

ORACLE SQL_PL/SQL ABBREVIATED

By Mark E. Donaldson

DATABASE OBJECTS

An Oracle database can contain multiple data structures (Objects). Each structure should be outlined in the database design so that it can be created during the build stage of database development.

DATABASE OBJECTS	
Object	Description
Table	Basic unit of storage. Composed of rows and columns.
View	Logically represents subsets of data from one or more tables.
Sequence	Generates primary key values.
Index	Improves the performance of some queries.
Synonym	Gives alternative names to objects.

SQL

SQL STATEMENTS	
Statement	General Description
SELECT	Data Retrieval
INSERT UPDATE DELETE	Data Manipulation Language (DML)
CREATE ALTER DROP RENAME TRUNCATE	Data Definition Language (DDL)
COMMIT ROLLBACK SAVEPOINT	Transaction Control
GRANT REVOKE	Data Control Language (DCL)

Statement Attributes:

- Are not case sensitive.
- Can be on one or more lines.
- Cannot abbreviate keywords
- Place clauses on separate lines.
- Use tabs and indents to enhance readability.
- End with a semicolon (;).

SELECT STATEMENT

Retrieves data from the database. You can do the following:

- Selection – Choose rows you want returned.
- Projection – Choose columns you want returned.
- Join – Bring together data stored in different tables by creating a link between them.

A SELECT statement must have the following:

- A SELECT clause, which specifies columns.
- A FROM clause, which specifies the table.

SELECT Statement Syntax

```
SELECT [DISTINCT] {*, column [alias], ...
FROM table;
```

where:

SELECT	a list of one or more columns.
DISTINCT	suppresses duplicates.
*	selects all columns.
column	select the named column.
alias	gives selected cols different headings.
FROM table	specifies table containing the columns.

SELECT Examples

```
SELECT *
FROM dept;
```

```
SELECT deptno, dname, loc
FROM dept;
```

```
SELECT ename, sal, sal + 300
FROM emp;
```

```
SELECT ename, AS name, sal salary or "Annual Salary"
FROM emp;
```

```
SELECT ename || job AS "Employees"
FROM emp;
```

```
SELECT ename || ' is a ' || job
AS "Employee Details"
FROM emp;
```

```
SELECT ename || ': ' || '1' || " Month salary = " || sal
FROM emp;
```

```
SELECT DISTINCT deptno
FROM emp;
```

COMPARISON OPERATORS

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ...AND...	Between two values
IN(list)	Match any of a list of values
LIKE	Match a character pattern. % represents any sequence of zero or more characters; _ represents any single character.
ESCAPE	Identifier used to search for "%" or "_".
IS NULL	Is a null value
AND	Returns TRUE if both component conditions are TRUE
OR	Returns TRUE if either component condition is TRUE
NOT	Returns TRUE if the following condition is

FALSE

Comparison Operator Examples

```
SELECT ename, sal
FROM emp
WHERE sal BETWEEN 1000 AND 1500;
```

```
SELECT empno, ename, sal, mgr
FROM emp
WHERE mgr IN (1902, 7566, 7788);
```

-- exact match

```
SELECT ename
FROM emp
WHERE ename LIKE 'S%';
```

-- A as second character

```
SELECT ename
FROM emp
WHERE ename LIKE '_A%';
```

```
SELECT ename
FROM emp
WHERE mgr IS NULL;
```

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 1100
AND job = "CLERK";
```

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 1100
OR job = "CLERK";
```

```
SELECT ename, job
FROM emp
WHERE job NOT IN (1902, 7566, 7788);
```

WHERE CLAUSE

Restricts the query to rows that meet a condition. The condition is composed of column names, expressions, constants, and a comparison operator. Can compare values in columns, literal values, arithmetic expressions, or functions. Consists of three elements:

- Column name
- Column operator
- Column name, constant, or list of values
- Character strings and date values and date values must be enclosed in single quotation marks ("")
- Character values are case sensitive and date values are format sensitive.
- The default date format is DD-MON-YY

WHERE Syntax
[WHERE conditions(s)]

WHERE Examples

```
SELECT ename, job, deptno
```

```
FROM emp
WHERE job='CLERK';
```

```
SELECT ename, sal
FROM emp
WHERE sal BETWEEN 1000 AND 1500;
```

```
SELECT empno, ename, sal mgr
FROM emp
WHERE mgr IN (1902, 7566, 6688);
```

```
SELECT ename
FROM emp
WHERE ename LIKE '_A%';
```

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 1100
AND job='CLERK';
```

```
SELECT *
FROM dept
WHERE dname LIKE '%\%' ESCAPE '\';
```

ORDER BY CLAUSE

Used to sort the rows in a specific order. If used, must be placed last in statement.

ORDER BY Syntax

[ORDER BY {column, expr} [ASC|DESC]]

where:

ORDER BY	specifies the order rows retrieved
ASC	ascending order (default)
DESC	descending order

ORDER BY Example

```
SELECT ename, job, deptno, hiredate
FROM emp
ORDER BY hiredate DESC, deptno
```

SQL DATA TYPES

Datatype	Description
NUMBER(p,s)	Number having precision p and scale s. Precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point. The precision can range from 1 to 38 and the scale can range from -84 to 127.
VARCHAR2(size)	Variable length character value of maximum size size. A maximum size must be specified. Default and minimum size is 1. Maximum size is 4000.
CHAR(size)	Fixed length character value of length size bytes. Default and minimum size is 1. Maximum size is 2000.
DATE	Date and time value between January 1, 4712 B.C. and December 31, 9999 A.D.
LONG	Variable length character data up to 2 gigabytes.

CLOB	Single byte character data up to 4 gigabytes.
RAW(size)	Raw binary data of length size. A maximum size must be specified. Maximum size is 2000.
LONG RAW	Raw binary data of variable length up to 2 gigabytes.
BLOB	Binary data up to 4 gigabytes.
BFILE	Binary data stored in an external file up to 4 gigabytes.

SQL FUNCTIONS

SINGLE ROW FUNCTIONS

Single row functions are used to manipulate data items, accept one or more arguments, and return one value for each row returned by the query. An argument can be one of the following:

- User supplied constant
- Variable value
- Column name
- Expression

Features of single row functions include:

- Act on each row returned in query
- Return one result per row
- May return a data value of a different type than that referenced
- May expect one or arguments
- Use in SELECT, WHERE, & ORDER BY clauses
- Can be nested

Single Row Function Syntax

function_name (column | expression, [arg1, arg2, . . .])

function_name	name of function
column	any named database column
expression	any character string or calc
arg1, arg2	any argument used by function

Character Functions	
Function	Purpose
LOWER (column expr)	Converts alpha characters values to lower case
UPPER (column expr)	Converts alpha characters to upper case
INITCAP (column expr)	Converts alpha character values to uppercase for the first letter of each word, all others in lower case
CONCAT (col1 expr1, col2 expr2)	Concatenates first character value to second character value; equiv to operator
SUBSTR (col expr, m[, n])	Returns specified characters from character value starting at character position m, n characters long (If m is negative, the count starts from the end of the character value. If n is omitted, all characters to the end of the string are returned).
LENGTH (column expr)	Returns the number of characters in value
INSTR (column expr, m)	Returns the numeric position of

	named character
LPAD (col expr, n, 'string')	Pads the character value right justified to a total of n character positions
RPAD (col expr, n, 'string')	Pads the character value left justified to a total of n character positions
TRIM (lead trail both, trim_character FROM trim_source)	Enables you to trim leading or trailing characters (or both) from a character string. If trim_character or trim_source is a character literal, you must enclose it in single quotes (")

Character Function Examples

```
SELECT "The job title for ' || INITCAP(ename) || ' is ' ||
LOWER(job) AS "EMPLOYEE DETAILS"
FROM emp;
```

```
SELECT empno, ename, deptno
FROM emp
WHERE ename = UPPER('blake');
```

```
SELECT ename, CONCAT(ename, job),
LENGTH(ename), INSTR(ename, 'A')
FROM emp
WHERE SUBSTR(job, 1, 5) = 'SALES';
WHERE job NOT IN (1902, 7566, 7788);
```

Character Manipulation Function Examples

Function	Result
CONCAT('Good', 'String')	GoodString
SUBSTR('String', 1, 3)	Str
LENGTH('String')	6
INSTR('String', 'r')	3
LPAD(sal, 10, '*')	*****5000
TRIM('S' FROM 'SSMITH')	mith

Number Functions

Function	Purpose
ROUND (column expr, n)	Rounds the column, expression, or value to n decimal places or if n is omitted, no decimal places (If n is negative, numbers to left of the decimal point are rounded).
TRUNC (column expr, n)	Truncates the column, expression, or value to n decimal places or if n is omitted, no decimal places (If n is negative, numbers to left of the decimal point are truncated to zero).
MOD (m, n)	Returns the remainder of m divided by n.

Number Function Examples

-- Outputs 45.92, 46, 50
 SELECT ROUND(45.923, 2) ROUND(45.923, 0),
 ROUND(45.923, -1)
 FROM DUAL;

-- Outputs 45.92, 45, 40
 SELECT TRUNC(45.923, 2), TRUNC(45.923),
 TRUNC(45.923, -1)
 FROM DUAL;

SELECT ename, sal, comm, MOD(sal, comm)
 FROM emp
 WHERE job = 'SALESMAN';

comparing it to each search value. If the expression is the same as search, result is returned. If the default value is omitted, a null value is returned where a search value does not match any of the result values.

Datatype Conversion Function Examples

SELECT ename TO_CHAR(hiredate, 'fmDD Month
 YYYY') HIREDATE
 FROM emp; → 17 November 1981

SELECT TO_CHAR(sal, '\$99,999') SALARY
 FROM emp
 WHERE ename = 'SCOTT'; → \$3,000

SELECT ename, hiredate
 FROM emp
 WHERE hiredate = TO_DATE('February 22, 1981',
 'Month dd, YYYY');

SELECT ename, sal, NVL(comm, 0)
 FROM emp;

-- DECODE is IF-THEN-ELSE & DEFAULT statement
 SELECT job, sal,
 DECODE(job, 'ANALYST', SAL * 1.1,
 'CLERK', SAL * 1.15,
 'MANAGER', SAL * 1.20,
 SAL) REVISED_SAL
 FROM emp;

SELECT ename, sal,
 DECODE(TRUNC(sal / 1000, 0),
 0, 0.00,
 1, 0.09,
 2, 0.20,
 3, 0.30,
 4, 0.40,
 5, 0.42,
 6, 0.44,
 0.45) TAX_RATE
 FROM emp
 WHERE deptno = 30;

Date Functions 'fmt' = special format model	
Function	Purpose
MONTHS_BETWEEN (date1, date2)	Number of months between two dates
ADD_MONTHS (date, n)	Add calendar months to date
NEXT_DAY (date, 'char')	Next day of the date specified
LAST_DAY (date)	Last day of the month
ROUND (date[, 'fmt'])	Round date
TRUNC (date[, 'fmt'])	Truncate date
SYSDATE	Returns date and time

Date Function Examples

MONTHS_BETWEEN('01-SEP-95', '11-JAN-94')
 → 19.6774194

ADD_MONTHS('11-JAN-94', 6) → '11-JUL-94'

NEXT_DAY('01-SEP-95', 'FRIDAY') → '08-SEP-95'

LAST_DAY('01-SEP-95') → '30-SEP-95'

ROUND('25-JUL-95', 'MONTH') → 01-AUG-95

ROUND('25-JUL-95', 'YEAR') → 01-JAN-96

ROUND('25-JUL-95', 'MONTH') → 01-JUL-95

ROUND('25-JUL-95', 'YEAR') → 01-JAN-95

Datatype Conversion Functions 'fmt' = special format model	
Function	Purpose
TO_CHAR (number[date, 'fmt'])	Convert number or date value as a character
TO_NUMBER (char[, 'fmt'])	Convert character string to a number
TO_DATE (char[, 'fmt'])	Convert character string to date
NVL (expr1, expr2)	Converts null to actual value. Can use date, character, and date datatypes. The datatypes must match
DECODE (col expression, search1, result1 [, search2, result2, . . .], [default])	Decodes an expression in a way similar to CASE or IF-THEN-ELSE statements. Decodes expression after

JOINS

A join is used to query data from more than one table.

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.

There are two main types of join conditions:

- Equijoins
- Non-equijoins

Additional joins methods include the following:

- Outer joins

- Self joins
- Set operators

JOIN SYNTAX

```
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column1 = table2.column2
```

EQUIJOIN SYNTAX

```
SELECT emp.ename, emp.deptno, dept.deptno, FROM
emp, dept
WHERE emp.deptno = dept.deptno;
```

```
SELECT c.name, o.ordid, i.itemtot, o.total
FROM customer c, ord o, item i
WHERE c.custid = o.custid
AND o.ordid = i.ordid
AND c.name = 'TKB SPORT SHOP';
```

NON-EQUIJOIN SYNTAX

```
SELECT e.ename, e.sal, s.grade
FROM emp e, salgrade s
WHERE e.sal
BETWEEN s.losal AND s.hisal;
```

OUTER JOIN SYNTAX

```
SELECT e.ename, d.deptno, d.dname
FROM emp e, dept d
WHERE e.deptno(+) = d.deptno
```

SELF JOIN SYNTAX

```
SELECT worker.ename|| ' works for '||manager .ename
FROM emp worker, emp manager
WHERE worker.mgr = manager.empno;
```

GROUP FUNCTIONS

Group functions operate on sets of rows to give one result per group.

Group Functions	
Function	Description
AVE ([DISTINCT ALL] n)	Average value of n, ignoring null values.
COUNT ({*[DISTINCT ALL] expr}) *returns number of rows in table or WHERE statement. expr returns number of non-null rows identified by expr	Number of rows, where expr evaluates to something other than null (Count all selected rows using *, including duplicates and rows with nulls). Can also use -1 to bypass index for faster search.
MAX ([DISTINCT ALL] expr)	Maximum value of expr, ignoring null values.
MIN ([DISTINCT ALL] expr)	Minimum value of expr, ignoring null values.
STDDEV ([DISTINCT ALL] x)	Standard deviation of n, ignoring null values.
SUM([DISTINT ALL] n)	Sum of values n, ignoring null values.
VARIANCE ([DISTINCT ALL]x)	Variance of n, ignoring null values.

GROUP FUNCTION EXAMPLES

```
SELECT AVE (sal), MAX (sal), MIN (sal), SUM(sal)
FROM emp
WHERE job LIKE "SALES%";
```

```
SELECT MIN (hiredate), MAX (hiredate)
FROM emp;
```

```
SELECT COUNT (*)
FROM emp
WHERE deptno = 30;
```

```
SELECT COUNT (DISTINCT (deptno) )
FROM emp;
```

```
SELECT AVE (NVL (comm, 0) )
FROM emp;
```

GROUP BY CLAUSE

Used to divide the rows in a table into groups. You can then use the group functions to return summary information by each group.

GROUP BY Syntax

where:
group_by_expression

specifies columns whose values determine the basis for grouping rows.

- When using GROUP BY make sure all columns in SELECT list that are not in the group functions are included in the GROUP BY clause:
- The SELECT clause specifies the column to be retrieved.
- The FROM clause specifies the tables to access.
- The WHERE clause specifies the rows to be retrieved. If no WHERE all rows retrieved.
- The GROUP BY clause specifies how the rows should be grouped.
- You cannot use the WHERE clause to restrict groups. Use the HAVING clause to restrict groups.

GROUP BY Examples

```
SELECT deptno, AVE (sal)
FROM emp
GROUP BY deptno
```

```
SELECT deptno, COUNT (ename)
FROM emp
GROUP BY deptno;
```

```
SELECT deptno, AVE (sal)
FROM emp
GROUP BY deptno
HAVING AVE (sal) > 2000;
```

HAVING CLAUSE

Used to restrict groups or specify which groups are to be displayed.

HAVING Syntax

where:

group_condition

restrict the groups of rows returned to those groups for which the specified condition is TRUE. The Oracle Server performs the following when HAVING clause is used:

- Rows are grouped.
- The group function is applied to the group.
- The groups that match the criteria in the HAVING clause are displayed.

```
SELECT column, group_function
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY column];
```

HAVING Examples

```
SELECT deptno, MAX (sal)
FROM emp
GROUP BY deptno
HAVING MAX (sal) > 2000;
```

```
SELECT job, SUM (sal) PAYROLL
FROM emp
WHERE job NOT LIKE 'SALES%'
GROUP BY job
HAVING SUM (sal) > 5000
ORDER BY SUM (sal);
```

SUBQUERIES

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. Can be placed in the following SQL clauses:

- WHERE
- HAVING
- FROM

Subquery Guidelines:

- Must be enclosed in parentheses.
- Must appear on the right side of the comparison operator.
- Subqueries cannot contain an ORDER BY clause.
- Three classes of comparison operators are used in subqueries: single row operators, multiple row operators, and multiple column operators.
- IN, ANY, & ALL are the three multiple row operators.
- **The subquery (inner query) executes once before the main query.**
- **The result of the subquery is used by the main query (outer query).**

SUBQUERY Syntax

Single Row:

```
SELECT select_list
FROM table
WHERE expr operator
                (SELECT select_list
                 FROM table);
```

Multiple Column:

```
SELECT column, column . . .
FROM table
WHERE (column, column, . . . ) IN
                (SELECT column, column . . .
                 FROM table
                 WHERE condition);
```

SUBQUERY Examples

Single Row:

```
SELECT ename
FROM emp
WHERE sal >
                (SELECT sal
                 FROM emp
                 WHERE empno = 7566);
```

Single Row IN FROM Clause:

```
SELECT a.ename, a.sal, a.deptno, b.salary
FROM emp a, (SELECT deptno, AVG(sal) salavg
             FROM emp
             GROUP BY deptno) b
WHERE a.deptno = b.deptno
AND a.sal > b.salavg;
```

Single Row With Group Function:

```
SELECT deptno, MIN(sal)
FROM emp
GROUP BY deptno
HAVING MIN(sal) >
                (SELECT MIN(sal)
                 FROM emp
                 WHERE deptno = 20
                 GROUP BY deptno);
```

Multiple Row:

```
SELECT empno, ename, job
FROM emp
WHERE sal < ANY
                (SELECT sal
                 FROM emp
                 WHERE job = 'CLERK')
AND job <> 'CLERK';
```

Multiple Column: Pairwise

```
SELECT ordid prodid, qty
FROM item
WHERE (prodid, qty) IN
                (SELECT prodid, qty
                 FROM item
                 WHERE ordid = 605)
AND ordid <> 605;
```

Multiple Column: Non-Pairwise

```
SELECT ordid prodid, qty
FROM item
```

```
WHERE prodid) IN
      (SELECT prodid
       FROM item
       WHERE ordid = 605)
AND qty IN
      (SELECT qty
       FROM item
       WHERE ordid = 605);
AND ordid <> 605;
```

INSERT INTO STATEMENT

Enters new rows into a table one row at a time into a database. Uses the VALUES clause to describe new data. The INSERT statement can be used to copy rows from one table to another. Instead of using the VALUES clause, you use a subquery.

INSERT INTO Statement Syntax

```
INSERT INTO table [(column {, column . . . } ) ]
VALUES (value [, value . . . ] );
```

where:

table is the name of the table
column is the name of the column in the table to populate
value is the corresponding value for the column

INSERT INTO Statement Examples

```
INSERT INTO dept (deptno, dname, loc)
VALUES (50, 'DEVELOPMENT', 'DETROIT');
```

```
INSERT INTO dept (deptno, dname)
VALUES (60, 'MIS');
```

```
INSERT INTO dept (deptno, dname, loc)
VALUES (&department_id, "&department_name");
```

INSERT INTO Statement Syntax (For Copying Rows)

```
INSERT INTO table [column (, column) } subquery;
```

where:

table is the table name
column is the name of the column in the table to populate
subquery is the subquery that returns to the table

UPDATE STATEMENT

Used to modify existing rows.

UPDATE Statement Syntax

```
UPDATE table
SET column = value [, column = value, . . . ]
[WHERE condition];
```

where:

table is the name of the table
column is the name of the column in the table
value is the corresponding value or query for column
condition identifies the rows to be updated and is

composed of column names, expressions, constants, subqueries, and comparison operators.

All rows in table are modified if the WHERE clause is omitted.

UPDATE Statement Examples

Single Column Subquery

```
UPDATE emp
SET deptno = 20
WHERE empno = 7782;
```

Multiple Column Subquery

```
UPDATE emp
SET (job, deptno) = (SELECT job, deptno
                    FROM emp
                    WHERE empno = 7499)
WHERE empno = 7698;
```

Based On Another Table

```
UPDATE employee
SET deptno = (SELECT deptno
              FROM emp
              WHERE empno = 7789)
WHERE job = (SELECT job
             FROM emp
             WHERE empno = 7788);
```

DELETE STATEMENT

Remove existing rows from a table with the DELETE statement.

DELETE Statement Syntax

```
DELETE table
[WHERE condition];
```

where:

table is the table name
condition identifies the rows to be deleted and is composed of column names, expressions, constants, subqueries, and comparison operators

If the WHERE clause is omitted, all rows are deleted. You can confirm the delete operation by displaying the deleted rows using the SELECT statement.

DELETE Statement Examples

```
DELETE FROM department
WHERE dname = 'DEVELOPMENT';
```

```
DELETE FROM emp
WHERE hiredate >TO_DATE('01.01.1997',
'DD.MM.YY');
```

DATABASE TRANSACTIONS

Transactions consist of DML statements that make up one consistent change to the data. A transaction begins

when the first executable SQL statement is encountered and terminates when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued.
- A DDL statement, such as CREATE, is issued.
- A DCL statement is issued.
- The user exits SQL *Plus.
- A machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction. A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Transaction Control Statements COMMIT, ROLLBACK, & SAVEPOINT STATEMENTS	
Statement	Description
COMMIT	Ends the current transaction by making all pending data changes.
SAVEPOINT name	Mark a savepoint within the current transaction.
ROLLBACK [TO SAVEPOINT name]	ROLLBACK ends the current transaction by discarding all pending data changes. ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding the savepoint and any subsequent changes. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

COMMIT, ROLLBACK, SAVEPOINT Examples
DELETE FROM employee; ROLLBACK;
UPDATE SAVEPOINT update_done; INSERT ROLLBACK TO update_done;

DATA DICTIONARY

A collection of tables and views in the Oracle database. This collection is created and maintained by the Oracle Server and contains information about the database.

All data dictionary tables are owned by the SYS user. Information stored in the data dictionary include names of the Oracle Server users, privileges granted to users, database object names, table constraints, and auditing information.

There are four categories of data dictionary views, each having a distinct prefix which reflects their intended use:

DATA DICTIONARY VIEW TYPES	
Prefix	Description
USER_	Contain information about objects owned by the user.
ALL_	Contain information about all of the tables (object tables and relational tables) accessible to the user.
DBA_	Restricted views that can be accessed only by people who have been assigned the role DBA.

V\$_	Contain information about dynamic performance views, database server performance and locking.
------	---

DATA DICTIONARY VIEW TYPES FOR: USER_, ALL_, DBA_
USER_TABLES USER_VIEWS USER_TAB_COLUMNS USER_CONSTRAINTS USER_CONS_COLUMNS USER_INDEXES USER_IND_COLUMNS USER_SYNONYMS USER_TRIGGERS USER_TAB_PRIVS USER_OBJECTS USER_CATALOG USER_ERRORS

Querying Data Dictionary Examples
SELECT * FROM user_tables;
SELECT * FROM all_tables;
SELECT * FROM dba_tables;
SELECT DISTINCT object_type FROM user_objects;
SELECT * FROM user_errors;
SELECT * FROM user_catalog;
USER_CATALOG has a synonym called CAT that can be used instead of USER_CATALOG:
SELECT * FROM CAT;
To view comments use:
<ul style="list-style-type: none"> • ALL_COL_COMMENTS • USER_COL_COMMENTS • ALL_TAB_COMMENTS • USER_TAB_COMMENTS
DESCRIBE USER_CATALOG DESCRIBE user_errors;

CREATE TABLE STATEMENT
CREATE TABLE Syntax
CREATE [GLOBAL TEMPORARY] TABLE [schema.] table (column datatype [DEFAULT expr] [, . . .])

where:

GLOBAL TEMPORARY specifies that the table is temporary and that its definition is visible to all sessions. The data in a temporary table is visible only to the session that inserts the data into the table.

schema is the same as the owner's name.
table is the name of the table.

DEFAULT expr specifies a default value if a value is omitted in the INSERT command.

datatype is the column's datatype and length.

Using a subquery:

```
CREATE TABLE table [ (column, column . . . )]  
AS subquery;
```

CREATE TABLE Examples

```
CREATE TABLE dept  
(deptno NUMBER(2),  
      dname VARCHAR2(15),  
      loc VARCHAR2(13) );
```

```
CREATE TABLE dept30  
AS  
  SELECT empno, ename, sal * 12 ANNSAL, hiredate  
 FROM emp  
 WHERE deptno = 30;
```

ALTER TABLE STATEMENT

Used to add new columns, modify existing columns, and define default values for a new column.

ALTER TABLE Syntax

```
ALTER TABLE table  
ADD      (column datatype [DEFAULT expr]  
          [, column datatype] . . . );
```

```
ALTER TABLE table  
MODIFY  (column datatype [DEFAULT expr]  
          [, column datatype] . . . );
```

```
ALTER TABLE table  
DROP COLUMN column;
```

where:

table is the name of the table.
column is the name of the new column.
datatype is the datatype and length of the new column.
DEFAULT expr specifies the default value for the new column.

ALTER TABLE Examples

```
ALTER TABLE dept30  
ADD      (job VARCHAR2(9) );
```

```
ALTER TABLE dept30  
MODIFY      (ename VARCHAR2(15) );
```

```
ALTER TABLE dept30  
DROP COLUMN job;
```

```
ALTER TABLE table  
SET UNUSED (column);  
      Or  
ALTER TABLE table  
SET UNUSED COLUMN column;
```

```
ALTER TABLE table  
DROP UNUSED COLUMNS;
```

DROP TABLE STATEMENT

When a table is dropped:

- All data and structure in the table is deleted.
- Any views and synonyms will remain but are invalid.
- Any pending transactions are committed.
- All indexes are dropped.
- You cannot roll back the statement.

DROP TABLE Syntax

```
DROP TABLE table;
```

where:

table is the name of the table.

DROP TABLE Examples

```
DROP TABLE dept30;
```

RENAME TABLE STATEMENT

RENAME TABLE Syntax

```
RENAME old_name TO new_name;
```

where:

old_name is the old name of the table, view, sequence, or synonym.

new_name is the new name of the table, view, sequence, or synonym.

RENAME TABLE Examples

```
RENAME dept TO department;
```

TRUNCATE TABLE STATEMENT

TRUNCATE TABLE does the following:

- Removes all rows from a table
- Releases the storage space used by that table.
- Cannot be rolled back.
- Rows can alternatively be removed using the DELETE statement.

TRUNCATE TABLE Syntax

TRUNCATE TABLE table;

where:

table is the name of the table.

TRUNCATE TABLE Example

TRUNCATE TABLE department;

COMMENT ON TABLE STATEMENT

You can add a comment of up to 2,000 bytes about a column, table, view, or snapshot by using the COMMENT statement. The command is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

COMMENT ON TABLE Syntax

COMMENT ON TABLE table | COLUMN table.column
IS 'text';

where:

table is the name of the table.

column is the name of a column in a table.

text is the text of the comment.

COMMENT ON TABLE Examples

COMMENT ON TABLE emp
IS 'Employee Information';

CONSTRAINTS

The Oracle Server uses constraints to prevent invalid data entry into tables. You can use constraints to do the following:

- Enforce rules at the table level whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer

Data Integrity Constraints	
Constraint	Description
NOT NULL	Specifies that its column may not contain a null value.
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table.
PRIMARY KEY	Uniquely identifies each row of the table.
FOREIGN KEY	Establishes and enforces a foreign key relationship between the column and a

	column of the referenced table.
CHECK	Specifies a condition that must be true.

Constraint Guidelines

- Name a constraint or the Oracle Server will generate a name by using the SYS_Cn format.
- Create a constraint at the same time a table is created or after the table has been created.
- Define a constraint at the table or column level
- View a constraint in the data dictionary.

CONSTRAINT Syntax

```
CREATE TABLE [schema.] table  
            (column datatype [DEFAULT expr]  
            [column_constraint],  
            ...  
            [table_constraint] [, . . . ]);
```

where:

schema is the same as the owner's name.

table is the name of the table.

DEFAULT expr specifies a default value if a value is omitted in the INSERT command.

column is the name of the column.

datatype is the column's datatype and length.

column_constraint is an integrity constraint as part of the column definition

table_constraint is an integrity constraint as part of the table definition.

CONSTRAINT Examples

```
CREATE TABLE emp (  
    empno NUMBER (4)  
    ename VARCHAR2(10)  
    deptno NUMBER(2) NOT NULL,  
    CONSTRAINT emp_empno_pk PRIMARY KEY  
                                (EMPNO);
```

```
CREATE TABLE emp (  
    empno NUMBER(4),  
    ename VARCHAR2(10) NOT NULL,  
    deptno NUMBER(7, 2) NOT NULL;  
    CONSTRAINT emp_deptno_ck  
        CHECK (DEPTNO BETWEEN 10 AND 99);
```

```
CREATE TABLE dept (  
    deptno NUMBER(2),  
    dname VARCHAR2(14),  
    loc VARCHAR2(13),  
    CONSTRAINT dept_dname_uk UNIQUE (dname);  
    CONSTRAINT dept_deptno_pk PRIMARY KEY  
                                (deptno);
```

```
CREATE TABLE emp (  
    empno NUMBER(4),  
    ename VARCHAR2(14) NOT NULL,  
    deptno NUMBER (7, 2),  
    CONSTRAINT emp_deptno_fk FOREIGN KEY
```

```
(deptno)
REFERENCES dept (deptno);
```

ADDING CONSTRAINT Syntax

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

ADDING CONSTRAINT Examples

```
ALTER TABLE emp
ADD CONSTRAINT emp_mgr_fk
FOREIGN KEY (mgr)
REFERENCES emp (empno);
```

DROPPING CONSTRAINT Examples

```
ALTER TABLE emp
DROP CONSTRAINT emp_mgr_fk;

ALTER TABLE dept
DROP PRIMARY KEY CASCADE;
```

DISABLING & ENABLING CONSTRAINT Examples

```
ALTER TABLE emp
DISABLE CONSTRAINT emp_empno_pk CASCADE

ALTER TABLE emp
ENABLE CONSTRAINT emp_empno_pk;
```

VIEWING CONSTRAINT Examples

```
SELECT constraint_name, constraint_type,
       search_condition
FROM user_constraints
WHERE table_name = 'EMP';

SELECT constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'EMP';
```

VIEW OBJECTS

A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The view is stored as a SELECT statement in the data dictionary.

Views are used for the following reasons:

- Restrict data access.
- Make complex queries easy. Information can be retrieved from multiple tables without writing a join statement.
- Provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Provide groups of users access to data according to particular criteria.
- Views can be simple or complex.
- Views can be inline where the FROM clause of a SELECT statement defines a data source for the SELECT statement.

Simple & Complex Views		
Feature	Simple	Complex
Number of tables	One	One or more
Contain functions	No	Yes
Groups of data	No	Yes
DML available	Yes	Not always

Guidelines For View Creation

- The subquery that defines a view can contain complex SELECT syntax, including joins, groups, and subqueries. The subquery that defines the view cannot contain an ORDER BY clause. The ORDER BY clause is specified when you retrieve data from the view.
- If you do not specify a constraint name for a view created with CHECK OPTION, the system will assign a default name in the format SYS_Cn.
- You can use the OR REPLACE option to change the definition of the view without dropping and re-creating it or regranting object privileges previously granted on it.

Guidelines For View DML Operations

1. You can remove a row from a view unless it contains any of the following:
 - Group functions
 - GROUP BY clause
 - The pseudocolumn ROWNUM keyword
2. You cannot modify data in a view if it contains:
 - Any of the conditions listed above
 - Columns defined by expressions
 - The ROWNUM pseudocolumn
3. You cannot add data if:
 - The view contains any of the conditions listed above
 - There are NOT NULL columns in the base tables that are not selected by the view.

CREATE VIEW Syntax

```
CREATE [OR REPLACE][FORCE|NOFORCE VIEW
view
[ (alias [, alias] . . . ) ]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint] ]
[WITH READ ONLY];
```

where:

OR REPLACE re-creates the view if it already exists. FORCE creates the view whether base table or not. view is the name of the view.

alias specifies names for the expressions selected by the views query (number of aliases must match the number of expressions).

subquery is a complete SELECT statement.

WITH CHECK OPTION specifies that only rows accessible to the view can be inserted or updated.

constraint is the name assigned to the CHECK OPTION constraint.

WITH READ ONLY ensures that no DML operations can be performed on this view.

CREATE VIEW Examples

```
CREATE VIEW salvu30
AS SELECT empno EMPLOYEE_NUMBER,
       ename NAME, sal SALARY
FROM emp
WHERE deptno = 30;
```

To use the view:
SELECT *
FROM salvy30;

```
-- add an alias for each column name
CREATE OR REPLACE VIEW empvu10
       (employee_number, employee_name, job_title)
AS SELECT empno, ename, job
FROM emp
WHERE deptno = 10;
```

```
-- create a complex view
CREATE VIEW dept_sum_vu
       (name, minsal, maxsal, avgsal)
AS SELECT d.dname, MIN(e.sal),
       MAX(e.sal), AVG(e.sal)
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY d.dname;
```

```
CREATE OR REPLACE VIEW empvy20
AS SELECT *
FROM emp
WHERE deptno = 20
WITH CHECK OPTION CONSTRAINT empvy20_ck;
```

```
CREATE OR REPLACE VIEW empvy10
       (employee_number, employee_name, job_title)
AS SELECT empno, ename, job
FROM emp
WHERE deptno = 10
WITH READ ONLY;
```

```
-- create an inline view
SELECT a.ename, a.sal, a.deptno, b.maxsal
FROM emp a,
       (SELECT deptno, max(sal) maxsal
        FROM emp
        GROUP BY deptno) b
WHERE a.deptno = b.deptno
AND a.sal < b.maxsal;
```

DROP VIEW Syntax

```
DROP VIEW view;
```

where:

View is the name of the view.

“TOP-N” ANALYSIS

Used to display only the n top-most or the n bottom-most records of a table based on a condition. This result set can be used for further analysis.

“TOP-N” ANALYSIS Syntax

```
SELECT [column_list], ROWNUM
FROM (SELECT [column_list] FROM table
      ORDER BY Top-N_column)
WHERE ROWNUM <= N
```

“TOP-N” ANALYSIS Examples

```
SELECT ROWNUM as RANK, ename, sal
FROM (SELECT ename, sal FROM emp
      ORDER BY sal DESC)
WHERE ROWNUM <= 3;
```

```
SELECT ROWNUM as SENIOR, E.ename, E.hiredate
FROM (SELECT ename, hiredate FROM emp
      ORDER BY hiredate) E
WHERE rownum <= 4;
```

SEQUENCE OBJECTS

A sequence is a database object created by a user, and can be shared by multiple users to generate unique integers. You can use sequences to automatically generate primary key values.

Sequence NEXTVAL & CURRVAL Pseudocolumns

Once you create a sequence, you can use it to generate sequential numbers for use in tables. The **NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. When you reference sequence. NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.**

The CURRVAL pseudocolumn is used refer to a sequence number that the current user has just generated. **NEXTVAL must be used to generate a sequence number in the current session before CURRVAL can be referenced.** When sequence.CURRVAL is referenced, the last value returned to the user's process is displayed.

Both NEXTVAL and CURRVAL must be referenced with the sequence name: sequence.NEXTVAL.

Rules For Using NEXTVAL & CURRVAL

You can use NEXTVAL & CURRVAL in the following:

- The SELECT list of a SELECT statement that is not part of a subquery.
- The SELECT list of a subquery in an INSERT statement.
- The VALUES clause of an INSERT statement.
- The SET clause of an UPDATE statement.

You cannot use NEXTVAL & CURRVAL in the following:

- A SELECT list of a view.
- A SELECT statement with the DISTINCT keyword.
- A SELECT statement with the GROUP BY, HAVING, or ORDER BY clauses.
- A subquery in a SELECT, DELETE, or UPDATE statement.
- A DEFAULT expression in a CREATE TABLE or ALTER TABLE statement.

SEQUENCE Syntax

```
CREATE SEQUENCE sequence
[INCREMENT BY n]
[START WITH n]
[ {MAXVALUE n | NOMAXVALUE} ]
[ {MINVALUE n | NOMINVALUE} ]
[ {CYCLE | NOCYCLE} ]
[ {CACHE n | NOCACHE} ];
```

where:

sequence is the name of the sequence generator.

INCREMENT BY n specifies the interval between sequence numbers where n is an integer (If this clause is omitted, the sequence will increment by 1).

START WITH n specifies the first sequence number to be generated (If the clause is omitted, the sequence will start with 1).

MAXVALUE n specifies the maximum value the sequence can generate.

NOMAXVALUE specifies a maximum value of 10²⁷ for an ascending sequence and -1 for a descending sequence (default option).

MINVALUE n specifies the minimum sequence value.

NOMINVALUE specifies a minimum value of 1 for an ascending sequence and -(10²⁶) for a descending sequence (default option).

CYCLE | NOCYCLE specifies that the sequence continues to generate values after reaching either its maximum or minimum value or does not generate additional values (NOCYCLE is the default option).

CACHE n | NOCACHE specifies how many values the Oracle Server will preallocate and keep in memory (20 values default).

SEQUENCE Examples

Create A Sequence

```
CREATE SEQUENCE dept_deptno
  INCREMENT BY 1
  START WITH 91
  MAXVALUE 100
  NOCACHE
  NOCYCLE;
```

--do not use CYCLE if sequence is to generate primary key values.

Confirm A Sequence

```
SELECT sequence_name, min_value, max_value,
       increment_by, last_number
FROM user_sequences;
```

Use A Sequence

```
INSERT INTO dept(deptno, dname, loc)
VALUES (dept_deptno.NEXTVAL, 'MARKETING',
       'SAN DIEGO');
```

```
--return current sequence value
SELECT dept_deptno.CURRVAL
FROM dual;
```

--repeated execution

```
INSERT INTO emp . . .
VALUES . . . (emp_deptno.NEXTVAL,
             dept_deptno.CURRVAL, . . . );
```

Modify A Sequence

```
ALTER SEQUENCE dept_deptno
  INCREMENT BY 1
  MAXVALUE 999999
  NOCACHE
  NOCYCLE;
```

Remove A Sequence

```
DROP SEQUENCE dept_deptno;
```

INDEX OBJECTS

An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer. If you do not have an index, a full table scan will occur. Indexes are logically and physically independent of the table they index so they can be created and dropped without effect on the base tables or other indexes.

Function based indexes are possible where an index is based on expressions. The index expression is built from table columns, constraints, SQL functions, and user-defined functions.

INDEX Syntax

```
CREATE INDEX index
ON table (column[, column . . . ] );
```

where:

index is the name of the index.

table is the name of the table.

column is the name of the column in the table to be indexed.

INDEX Examples

Create An Index

```
CREATE INDEX emp_ename_idx
ON emp(ename);
```

Confirm An Index

```
SELECT ic.index_name, ic.column_name,
       ic.column_position col_pos, ix.uniqueness
```

```
FROM user_indexes ix, user_ind_columns ic
WHERE ic.index_name = ix.index_name
AND ic.table_name = 'EMP';
```

Function Based Indexes

```
CREATE INDEX uppercase_idx
ON emp (UPPER (ename));
```

```
SELECT *
FROM emp
WHERE UPPER (ename) = 'KING';
```

```
SELECT *
FROM emp
WHERE UPPER (ename) IS NOT NULL
ORDER BY UPPER (ename);
```

Remove A Sequence

```
DROP INDEX emp_ename_idx;
```

SYNONYM OBJECTS

A Synonym is another name for an object. It is used to simplify access to objects such as tables owned by another user or to shorten lengthy object name.

SYNONYM Syntax

```
CREATE [PUBLIC] SYNONYM synonym
FOR object;
```

where:

PUBLIC creates a synonym accessible to all users.

synonym is the name of the synonym to be created.

object identifies the object for which the synonym is created.

SYNONYM Examples

Create A Synonym

```
CREATE SYNONYM d_sum
FOR dept_sum_vu;
```

```
CREATE PUBLIC SYNONYM dept
FOR alice.dept;
```

Remove A Sequence

```
DROP SYNONYM d_sum;
DROP SYNONYM dept;
```

CONTROLLING USER ACCESS - SECURITY Schema

A schema is a collection of objects, such as tables, views, and sequences. **The schema is owned by a database user and has the same name as that user.**

Privileges

Privileges are the right to execute particular SQL statements. Users require **system privileges** to gain access to the database and **object privileges** to

manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users, or to **roles**, which are named groups of related privileges. Once a user is created, the DBA can grant specific system privileges to the user.

PRIVILEGES	
DBA System Privilege	Operations Authorized
CREATE USER	Allows grantee to create other Oracle users.
DROP USER	Drops another user
DROP ANY TABLE	Drops a table in any schema
BACKUP ANY TABLE	Backs up any table in any schema with the export utility.
User System Privilege	Operations Authorized
CREATE SESSION	Connect to the database
CREATE TABLE	Create tables in the user's schema.
CREATE SEQUENCE	Create a sequence in the user's schema.
CREATE VIEW	Create a view in the user's schema.
CREATE PROCEDURE	Create stored procedure, function, or package in the user's schema.

CREATE USER Syntax & Example

```
CREATE USER user
IDENTIFIED BY password;
```

where:

user is the name of the user to be created.

password specifies that the user must log in with this password.

```
CREATE USER scott
IDENTIFIED BY tiger;
```

```
GRANT create table, create sequence, create view
TO scott;
```

Changing The Password

```
ALTER USER scott
IDENTIFIED BY lion;
```

Roles

A role is a named group of related privileges that can be granted to a user. A user can have access to several roles, and several users can be assigned to the same role.

CREATE ROLE Syntax & Example

```
CREATE ROLE manager;
```

```
GRANT create table, create view
TO manager;
```

```
GRANT manager
TO BLAKE, CLARK;
```

GRANT STATEMENT

Used to grant object privileges to users. SELECT, ALTER, DELETE, EXECUTE, INDEX, INSERT, REFERENCES, And UPDATE privileges can be granted to or revoked from a user.

GRANT Syntax

```
GRANT object_priv [ (columns) ]
ON object
TO {user | role | PUBLIC}
[WITH GRANT OPTION];
```

where:

object_priv is an object privilege to be granted.
ALL specifies all object privileges.

columns specifies the column from a table or view on which privileges are granted.

ON object is object on which the privileges are granted.
TO identifies to whom privilege is granted.
PUBLIC grants object privilege to all users.

WITH GRANT OPTION allows the grantee to grant the object privileges to other users and roles.

GRANT Example

```
GRANT select
ON emp
TO sue, rich;
```

```
GRANT update (dname, loc)
ON dept
TO scott, manager;
```

```
GRANT select, insert
ON dept
TO scott
WITH GRANT OPTION;
```

```
GRANT select
ON alice.dept
TO PUBLIC;
```

Confirming Privileges Granted

Data Dictionary Table	Description
ROLE_SYS_PRIVS	System privileges granted to roles.
ROLE_TAB_PRIVS	Table privileges granted to roles.
USER_ROLE_PRIVS	Roles accessible by user.
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects.
USER_TAB_PRIVS_RECD	Object privileges granted to the user.
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects.
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns.

REVOKE STATEMENT

REVOKE Syntax and Example

```
REVOKE {privilege [, privilege . . . ] | ALL}
ON object
FROM {user [, user . . . ] | role | PUBLIC}
[CASCADE CONSTRAINTS];
```

where:

CASCADE CONSTRAINTS is required to remove any referential constraints made to the object by means of the REFERENCES privilege.

```
REVOKE select, insert
ON dept
FROM scott;
```

EXPLAIN PLAN STATEMENT

An SQL statement that causes Oracle to report the execution plan it would choose for any SELECT, INSERT, UPDATE, or DELETE statement. An execution plan refers to the approach Oracle will take to retrieve the necessary data for a statement. If you have a poorly performing SQL statement, you can use EXPLAIN PLAN to find out how Oracle is processing it.

When you use EXPLAIN PLAN, Oracle inserts rows into the **plan table**, and you must query it to see the results. Oracle provides the script **UTLXPLAN.SQL** to create the plan table for you.

AUTOTRACE is provided by Oracle to automatically display the execution plan for any query, plus other statistics such as disk I/O and network traffic that occurred during the execution. AUTOTRACE uses the **plan table**.

To use AUTOTRACE a user must have SELECT access to Oracle's dynamic performance views. Oracle provides the file **PLUSTRCE.SQL** (SQLPLUS\ADMIN) to simplify granting such access. Before PLUSTRCE.SQL can be run, you must first run **CATALOG.SQL** (RDBMS\ADMIN).

EXPLAIN PLAN Syntax and Example

```
EXPLAIN PLAN
[SET STATEMENT_ID = 'statement_id']
[INTO table_name]
FOR statement;
```

where:

statement_id can be anything you like and is stored in the STATEMENT_ID field of all plan table records related to the query you are explaining. It defaults to null.

table_name is the name of the plan table that defaults to PLAN_TABLE. Only need to use this value if plan table is something other than default.

statement is the DML statement to be explained. This can be an INSERT, UPDATE, DELETE, or SELECT statement, but it must not reference any data dictionary views or dynamic performance tables.

```
DELETE FROM plan_table
WHERE statement_id = 'HOURS_BY_PROJECT';
```

```
EXPLAIN PLAN
SET STATEMENT_ID = 'HOURS_BY_PROJECT
FOR
SELECT employee_name, project_name, SUM(hours)
FROM employee, project, project_hours
WHERE employee.employee_id =
      project_hours.employee_id
AND project.project_id = project_hours.project_id
GROUP BY employee_name, project_name;
```

REMAINING SQL STATEMENT CLAUSES

ANALYZE TABLE STATEMENT

AUDIT STATEMENT

LOCK TABLE STATEMENT

NOAUDIT STATEMENT

SET ROLE STATEMENT

SET TRANSACTION STATEMENT

SQL *Plus

SQL	SQL *Plus
Is a language for communicating with the Oracle Server to access data	Recognizes SQL statements and send them to the Server
Is based on ANSI standard SQL	Is the Oracle proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time and not stored in SQL buffer
Does not have a continuation character	Has dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute command immediately (:)	Does not require termination characters and command are executed immediately
Uses functions to perform some formatting	Use commands to format data
	Accepts SQL input from files and provides a line editor for modifying SQL statements

SQL *Plus LOG-ON & LOG-OFF

SQLPLUS [[-S[ILENT]] [logon] [start]] | - | -?

Where *logon* requires the following syntax:

username[/password][@net_service_name]//nolog

and where *start* requires the following syntax:

@file_name[.ext] [arg ...]

Revised February 1, 2009

STARTUP [FORCE] [RESTRICT] [PFILE-filename] [MOUNT[OPEN[RECOVER]]][database] [mount_options] | [NOMOUNT]

Where *mount_options* requires the following syntax:

[EXCLUSIVE | [PARALLEL | SHARED] [RETRY]]

Starts an Oracle instance with several options, including mounting and opening a database.

SHUTDOWN [ABORT | IMMEDIATE | NORMAL | TRANSACTIONAL]

Shuts down a currently running Oracle instance, optionally closing and dismounting a database.

{EXIT | QUIT} [SUCCESS | FAILURE | WARNING | n | variable | :BindVariable] [COMMIT | ROLLBACK]

Commits all pending changes, terminates SQL *Plus, and returns control to the operating system.

SQL *Plus COMMANDS

SQL *Plus General Commands	
Command	Description
DESC[RIBE] tablename	Displays the structure of a table including column names, datatypes, and whether a column must contain data. DESCRIBE can also be used against procedures, functions, packages, synonyms, and object types.
DESC[RIBE] objectname	
PASSW[ORD] username	Allows you to change database password. You do not need to supply the username when changing your own password. New to SQL*Plus 8. Used to have to use SQL ALTER USER username IDENTIFIED BY password to change the password.
HELP topic	Help for SQL statements, SQL*Plus commands, and PL/SQL commands. Not available with Windows platform.
CONN[ECT]	Allows you to log into the database as a different user, or log into a different database. CONNECT username/password @connect
DISC[ONNECT]	Disconnects you from the database while leaving you in SQL*Plus.
WHENEVER	Controls the behavior of SQL *Plus when an operating system or SQL error occurs.
QUIT	Terminates SQL*Plus session.
EXIT	Terminates SQL*Plus session.

SQL *Plus Execution Commands	
Command	Description
/(slash)	Executes the SQL command or PL/SQL block currently stored in

	the SQL buffer.
EXEC[UTE] statement	Executes a single PL/SQL statement.
R[UN]	Displays and runs the current SQL statement in the buffer.
GET filename[.ext]	Writes the contents of a previously saved file to the SQL buffer
STA[RT] filename[.ext]	Runs previously saved command file.
@ PATH\filename[.ext]	Runs previously saved command file. Same as START
TIMI[NG] [START] text SHOW STOP]	Records timing data for an elapsed period of time, lists the current timer's name and timing data, or lists the number of active timers.
HO[ST] [command]	Executes a host operating system command without leaving SQL *Plus.

SQL *Plus File Commands

Command	Description
SAVE[E] filename[.ext]	Saves current contents of SQL buffer to file
APP[END]	Add to an existing file
RE[PLACE]	Overwrite an existing file
ED[IT]	Invokes the editor and saves the buffer contents to a file named afiedt.buf
SPO[OL] filename[.ext]	Stores query results in a file
SPO[OL] OFF	Closes the spool file
SPO[OL] OUT	Closes the spool file and sends the file results to the system printer
PRINT	Prints variable(s) to screen

SQL *Plus Editing Commands

Command	Description
ED[IT] ED[IT] filename	Invokes default editor if there is data in SQL*Plus buffer, or opens the specified file.
A[PPEND] text	Adds text to end of current line
C[HANGE] /old /new	Changes old text to new in the current line
C[HANGE] /text /	Deletes text from the current line
CL[EAR] BUFF[ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
I[INPUT]	Inserts an indefinite number of lines
I[INPUT] text	Inserts a line consisting of text
L[IST]	List all lines in the SQL buffer
L[IST] n	List one line specified by n
L[IST] m n	List a range of lines m to n
n	Specifies the line to make the current line
n text	Replaces line n with text
() text	Inserts a line before line 1

SQL *Plus Substitution & User Defined Variables

Command	Description
&user_variable	Indicates a variable in a SQL statement. If the variable does not exist, SQL *Plus prompts the user for a value (SQL *Plus discards a new variable once it is used).
&&	Use to reuse the variable value without prompting the user each time.
SET VERIFY	Used to prompt user to confirm

	changes in the SQL statement. Setting SET VERIFY ON forces SQL *Plus to display the text of a command before and after it replaces substitution variables with values.
DEFINE	Displays all user variables with value and datatype.
DEFINE variable value	Creates a CHAR datatype user variable and assigns a value to it.
DEFINE variable	Displays the variable, its value, and its datatype.
UNDEFINE	Erases DEFINED variable.
ACCEPT variable	Reads a line of user input and stores it in a variable.
SET system_variable value	Used to set environmental variables in current session.

SQL *Plus ACCEPT Command Syntax

ACCEPT variable [datatype] [FORMAT format] [PROMPT text] [HIDE]

where:

variable – the name for the variable that stores the value (If it does not exist, SQL *Plus creates it).

datatype – is NUMBER, CHAR, or DATE (CHAR has a maximum length limit of 240 bytes. DATE checks against a format model, and the datatype is CHAR).

FOR[MAT] *format* – specifies the format model.

PROMPT *text* – displays the text before the user can enter the value.

HIDE – suppresses what the user enters such as a password.

SQL *Plus Substitution & User Defined Variables Examples

```
SET VERIFY ON
SELECT empno, ename, sal, deptno
FROM emp
WHERE empno = &employee_num;
Enter value for employee_num:
```

```
SELECT ename, deptno, sal * 12
FROM emp
WHERE job = '&job_title';
Enter value for job_title:
```

```
ACCEPT dept PROMPT "Provide department name: "
SELECT *
FROM dept
WHERE dname = UPPER ('&dept');
Provide department name:
```

```
DEFINE deptname = sales
DEFINE deptname
```

```

DEFINE deptname = "sales" (CHAR)

SELECT *
FROM dept
WHERE dname = UPPER ('&deptname');

UNDEFINE deptname

```

SQL *Plus Format Command Variables

Command	Description
COL[UMN] [column option]	Controls column formats.
TTI[TLE] text ON OFF	Specifies a header to appear at the top of each page.
BTI[TLE] text ON OFF	Specifies a footer to appear at the bottom of each page.
BRE[AK] [ON report element]	Suppresses duplicate values and sections rows of data with line feeds.

SQL *Plus COLUMN Command Options

Option	Description
CLE[AR]	Clears any column formats.
FOR[MAT] format format = special format model	Changes the display of the column data.
HEA[DING] text	Sets the column heading (A vertical line () force a line feed in the heading if you do not use justification).
JUS[TIFY] {align}	Justifies the column heading (not the data) to be left, center, or right.
NOPRI[NT]	Hides the column.
NUL[L] text	Specifies text to be displayed for null values.
PRI[NT]	Shows the column.
TRU[NCATED]	Truncates the string at the end of the first line of display.
WRA[PPED]	Wraps the end of the string to the next line.

SQL *Plus COLUMN Command Variable Syntax

```
COL[UMN] [{column | alias} [option]]
```

SQL *Plus COLUMN Command Variable Examples

```

COLUMN ename HEADING 'Employee|Name' FORMAT A15
COLUMN sal JUSTIFY LEFT FORMAT $99,990.00
COLUMN mgr FORMAT 99999999 NULL 'No manager'
COLUMN ename
COLUMN ename CLEAR

```

SQL *Plus BREAK Command Variable Syntax

```
BREAK on column [ | alias | row] [skip n|dup|page] on .. [on report]
```

where:

page – throws a new page when the break value changes.

skip n – skips n number of lines when the break value changes. Breaks can be active on:

- Column
- Row
- Page
- Report

duplicate – displays duplicate values.

Clear all BREAK settings using the CLEAR command.

SQL *Plus BREAK Command Variable Examples

```

CLEAR BREAK
BREAK ON ename ON job
BREAK ON ename SKIP 4 ON job SKIP 2

```

SQL *Plus TTILE & BTITLE Command Variable Syntax

```
TTI[TLE] {text | OFF | ON}
```

SQL *Plus TTILE & BTITLE Command Variable Examples

```

TTITLE 'Salary | Report'
BTITLE 'Confidential'

```

SQL *Plus SET Command Variable Syntax

```
SET system_variable value
```

Where:

system_variable – is a variable that controls one aspect of the session environment.

value – is a value for the system variable.

SQL *Plus SET Command Variables Customizes SQL *Plus Environment

Command	Description
Operational Settings	
APPI[NFO] {OFF ON app_text}	Controls whether or not SQL*Plus automatically registers command files using the DBMS_APPLICATION_INFO package
ARRAY[SIZE] array_size	Controls the number of rows SQL*Plus fetches from the database at one time.
AUTO[COMMIT] {OFF ON IMMEDIATE statement_count}	Controls whether or not SQL*Plus automatically commits changes you make to the database and how often changes are committed.
CLOSECUR[SOR] {OFF ON}	Controls whether or not SQL*Plus closes the cursor used to execute an SQL statement after the statement has executed.
COM[PATIBILITY] {V7 V8 NATIVE}	Tells SQL*Plus the version of Oracle to which connected.

COPYC[OMMIT] batch_count	Controls how often SQL*Plus commits during execution of a COPY command. Works in conjunction with the arraysize setting.
FLAGGER {OFF ENTRY INTERMED[ATE] FULL}	Checks for conformance to ANSI/ISO SQL92 syntax.
FLU[SH] {ON OFF}	Indicates whether or not the host operating system is allowed to buffer output.
Substitution Variable Settings	
CON[CAT] {OFF ON concat_char}	Allows you to change the character used to terminate a substitution variable reference. You can also use the command to turn the feature off so that SQL*Plus doesn't recognize any character as the terminator.
DEF[INE] {OFF ON prefix_char}	Allows you to change the prefix character used to mark substitution variables. You can also use SET DEFINE to turn variable substitution off. Variable substitution is on by default, and the default prefix character is an ampersand.
ESC[APE] {OFF ON escape_char}	Used to specify the character used to escape the substitution variable prefix. Can also be used to change the escape character to something other than a backslash.
SCAN {OFF ON}	Obsolete command that allows you to choose whether or not SQL*Plus scans for substitution variables.
Large Object Settings	
LOBOF[FSET] offset	Represents an index into a LONG column. When SQL*Plus displays a LONG, it begins with the character pointed to by LOBOFFSET.
LONG long_length	Controls the number of characters displayed by SQL*Plus from any LONG columns returned by a query.
LONGC[HUNKSIZE] size	A performance related setting that controls the number of characters retrieved at one time from a long column.
Tuning & Timing Settings	
AUTOT[RACE] {OFF ON TRACE[ONLY]} [EXPLAIN] [STATISTICS]	Control whether or not SQL*Plus displays the execution plan and statistics for each SQL statement as it is executed.
TIMI[NG] {OFF ON}	Control whether or not SQL*Plus displays the elapsed time for each SQL statement or PL/SQL block executed.
Database Administration Settings	
AUTORECOVERY {OFF ON}	Causes the RECOVER command to run without user intervention as long as the archived log files are in the destination pointed to by the LOG_ARCHIVE_DEST parameter and the names conform to the

	LOG_ARCHIVE_FORMAT parameter.
LOGSOURCE logpath	Specifies the location of the archive log files and is referenced during recovery.
Miscellaneous Settings	
COPYTYPECHECK {OFF ON}	Controls whether or not SQL*Plus checks the datatypes when you use the COPY command to move data between two databases.
EDITF[ILE] edit_filename	Lets you change the name of the work file that is created when you use the EDIT command to edit the SQL statement in the buffer.
INSTANCE [service_name LOCAL]	Allows you to specify a default database to connect to when you use the CONNECT command without specifying a service name.
Report Output & Format Settings	
COLSEP column_separator	Changes the text that prints between columns of data.
EMB[EDDED] {ON OFF}	Controls the printing of embedded reports, which are those that print as if it were the continuation of a previous report.
HEADS[EP] heading_separator	Used to change the character used when defining a two-line column heading. Normally this character is a vertical bar and marks the place in a column's heading where you want a line break to occur.
HEA[DING] {ON OFF}	Controls whether or not column headings print when you SELECT or PRINT data.
LIN[ESIZE] line_width	Controls the number of characters SQL*Plus prints on one physical line. The default is 80.
NEWP[AGE] {lines_to_print NONE}	Controls the manner in which the transition from one page to the next is marked.
NULL null_text	Allows you to change the text SQL*Plus prints in a column when the value for that column is NULL.
NUMF[ORMAT] format_spec	Allows you to specify the default formatting of numeric values returned from a SELECT statement.
NUM[WIDTH] width	Controls the default width used when displaying numeric values.
PAGES[IZE] lines_on_page	Specifies the number of lines per page of output.
RECSEP {WR[APPED] EA[CH] OFF}	Tells SQL*Plus whether or not to print a record separator between each record displayed as the result of a query.
RECSEPCHAR separator_char	Changes the record separator to something other than a line of space characters.
SHIFT[INOUT] {VIS[IBLE] INV[ISIBLE]}	Controls whether or not shift characters are displayed as part of the output.
TAB {OFF ON}	Controls whether or not SQL*Plus uses tab characters when generating white space

	in terminal output.
TRIM[OUT] {ON OFF}	Controls whether or not SQL*Plus displays any trailing spaces that may occur at the end of a line.
TRIMS[POOL] {ON OFF}	Controls whether or not SQL*Plus writes trailing spaces when spooling data to a file.
TRU[NCATE] {OFF ON}	Obsolete command.
UND[ERLINE] {underline_char}{ON OFF}}	Controls the character used to underline column headings.
WRA[P] {ON OFF}	Controls how SQL*Plus prints lines that contain more characters than the current LINESIZE setting allows. With WRAP OFF, lines are truncated to match the LINESIZE setting.
Feedback Settings	
AUTOP[RINT] {OFF ON}	Controls whether or not SQL*Plus automatically prints the contents of any bind variables referenced by a PL/SQL block after it executes.
DOC[UMENT] {ON OFF}	Controls whether or not SQL*Plus prints text created with the DOCUMENT command.
ECHO {ON OFF}	Tells SQL*Plus whether or not you want the contents of command files to be echoed to the screen as they are executed.
SET ERRORS {ON OFF}	Shows compiler errors.
FEED[BACK] {OFF ON row_threshold}	Controls whether or not SQL*Plus displays the number of records returned by a query when the query selects at least n records.
PAU[SE] {ON OFF text}	Allows you to control scrolling of your terminal (You must press [Return] after seeing each pause).
SERVEROUT[PUT] {OFF ON} [SIZE buffer_size] [FORM[MAT] {WRA[PPED] TRU[NCATED]}]	Controls whether or not SQL*Plus prints the output generated by the DBMS_OUTPUT package from PL/SQL procedures.
SHOW[MODE] {ON OFF BOTH}	Controls the feedback you get when you use the SET command to change a setting.
SQLP[ROMPT] Prompt_text	Changes the SQL*Plus command prompt.
TERM[OUT] {OFF ON}	Determines whether output generated by SQL statements, PL/SQL blocks, and SQL*Plus commands are displayed on screen.
TI[ME] {OFF ON}	Controls whether or not SQL*Plus displays the current time with each command prompt.
VER[IFY] {OFF ON}	Controls whether or not SQL*Plus displays before and after images of each line that

	contains a substitution variable.
Input Settings	
BLO[CKTERMINATOR] block_term_char	Controls the character used to terminate a PL/SQL block that is being entered into the buffer for editing.
BU[FER] {buffer_name SQL}	Allows you to switch to another buffer for editing purposes.
CMDS[EP] {OFF ON separator_char}	Controls whether or not you can enter multiple commands on one line.
SQLBLANKLINES {OFF ON}	Allows SQL statements to contain embedded blank lines.
SQLC[ASE] {MIXED UPPER LOWER}	Controls whether or not SQL*Plus automatically uppercases or lowercases SQL statements and PL/SQL blocks as they are transmitted to the server for execution.
SQLCO[NTINUE] continue_prompt	Controls the prompts used when you continue a statement to a second line using the SQL*Plus continuation character.
SQLN[UMBER] {OFF ON}	Controls whether or not SQL*Plus uses the line number as a prompt when you enter a multiline SQL statement.
SQLPRE[FIX] prefix_char	Controls the SQL*Plus prefix character. The prefix character allows you to execute a SQL*Plus command while in the middle of entering an SQL statement or PL/SQL block.
SQLT[ERMINATOR] {OFF ON term_char}	Controls whether or not SQL*Plus allows you to use a semicolon to terminate and execute a SQL statement. This setting also controls the specific character used for this purpose.
SUF[IX] extension	Controls the default extension used for command files.
SHOW setting	Tells user what current SET setting is.
SQL *Plus SHOW Command Condition of SQL *Plus Environment Review Current State of Database	
Command	Description
SHOW setting	Tells user what current SET setting is.
SHOW SGA	Display information regarding the size of the System Global Area (SGA).
SHOW PARAMETER	Current settings for database initialization parameters. The parameter can be narrowed down by using the parameter name as an argument.
SHOW LOGSOURCE	Displays directory that database archive log files are being written to. Use the command ARCHIVE LOG LIST to retrieve details.
SHOW AUTORECOVERY	Displays information about archive logging and recovery.

SCRIPS & REPORTS

Sample Report Created By Script File

```
SET PAGESIZE 37
SET LINESIZE 60
SET FEEDBACK OFF
TTITLE 'Employee | Report'
BTITLE 'Confidential'
BREAK ON job
COLUMN job HEADING 'Job | Category' FORMAT A15
COLUMN ename HEADING 'Employee' FORMAT A15
COLUMN sal HEADING 'Salary' FORMAT $99,999.99
REM ** Insert SELECT statement
SELECT job, ename
FROM emp
WHERE sal < 3000
ORDER BY job, ename
/
SET FEEDBACK ON
REM clear all formatting commands
```

PL/SQL

PL/SQL BLOCK STRUCTURE

PL/SQL Block Structure		
Section	Description	Inclusion
Declarative	Contains all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and declarative sections.	Optional
Executable	Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block.	Mandatory
Exception	Specifies the actions to perform when errors and abnormal conditions arise in the executable section.	Optional

PL/SQL Block Structure Syntax

```
DECLARE
  v_variable VARCHAR2(5);
BEGIN
  SELECT column_name
  INTO v_variable
  FROM table_name;
EXCEPTION
  WHEN exception_name
  THEN .....
END;
```

Executing Statements & PL/SQL Blocks From SQL *Plus

- Place a semicolon (;) at the end of a SQL statement or PL/SQL control statement.
- Use a slash (/) to run the anonymous PL/SQL block in the SQL *Plus buffer.
- Place a period (.) to close a SQL *Plus buffer. A PL/SQL block is treated as one continuous

statement in the buffer, and the semicolons within the block do not close or run the buffer.

BLOCK TYPES

There are three types of block types:

- Anonymous Blocks
- Subprograms (Procedures and Functions)
- Packages

Anonymous

```
[DECLARE]
BEGIN
  statements
[EXCEPTION]
END;
```

Procedure

```
PROCEDURE name
IS
BEGIN
  Statements
[EXCEPTION]
END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
  statements
  RETURN value;
[EXCEPTION]
END:
```

Procedures and Functions can also be declared as follows:

```
CREATE OR REPLACE PROCEDURE [FUNCTION]
name
IS
```

VARIABLES

PL/SQL supports five datatype categories:

- Scalar
- Composite
- Reference (Pointer)
- LOB (Large Objects)
- SQL *Plus bind and host variables

Declaring PL/SQL Variables Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
[:= | DEFAULT expr];
```

where:

identifier is the name of the variable

CONSTANT constrains the variable so that its value cannot change. Constants must be initialized.

datatype is a scalar, composite, reference, or LOB.

NOT NULL constrains the variable so that it must

contain a value.

expr is any PL/SQL expression that can be literal, another variable, or an expression involving operators and functions.

Declaring PL/SQL Variables Examples

```

DECLARE
v_hiredate DATE
v_deptno NUMBER(2) NOT NULL := 10;
v_location VARCHAR2(13) := 'Atlanta';
c_comm. CONSTANT NUMBER := 1400;
v_job VARCHAR2(9);
v_count BINARY_INTEGER := 0;
v_total_sal NUMBER(9, 2) := 0;
v_orderdate DATE := SYSDATE + 7;
c_tax_rate CONSTANT NUMBER(3, 2) := 8.25;
v_valid BOOLEAN NOT NULL := TRUE;

```

PL/SQL DATA TYPES

Scalar Datatype Variables	
Datatype	Description
NUMBER(p,s)	Base type for fixed and floating point numbers.
VARCHAR2(size)	Variable length character value of maximum size of 32,767 bytes. There is no default size.
CHAR(size)	Base type for fixed-length character data up to 32, 767 bytes. If maximum length is not specified, it is set to 1.
DATE	Date and time value between January 1, 4712 B.C. and December 31, 9999 A.D.
LONG	Variable length character data up to 2 gigabytes.
LONG RAW	Base type for binary data and byte strings up to 32,760 bytes. LONG RAW data is not interpreted by PL/SQL.
BOOLEAN	Base type that stores one of three possible values used for logical calculations: TRUE, FALSE, or NULL.
BINARY_INTEGER	Base type for integers between -2,147, 483,647 and 2,147,483,647.
PLS_INTEGER	Base type for signed integers between -2,147, 483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER and BINARY_INTEGER values.
LOB Datatype Variables	
CLOB	Book or text, the CLOB (character large object) datatype is used to store large blocks of single-byte character data in the database.
BLOB	Photo, the BLOB (binary large object) datatype is used to store large binary objects in the database in line (inside the row) or out of line (outside the row).
BFILE	Movie, the BFILE (binary file) datatype is used to store large binary objects in operating system files outside the database.

NCLOB

NCLOB (national language character large object) datatype is used to store large blocks of single-byte or fixed-width multibyte NCHAR data in the database, in line or out of line.

%TYPE Attribute

You can use the %TYPE attribute to declare a variable according to another previously declared variable or database column.

Examples:

```

v_ename emp.ename%TYPE
v_balance NUMBER(7, 2);
v_nim_balance v_balance%TYPE := 10;

```

BIND Variables

A BIND variable is a variable declared in the host environment and then used to pass runtime values, either number or character, into or out of one or more PL/SQL programs, which can use it as they would use any variable. **You can reference variables declared in the host or calling environment in PL/SQL statements, unless the statement is in a procedure, function, or package.** A BIND variable is referenced by prefixing with a colon (:). To display the current value of a BIND variable in the SQL *Plus environment, use the PRINT command:

BIND Variable Example:

```

VARIABLE g_monthly_sal NUMBER
ACCEPT p_monthly_sal PROMPT 'Enter: '

DECLARE
    v_sal NUMBER(9, 2) := &p_annual_sal;
BEGIN
    :g_monthly_sal := v_sal / 12;
END;
/
PRINT g_monthly_sal

```

DBMS_OUTPUT.PUT_LINE

- An alternative to BIND variables.
- An Oracle supplied packages procedure.
- Must be enabled in SQL *Plus with SET SERVEROUTPUT ON

DBMS_OUTPUT.PUT_LINE Example:

```

SET SERVEROUTPUT ON
ACCEPT p_annual_sal PROMPT 'Enter: '

DECLARE
    v_sal NUMBER(9, 2) := &p_annual_sal;
BEGIN
    v_sal := v_sal / 12;
    DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' || v_sal);

```

```
END;  
/
```

SELECT Statement & INTO Clause

SELECT Statement Syntax

```
SELECT select_list  
INTO {variable_name[, variable_name] . . .  
      [record_name]}  
FROM table  
WHERE condition;
```

where:

select_list is a list of at least one column and can include SQL expressions, row functions, or group functions

variable_name is the scalar variable to hold the retrieved value.

record_name is the PL/SQL RECORD to hold retrieved values.

table specifies the database table name.

condition is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants.

- The INTO clause is required and must be between the SELECT and FROM clauses.
- **Specify the same number of output variables in the INTO clause as database columns in the SELECT clause, and be sure they correspond positionally and that datatypes are compatible. The %TYPE attribute is useful to assure this.**
- Queries must return one row and only one row.
- Terminate each SQL statement with a semicolon(;);

SELECT Statement Examples

```
DECLARE  
  v_deptno NUMBER(2);  
  v_loc VARCHAR2(15);  
  
BEGIN  
  SELECT deptno, loc  
  INTO v_deptno, v_loc  
  FROM dept  
  WHERE dname = 'SALES';  
  
END;
```

```
DECLARE  
  v_sum_sal emp.sal%TYPE;  
  v_deptno NUMBER NOT NULL := 10;
```

```
BEGIN  
  SELECT SUM(sal)  
  INTO v_sum_sal
```

```
FROM emp  
WHERE deptno = v_deptno;
```

```
END;
```

INSERT, UPDATE, & DELETE Statements

INSERT, UPDATE, DELETE Statement Examples

```
BEGIN  
  INSERT INTO emp(empno, ename, job, deptno)  
  VALUES( empno_sequence.NEXTVAL, 'HARDING',  
  'CLERK', 10);  
  
END;
```

```
DECLARE  
  v_sal_increase emp.sal%TYPE;  
  v_deptno NUMBER NOT NULL := 10;
```

```
BEGIN  
  UPDATE emp  
  SET sal = sal + v_sal_increase  
  WHERE job = 'ANALYST';  
  
END;
```

```
DECLARE  
  v_deptno emp.deptno%TYPE := 10;
```

```
BEGIN  
  DELETE FROM emp  
  WHERE deptno = v_deptno;
```

```
END;
```

PROGRAM FLOW CONTROL

IF Statement Syntax

```
IF condition THEN  
  statements;  
[ELSIF condition THEN  
  statements;]  
[ELSE  
  statements;]  
END IF;
```

where:

condition is a Boolean variable or expression (TRUE, FALSE, or NULL).

THEN is a clause that associates the Boolean expression that precedes it with the sequence of statements that follow it.

Statements can be one or more PL/SQL or SQL statements (They may include further IF statements containing several nested ifs, ELSEs, and ELSIFs).

ELSIF is a keyword that introduces a Boolean

expression (If the first condition yields FALSE or NULL then the ELSIF keyword introduces additional conditions).

ELSE is a keyword that if control reaches it, the sequence of statements that follows it is executed.

IF Statement Example

```
IF v_name = 'MILLER' THEN
  v_job := 'SALESMAN';
  v_deptno := 35;
  v_new_comm. := sal * 0.20;
END IF;
```

IF-THEN-ELSE Statement Example

```
IF v_ename = 'KING' THEN
  v_job := 'MANAGER';
ELSE
  v_job := 'CLERK';
END IF;
```

IF-THEN-ELSIF Statement Example

```
IF v_deptno = 10 THEN
  v_comm. := 5000;
ELSIF v_deptno = 20 THEN
  v_comm. := 7500;
ELSE
  v_comm. := 2000;
END IF;
```

LOOP Statement Syntax

```
LOOP                               -- delimiter
  statement1                         -- statements
  ...
  EXIT [WHEN condition];           -- EXIT statement
END LOOP;                           -- delimiter
```

where:

Condition is a Boolean variable or expression (TRUE, FALSE, or NULL);

LOOP Statement Example

```
DECLARE
  v_orcid item.ordid%TYPE := 601;
  v_counter NUMBER(2) := 1;

BEGIN
  LOOP
    INSERT INTO item(ordid, itemid)
    VALUES(v_orcid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP
END;
```

FOR LOOP Statement Syntax

```
FOR counter in {REVERSE}
  lower_bound . . . . upper_bound LOOP
  statement1;
  statement2;
  ...
END LOOP;
```

where:

counter is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached.

REVERSE causes the counter to decrement with each iteration from the upper bound to the lower bound (lower bound is still referenced first).

lower_bound specifies the lower bound for the range of counter values.

upper_bound specifies the upper bound for the range of counter values.

FOR LOOP Statement Examples

```
DECLARE
  v_lower NUMBER := 1;
  v_upper NUMBER := 100;

BEGIN
  FOR I IN v_lower . . v_upper LOOP
    ...
  END LOOP;
END;
```

```
DECLARE
  v_orcid item.ordid%TYPE := 601;

BEGIN
  FOR I IN 1 . . 10 LOOP
    INSERT INTO item(ordid, itemid)
    VALUES (v_orcid, I);
  END LOOP;
END;
```

WHILE LOOP Statement Syntax

```
WHILE condition LOOP
  Statement1;
  Statement2;
  ...
END LOOP;
```

where:

condition is a Boolean variable or expression (TRUE, FALSE, or NULL).

statement can be one or more PL/SQL or SQL statement.

If the variable involved in the conditions do not change during the body of the loop, then the condition remains TRUE and the loop does not terminate. If the condition yields NULL, the loop is bypassed and control is passed to the next statement.

WHILE LOOP Statement Example

```
ACCEPT p_new_order PROMPT 'Enter the order
number: '
ACCEPT p_items -
  PROMPT 'Enter the number of items in this order: '

DECLARE
v_count NUMBER(2) := 1;

BEGIN
  WHILE v_count <= &p_items LOOP
    INSERT INTO item(ordid, itemid)
      VALUES (&p_new_order, v_count);
    v_count := v_count + 1;
  END LOOP;
  COMMIT;
END;
/
```

NESTED LOOP Statement Example

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ....
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
```

FOR LOOP Statement Examples

```
DECLARE
  v_lower NUMBER := 1;
  v_upper NUMBER := 100;

BEGIN
  FOR I IN v_lower .. v_upper LOOP
    ....
  END LOOP;
END;

DECLARE
  v_ordid item.ordid%TYPE := 601;
```

```
BEGIN
  FOR I IN 1 .. 10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES (v_ordid, I);
  END LOOP;
END;
```

COMPOSITE DATATYPES

There are four COMPOSITE datatypes:

- TABLE
- RECORD
- NESTED TABLE
- VARRAY

You use the RECORD datatype to treat related but dissimilar data as a logical unit. You use the TABLE datatype to reference and manipulate collections of data as a whole object.

PL/SQL RECORDS (Structures)

A record is a group of related data items stored in fields, each with its own name and datatype. A record lets you treat data as a logical unit and manipulate it as a unit. A record:

- Can have as many fields as necessary.
- Can be assigned initial values and can be defined as NOT NULL.
- Fields without initial values are initialized to NULL.
- Fields can be defined using the DEFAULT keyword.
- You can define RECORD types and declare user-defined records in the declarative part of any block, subprogram, or package.
- You can declare and reference nested records. A record can be the component of another record.
- You can assign a list of common values to a record by using the SELECT or FETCH statement. Make sure that the column names appear in the same order as the fields in the record. You can also assign one record to another if they have the same datatype. A user-defined record and a %ROWTYPE record never have the same datatype.

RECORD Syntax

```
DECLARE
TYPE type_name IS RECORD
  (field_declaration [, field_declaration] . . . );
```

Where field_declaration is:

```
field_name {field_type | variable%TYPE
  | table.column%TYPE | table%ROWTYPE}
  [ [NOT NULL] {:= | DEFAULT} expr]
```

where:

type_name is the name of the RECORD type (This identifier is used to declare records).

field_name is the name of a field within a record.

field_type is the datatype of the field (It represents any

PL/SQL datatype except REF CURSOR. You can use %TYPE and %ROWTYPE attributes).

expr is the field_type or initial value.

The NOT NULL constraint prevents the assigning of nulls to those fields. **Be sure to initialize NOT NULL fields.**

RECORD Example

DECLARE

```
TYPE emp_record_type IS RECORD
  (ename VARCHAR2(10),
   job emp.job%TYPE,
   sal NUMBER(7, 2) );
emp_record emp_record_type;
```

Fields in a record are reference and accessed by name using the dot notation:

```
emp_record.job := 'CLERK';
```

%ROWTYPE Attribute

To declare a record based on a collection of columns in a database table or view, you use the %ROWTYPE attribute. The fields in the record take their names and datatypes from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

Syntax:

```
identifier reference%ROWTYPE;
```

where:

identifier is the name chosen for the record as a whole.

reference is the name of the table, view, cursor, or cursor variable on which the record is to be based.

Fields in a ROWTYPE% record are reference and accessed by name using the dot notation:

```
emp_record.comm := 750;
```

Examples:

```
emp_record emp%ROWTYPE
dept_record dept%ROWTYPE
```

```
DECLARE
emp_rec emp%ROWTYPE
```

```
BEGIN
SELECT * INTO emp_rec
FROM emp
WHERE empno = &employee_number;
```

```
INSERT INTO retired_emps(empno, ename, job, mgr,
```

```
hiredate, leavedate, sal, comm., deptno)
VALUES(emp_rec.empno, emp_rec.ename,
emp_rec.job, emp_rec.mgr, emp_rec.hiredate,
SYSDATE, emp_rec.sal, emp_rec.comm.,
emp_rec.deptno);
COMMIT;
END;
```

PL/SQL TABLES (Hash or Array)

A PL/SQL table:

- Is similar to an array.
- Must contain two components: 1) A primary key of datatype BINARY_INTEGER that indexes the PL/SQL TABLE and 2) A column of a scalar or record datatype, which stores the PL/SQL TABLE elements.
- Can increase dynamically because it is unconstrained.

TABLE Syntax

DECLARE

```
TYPE type_name IS TABLE OF
  {column_type | variable%TYPE
  | table.column%TYPE} [NOT NULL]
  [INDEX BY BINARY_INTEGER];
```

```
identifier type_name;
```

where:

type_name is the name of the TABLE type (It is a type specifier used in subsequent declarations of PL/SQL tables).

column_type is any scalar (not composite) datatype such as VARCHAR2, DATE, NUMBER or %TYPE attribute.

identifier is the name of the identifier that represents an entire PL/SQL table.

The NOT NULL constraint prevents nulls from being assigned to the PL/SQL TABLE of that type. **Do not initialize the PL/SQL TABLE.**

TABLE Example

DECLARE

```
TYPE ename_table_type IS TABLE OF
  emp.ename%TYPE
  INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
```

```
TYPE date_table_type IS TABLE OF DATE
  INDEX BY BINARY_INTEGER;
date_table date_table_type;
```

Fields in a record are reference and accessed by the PL/SQL tablename and primary key value:

```
ename_table(3);
```

```

DECLARE
TYPE emp_name_table_type IS TABLE OF
    emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
emp_name_table emp_name_table_type hiredate_table
hiredate_table_type;
BEGIN
emp_name_table(1) := 'CAMERON';
hiredate_table(8) := SYSDATE + 7;
IF emp_name_table.EXISTS(1) THEN
INSERT INTO . . . . .
. . . . .
END;

```

The table EXISTS(i) statement returns TRUE if at least one row with index I is returned. Use the EXISTS statement to prevent an error that is raised in reference to a non-existing table element.

record that represents a row in a database table. The difference between the %ROWTYPE attribute and the composite datatype RECORD is that RECORD allows you to specify the datatypes of fields in the record or to declare fields of your own.

```

DECLARE
TYPE e_table_type IS TABLE OF emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
e_tab e_table_type;

BEGIN
e_tab(1) := 'SMITH';
UPDATE emp
SET sal = 1.1 * sal
WHERE Ename = e_tab(1);
COMMIT;
END;
/

```

PL/SQL TABLE METHODS	
table_name.method_name[(parameters)]	
Method	Description
EXISTS(n)	Returns TRUE if the nth element in a PL/SQL table exists.
COUNT	Returns the number of elements that a PL/SQL table currently contains.
FIRST LAST	Returns the first and last (smallest and largest) index numbers in a PL/SQL table. Returns NULL if the PL/SQL table is empty.
PRIOR(n)	Returns the index number that precedes index n in a PL/SQL table.
NEXT(n)	Returns the index number that succeeds index n in a PL/SQL table.
EXTEND(n, I)	Increases the size of a PL/SQL table. EXTEND appends one null element to a PL/SQL table. EXTEND(n) appends n null elements to a PL/SQL table. EXTEND(n, I) appends n copies of the ith element to a PL/SQL table.
TRIM	TRIM removes one element from the end of a PL/SQL table. TRIM(n) removes n elements from the end of a PL/SQL table.
DELETE	DELETE removes all elements from a PL/SQL table. DELETE(n) removes the nth element from a PL/SQL table. DELETE(m, n) removes all elements in the range m . . n from a PL/SQL table.

SQL CURSOR

Whenever you issue a SQL statement, the Oracle Server opens a work area called private SQL areas in memory in which the command is parsed and executed. This area is called a cursor.

When the executable part of the block issues a SQL statement, PL/SQL creates an **implicit** cursor, which has the SQL identifier. PL/SQL manages this cursor automatically. The programmer can **explicitly** declare and name an explicit cursor.

You can use PL/SQL cursors to name a private SQL area and access its stored information. The cursor directs all phases of processing.

PL/SQL Cursors	
Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row. PL/SQL lets you refer to the most recent implicit cursor as the SQL cursor. You cannot use OPEN, FETCH, and CLOSE statements to control the SQL cursor, but you can use cursor attributes to get information about the most recently executed SQL statement.
Explicit	For queries that return more than one row. Explicit cursors are declared and named by the programmer and manipulated through specific statements in the block's executable actions.

TABLE OF RECORDS Syntax and Examples

```

table(index).field
dept_table(15).loc := 'Atlanta';
DECLARE
TYPE dept_table_type IS TABLE OF dept%ROWTYPE
    INDEX BY BINARY_INTEGER;
dept_table dept_table_type;
-- each element of dept_table is a record

```

You can use the %ROWTYPE attribute to declare a

There are four attributes available in PL/SQL that can be applied to cursors:

SQL Cursor Attributes	
SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value).
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.

SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows.
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.

SQL Cursor Examples

```
VARIABLE rows_deleted VARCHAR2(30)

DECLARE
  v_ordid NUMBER := 605;
BEGIN
  DELETE FROM item
  WHERE ordid = v_ordid;
  :rows_deleted := (SQL%ROWCOUNT || ' rows
deleted. ');
END;
/
PRINT rows_deleted

DECLARE
  v_sal_increase emp.sal%TYPE;
  v_deptno NUMBER NOT NULL := 10;
BEGIN
  UPDATE emp
  SET sal = sal + v_sal_increase
  WHERE job = 'ANALYST';
END;

DECLARE
  v_deptno emp.deptno%TYPE := 10;
BEGIN
  DELETE FROM emp
  WHERE deptno = v_deptno;
END;
```

EXPLICIT CURSORS

Use explicit cursors to individually process each row returned by a multiple-row SELECT statement. The set of rows returned by a multiple-row query is called the active set. Its size is the number of rows that meet the search criteria. An explicit cursor points to the current row in the active set. This allows the program to process the rows one at a time.

A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in the active set.

Explicit cursors are controlled using four commands:

- 1) Declare the cursor by naming it and defining the structure of the query to be performed within it.
- 2) Open the cursor. The OPEN statement executes the query and binds any variables that are referenced. Rows identified by the query are called the active set and are now available for fetching.
- 3) Fetch data from the cursor using the FETCH command. After each fetch you test the cursor for any existing row. If there are no more rows to process, you need to close the cursor.

- 4) Close the cursor. The CLOSE command releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

OPEN STATEMENT

Open the cursor to execute the query and identify the active set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the active set.

OPEN is an executable statement that performs the following operations:

- Dynamically allocates memory for a context area that eventually contains crucial processing information.
- Parses the SELECT statement.
- Binds the input variable, or sets the value for the input variables by obtaining their memory addresses.
- Identifies the active set, or the set of rows that satisfy the search criteria. Rows in the active set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
- Positions the pointer just before the first row in the active set.

FETCH INTO STATEMENT

The FETCH statement retrieves the rows in the active set one at a time. After each fetch, the cursor advances to the next row in the active set.

Guidelines for FETCH:

- Include the same number of variables in the INTO clause of the FETCH statement as columns in the SELECT statement., and be sure that the datatypes are compatible.
- Match each variable to correspond to the columns positionally.
- Alternatively, define a record for the cursor and reference the record in the FETCH INTO clause.
- Test to see if the cursor contains rows. If a fetch acquires no values, there are no rows left to process in the active set and no error is recorded.

The FETCH statement performs the following operations:

- Advances the pointer to the next row in the active set.
- Reads the data for the current row into the output PL/SQL variables.

You use the FETCH statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their datatypes must be compatible.

CLOSE STATEMENT

The **CLOSE** statement disables the cursor, and the active set becomes undefined. Close the cursor after completing the processing of the **SELECT** statement. This step allows the cursor to be reopened if required. Therefore, you can establish an active set several times.

Explicit SQL Cursor Examples

-- syntax

```
DECLARE
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
  -- declare the cursor
  CURSOR emp_cursor IS
    SELECT empno, ename
    FROM emp;
BEGIN
  -- open the cursor
  OPEN emp_cursor;
  FOR I IN 1..10 LOOP
  -- fetch the rows
    FETCH emp_cursor INTO v_empno, v_ename;
  END LOOP;
  -- close the cursor
  CLOSE emp_cursor;
END;
```

-- utilizing cursor attributes

```
DECLARE
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
  CURSOR emp_cursor IS
    SELECT empno, ename
    FROM emp;
BEGIN
  -- test cursor to see if open
  IF NOT emp_cursor%ISOPEN THEN
    OPEN emp_cursor;
  END IF;
  -- use %ROWCOUNT & %NOTFOUND to end loop
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename;
    EXIT WHEN emp_cursor%ROWCOUNT > 10
    OR emp_cursor %NOTFOUND;
  -- alternative
  EXIT WHEN emp_cursor%NOTFOUND
  OR emp_cursor %NOTFOUND IS NULL;
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

Explicit SQL Cursor Example

```
SET SERVEROUTP ON
ACCEPT p_ordid PROMPT 'Enter order number: '

DECLARE
  v_prodid item.prodid%TYPE;
```

```
v_item_total NUMBER(11, 2);
v_order_total NUMBER(11, 2) := 0;
-- declare the cursor
CURSOR item_cursor IS
  SELECT prodid, actualprice * qty
  FROM item
  WHERE ordid = &pordid;
BEGIN
  -- open the cursor
  OPEN item_cursor;
  LOOP
    FETCH item_cursor INTO v_prodid, v_item_total;
    EXIT WHEN item_cursor%ROWCOUNT > 5 OR
item_xursor%NOTFOUND;
    v_order_total := v_order_total + v_item_total;
    DBMS_OUTPUT.PUT_LINE( 'Product number ' ||
      TO_CHAR(v_prodid) ||
      ' brings this order to a total of ||
      TO_CHAR(v_order_total, '$999,9999.99') );
  END LOOP
  -- close the cursor
  CLOSE item_cursor;
END;
/
```

Cursors and Records

You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set because you can simply fetch into the record. Therefore, the values of the rows are loaded directly into the corresponding fields of the record.

Explicit SQL Cursor Record Example

-- utilize records in a cursor

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename
    FROM emp;
  emp_record emp_cursor%ROWTYPE
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    INSERT INTO temp_list (empid, empname)
    VALUES(emp_record.empno,
      emp_record.ename);
  END LOOP;
  COMMIT;
  CLOSE emp_cursor;
END;
```

CURSOR FOR LOOPS

A cursor **FOR** loop processes rows in an explicit cursor. It is a shortcut because the cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched.

Guidelines:

- Do not declare the record that controls the loop. Its scope is only in the loop.
- Test the cursor attributes during the loop if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement.
- Do not use a cursor FOR loop when the cursor operations must be handled manually.
- You can define a query at the start of the loop itself. The query expression is called a SELECT sub statement, and the cursor is internal to the FOR loop. Because the cursor is not declared with a name, you cannot test its attributes.

Explicit SQL Cursor FOR LOOP Examples

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
-- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' ||
        emp_record.ename ||
        ' works in the sales Dept. ');
    END IF;
  END LOOP; -- implicit close occurs
END;
/
-- using a subquery
SET SERVEROUTPUT ON
BEGIN
  FOR emp_record IN (SELECT ename, deptno
    FROM emp) LOOP
-- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' ||
        emp_record.ename ||
        ' works in the sales Dept. ');
    END IF;
  END LOOP; -- implicit close occurs
END;
/
```

CURSORS WITH PARAMETERS

Parameters allow values to be passed to a cursor when it is opened and to be used in the query when it executes. This means you can open and close an explicit cursor several times in a block, returning a different active set on each occasion.

Each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. Parameter datatypes are the same as those for scalar variables but you don't give them sizes. The

parameter names are for references in the query expression of the cursor.

When the cursor is opened, you pass values to each of the parameters positionally. You can pass values from PL/SQL or host variables as well from literals. You can also pass parameters to the cursor used in a cursor FOR loop.

The use of parameters is useful when the same cursor is referenced repeatedly.

CURSOR PARAMETERS Syntax

```
CURSOR cursor_name
  [ (parameter_name datatype, . . . ) ]
IS
select statement;
```

where:

cursor_name is a PL/SQL identifier for the previously declared cursor.

parameter_name is the name of a parameter. (Parameter stands for the following syntax:

```
cursor_parameter_name [IN] datatype [(:= | DEFAULT)
expr]
```

datatype is a scalar datatype of the parameter.

select_statement is a SELECT statement without the INTO clause.

CURSOR PARAMETERS Examples

```
DECLARE
  CURSOR emp_cursor (p_deptno NUMBER,
p_job VARCHAR2) IS
  SELECT empno, ename
  FROM emp
  WHERE deptno = v_deptno
  AND job = v_job;
BEGIN
  OPEN emp_cursor(10, 'CLERK');

DECLARE
  v_emp_job emp.job%TYPE := 'CLERK';
  v_ename emp.ename%TYPE;
  CURSOR emp_cursor(p_deptno NUMBER,
  p_job VARCHAR2) IS
  SELECT . . .
```

-- open the cursor

```
OPEN emp_cursor(10, v_emp_job);
OPEN emp_cursor(20, 'ANALYST');
```

-- pass parameters to cursor in cursor FOR LOOP

```
DECLARE
  CURSOR emp_cursor(p_deptno NUMBER,
  p_job VARCHAR2) IS
```

```

SELECT ...
BEGIN
  FOR emp_record IN emp_cursor(10, 'CLERK') LOOP
  .....
```

FOR UPDATE OF CLAUSE

Add the FOR UPDATE clause in the cursor query to lock affected rows when the cursor is opened. Because the Oracle Server releases locks at the end of the transaction, you should not commit across fetches from an explicit cursor if FOR UPDATE is used. FOR UPDATE is the last clause in a SELECT statement, even after the ORDER BY clause.

When querying multiple tables, you can use the FOR UPDATE clause to confine row locking to particular tables. Rows in a table are locked only if the FOR UPDATE clause refers to a column in that table. **Exclusive row locks are taken on the rows in the active set before the OPEN returns when the FOR UPDATE clause is used.**

If the Oracle Server cannot acquire the locks on the rows it needs in a SELECT FOR UPDATE, it waits indefinitely. You can use the NOWAIT clause in the SELECT FOR UPDATE statement and text for the error code that returns because of failure to acquire the locks in a loop. If you have a large table, you can achieve better performance by using the LOCK TABLE statement to lock all rows in the table. When you use LOCK TABLE, you cannot use the WHERE CURRENT OF clause and you must use the notation WHERE column = identifier.

It is not mandatory that the FOR UPDATE OF clause refers to a column, but it is recommended for readability and maintenance.

FOR UPDATE OF CLAUSE Syntax

```

SELECT ....
FROM FOR UPDATE [OF column_reference]
[NOWAIT];
```

where:

column_reference is a column in the table against which the query is performed (list of columns may also be used).

NOWAIT returns an Oracle error if the rows are locked by another session.

FOR UPDATE OF CLAUSE Example

```

DECLARE
  CURSOR emp_cursor IS
    SELECT emp_cursor
  FROM emp
  WHERE deptno = 30
  FOR UPDATE OF sal NOWAIT;
```

WHERE CURRENT OF CLAUSE

When referencing the current row from an explicit cursor, use the WHERE CURRENT OF clause. This allows you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference ROWID. You must include the FOR UPDATE clause in the cursor query so that the rows are locked on OPEN.

The WHERE CURRENT OF clause in the UPDATE or DELETE statements refers to the currently fetched record.

WHERE CURRENT OF Syntax and Example

WHERE CURRENT OF cursor;

where:

cursor is the name of a declared cursor (the cursor must have been declared with the FOR UPDATE clause).

```

DECLARE
  CURSOR sal_cursor IS
    SELECT sal
  FROM emp
  WHERE deptno = 30
  FOR UPDATE OF sal NOWAIT;
BEGIN
  FOR emp_record IN sal_cursor LOOP
    UPDATE emp
      SET sal = emp_record.sal * 1.10
      WHERE CURRENT OF sal_cursor;
  END LOOP;
END;
/
```

HANDLING EXCEPTIONS

EXCEPTION TYPES		
Exception	Description	Handling
Predefined Oracle Server error Implicitly Raised	One of about 20 errors that occur most often in PL/SQL code.	Do not declare and allow the Oracle Server to raise them implicitly.
Non-predefined Oracle Server error Implicitly Raised	Any other standard Oracle Server error.	Declare within the declarative section and allow the Oracle Server to raise them implicitly.
User-defined error Explicitly Raised	A condition the developer determines is abnormal.	

Trapping Exceptions Syntax

```

EXCEPTION
  WHEN exception1 [OR exception2 . . . ] THEN
    statement1;
    statement2;
  .....
```

```
[WHEN exception3 [OR exception4 . . . ] THEN
  statement1;
  statement2;
  . . . .
[WHEN OTHERS THEN
  statement1;
  statement2;
  . . . .]
```

		returned more than one row.
VALUE_ERROR	ORA-06502	Arithmetic conversion, truncation, or size constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero.

Trapping Predefined Oracle Server Errors

Trap a predefined Oracle Server error by referencing its standard name within the corresponding exception-handling routine. It is good to always consider the NO_DATA_FOUND and TOO_MANY_ROWS exceptions which are most common.

Predefined Exceptions Syntax

```
BEGIN
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
    statement2;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement2;
END;
```

Trapping Non-Predefined Oracle Server Errors

PREDEFINED ORACLE SERVER EXCEPTIONS		
Exception Name	Oracle Error #	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor.
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value.
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails.
LOGIN_DENIED	ORA-01017	Logging on to Oracle with an invalid username or password.
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data.
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to Oracle.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	Host error variable and PL/SQL cursor variable involved in an assignment have incompatible return types.
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or varray element using an index number that is outside the legal range (-1 for example).
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while Oracle is waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single row SELECT

Non-Predefined Exceptions Syntax and Example

1. Declare the name for the exception within the declarative section:

exception EXCEPTION;
2. Associate the declared exception with the standard Oracle Server error number using the PRAGMA EXCEPTION_INIT statement:

PRAGMA EXCEPTION_INIT(exception, error_number);
3. Reference the declared exception within the corresponding exception-handling routine.

```
DECLARE
  e_emps_remaining EXCEPTION;
  PRAGMA EXCEPTION_INIT(
    e_emps_remaining, -2292);

BEGIN
  DELETE FROM dept
  WHERE deptno = v_deptno;
  COMMIT;

EXCEPTION
  WHEN e_emps_remaining THEN
    DBMS_OUTPUT.PUT_LINE('Error Message');
END;
```

When an exception occurs, you can identify the associated error code or error message by using two

functions. Based on the values of the code message, you can decide what action to take based on the error.

ERROR TRAPPING FUNCTIONS	
Function	Description
SQLCODE	Returns the numeric value for the error code. This can be assigned to a NUMBER variable.
SQLERRM	Returns character data containing the message associated with the error number. TRUNCATE the value of SQLERRM to a known length before attempting to write it to a variable.

Functions For Trapping Exceptions Example

```

DECLARE
  v_error_code NUMBER;
  v_error_message VARCHAR2(255);

BEGIN
  .....
EXCEPTION
  .....
WHEN OTHERS THEN
  ROLLBACK;
  v_error_code := SQLCODE;
  v_error_message := SQLERRM;

  -- place error data in user_errors table
  INSERT INTO errors
  VALUES(v_error_code, v_error_message);
END;
```

Trapping User-Defined Exceptions

User-Defined Exceptions Syntax and Example

1. Declare the name for the user-defined exception within the declarative section:

```
Exception EXCEPTION;
```

2. Use the RAISE statement to raise the exception explicitly within the executable section:

```
RAISE exception;
```

3. Reference the declared exception within the corresponding exception handling routine.

```

DECLARE
  e_invalid_product EXCEPTION;

BEGIN
  UPDATE product;
  SET descrip = '&product_description'
  WHERE prodid = &product_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_product;
  END IF;
  COMMIT;
```

```

EXCEPTION
  WHEN e_invalid_product THEN
    DBMS_OUTPUT.PUT_LINE('Error Message');
END;
```

Propagating Exceptions

When a subblock handles an exception, it terminates normally, and control resumes in the enclosing block immediately after the subblock END statement.

If PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates in successive enclosing blocks until it finds a handler. If none of these blocks handle the exception, an unhandled exception in the host environment results. When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

RAISE_APPLICATION_ERROR Procedure

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

RAISE_APPLICATION_ERROR may be used in either the executable section or exception section.

RAISE_APPLICATION_ERROR Syntax and Example

```
raise_application_error(error_number, message[ , {TRUE | FALSE}]);
```

where:

error_number is a user specified number for the exception between -20000 and -20999.

message is the user-specified message for the exception. It is a character string up to 2,048 bytes long.

TRUE | FALSE is an optional Boolean parameter (If TRUE, the error is placed on the stack of previous errors. If FALSE, the default, the error replaces all previous errors).

```

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20201,
    'Error Message.')
```

END;

```
DELETE FROM emp
WHERE mgr = v_mgr;
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20202, 'Message');
END IF;
```