL Triggers

Testing T001
Add employee 1 to project 10 for a total of 9 hours/week.

CSUPERSON SQL> insert into works_on values (1,10,9);
insert into works_on values (1,10,9)
* ERROR at line 1:
ORA-20001: Proj. 10 load must be between 4-6 hours/week
ORA-06512: at "CSUPERSON.T001", line 4
ORA-04088: error during execution of trigger 'CSUPERSON.T001'
PL/SQL Triggers

EXAMPLE 2

• The following trigger is fired when a new dependent record is inserted.
  • If the record corresponds to a new born child (< 1 year) several gifts are ordered for the baby.
  • If the new entry is not related to an existing employee the insertion is rejected (referential integrity).
  • Assume there is a logging table MYLOG(a,b)

```plsql
TRIGGER T002
BEFORE INSERT on Dependent
FOR EACH ROW
DECLARE
  mName varchar(20) := NULL;
BEGIN
  select Lname into mName from employee
  where ssn = :new.Essn;
  if (months_between(sysdate,:new.Bdate) < 12 ) then
    insert into Mylog values (sysdate || ' Personnel Dept-->', 'Send NEW_BABY greeting card to '|| mName);
    insert into Mylog values (sysdate || ' Marketing Dept.-->', 'Mail $100 company stock to '|| mName);
    insert into mylog values (sysdate || ' Purchasing Dpt.-->', 'Order one-year diapers for '|| mName);
  end if;
EXCEPTION
  when others then
    raise_application_error(-20002,'Dependent problem: ' || :new.essn);
END;
```
PL/SQL Triggers

Testing T002

-- Anonymous block to test T002
DECLARE
BEGIN
-- Intentionally done! this insertion will be rejected
BEGIN
    INSERT INTO dependent VALUES
    (777000777, 'Angel', 'M', '03-SEP-96', 'Son');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(sqlErrm);
END;
-- This second insertion is valid and will be accepted
-- all the gifts will be ordered for the new baby.
BEGIN
    INSERT INTO dependent VALUES
    (123456789, 'Jose', 'M', '01-JAN-2005', 'Son');
END;
-- This third insertion is valid and will be accepted
-- not a baby.
BEGIN
    INSERT INTO dependent VALUES
    (123456789, 'Joe', 'M', '04-JUL-2001', 'Son');
END;
END;

ORA-20002: Dependent problem: 777000777
ORA-06512: at "CSUPERSON.T002", line 24
ORA-04088: error during execution of trigger 'CSUPERSON.T002'

MYLOG

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15-FEB-05</td>
<td></td>
</tr>
<tr>
<td>Personnel Dept.---&gt;</td>
<td>Send NEW_BABY greeting card to Smith</td>
</tr>
<tr>
<td>15-FEB-05</td>
<td></td>
</tr>
<tr>
<td>Marketing Dept.---&gt;</td>
<td>Mail $100 company stock to Smith</td>
</tr>
<tr>
<td>15-FEB-05</td>
<td></td>
</tr>
<tr>
<td>Purchasing Dpt.---&gt;</td>
<td>Order one-year diapers for Smith</td>
</tr>
</tbody>
</table>

DEPENDENT

<table>
<thead>
<tr>
<th>SSN</th>
<th>DEPENDENT_ SSN</th>
<th>DEPENDENT_ NAME</th>
<th>RELATIONSHIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>333465555 Alice</td>
<td>F 05-APR-76</td>
<td>Daughter</td>
<td></td>
</tr>
<tr>
<td>333465555 Theodore</td>
<td>M 02-OCT-73</td>
<td>Son</td>
<td></td>
</tr>
<tr>
<td>333465555 Joy</td>
<td>F 03-MAY-48</td>
<td>Spouse</td>
<td></td>
</tr>
<tr>
<td>987654321 Amber</td>
<td>M 29-FEB-32</td>
<td>Spouse</td>
<td></td>
</tr>
<tr>
<td>123456789 Michael</td>
<td>M 01-JAN-78</td>
<td>Son</td>
<td></td>
</tr>
<tr>
<td>123456789 Alice</td>
<td>F 31-DEC-78</td>
<td>Daughter</td>
<td></td>
</tr>
<tr>
<td>123456789 Elizabeth</td>
<td>F 01-MAY-57</td>
<td>Spouse</td>
<td></td>
</tr>
<tr>
<td>123456789 Joe</td>
<td>M 01-JAN-05</td>
<td>Son</td>
<td></td>
</tr>
<tr>
<td>123456789 Joe</td>
<td>M 04-JUL-01</td>
<td>Son</td>
<td></td>
</tr>
</tbody>
</table>
PL/SQL Triggers

Statement Level trigger.
Execute only once for the entire statement (rather than on a row-by-row basis).

```
TRIGGER "CSUPERSON"."T003"
BEFORE INSERT OR UPDATE OR DELETE ON "CSUPERSON"."WORKS_ON"
BEGIN
  -- RULE: Only CSUPERSON is allowed to change the works_on table
  if user != 'CSUPERSON' then
    raise_application_error(-20004,
      'T003. You are not authorized to make changes on Works_On');
  end if;

  -- RULE: changes to WORKS_ON are accepted only from Mon-Fri between
  -- working hours 8:00 AM and 6:00 PM. Reject otherwise.
  if (to_char(sysdate,'HH24:MI') not between '08:00' and '18:00' )
    or (to_char(sysdate,'DY') in ('SAT', 'SUN') )
  then
    raise_application_error(-20005,
      'T003. Changes are only accepted during normal office hours');
  end if;
END;
```

What is going on?
You may use the following predicates to exactly determine what is happening in the trigger’s code:

- if `INSERTING` then …
- if `DELETING` then …
- if `UPDATING` then …
PL/SQL Triggers

T004. Testing the action Predicates.

TRIGGER "CSUPERSON"."T004" BEFORE
INSERT OR UPDATE OR DELETE ON "CSUPERSON"."PROJECT"
FOR EACH ROW
BEGIN
  -- tell us what is going on during the execution of the trigger
  /***********************************************************************
  INSERTING (only :NEW data exists for the record)
  *****************************************************************************/
  if INSERTING then
    dbms_output.put_line('Inserting a new Project ');
    dbms_output.put_line (' :NEW.Pname   |  '-' | :NEW.Pnumber |  '-'
      | :NEW.Plocation |  '-'
      | :NEW.Dnum    )
    return;
  end if;
  /***********************************************************************
  DELETING (there is no :NEW image - ask only for :OLD)
  *****************************************************************************/
  if DELETING then
    dbms_output.put_line('Deleting an old Project ');
    dbms_output.put_line (' :OLD.Pname   |  '-' | :OLD.Pnumber |  '-'
      | :OLD.Plocation |  '-'
      | :OLD.Dnum    )
    return;
  end if;
  /***********************************************************************
  UPDATING (there are :NEW and :OLD images for the record
  *****************************************************************************/
  if UPDATING then
    if (:NEW.Pname != :OLD.Pname ) then
      dbms_output.put_line ('Updating PNAME of an old Project ');
      dbms_output.put_line (' :NEW.Pname   |  '-' | :NEW.Dnum    )
      dbms_output.put_line (' Old Pname: ' || :OLD.Pname );
    end if;
    if (:NEW.Plocation != :OLD.Plocation ) then
      dbms_output.put_line ('Updating PLOCATION of an old Project ');
      dbms_output.put_line (' :NEW.Plocation |  '-' | :NEW.Dnum    )
      dbms_output.put_line (' Old Plocation: ' || :OLD.Plocation );
    end if;
    if (:NEW.Dnum != :OLD.Dnum ) then
      dbms_output.put_line ('Updating DNUM of an old Project ');
      dbms_output.put_line (' :NEW.Pname   |  '-'
      | :NEW.Dnum    )
      dbms_output.put_line (' Old DNUM: ' || :OLD.Dnum );
    end if;
  end if;
  -- updating
END;
PL/SQL Triggers

1. Testing triggering predicates

```
PL/SQL Triggers
1. Testing triggering predicates

PROJECT Table

<table>
<thead>
<tr>
<th>PNAME</th>
<th>PNUMBER</th>
<th>PLOCATION</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductX</td>
<td>1</td>
<td>Bellaire</td>
<td>5</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td>Sugarland</td>
<td>5</td>
</tr>
<tr>
<td>ProductZ</td>
<td>3</td>
<td>Houston</td>
<td>5</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td>Houston</td>
<td>1</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td>Stafford</td>
<td>4</td>
</tr>
</tbody>
</table>

SQL> insert into project values ('XYZ', 40, 'Cleveland', 4);
Inserting a new Project
XYZ-40-Cleveland-4
1 row created.

CSUPERSON SQL> select * from project;

<table>
<thead>
<tr>
<th>PNAME</th>
<th>PNUMBER</th>
<th>PLOCATION</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductX</td>
<td>1</td>
<td>Bellaire</td>
<td>5</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td>Sugarland</td>
<td>5</td>
</tr>
<tr>
<td>ProductZ</td>
<td>3</td>
<td>Houston</td>
<td>5</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td>Houston</td>
<td>1</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>XYZ</td>
<td>40</td>
<td>Cleveland</td>
<td>4</td>
</tr>
</tbody>
</table>
```

PL/SQL Triggers

2. Testing triggering predicates

```
SQL> update project set pname='Rock and Roll', Dnum=1 where pnumber=40;
Updating PNAME of an old Project
Rock and Roll-40-Cleveland-1
Old Pname: XYZ
Updating DNUM of an old Project
Rock and Roll-40-Cleveland-1
Old DNUM: 4
1 row updated.

PL/SQL Triggers

2. Testing triggering predicates

```
SQL> update project set pname='Rock and Roll', Dnum=1 where pnumber=40;
Updating PNAME of an old Project
Rock and Roll-40-Cleveland-1
Old Pname: XYZ
Updating DNUM of an old Project
Rock and Roll-40-Cleveland-1
Old DNUM: 4
1 row updated.
```