CHAPTER 1 : SQL BASICS

SQL - Overview

SQL tutorial gives unique learning on **Structured Query Language** and it helps to make practice on SQL commands which provides immediate results. SQL is a language of database, it includes database creation, deletion, fetching rows and modifying rows etc.
SQL is an ANSI (American National Standards Institute) standard but there are many different versions of the SQL language.

What is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in relational database.
SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server use SQL as standard database language.
Also, they are using different dialects, such as:

- MS SQL Server using T-SQL,
- Oracle using PL/SQL,
- MS Access version of SQL is called JET SQL (native format) etc.

Why SQL?

- Allows users to access data in relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in database and manipulate that data.
- Allows to embed within other languages using SQL modules, libraries & pre-compilers.
- Allows users to create and drop databases and tables.
- Allows users to create view, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures, and views

History

- **1970 --** Dr. Edgar F. "Ted" Codd of IBM is known as the father of relational databases. He described a relational model for databases.
- **1974 --** Structured Query Language appeared.
- **1978 --** IBM worked to develop Codd's ideas and released a product named System/R.

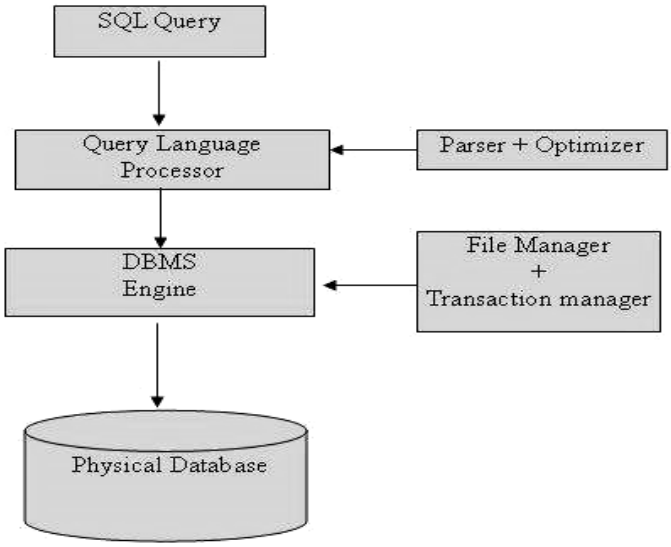
- **1986 --** IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software and its later becoming Oracle.

SQL Process

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries but SQL query engine won't handle logical files.

Following is a simple diagram showing SQL Architecture



SQL Commands

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their nature:

DDL - Data Definition Language:

Command	Description
CREATE	Creates a new table, a view of a table, or other object in database
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other object in the database.

DML - Data Manipulation Language:

Command	Description
SELECT	Retrieves certain records from one or more tables
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records

DCL - Data Control Language:

Command	Description
GRANT	Gives a privilege to user
REVOKE	Takes back privileges granted from user

SQL is followed by unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax:
All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and the entire statements end with a semicolon (;).
Important point to be noted is that SQL is **case insensitive**, which means SELECT and select have same meaning in SQL statements, but MySQL makes difference in table names. So if you are working with MySQL, then you need to give table names as they exist in the database.

SQL SELECT Statement

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQL DISTINCT Clause

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQL WHERE Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE CONDITION;
```

SQL AND/OR Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE CONDITION-1 {AND|OR} CONDITION-2;
```

SQL IN Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE column_name IN (val-1, val-2,...val-N);
```

SQL BETWEEN Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE column_name BETWEEN val-1 AND val-2;
```

SQL LIKE Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE column_name LIKE { PATTERN };
```

SQL ORDER BY Clause

```
SELECT column1, column2....columnN
FROM   table_name
WHERE CONDITION
ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY Clause

```
SELECT SUM(column_name)
FROM   table_name
WHERE CONDITION
GROUP BY column_name;
```

SQL COUNT Clause

```
SELECT COUNT(column_name)
FROM   table_name
WHERE CONDITION;
```

SQL HAVING Clause

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQL CREATE TABLE Statement

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

SQL DROP TABLE Statement

```
DROP TABLE table_name;
```

SQL CREATE INDEX Statement

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQL DROP INDEX Statement

```
ALTER TABLE table_name
DROP INDEX index_name;
```

SQL DESC Statement

```
DESC table_name;
```

SQL TRUNCATE TABLE Statement

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE Statement

```
ALTER TABLE table_name {ADD|DROP|MODIFY} column_name
{data_ype};
```

SQL ALTER TABLE Statement (Rename)

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQL INSERT INTO Statement

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQL UPDATE Statement

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE  CONDITION ];
```

SQL DELETE Statement

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQL CREATE DATABASE Statement

```
CREATE DATABASE database_name;
```

SQL DROP DATABASE Statement

```
DROP DATABASE database_name;
```

SQL USE Statement

```
USE database_name;
```

SQL COMMIT Statement

```
COMMIT;
```

SQL ROLLBACK Statement

```
ROLLBACK;
```

SQL data type is an attribute that specifies type of data of any object. Each column, variable and expression has related data type in SQL.
You would use these data types while creating your tables. You would choose a particular data type for a table column based on your requirement.
SQL Server offers six categories of data types for your use –

Exact Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

DATA TYPE	FROM	TO
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Note – Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types

DATA TYPE	Description
char	Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
varchar	Maximum of 8,000 characters.(Variable-length non-Unicode data).
varchar(max)	Maximum length of 231characters, Variable-length non-Unicode data (SQL Server 2005 only).
text	Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types

DATA TYPE	Description
nchar	Maximum length of 4,000 characters.(Fixed length Unicode)
nvarchar	Maximum length of 4,000 characters.(Variable length Unicode)
nvarchar(max)	Maximum length of 231characters (SQL Server 2005 only).(Variable length Unicode)
ntext	Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Types

DATA TYPE	Description
binary	Maximum length of 8,000 bytes(Fixed-length binary data)
varbinary	Maximum length of 8,000 bytes.(Variable length binary data)
varbinary(max)	Maximum length of 231 bytes (SQL Server 2005 only). (Variable length Binary data)
image	Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

Misc Data Types

DATA TYPE	Description
sql_variant	Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
timestamp	Stores a database-wide unique number that gets updated every time a row gets updated
uniqueidentifier	Stores a globally unique identifier (GUID)
xml	Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
cursor	Reference to a cursor object
table	Stores a result set for later processing

Practice Excercises
VARIOUS RELATIONS IN SQL

STEP1: CREATE A TABLE

```
SQL> CREATE TABLE STUDENT(RNO VARCHAR(12),SNAME
VARCHAR(20), COURSE VARCHAR(5),ADDR VARCHAR(10),PHNO
NUMBER(10));
Table created.
```

STEP2: VIEW OF THE TABLE STRUCTURE:-

```
SQL> DESC STUDENT;
 Name                  Null?        Type
 ----------------------------------- --------
 RNO                                VARCHAR2(12)
 SNAME                              VARCHAR2(20)
 COURSE                             VARCHAR2(5)
 ADDR                               VARCHAR2(10)
 PHNO                               NUMBER(10)
```

STEP3: INSERT VALUES INTO A TABLE:-

```
SQL> INSERT INTO STUDENT VALUES ('&RNO', '&SNAME', '&COURSE',
'&ADDR', &PHNO);
Enter value for rno: 085N1F0049
Enter value for sname: NOORJAHAN
Enter value for course: MCA
Enter value for addr: PVL
Enter value for phno: 9849804997
old   1: insert into student
values('&rno','&sname','&course','&addr',&phno)
new   1: insert into student
values('085N1F0049','NOORJAHAN','MCA','PVL',9849804997)
1 row created.
SQL> /
Enter value for rno: 085N1F0048
Enter value for sname: NIRANJAN
Enter value for course: MCA
Enter value for addr: KNL
Enter value for phno: 9030245678
old   1: insert into student
values('&rno','&sname','&course','&addr',&phno)
new   1: insert into student
values('085N1F0048','NIRANJAN','MCA','KNL',9030245678)
1 row created.
```

STEP4: VIEW THE TABLE CONTENTS OR QUERYING THE DATA FROM THE TABLE:-

```
SQL> SELECT * FROM STUDENT;
RNO            SNAME      COURS   ADDR      PHNO
-----------  ----------------      ----- - ----------   --------
085N1F0049  NOORJAHAN  MCA      PVL     9849804997
085N1F0048  NIRANJAN      MCA      KNL     9030245678
08701F0049  SUJATHA    MBA   REPALLE 9849667789
08701F0036  SHANTHI    MBA   ONGOLE  9342137151
SQL> SELECT * FROM STUDENT WHERE SNAME='NOORJAHAN';
RNO          SNAME      COURS   ADDR      PHNO
------------------------------ -- - -- ------  -----
085N1F0049   NOORJAHAN    MCA    PVL   9849804997
SQL> SELECT SNAME, PHNO FROM STUDENT;
SNAME             PHNO
---------------   --------------
NOORJAHAN     9849804997
NIRANJAN        9030245678
SUJATHA         9849667789
SHANTHI        9342137151
SQL> SELECT DISTINCT COURSE FROM STUDENT;
COURSE
--------
MBA
MCA
```

STEP5: ALTER:-

SYNTAX1:- TO ADD A FIELD

```
SQL> ALTER TABLE STUDENT ADD (DOB DATE);
Table altered.
SQL> SELECT * FROM STUDENT;
RNO            SNAME      COURS     ADDR     PHNO            DOB
-----------  ----------   ---------    ---------         ---------          --------
085N1F0049 NOORJAHAN  MCA   PVL      9849804997
085N1F0048 NIRANJAN   MCA      KNL      9030245678
08701F0049 SUJATHA    MBA    REPALLE  9849667789
08701F0036 SHANTHI    MBA    ONGOLE   9342137151
```

SYNTAX2:- TO MODIFY A FIELD

```
SQL> ALTER TABLE STUDENT MODIFY (COURSE VARCHAR (8));
Table altered.
```

```
SQL> DESC STUDENT;
 Name                  Null?       Type
 ----------------------------------- --------
 RNO                               VARCHAR2(12)
 SNAME                              VARCHAR2(20)
 COURSE                             VARCHAR2(8)
 ADDR                              VARCHAR2(10)
 PHNO                              NUMBER(10)
 DOB                                    DATE
```

SYNTAX3:- DROP

```
SQL> ALTER TABLE STUDENT DROP COLUMN DOB;
Table altered.
SQL> SELECT * FROM STUDENT;
RNO       SNAME       COURS  ADDR       PHNO
--------------------------------------------------
085N1F0049 NOORJAHAN   MCA   PVL       9849804997
085N1F0048 NIRANJAN    MCA   KNL       9030245678
08701F0049 SUJATHA     MBA   REPALLE   9849667789
08701F0036 SHANTHI     MBA   ONGOLE    9342137151
```

STEP6: DROP

```
SQL> DROP TABLE STUDENT1;
Table dropped.
```

STEP7: DELETE

```
SQL> DELETE FROM STUDENT WHERE RNO='08701F0036';
1 row deleted.
SQL> SELECT * FROM STUDENT;
RNO         SNAME              COURS ADDR     PHNO
----------- -------------      ---------- -------    ---------
085N1F0049   NOORJAHAN   MCA     PVL       9849804997
085N1F0048   NIRANJAN      MCA     KNL       9030245678
08701F0049   SUJATHA       MBA     REPALLE   9849667789
```

STEP8: UPDATE

```
SQL> UPDATE STUDENT SET ADDR='CHITTOOR' WHERE
SNAME='NIRANJAN';
1 row updated.
```

```
SQL> SELECT * FROM STUDENT;
RNO         SNAME     COURS   ADDR      PHNO
----------- ----------------------------------------
085N1F0049   NOORJAHAN  MCA    PVL      9849804997
085N1F0048   NIRANJAN   MCA   CHITTOOR  9030245678
08701F0049   SUJATHA    MBA   REPALLE   9849667789
```

STEP9: ORDER BY

```
SQL> SELECT * FROM STUDENT ORDER BY RNO;
RNO         SNAME        COURSE     ADDR            PHNO
----------- ------------   -------------    --------        -----------
085N1F0048   NIRANJAN     MCA      CHITTOOR     9030245678
085N1F0049   NOORJAHAN    MCA        PVL       9849804997
08701F0049   SUJATHA      MBA        REPALLE    9849667789


SQL> SELECT * FROM STUDENT ORDER BY SNAME DESC;
RNO         SNAME         COURSE       ADDR          PHNO
----------- --------------   -------------  -------------        -----
08701F0049   SUJATHA      MBA      REPALLE   9849667789
085N1F0049   NOORJAHAN    MCA      PVL       9849804997
085N1F0048   NIRANJAN     MCA     CHITTOOR   9030245678
```

STEP10: GROUP BY

```
SQL> SELECT COURSE FROM STUDENT GROUP BY COURSE;
COURSE
--------
MBA
MCA
SQL> SELECT COURSE FROM STUDENT GROUP BY COURSE HAVING
COURSE='MCA';
COURSE
--------
MCA
```

Operators in SQL

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.
Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

SQL Comparison Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.

!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQL Logical Operators

Here is a list of all the logical operators available in SQL.

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.

BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

Practice Excercise
SQL OPERATORS

```
SQL> select *from emp;
EMP_NO    EMP_NAME          SAL          DNO        LOCATION
---------- -  ------------------  ----------  -----  ----------------
1          ramu              10000        256          rjp
2          kumar             15000         257         kdp
3          raju               7000        256          rjp
4          balu              15000         254         mpl
5          hari               8000        258          kdp
```

ARITHMETIC OPERATORS

+unary:-

```
SQL> select +3 from dual;
      +3
----------
       3
```

-unary:-

```
SQL> select -4 from dual;
      -4
----------
      -4
```

/ operator:-

```
SQL> select sal/10 from emp;
   SAL/10
----------
      1000
      1500
       700
      1500
       800
```

***operator:-**

```
SQL> select sal*5 from emp;
 SAL*5
----------
     50000
     75000
     35000
     75000
     40000
```

+oprator:-

```
SQL> select sal +200 from emp;
   SAL+200
----------
     10200
     15200
```

7200
15200
8200

-oprator:-

```
SQL> select sal -100 from emp;
 SAL-100
----------
      9900
     14900
      6900
     14900
      7900
```

Character oprator

|| oprator:-

```
SQL> select 'the name of the employee is:'||emp_name from emp;
'THENAMEOFTHEEMPLOYEEIS:'||EMP_NAME
---------------------------------------
the name of the employee is:ramu
the name of the employee is:kumar
the name of the employee is:raju
the name of the employee is:balu
the name of the employee is:hari
```

COMPARISION OPERATOR

=oprator:-

```
SQL> select emp_name "employee" from emp where sal=15000;
employee
--------------------
kumar
balu
```

^=oprator:-

```
SQL> select emp_name from emp where sal ^=8000;
EMP_NAME
--------------------
ramu
kumar
raju
balu
```

>operator :-

```
SQL> select emp_name "employee" from emp where sal>3000;
employee
--------------------
ramu
kumar
raju
balu
```

hari

<= operator :-

```
SQL> select emp_name from emp where sal <= 8000;
EMP_NAME
--------------------
raju
hari
```

IN operator :-

```
SQL> select *from emp where emp_name in('kumar','kdp');
EMP_NO     EMP_NAME              SAL                DNO   LOCATION
---------- -------------------- --------------------  -----  --------------------
2                  kumar              15000               257        kdp
```

Any oprator :-

```
SQL> select *from emp where location=some('257','kdp');
EMP_NO     EMP_NAME           SAL             DNO        LOCATION
---------- -------------------- --------------------   ----- --------------------
2              kumar            15000             257               kdp
5              hari             8000              258               kdp
```

Not in operator :-

```
SQL> select *from emp where location not in('257','kdp');
EMP_NO     EMP_NAME           SAL                DNO   LOCATION
---------- -------------------- --------------------         ----- --------------------
1         ramu               10000                        256     rjp
3         raju              7000                         256     rjp
4         balu              15000                        254     mpl
```

All oprator :-

```
SQL> select *from emp where sal >= all(15000,8000);
EMP_NO     EMP_NAME           SAL                DNO   LOCATION
---------- -------------------- --------------------         ----- --------------------
2         kumar              15000                        257    kdp
4         balu              15000                        254     mpl
SQL> select *from emp where sal >= all(7000,8000);
EMP_NO     EMP_NAME     SAL                      DNO   LOCATION
---------- -------------------- -------------------- ----- --------------------
1         ramu               10000                        256    rjp
2         kumar              15000                        257     kdp
4         balu              15000                        254      mpl
5         hari              8000                         258    kdp
```

[Not] between X and Y:-

```
SQL> select emp_name,location from emp where sal between 8000 and
10000;
EMP_NAME           LOCATION
-------------------- --------------------
ramu                          rjp
hari                          kdp
```

exists operator:-

```
SQL> select *from emp where exists(select emp_name from emp where
256 is not null);
EMP_NO     EMP_NAME        SAL                 DNO     LOCATION
---------- -------------------- --------------------  ----- --------------------
1          ramu                 10000                 256            rjp
2          kumar               15000                 257           kdp
3          raju                7000                  256           rjp
4          balu                 15000                 254            mpl
5          hari                 8000                  258           kdp
```

like oprator :-

```
SQL> select *from emp where emp_name like'%u%';
EMP_NO     EMP_NAME       SAL                 DNO   LOCATION
---------- -------------------- -------------------- ---- --------    ------------
1          ramu                 10000                 256           rjp
2          kumar               15000                 257           kdp
3          raju                7000                  256           rjp
4          balu                15000                   254          mpl
SQL> select *from emp where emp_name like'%u';
EMP_NO     EMP_NAME      SAL                  DNO      LOCATION
---------- -------------------- -------------------- --- -------    -------------
1          ramu                10000                 256           rjp
3          raju               7000                  256            rjp
4          balu                15000                 254            mpl
SQL> select *from emp where emp_name like'r%';
EMP_NO         EMP_NAME           SAL               DNO   LOCATION
----------          -------------------- --------------------  ----- --------------------
1                    ramu               10000              256         rjp
3                    raju               7000               256          rjp
```

is[not]null :-

```
SQL> select *from emp where dno is not null and sal >10000;
EMP_NO     EMP_NAME           SAL               DNO   LOCATION
---------- --------------------          --------------------  -----  --------------------
-
2                   kumar              15000              257         kdp
4                    balu              15000              254         mpl
SQL> select *from emp where not(dno is null);
EMP_NO     EMP_NAME             SAL               DNO   LOCATION
---------- --------------------       -------------------- ----- --------------------
1                    ramu               10000              256         rjp
2                    kumar               15000              257        kdp
3                    raju                7000               256         rjp
4                    balu                15000              254         mpl
5                    hari                 8000               258        kdp
```

LOGICAL OPERATORS

NOT:-

```
SQL> select *from emp where not(dno is not null);
no rows selected
```

AND:-

```
SQL> select *from emp where sal=7000 and dno=256;
EMP_NO     EMP_NAME                SAL                DNO   LOCATION
---------- --------------------      -------------------- ----- --------------------
3                     raju                 7000              256         rjp
```

OR:-

```
SQL> select *from  emp where sal=8000 or dno=258;
EMP_NO     EMP_NAME           SAL                DNO      LOCATION
---------- --------------------    -------------------- ----- --------------------
5                       hari              8000              258          kdp
```

SET OPERATORS

```
SQL> SELECT * FROM STUDENT1;
      SNO  SNAME            COURSE
---------- ---------------      ---------
      101 ABLE              BTECH
      102 BRAVO              MCA
      103 CHARLIE            MCA
      104 DECON              BTECH
      105 EXITOR             MBA
      106 FUBAR              MBA
      107 GOOBER             MCA
7 rows selected.
SQL> SELECT * FROM STUDENT2;
      SNO SNAME               COURSE
---------- ---------------     ---------
      201 ABLE              MTECH
      202 BAKER              BTECH
      203 CHARLIE            MCA
      204 DEAN               BTECH
      205 EXITOR             MBA
      206 FALCONER             MTECH
      207 GOOBER             MCA

7 rows selected.
```

UNION:-

SQL> SELECT SNAME FROM STUDENT1 **UNION** SELECT SNAME FROM STUDENT2;

```
SNAME
---------------
ABLE
BAKER
BRAVO
CHARLIE
DEAN
DECON
EXITOR
FALCONER
FUBAR
GOOBER

10 rows selected.
```

UNION ALL:-

SQL> SELECT SNAME FROM STUDENT1 UNION ALL SELECT SNAME FROM STUDENT2;

```
SNAME
---------------
ABLE
BRAVO
CHARLIE
DECON
EXITOR
FUBAR
GOOBER
ABLE
BAKER
CHARLIE
DEAN
EXITOR
FALCONER
GOOBER
14 rows selected.
```

INTERSECT:-

```
SQL> SELECT SNAME FROM STUDENT1 INTERSECT SELECT SNAME
FROM STUDENT2;

SNAME
---------------
ABLE
CHARLIE
EXITOR
GOOBER
```

MINUS:-

```
SQL> SELECT SNAME FROM STUDENT1 MINUS SELECT SNAME FROM
STUDENT2;

SNAME
---------------
BRAVO
DECON
FUBAR
SQL> SELECT SNAME FROM STUDENT2 MINUS SELECT SNAME FROM
STUDENT1;
SNAME
---------------
BAKER
DEAN
FALCONER
```

SQL Constraints

SQL constraints are used to specify rules for the data in a table.
If there is any violation between the constraint and the data action, the action is aborted by the constraint.
Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).

SQL CREATE TABLE + CONSTRAINT Syntax

CREATE TABLE *table_name*
(
column_name1 data_type(*size*) *constraint_name*,
column_name2 data_type(*size*) *constraint_name*,
column_name3 data_type(*size*) *constraint_name*,
....
);

In SQL, we have the following constraints:

- NOT NULL - Indicates that a column cannot store NULL value
- UNIQUE - Ensures that each row for a column must have a unique value
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have a unique identity which helps to find a particular record in a table more easily and quickly
- FOREIGN KEY - Ensure the referential integrity of the data in one table to match values in another table
- CHECK - Ensures that the value in a column meets a specific condition
- DEFAULT - Specifies a default value for a column

1. **NOT NULL CONSTRAINT**
   The NOT NULL constraint enforces a column to NOT accept NULL values.
   The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

```
SQL> CREATE TABLE STUD1(SNO VARCHAR2(10) NOT NULL,
SNAME VARCHAR2(10) NOT NULL,
COURSE VARCHAR2(8) NOT NULL,
PHNO NUMBER(10) NOT NULL);
Table created.
```

```
SQL> INSERT INTO STUD1
VALUES('085N49','NOOR','MCA',9989192636);
1 row created.
SQL> INSERT INTO STUD1
VALUES('081F36','SOWMYA','MCA',9703522331);
1 row created.
SQL> SELECT * FROM STUD1;
SNO      SNAME    COURSE     PHNO
-------------------------- ----------
085N49    NOOR      MCA      9989192636
081F36    SOWMYA    MCA      9703522331
SQL> INSERT INTO STUD1 VALUES(‘’,'MAHESH','MBA',9000463529);
INSERT INTO STUD1 VALUES(NULL,'MAHESH','MBA',9000463529)
ERROR at line 1:
ORA-01400: cannot insert NULL into ("MCA49"."STUD1"."SNO")
SQL> INSERT INTO STUD1 VALUES('081F45',NULL,'MCA',NULL);
INSERT INTO STUD1 VALUES('081F45',NULL,'MCA',NULL)
ERROR at line 1:
ORA-01400: cannot insert NULL into ("MCA49"."STUD1"."SNAME")
```

2. PRIMARY KEY CONSTAINT

The PRIMARY KEY constraint uniquely identifies each record in a database table.
Primary keys must contain UNIQUE values.
A primary key column cannot contain NULL values.
Most tables should have a primary key, and each table can have only ONE primary key.

```
SQL> CREATE TABLE STUD2(SNO VARCHAR2(10),
SNAME VARCHAR2(10),
COURSE VARCHAR2(8),
PHNO NUMBER(10),
PRIMARY KEY (SNO));
 Table created.
SQL> INSERT INTO STUD2
VALUES('0801','ANNAYYA','MCA',9999966666);
1 row created.
SQL> INSERT INTO STUD2
VALUES('0809','SREEDHAR','MBA',9876543210);
1 row created.
SQL> SELECT * FROM STUD2;
SNO      SNAME      COURSE      PHNO
-----------------------------------------------------
0801     ANNAYYA     MCA      9999966666
0809     SREEDHAR    MBA      9876543210
```

SQL> INSERT INTO STUD2 ALUES('0801','APARNA','MCA',9077777777);
INSERT INTO STUD2 VALUES('0801','APARNA','MCA',9077777777)
ERROR at line 1:
ORA-00001: unique constraint (MCA49.SYS_C003648) violated

SQL> INSERT INTO STUD2 VALUES(NULL,'SIRI','MBA',9765768888);
INSERT INTO STUD2 VALUES(NULL,'SIRI','MBA',9765768888)
ERROR at line 1:
ORA-01400: cannot insert NULL into ("MCA49"."STUD2"."SNO")

3. UNIQUE CONSTRAINT

The UNIQUE constraint uniquely identifies each record in a database table.
The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.
A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.
Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

SQL> CREATE TABLE STUD3 (SNO VARCHAR2(10),
SNAME VARCHAR2(10),
COURSE VARCHAR2(8),
PHNO NUMBER(10),
UNIQUE(SNAME,PHNO));
Table created.
SQL> INSERT INTO STUD3
VALUES('0809','SREEDHAR','MBA',9876543210);
1 row created.
SQL> INSERT INTO STUD3
VALUES('0801','APARNA','MBA',9077777777);
1 row created.
SQL> INSERT INTO STUD3
VALUES('0805','APARNA','MCA',9898989898);
1 row created.
SQL> SELECT * FROM STUD3;

SNO	SNAME	COURSE	PHNO
0809	SREEDHAR	MBA	9876543210
0801	APARNA	MBA	9077777777
0805	APARNA	MCA	9898989898

```
SQL> INSERT INTO STUD3
VALUES('0807','APARNA','BTECH',9077777777);
INSERT INTO STUD3 VALUES('0807','APARNA','BTECH',9077777777)
```
ERROR at line 1:
ORA-00001: unique constraint (MCA49.SYS_C003711) violated

4. CHECK CONSTRAINT:-

The CHECK constraint is used to limit the value range that can be placed in a column.
If you define a CHECK constraint on a single column it allows only certain values for this column.
If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
SQL>  CREATE TABLE STUD4(
SNO NUMBER(10) PRIMARY KEY CONSTRAINT S4 CHECK( SNO>10000
AND SNO <10100), NAME VARCHAR2(10) NOT NULL CONSTRAINT S5
CHECK(NAME=UPPER(NAME)), COURSE VARCHAR2(8), PHNO
NUMBER(10));
Table created.
SQL> INSERT INTO STUD4 VALUES(10001,'RAM','BTECH',9888888888);
1 row created.
SQL> INSERT INTO STUD4
VALUES(10002,'NEELAM','E.E.E',9888884567);
1 row created.
SQL> SELECT * FROM STUD4;
      SNO  NAME       COURSE     PHNO
    ---------- -------- ----------
    10001  RAM        BTECH    9888888888
    10002  NEELAM     E.E.E     9888884567
SQL> INSERT INTO STUD4
VALUES(21345,'KARUNA','CSE',9000094969);
INSERT INTO STUD4 VALUES(21345,'KARUNA','CSE',9000094969)
```
ERROR at line 1:
ORA-02290: check constraint (MCA49.S4) violated
```
SQL> INSERT INTO STUD4 VALUES(10345,'sneha','CSE',9000094954);
INSERT INTO STUD4 VALUES(10345,'sneha','CSE',9000094954)
```
ERROR at line 1:
ORA-02290: check constraint (MCA49.S5) violated

5. DEFAULT

The DEFAULT constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified.

```
SQL> CREATE TABLE STUD5(SNO VARCHAR2(10),NAME VARCHAR2(15), COURSE VARCHAR2(8) DEFAULT ('BTECH'), PHNO NUMBER(10));
Table created.
SQL> INSERT INTO STUD5(SNO,NAME,PHNO) VALUES ('&SNO', '&NAME', &PHNO);
Enter value for sno: 1002
Enter value for name: SKDFJLSKD
Enter value for phno: 6566555
old 1: INSERT INTO STUD5(SNO,NAME,PHNO) VALUES ('&SNO','&NAME', &PHNO)
new 1: INSERT INTO STUD5(SNO,NAME,PHNO) VALUES ('1002', 'SKDFJLSKD', 6566555)
1 row created.
SQL> /
Enter value for sno: 555
Enter value for name: DFGDFG
Enter value for phno: 88888
old 1: INSERT INTO STUD5(SNO,NAME,PHNO) VALUES ('&SNO','&NAME', &PHNO)
new 1: INSERT INTO STUD5(SNO,NAME,PHNO) VALUES ('555', 'DFGDFG', 88888)
1 row created.
SQL> SELECT * FROM STUD5;
SNO        NAME             COURSE        PHNO
---------- -------------------- ----------------------
1002       SKDFJLSKD        BTECH       6566555
555        DFGDFG           BTECH       88888
```

6. FOREIGN KEY

A FOREIGN KEY in one table points to a PRIMARY KEY in another table. Let's illustrate the foreign key with an example. Look at the following two tables:

CREATE MASTER TABLE:-

```
SQL> CREATE TABLE STUDENTS(SNO NUMBER(10) CONSTRAINT S11 PRIMARY KEY,NAME VARCHAR2(10) CONSTRAINT S12 NOT NULL, COURSE VARCHAR2(8) CONSTRAINT S13 NOT NULL, PHNO NUMBER(10));
Table created.
```

CREATE CHILD TABLE:-

```
SQL> CREATE TABLE ENROLLED(STUDID NUMBER(10),CID VARCHAR2(10),GRADE CHAR(1),PRIMARY KEY(STUDID,CID),FOREIGN KEY(STUDID) REFERENCES STUDENTS(SNO));
Table created.
SQL> INSERT INTO ENROLLED VALUES(5003,'MATHS','A');
INSERT INTO ENROLLED VALUES(5003,'MATHS','A')
ERROR at line 1:
ORA-02291: integrity constraint (MCA49.SYS_C004233) violated - parent key not Found
SQL> INSERT INTO STUDENTS VALUES(5001,'GAY','MCA',9899999);
1 row created.
SQL> INSERT INTO ENROLLED VALUES(5001,'MATHS','A');
1 row created.
SQL> DELETE FROM STUDENTS WHERE SNO=5001;
DELETE FROM STUDENTS WHERE SNO=5001
ERROR at line 1:
ORA-02292: integrity constraint (MCA49.SYS_C004233) violated - child record Found
```

ON DELETE CASCADE

CREATE MASTER TABLE:-

```
SQL> CREATE TABLE STUDENTS1(SNO NUMBER(10) CONSTRAINT S21 PRIMARY KEY,NAME VARCHAR2(10) CONSTRAINT S22 NOT NULL,COURSE VARCHAR2(8) CONSTRAINT S23 NOT NULL,PHNO NUMBER(10));
Table created.
SQL> INSERT INTO STUDENTS1 VALUES (1001,'MANASA','TPT',99999);
1 row created.
SQL> INSERT INTO STUDENTS1 VALUES (1002,'BINDU','GDL',987677);
1 row created.
```

CREATE CHILD TABLE:-

```
SQL> CREATE TABLE ENROLLED1(STUDID NUMBER(10),CID VARCHAR2(10),GRADE CHAR(1), PRIMARY KEY(STUDID,CID), FOREIGN KEY(STUDID) REFERENCES STUDENTS1(SNO) ON DELETE CASCADE);
Table created.
SQL> INSERT INTO ENROLLED1 VALUES(1002,'ARTS','A');
1 row created.
SQL> SELECT * FROM STUDENTS1;
     SNO   NAME     COURSE        PHNO
       ---------- -----      -----       --------
     1001 MANASA     TPT          99999
     1002 BINDU      GDL          987677

SQL> SELECT * FROM ENROLLED1;
   STUDID     CID      G
   ---------- ------------------
     1002      ARTS    A
SQL> DELETE FROM STUDENTS1 WHERE SNO=1002;
1 row deleted.
SQL> SELECT * FROM STUDENTS1;
      SNO  NAME       COURSE     PHNO
---------- ---------- -------- ----------
     1001 MANASA      TPT       99999

SQL> SELECT * FROM ENROLLED1;
        no rows selected
```

ON DELETE SET NULL

CREATE MASTER TABLE:-

```
SQL> CREATE TABLE STUDENTS2(SNO NUMBER(10) CONSTRAINT S31 PRIMARY KEY, NAME VARCHAR2(10) CONSTRAINT S32 NOT NULL, COURSE VARCHAR2(8) CONSTRAINT S33 NOT NULL,PHNO NUMBER(10));
Table created.
```

CREATE CHILD TABLE:-

```
SQL>CREATE TABLE ENROLLED2(STUDID NUMBER(10),CID VARCHAR2(10) , GRADE CHAR(1),PRIMARY KEY (STUDID,CID),FOREIGN KEY(STUDID) REFERENCES STUDENTS2(SNO) ON DELETE SET NULL)
Table created.
SQL> INSERT INTO STUDENTS2 VALUES(9001,'RAJU','MBA',98788);
1 row created.
SQL> INSERT INTO STUDENTS2 VALUES(9002,'KALA','MCA',98777);
1 row created.
SQL> INSERT INTO ENROLLED2 VALUES(9001,'ARTS','B');
1 row created.
```

```
SQL> INSERT INTO ENROLLED2 VALUES(9002,'ARTS','A');
1 row created.
SQL> SELECT * FROM STUDENTS2;
      SNO   NAME    COURSE   PHNO
     ------------------------------------
     9001   RAJU    MBA      98788
    9002   KALA     MCA      98777

SQL> SELECT * FROM ENROLLED2;
   STUDID     CID      G
   ---------------------------
     9001     ARTS     B
     9002     ARTS     A

SQL> DELETE FROM STUDENTS2 WHERE SNO=9001;
1 row deleted.

SQL> SELECT * FROM STUDENTS2;
      SNO NAME     COURSE     PHNO
---------- --------- -------- ----------
     9002 KALA       MCA     98777

SQL> SELECT * FROM ENROLLED2;
   STUDID     CID       G
    ------------------------
              ARTS      B
     9002     ARTS      A
```

SUBQUERIES - CORRELATED QUERIES – NESTED QUERIES

Subquery is a query inside a main query.
SQL subquery can be embedded:

- As a column expression in the main SQL statement
- As a filter inside the WHERE (or HAVING) clause in the main SQL statement
- As a datasource inside the FROM clause in the main SQL statement

While working with SQL Subqueries you must:

- Enclose the subquery in parentheses
- Do not use a semicolon at the end of the subquery statement

```
SQL> SELECT * FROM TBLPROD;
        ID NAME
---------- --------------------
         1 TV
         2 LAPTAP
         3 PENDRIVE
SQL> SELECT * FROM TBLPRODSALES;
        ID     PRODID  UNITPRICE    QTYSOLD
---------- ---------- ---------- ----------
         1          3       4500          4
         2          2       5600          3
         3          3       3430          7
         4          3       2340          2
```

SINGLE ROW SUB-QUERIES:-

```
SQL> SELECT ID,NAME FROM TBLPROD WHERE ID IN(SELECT
PRODID FROM TBLPRODSALES);
        ID NAME
---------- --------------------
         3 PENDRIVE
         2 LAPTAP
SQL> SELECT ID,NAME FROM TBLPROD WHERE ID >(SELECT ID
FROM TBLPRODSALES WHERE PRODID=2);
        ID NAME
---------- --------------------
         3 PENDRIVE
SQL> SELECT ID,NAME FROM TBLPROD WHERE ID  NOT IN(SELECT
PRODID FROM TBLPRODSALES);
        ID NAME
---------- --------------------
         1 TV
```

```
SQL> SELECT ID,PRODID FROM TBLPRODSALES WHERE ID =(SELECT
MAX(ID) FROM TBLPROD);

        ID     PRODID
---------- ----------
         3          3
```

MULTIPLE ROW SUB-QUERIES:-

```
SQL> SELECT ID,NAME FROM TBLPROD WHERE ID IN(SELECT
MIN(ID) FROM TBLPRODSALES GROUP BY PRODID);

        ID NAME
---------- --------------------
         1 TV
         2 LAPTAP

SQL> SELECT ID,NAME FROM TBLPROD WHERE ID IN(SELECT ID
FROM TBLPRODSALES WHERE PRODID IN(2));

        ID NAME
---------- --------------------
         2 LAPTAP

SQL> SELECT ID,NAME FROM TBLPROD WHERE ID NOT  IN(SELECT
ID FROM TBLPRODSALES WHERE PRODID IN(2));

        ID NAME
---------- --------------------
         1 TV
         3 PENDRIVE
```

CORRELATED SUBQUERIES:-

When you reference a column from the table in the parent query in the subquery, it is known as a correlated subquery. For each row processed in the parent query, the correlated subquery is evaluated once.

While processing Correlated subquery:

- The first row of the outer query is fetched.
- The inner query is processed using the outer query's value or values.
- The outer query is processed using the inner query's value or values.
- This process is continued until the outer query is done.

```
SQL> SELECT NAME, (SELECT SUM(QTYSOLD) FROM TBLPRODSALES
WHERE PRODID=TBLPROD.ID) AS QTYSOLD FROM TBLPROD;

NAME                 QTYSOLD
-------------------  ----------
TV
LAPTAP                    3
PENDRIVE                 13

SQL> SELECT ID,NAME FROM TBLPROD T WHERE EXISTS(SELECT 1
FROM TBLPRODSALES B WHERE T.ID=B.PRODID);

        ID NAME
---------- --------------------
         3 PENDRIVE
         2 LAPTAP

SQL> SELECT ID,NAME FROM TBLPROD T WHERE NOT
EXISTS(SELECT 1 FROM TBLPRODSALES B WHERE T.ID=B.PRODID);

        ID NAME
---------- --------------------
         1 TV
```

NESTED QUERIES:-

A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.

```
SQL> SELECT * FROM TBLPROD WHERE ID IN(SELECT ID FROM
TBLPRODSALES WHERE PRODID=(SELECT PRODID FROM
TBLPRODSALES WHERE ID=3));

        ID NAME
---------- --------------------
         1 TV
         3 PENDRIVE
```

SQL FUNCTIONS

QUERIES USING AGGREGATE FUNCTIONS

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT() - Returns the number of rows
- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

```
SQL> SELECT * FROM EMPLOYEE;
     ENO    NAME     SAL       JOB
    ------------------------------
     1001   MURALI   35000    C.A.
     1002   LATHA    23000    S.E.
     1003   REKHA    27000    DBA
     1004   PRADEEP  30000    S.E.
```

1. AGGREGATE FUNCTIONS:-

```
SQL> SELECT COUNT(*) FROM EMPLOYEE;
    COUNT(*)
   ----------
         4
SQL> SELECT SUM(SAL) FROM EMPLOYEE;
   SUM(SAL)
  ------------
    115000
SQL> SELECT AVG(SAL) FROM EMPLOYEE;
    AVG(SAL)
  -------------
     28750
SQL> SELECT MAX(SAL) FROM EMPLOYEE;
   MAX(SAL)
  -------------
     35000
SQL> SELECT MIN(SAL) FROM EMPLOYEE;
   MIN(SAL)
  ------------
     23000
```

QUERIES USING CONVERSION FUNCTIONS

What is a DUAL Table in Oracle?

This is a single row and single column dummy table provided by oracle. This is used to perform mathematical calculations without using a table.

DUAL TALBE:-

```
SQL> SELECT * FROM DUAL;
D
-
X
SQL> DESC DUAL;
 Name        Null?    Type
 ----------------------------- --------
 DUMMY                VARCHAR2(1)
```

2. DATE AND TIME FUNCTIONS:-

These are functions that take values that are of datatype DATE as input and return values of datatypes DATE, except for the MONTHS_BETWEEN function, which returns a number as output. Few date functions are as given below.

Function Name	Return Value
ADD_MONTHS (date, n)	Returns a date value after adding *'n'* months to the date *'x'*.
MONTHS_BETWEEN (x1, x2)	Returns the number of months between dates x1 and x2.
ROUND (x, date_format)	Returns the date *'x'* rounded off to the nearest century, year, month, date, hour, minute, or second as specified by the *'date_format'*.
TRUNC (x, date_format)	Returns the date *'x'* lesser than or equal to the nearest century, year, month, date, hour, minute, or second as specified by the 'date_format'.
NEXT_DAY (x, week_day)	Returns the next date of the *'week_day'* on or after the date *'x'* occurs.
LAST_DAY (x)	It is used to determine the number of days remaining in a month from the date *'x'* specified.
SYSDATE	Returns the systems current date and time.
NEW_TIME (x, zone1, zone2)	Returns the date and time in zone2 if date 'x' represents the time in zone1.

The below table provides the examples for the above functions

Function Name	Examples	Return Value
ADD_MONTHS ( )	ADD_MONTHS ('16-Sep-81', 3)	16-Dec-81
MONTHS_BETWEEN( )	MONTHS_BETWEEN ('16-Sep-81', '16-Dec-81')	3
NEXT_DAY( )	NEXT_DAY ('01-Jun-08', 'Wednesday')	04-JUN-08
LAST_DAY( )	LAST_DAY ('01-Jun-08')	30-Jun-08
NEW_TIME( )	NEW_TIME ('01-Jun-08', 'IST', 'EST')	31-May-08

```
SQL> SELECT SYSDATE FROM DUAL;
SYSDATE
---------
05-OCT-09
SQL> SELECT ADD_MONTHS(SYSDATE,2) FROM DUAL;
ADD_MONTH
---------
05-DEC-09
SQL> SELECT MONTHS_BETWEEN('01-MAY-06','01-JUN-09') FROM
DUAL;
MONTHS_BETWEEN('01-MAY-06','01-JUN-09')
---------------------------------------
                  -37
SQL> SELECT MONTHS_BETWEEN('01-MAY-09','01-JUN-06') FROM
DUAL;
MONTHS_BETWEEN('01-MAY-09','01-JUN-06')
---------------------------------------
                    35
SQL> SELECT NEXT_DAY(SYSDATE,'FRI') FROM DUAL;
NEXT_DAY(
---------
09-OCT-09
SQL> SELECT LAST_DAY(SYSDATE) FROM DUAL;
LAST_DAY(
---------
31-OCT-09
SQL> SELECT LAST_DAY('14-AUG-2006') FROM DUAL;
LAST_DAY(
---------
31-AUG-06
```

3. ARITHMETIC OR NUMERIC FUNCTIONS:-

Numeric functions are used to perform operations on numbers. They accept numeric values as input and return numeric values as output. Few of the Numeric functions are:

Function Name	Return Value
ABS (x)	Absolute value of the number '*x*'
CEIL (x)	Integer value that is Greater than or equal to the number '*x*'
FLOOR (x)	Integer value that is Less than or equal to the number '*x*'
TRUNC (x, y)	Truncates value of number '*x*' up to '*y*' decimal places
ROUND (x, y)	Rounded off value of the number '*x*' up to the number '*y*' decimal places

The following examples explains the usage of the above numeric functions

Function Name	**Examples**	**Return Value**
ABS (x)	ABS (1) ABS (-1)	1 -1
CEIL (x)	CEIL (2.83) CEIL (2.49) CEIL (-1.6)	3 3 -1
FLOOR (x)	FLOOR (2.83) FLOOR (2.49) FLOOR (-1.6)	2 2 -2
TRUNC (x, y)	ROUND (125.456, 1) ROUND (125.456, 0) ROUND (124.456, -1)	125.4 125 120
ROUND (x, y)	TRUNC (140.234, 2) TRUNC (-54, 1) TRUNC (5.7) TRUNC (142, -1)	140.23 54 5 140

These functions can be used on database columns.

```
SQL> SELECT ROUND(65.836) FROM DUAL;
ROUND(65.836)
-------------
    66
SQL> SELECT ROUND(65.836,2) FROM DUAL;
ROUND(65.836,2)
---------------
     65.84
```

```
SQL> SELECT CEIL(65.368) FROM DUAL;
CEIL(65.368)
------------
          66
SQL> SELECT FLOOR(65.368) FROM DUAL;

FLOOR(65.368)
-------------
       65
SQL> SELECT MOD(11,4),MOD(45,8) FROM DUAL;

 MOD(11,4)  MOD(45,8)
---------- ----------
     3          5
SQL> SELECT POWER(13,5) FROM DUAL;
POWER(13,5)
-----------
   371293
SQL> SELECT SQRT(625) FROM DUAL;
 SQRT(625)
----------
    25
SQL> SELECT SQRT(9.25) FROM DUAL;

SQRT(9.25)
----------
3.04138127

SQL> SELECT ABS(-62) FROM DUAL;
  ABS(-62)
----------
    62
SQL> SELECT SIN(90) FROM DUAL;
  SIN(90)
----------
.893996664

SQL> SELECT COS(0) FROM DUAL;
  COS(0)
----------
   1
SQL> SELECT TAN(45) FROM DUAL;
   TAN(45)
----------
1.61977519
```

4. CHARACTER FUNCTIONS:-

Character or text functions are used to manipulate text strings. They accept strings or characters as input and can return both character and number values as output.
Few of the character or text functions are as given below:

Function Name	Return Value
LOWER (string_value)	All the letters in *'string_value'* is converted to lowercase.
UPPER (string_value)	All the letters in *'string_value'* is converted to uppercase.
INITCAP (string_value)	All the letters in *'string_value'* is converted to mixed case.
LTRIM (string_value, trim_text)	All occurrences of *'trim_text'* is removed from the left of *'string_value'*.
RTRIM (string_value, trim_text)	All occurrences of *'trim_text'* is removed from the right of *'string_value'* .
TRIM (trim_text FROM string_value)	All occurrences of *'trim_text'* from the left and right of *'string_value'* , *'trim_text'* can also be only one character long .
SUBSTR (string_value, m, n)	Returns *'n'* number of characters from *'string_value'* starting from the *'m'* position.
LENGTH (string_value)	Number of characters in *'string_value'* in returned.
LPAD (string_value, n, pad_value)	Returns *'string_value'* left-padded with *'pad_value'* . The length of the whole string will be of *'n'* characters.
RPAD (string_value, n, pad_value)	Returns *'string_value'* right-padded with *'pad_value'* . The length of the whole string will be of *'n'* characters.

For Example, we can use the above UPPER() text function with the column value as follows.

```
SELECT UPPER (product_name) FROM product;
```

The following examples explains the usage of the above character or text functions

Function Name	Examples	Return Value
LOWER(string_value)	LOWER('Good Morning')	good morning
UPPER(string_value)	UPPER('Good Morning')	GOOD MORNING
INITCAP(string_value)	INITCAP('GOOD MORNING')	Good Morning
LTRIM(string_value, trim_text)	LTRIM ('Good Morning', 'Good)	Morning
RTRIM (string_value, trim_text)	RTRIM ('Good Morning', ' Morning')	Good
TRIM (trim_text FROM string_value)	TRIM ('o' FROM 'Good Morning')	Gd Mrning
SUBSTR (string_value, m, n)	SUBSTR ('Good Morning', 6, 7)	Morning
LENGTH (string_value)	LENGTH ('Good Morning')	12
LPAD (string_value, n, pad_value)	LPAD ('Good', 6, '*')	**Good
RPAD (string_value, n, pad_value)	RPAD ('Good', 6, '*')	Good**

```
SQL> SELECT CHR(65) FROM DUAL;
C
------
A
SQL> SELECT ASCII('a') FROM DUAL;
ASCII('A')
----------
    97
SQL> SELECT CONCAT('ORACLE','CORPORATION') FROM DUAL;
CONCAT('ORACLE','
-----------------
ORACLECORPORATION
SQL> SELECT INITCAP('noorjahan') FROM DUAL;
INITCAP('
---------
Noorjahan
SQL> SELECT LOWER('NOOR') FROM DUAL;
LOWE
----
noor
```

```
SQL> SELECT UPPER('noor') FROM DUAL;
UPPE
----
NOOR
SQL> SELECT LPAD('DBMS',10,'*') FROM DUAL;
LPAD('DBMS
----------
******DBMS
SQL> SELECT RPAD('DBMS',10,'*') FROM DUAL;
RPAD('DBMS
----------
DBMS******
SQL> SELECT LTRIM('DBMS','D') FROM DUAL;
LTR
---
BMS
SQL> SELECT RTRIM('DBMS','S') FROM DUAL;
RTR
---
DBM
SQL> SELECT REPLACE('JACK AND JUE','J','BL') FROM DUAL;
REPLACE('JACKA
--------------
BLACK AND BLUE
SQL> SELECT SUBSTR('DBMS',1,3) FROM DUAL;
SUB
---
DBM
SQL> SELECT TRANSLATE (JOB,'S','E') FROM EMPLOYEE WHERE
JOB='S.E.';
TRANSLATE(JO
------------
E.E.
E.E.
SQL> SELECT INSTR('STRING','R') FROM DUAL;
INSTR('STRING','R')
-------------------
   3
SQL> SELECT LENGTH('SHAIK.NOORJAHAN') FROM DUAL;
LENGTH('SHAIK.NOORJAHAN')
-------------------------
         15
```

5. CONVERSION FUNCTIONS:-

These are functions that help us to convert a value in one form to another form. For Ex: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE.

Few of the conversion functions available in oracle are:

Function Name	Return Value
TO_CHAR (x [,y])	Converts Numeric and Date values to a character string value. It cannot be used for calculations since it is a string value.
TO_DATE (x [, date_format])	Converts a valid Numeric and Character values to a Date value. Date is formatted to the format specified by *'date_format'*.
NVL (x, y)	If *'x'* is NULL, replace it with *'y'*. *'x'*and *'y'* must be of the same datatype.
DECODE (a, b, c, d, e, default_value)	Checks the value of *'a'*, if *a = b*, then returns *'c'*. If *a = d*, then returns *'e'*. Else, returns *default_value*.

The below table provides the examples for the above functions

Function Name	Examples	Return Value
TO_CHAR ()	TO_CHAR (3000, '$9999') TO_CHAR (SYSDATE, 'Day, Month YYYY')	$3000 Monday, June 2008
TO_DATE ()	TO_DATE ('01-Jun-08')	01-Jun-08
NVL ()	NVL (null, 1)	1

```
SQL> SELECT TO_CHAR(SYSDATE,'DD') FROM DUAL;
TO
--
05
SQL> SELECT TO_DATE('01-FEB-99')+100 FROM DUAL;
TO_DATE('
---------
12-MAY-99
```

JOINS

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them.
The most common type of join is: **SQL INNER JOIN (simple join)**. An SQL INNER JOIN returns all rows from multiple tables where the join condition is met.
Different SQL JOINs
Before we continue with examples, we will list the types of the different SQL JOINs you can use:

- **INNER JOIN**: Returns all rows when there is at least one match in BOTH tables
- **LEFT JOIN**: Return all rows from the left table, and the matched rows from the right table
- **RIGHT JOIN**: Return all rows from the right table, and the matched rows from the left table
- **FULL JOIN**: Return all rows when there is a match in ONE of the tables

Practice Exercise on Joins

CREATE FIRST TABLE:-

```
SQL> SELECT * FROM S1;
     SID  SNAME   RATING   AGE
---------- ---------- ---------- ----------
      22  DUSTIN         7        45
      31  LUBBER         8        55
      58  RUSTY         10        35
```

CREATE SECOND TABLE:-

```
SQL> SELECT  * FROM R1;
     SID      BID    DAY
---------- ---------- ----------
      22       101   10/10/96
      58       103   11/12/96
      69       105   09/07/88
```

1.EQUI OR SIMPLE OR INNER JOIN:-

```
SQL> SELECT S.SID,S.SNAME,S.RATING,S.AGE,R.BID,R.DAY FROM S1
S,R1 R WHERE S.SID=R.SID;
      SID SNAME   RATING  AGE       BID  DAY
---------- ---------- ---------- ---------- ---------- ----------
       22 DUSTIN           7        45       101   10/10/96
       58 RUSTY           10        35       103   11/12/96
SQL> SELECT SID,SNAME,RATING,AGE,BID,DAY FROM S1 NATURAL
JOIN  R1;
           SID SNAME   RATING  AGE       BID  DAY
---------- ---------- ---------- ---------- ---------- ----------
       22  DUSTIN           7        45       101   10/10/96
       58  RUSTY           10        35       103   11/12/96
```

2.LEFT OUTER JOIN:-

```
SQL> SELECT SID,SNAME,BID FROM S1 NATURAL LEFT OUTER JOIN
R1;
         SID  SNAME           BID
---------- ---------- ----------
       22  DUSTIN          101
       58  RUSTY           103
       31  LUBBER
SQL> SELECT S.SID,S.SNAME,R.BID FROM S1 S,R1 R WHERE
S.SID=R.SID(+);
      SID   SNAME           BID
    ---------- ---------- ----------
       22   DUSTIN          101
       31   LUBBER
       58   RUSTY           103
```

3.RIGHT OUTER JOIN:-

```
SQL> SELECT SID,SNAME,BID FROM S1 NATURAL RIGHT OUTER JOIN
R1;
      SID   SNAME         BID
---------- ---------- ----------
       22   DUSTIN          101
       58   RUSTY           103
       69                    105
SQL> SELECT R.SID,S.SNAME,R.BID FROM S1 S,R1 R WHERE
S.SID(+)=R.SID;
      SID   SNAME        BID
      ---------- ---------- ----------
       22   DUSTIN          101
       58   RUSTY           103
       69                   105
```

4.SELF JOIN:-

```
SQL> SELECT * FROM EMP;
EMPNO      NAME       MGRNO
---------- ---------- ------------------
0001       BASU             0002
0002       RUKMINI       0005
0003       CAROL          0004
0004       CYNKURAN
0005        IVAN
6 rows selected.

SQL> SELECT E.NAME, MGR.NAME "MANAGER" FROM EMP E,EMP
MGR WHERE E.MGRNO=MGR.EMPNO;

NAME       MANAGER
---------- ------------------
BASU         RUKMINI
CAROL      CYNKURAN
RUKMINI    IVAN
```

VIEWS

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

SQL CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

CREATE TABLE:-

```
SQL> SELECT * FROM EMP;
  EMPNO  ENAME          SAL     JOB
---------- --------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1003    JANU          20000     DBA
```

CREATE VIEW:-

```
SQL> CREATE VIEW E1 AS SELECT * FROM EMP;
View created.
SQL> SELECT * FROM E1;
  EMPNO  ENAME          SAL     JOB
---------- --------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1003    JANU          20000     DBA
SQL> INSERT INTO E1 VALUES(1004,'ISHU',28000,'PRINCIPAL');
1 row created.
SQL> SELECT * FROM E1;
  EMPNO  ENAME          SAL     JOB
---------- --------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1003    JANU          20000     DBA
     1004    ISHU          28000     PRINCIPAL



SQL> SELECT * FROM EMP;
EMPNO   NAME          SAL     JOB
---------- --------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1003    JANU          20000     DBA
     1004    ISHU          28000     PRINCIPAL
```

OPEARATIONS ON VIEWS:-

```
SQL> SELECT ENAME, SAL FROM E1;

ENAME          SAL
--------------- ----------
NOOR           30000
SNH            35000
JANU           20000
ISHU           28000
SQL> SELECT * FROM E1 WHERE SAL>28000;

 EMPNO  ENAME          SAL     JOB
---------- --------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
```

```
SQL> DELETE FROM E1 WHERE SAL=20000;
1 row deleted.

SQL> SELECT * FROM E1;

EMPNO   NAME         SAL    JOB
---------- -------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1004    ISHU          28000     PRINCIPAL

SQL> SELECT * FROM EMP;

EMPNO   NAME         SAL    JOB
---------- -------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1004    ISHU          28000     PRINCIPAL


SQL> DROP VIEW E1;

View dropped.

SQL> SELECT * FROM EMP;

EMPNO   NAME         SAL    JOB
---------- -------------- ---------- ------------
     1001    NOOR          30000     S.E.
     1002    SNH           35000      P.P.
     1004    ISHU          28000     PRINCIPAL
```

CHAPTER 2
PL/SQL BASICS
CONTROL STRUCTURES, CURSORS AND EXCEPTIONS

PL/SQL stands for **Procedural Language** extension of SQL.
PL/SQL is a combination of SQL along with the procedural features of programming languages.
It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

The PL/SQL Engine:

Oracle uses a **PL/SQL** engine to processes the PL/SQL statements. A PL/SQL language code can be stored in the client system (client-side) or in the database (server-side).

This Oracle **PL SQL tutorial** teaches you the basics of database programming in PL/SQL with appropriate **PL/SQL tutorials** with coding examples. You can use these free online tutorials as your guide to practice, learn, for training, or reference while programming with PL SQL. I will be making more Oracle PL SQL programming tutorials as often as possible to share my knowledge in PL SQL and help you in learning PL SQL language and syntax better.
Even though the programming concepts discussed in this tutorial are specific to Oracle PL SQL. The concepts like *cursors*, *functions* and *stored procedures* can be used in other database systems like Sybase , Microsoft SQL server etc, with some change in **SQL syntax**. This PL/SQL tutorial will be growing regularly; let us know if any topic related to PL SQL needs to be added or you can also share your knowledge on PL SQL with us. Lets share our knowledge about PL SQL with others.

A Simple PL/SQL Block:
Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.

PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception Handling (or Error) section (optional).

Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

Execution Section:

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

Exception Section:

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

Every statement in the above three sections must end with a semicolon **;** . PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

How a Sample PL/SQL Block Looks

```
DECLARE
    Variable declaration
BEGIN
    Program Execution
EXCEPTION
    Exception handling
END;
```

SQL Command Categories

SQL commands are grouped into four major categories depending on their functionality. They are as follows:

Data Definition Language (DDL)

These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.

Data Manipulation Language (DML)

These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.

Transaction Control Language (TCL)

These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.

Data Control Language (DCL)

These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

Advantages of PL/SQL

- ***Block Structures*:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.

- ***Procedural Language Capability*:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).

- ***Better Performance*:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.

- ***Error Handling*:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

Conditional Statements in PL/SQL

- Conditional controls
  - IF-THEN
  - IF-THEN-ELSE, and
  - IF-THEN-ELSIF
- Iterative controls
  - Simple loops
  - WHILE loops
  - FOR loops
- Cursor Manipulation.
- Using Cursor For Loops.
- Using Parameters with Cursors.
- Exception Handling.
- Cursor variables

CONTROL STRUCTURES

According to the structure theorem, any computer program can be written using the basic control structures, which can be combined in any way necessary to deal with a given problem.

The **selection structure** tests a condition, and then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE, FALSE, or NULL).

The **iteration structure** executes a sequence of statements repeatedly as long as a condition holds true.

The **sequence structure** simply executes a sequence of statements in the order in which they occur.

CONDITIONAL CONTROL

Conditional control allows you to control the flow of the execution of the program based on a condition. In programming terms, it means that the statements in the program are not executed sequentially. Rather, one group of statements, or another will be executed, depending on how the condition is evaluated.

The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements - **IF-THEN**, **IF-THEN-ELSE**, and **IF-THEN-ELSIF**.

IF-THEN

This construct tests a simple condition. If the condition evaluates to TRUE, one or more lines of code are executed. If the condition evaluates to FALSE, program control is passed to the next statement after the test. The following code illustrates implementing this logic in PL/SQL.

```
If varl > 10 then
    var2 := varl + 20;
END IF;
```

The test, in this case ">", is a relational operator we discussed in the "PL/SQL Character Set”" section. The statement could have been using the following instead with the same result.

```
IF NOT(varl <= 10) THEN
    var2 := varl + 20;
END IF;
```

You may code nested IF-THEN statements as shown in the following.

```
IF varl > 10 THEN
    IF var2 < varl THEN
        var2 := varl + 20;
    END IF;
END IF;
```

Notice that there are two END IF in the above example - one for each IF. This leads us into two rules about implementing IF logic in PL/SQL:

1. Each IF statement is followed by its own THEN. There is no semicolon (;) terminator on the line that starts with IF.
2. Each IF statement block is terminated by a matching END IF.

IF-THEN-ELSE

This construct is similar to IF, except that when the condition evaluates to FALSE, one or more statements following the ELSE are executed. The following code illustrates implementing this logic in PL/SQL.

```
IF varl > 10 THEN
    var2 := varl + 20;
ELSE
    var2 := varl * varl;
END IF;
```

Note that the same logic can be expressed in the other way - adding 20 to *varl* with the ELSE and squaring *varl* with the IF branch of the statement.

```
IF varl <= 10 THEN
    var2 := varl * varl;
ELSE
    var2 := varl + 20;
END IF;
```

This statement can be nested also, as shown below.

```
IF varl > 10 THEN
    var2 := varl + 20;
ELSE
    IF varl BETWEEN 7 AND 8 THEN
        var2 := 2 * varl;
    ELSE
        var2 := varl * varl;
    END IF;
END IF;
```

This leads us to two more rules about implementing if logic in PL/SQL:

3. There can be one and only one ELSE with every IF statement.
4. There is no semicolon (;) terminator after ELSE.

IF-THEN-ELSIF

This format is an alternative to using the nested IF-THEN-ELSE construct. The code in the previous listing could be reworded to read:

```
IF varl > 10 THEN
    var2 := varl + 20;
ELSIF varl BETWEEN 7 AND 8 THEN
    var2 := var2 * varl;
ELSE
    var2 := varl * varl;
END IF;
```

NOTE : The third form of IF statement uses the keyword ELSIF (NOT ELSEIF) to introduce additional conditions.

This leads us to one final rule about implementing IF logic in PL/SQL.

5. There is no matching END IF with each ELSIF.

In the following code segment, the END IF appears to go with its preceding ELSIF:

```
IF varl > 10 THEN
    var2 := varl + 20;
ELSIF varl BETWEEN 7 AND 8 THEN
    var2 := 2 * varl;
END IF;
```

In fact, the END IF belongs to the IF that starts the whole block rather than the ELSIF keyword.

ITERATIVE CONTROL

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: **LOOP**, **WHILE-LOOP**, and **FOR-LOOP**.

LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
    statement1;
    statement2;
    statement3;
    …
END LOOP;
```

All the sequence of statements is executed for each iteration of the loop. Then, the control resumes at the top of the loop and the cycle starts again. If further processing is undesirable or impossible, you can use the EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop.
There are two forms of EXIT statements: **EXIT** and **EXIT-WHEN**.
The **EXIT** statement forces a loop to complete unconditionally. When an **EXIT** statement is encountered, the loop completes immediately and the control is passed to the next statement after the loop.

```
LOOP
    …
    IF … THEN
        …
        EXIT; -- exit loop immediately
    END IF;
END LOOP;
-- control resumes here
```

The **EXIT-WHEN** statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition evaluates to TRUE, the loop completes and the control is passed to the next statement after the loop.

```
LOOP
    FETCH c1 INTO …
    EXIT WHEN c1%NOTFOUND;  -- exit loop if condition is
true
    …
END LOOP;
```

Until the condition evaluates to TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition.

Like the PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
    statement1;
    statement2;
    statement3;
    ...
END LOOP [label_name];
```

Optionally, the label name can also appear at the end of the LOOP statement.
With either form of EXIT statement, you can complete not only the current loop, but also any enclosing loop. Simply label the enclosing loop that you want to complete, and then use the label in an EXIT statement.

```
<<outer>>
LOOP
    LOOP
        ...
        EXIT outer WHEN … -- exit both loops
    END LOOP;
END LOOP outer;
```

WHILE-LOOP
A WHILE loop has the following structure:

```
WHILE <condition> LOOP
    statement 1;
    statement 2;
    statement 3;
    ...
    statement N;
END LOOP;
```

The reserved word WHILE marks the beginning of a loop construct. The word "<condition>" is the test condition of the loop that evaluates to TRUE or FALSE. The result of this evaluation determines whether the loop is executed. Statements 1 through N are a sequence of statements that is executed repeatedly. The END LOOP is a reserved phrase that

indicates the end of the loop construct. The following is an example of using WHILE LOOP.

FOR-LOOP

Whereas the number of iteration through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

```
FOR counter IN [REVERSE] lower_bound..upper_bound LOOP
    statement 1;
    statement 2;
    statement 3;
    ...
    statement N;
END LOOP;
```

The lower bound may not be 1. However, the loop counter increment (or decrement) must be 1. Between the lower bound and the upper bound is a double dot (..), which serves as the range operator.

Practice Excercise PL/SQL PROGRAMS

1.Sample program using PL/SQL block:

```
Declare
        a number(5);
        b number(5);
        c number(5);
Begin
        a:=&a;
        b:=&b;
        c:=a+b;
          dbms_output.put_line('sum is:'||c);
        c:=a-b;
          dbms_output.put_line('substraction is:'||c);
        c:=a*b;
          dbms_output.put_line('multiplication is:'||c);
        c:=a/b;
          dbms_output.put_line('Division is:'||c);
 End;
```

OUTPUT:-

```
SQL>SET SERVEROUTPUT ON
SQL>@operations
12/
Enter value for a:8
```

```
Enter value for b:4
Sum is:12
Substraction is:4
Multiplication is:32
Division is:2
PL/SQL procedure is successfully completed
```

CONTROL STRUCTURES IN PL/SQL BLOCK

2.Simple if

```
Declare
       Day    varchar2(10):='&day';
 Begin
      If(day='sunday')then
        Dbms_output.put_line('Sunday is holiday');
          End if;
End;
```

Output:-

```
SQL>@sif
15/
Enter value for day:Sunday
Sunday is holiday
PL/SQL procedure is successfully completed
```

3.if-then-else

```
DECLARE
        N NUMBER(5):=&N;
BEGIN
     IF N MOD 2 = 0 THEN
    DBMS_OUTPUT.PUT_LINE(' THE NUMBER IS EVEN');
    ELSE
   DBMS_OUTPUT.PUT_LINE(' THE NUMBER IS ODD');
   END IF;
END;
```

OUTPUT:-

```
SQL>@ifelse
18/
Enter value for n:6
The number is even
PL/SQL procedure is successfully completed
```

4.elsif

```
Declare
        s number(5):=&s;
        grade char;
begin
       if(s>90 and s<100)then
```

```
            grade:='A'
       elsif (s>70 ands<90)then
          grade:='B';
     elsif(s>50 and s<70)then
          grade;='C';
     else
         grade:='F';
    end if;
  dbms_output.put_line('score is'||to_char(s));
  dbms_output.put_line('grade is'||grade);
end;
```

OUTPUT:-

```
SQL>@elsif
13/
Enter value for s:75
Score is:75
Grade is:B
PL/SQL procedure is successfully completed
```

5.nested-if

```
Declare
          gender char:='&gender';
         age number(2):=&age;
        charge number(3,1);
begin
        if(gender='m')then
           if(age>=25)then
              charge:=0.10;
           else
              charge:=0.5;
          end if;
       else
          if(age>=25)then
            charge:=0.5;
          else
              charge:=0.06;
        end if;
    end if;
           dbms_output.put_line('gender:'||gender);
          dbms_output.put_line('age:'||to_char(age));
          dbms_output.put_line('charge:'||to_char(charge));
end;
```

OUTPUT:-

```
SQL>@nesif
19/
Enter value for char:m
```

```
Enter value for age:26
Gender:m
Age:26
Charge:0.1
PL/SQL procedure is successfully completed
```

LOOPING STRUCTURES IN PL/SQL BLOCK

BASIC LOOP:

```
Declare
   Count1 number(2);
   Sum1 number(3):=0;
   Avg1 number (3,1);
Begin
       Count1:=1;
        Loop
             Sum1:=sum1+count1;
              Count1:=count1+1;
             Exit when count1>10;
         End loop;
             Avg1:=sum1/(count1-1);
            Dbms_output.put_line('avg of 1 to 10 is:'||to_char(avg1));
End;
```

OUTPUT;-

```
SQL>@basic
15/
Avg of 1 to 10 is:5.5
PL/SQL procedure is successfully completed.
```

WHILE LOOP:

```
DECLARE
 N NUMBER:=&N;
                 I NUMBER(5):=1;
                S NUMBER(5):=0;
BEGIN
                WHILE I<=10 LOOP
  S:=N*I;
  DBMS_OUTPUT.PUT_LINE(N||' * '||I||' = '||S);
  I:=I+1;
 END LOOP;
 END;
```

OUTPUT:-

```
SQL> @while1
17/

Enter value for n: 17
```

```
17 * 1 = 17
17 * 2 = 34
17 * 3 = 51
17 * 4 = 68
17 * 5 = 85
17 * 6 = 102
17 * 7 = 119
17 * 8 = 136
17 * 9 = 153
17 * 10 = 170
PL/SQL procedure successfully completed.
```

FOR LOOP:

```
Declare
   Count1 number(2):=1;
   n number(2);
   Sum1 number(3):=0;
   Avg1 number (3,1);
Begin
       For  count  in  1..10
          Loop
             Sum1:=sum1+count1;
              n:=count1;
          end loop;
             avg1:=sum1/n;
             dbms_output.put_line('avg is:'||avg1);
end;
```

OUTPUT:-

```
SQL>@for1
13/
Avg is:5.5
PL/SQL procedure is successfully completed
```

FOR IN REVERSE:

```
Declare
     n number(5);
      I number(5);
Begin
       n:=&n;
       for i in reverse 1..n-1
         loop
              n:=n*I;
          end loop;
        dbms_output.put_line('fact is:'||n);
end;
```

OUTPUT:-

```
SQL>@fact
17/
Enter value for n:5
Fact is:120
PL/SQL procedure is successfully completed
```

Stored Procedures

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.

1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]
IS
   Declaration section
BEGIN
   Execution section
EXCEPTION
  Exception section
END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

How to execute a Stored Procedure?

There are two ways to execute a procedure.

1) From the SQL prompt.
EXECUTE [or EXEC] procedure_name;
2) Within another procedure – simply use the procedure name.
procedure_name;

NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

Practice Excercise on Procedures

SQL> SELECT * FROM EMP;

EMPNO	ENAME	JOB	SALARY
7931	NOOR	P.L.	35000
7932	SNH	S.E.	28000
7933	SNJ	C.A.	40000
7934	ISHU	J.L.	25000

METHOD1:-

SQL> ED PR.SQL

```
CREATE OR REPLACE PROCEDURE FINDEMP (EMPNO1 IN NUMBER)
AS
 ENAME1 EMP.ENAME%TYPE;
 JOB1 EMP.JOB%TYPE;
BEGIN
 SELECT ENAME,JOB INTO ENAME1,JOB1 FROM EMP WHERE
EMPNO=EMPNO1;
 DBMS_OUTPUT.PUT_LINE('EMPLOYEE IS :'||ENAME1||'  '||JOB1);
EXCEPTION
 WHEN NO_DATA_FOUND THEN
 DBMS_OUTPUT.PUT_LINE('NOT FINDING THE EMPNO: '||EMPNO1);
END;
```

SQL> @PR

Procedure created.

TESTED DATA:-

```
SQL> EXECUTE FINDEMP(7932);
EMPLOYEE IS :SNH  S.E.
PL/SQL procedure successfully completed.
```

METHOD2:-

SQL> ED PROC.SQL

```
CREATE OR REPLACE PROCEDURE FINDEMP
(EMPNO1 IN NUMBER,
 ENAME1 OUT VARCHAR2,
 JOB1 OUT VARCHAR2) AS
BEGIN
  SELECT ENAME,JOB INTO ENAME1,JOB1 FROM EMP WHERE
EMPNO = EMPNO1;
```

```
EXCEPTION
 WHEN NO_DATA_FOUND THEN
 DBMS_OUTPUT.PUT_LINE('NOT FINDING THE EMPNO: '||EMPNO1);
END;
```
SQL> @PROC
Procedure created.

CALLING PROCEDURE:-
SQL> ED PROC1.SQL
```
DECLARE
 EMPNO2 EMP.EMPNO%TYPE:=&EMPNO2;
 ENAME2 EMP.ENAME%TYPE;
 JOB2 EMP.JOB%TYPE;
BEGIN
 FINDEMP(EMPNO2,ENAME2,JOB2);
 DBMS_OUTPUT.PUT_LINE('EMPLOYEE IS :'||ENAME2||'  '||JOB2);
 END;
```
TESTED DATA:-
SQL> @PROC1
```
Enter value for empno2: 7931
old   2:  EMPNO2 EMP.EMPNO%TYPE:=&EMPNO2;
new   2:  EMPNO2 EMP.EMPNO%TYPE:=7931;
EMPLOYEE IS :NOOR  P.L.
PL/SQL procedure successfully completed.
```

PL/SQL Functions

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

General Syntax to create a function is
```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS
Declaration_section
BEGIN
Execution_section
Return return_variable;
EXCEPTION
exception section
Return return_variable;
END;
```

1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
2) The execution and exception section both should return a value which is of the datatype defined in the header section.

Practice Excercise on FUNCTIONS

SQL> ED FUN.SQL

```
CREATE OR REPLACE FUNCTION FACT(N IN NUMBER) RETURN NUMBER
        IS
 F NUMBER:=1;
BEGIN
 FOR I IN 1 .. N
 LOOP
 F:=F*I;
 END LOOP;
 RETURN F;
END;
/
```

```
SQL> @FUN
Function created.
```

METHOD1:-

```
SQL> SELECT FACT(5) FROM DUAL;
   FACT(5)
   ----------
      120
```

METHOD2:-

CALLING FUNCTION:-

SQL> ED FUN1.SQL

```
DECLARE
 N NUMBER:=&N;
 F NUMBER;
BEGIN
 F:=FACT(N);
 DBMS_OUTPUT.PUT_LINE(N||' FACTORIAL IS: '||F);
 END;
```

TESTED DATA:-

METHOD2:-

```
SQL> @FUN1
Enter value for n: 4
old   2:  N NUMBER:=&N;
new   2:  N NUMBER:=4;
```

4 FACTORIAL IS: 24

CURSORS

In order for Oracle to process an SQL statement, it needs to create an area of memory known as the context area. This area contains the information needed to process the statement. The information includes the number of rows processed by the statement, and a pointer to the parsed representation of the statement (parsing an SQL statement is the process whereby information is transferred to the server, at which point the SQL statement is evaluated as being valid). In a query, the active set refers to the rows that will be returned.

A cursor is a handle, or a pointer, to the context area. Through the cursor, a PL/SQL program lets you control the context area, access the information, and process the rows individually.

TYPES OF CURSORS

There are two types of cursors:
An **Implicit** cursor is automatically declared by Oracle every time an SQL statement is executed.
An **Explicit** cursor is defined by the program for any query that returns more than one row of data.

IMPLICIT CURSORS

Whenever a SQL statement is issued, the Database server opens an area of memory in which the command is parsed and executed. This area is called a cursor. In Microsoft SQL Server, this refers to datasets. If a PL/SQL block executes a SELECT command that returns multiple rows, Oracle will displays an error message, which will also invoke the TOO_MANY_ROWS Exception (discussed later in the chapter). To get around this problem, Oracle uses a mechanism called CURSOR. Do not confuse the name CURSOR with the mouse pointer that appears on the screen.

A cursor may be like a temporary file, which stores and controls the rows returned by a SELECT command. SQL*PLUS automatically generates cursors for the queries executed. In PL/SQL, on the other hand, it is necessary for the user to create specific cursors.

When the executable part of a PL/SQL block issues an SQL command, PL/SQL creates an implicit cursor, which has the identifier SQL. PL/SQL manages this cursor for you.

PL/SQL provides some attributes, which allow you to evaluate what happened when the implicit cursor was last used. You can use these attributes in PL/SQL statements like some functions but you cannot use then within SQL statements.

The SQL cursor attributes are: -

%ROWCOUNT	When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row.
%FOUND	TRUE when a cursor has some remaining rows to fetch, and FALSE when a cursor has no rows left to fetch
%NOTFOUND	TRUE if a cursor has no rows to fetch, and FALSE when a cursor has some remaining rows to fetch.
%ISOPEN	TRUE if cursor is opened, or FALSE if cursor has not been opened or has been closed. Only used with explicit cursors.

An example follows: -

```
DECLARE
    row_del_no NUMBER(2);
BEGIN
    DELETE * FROM employee;
    row_del_no                    :=
SQL%ROWCOUNT;
END;
/
```

EXPLICIT CURSORS

The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly declare a cursor to process the rows. You can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package. The steps for using an Explicit Cursor are:

DECLARE	Declaring an explicit cursor names the cursor and defines the query associated with the cursor. The general format for this command is: **CURSOR <cursorname> IS <SELECT statement>;** Cursor name can be any valid PL/SQL variable name. You can use any legal SELECT statements except the one containing Union or Minus operators.

OPEN	Opening the cursor causes the SQL commands to parse the SQL Query (i.e. check for syntax errors). The general format for this command is: **OPEN <cursorname>;** The OPEN command causes the cursor to identify the data rows that satisfy SELECT query. However the data values are not actually retrieved.

FETCH	Loads the row addressed by the cursor pointer into variables and moves the cursor pointer on to the next row ready for the next fetch. The general format for this command is: **FETCH <cursorname> INTO <record variable(s)>;** The record variable is either a single variable or a list of variables that will receive data from the field or fields currently being processed.

CLOSE	Releases the data within the cursor and closes it. The cursor can be reopened to refresh its data. The general format for this command is: **CLOSE <cursorname>;**

Cursors are defined within a DECLARE section of a PL/SQL block. An example follows:

```
DECLARE
   CURSOR mycur IS SELECT emp_ssn, emp_last_name FROM employee;
   …
```

The cursor is defined using the CURSOR keyword, followed by the cursor identifier (MYCUR in this case), and the SELECT statement used to populate it. The SELECT statement can be any legal query. In the example shown above, a cursor is created to retrieve all the employee's SSNs and last names from the EMPLOYEE table.

An OPEN cursor statement is used to execute the SELECT statement, populate the cursor with data, and assign a pointer to the first record of the result set.

```
DECLARE
   CURSOR mycur IS SELECT emp_ssn, emp_last_name FROM employee;
BEGIN
   OPEN mycur;
   …
```

To access the rows of data within the cursor we use the FETCH statement.

```
DECLARE
   CURSOR mycur IS SELECT emp_ssn,emp_last_name FROM employee;
   thisemp_ssn number(10);
   thisempname varchar2(20);
BEGIN
   OPEN mycur;
   FETCH mycur INTO thisemp_ssn, thisemp_name;
   …
```

The FETCH statement reads one record at a time from the result set. In the above example, a FETCH statement is used to fetch the column values for the current cursor row (in this case, it is the 1st row) and puts them into either some declared variables (i.e. *thisemp_ssn* and *thisemp_name*), or a ROWTYPE variable (we will discuss this later). The cursor pointer is then updated to point at the next row. If the cursor has no returned row, the variables will be set to null on the first FETCH attempt, and subsequent FETCH attempts will raise an exception.

The CLOSE statement releases the cursor and any rows within it. The cursor can be re-opened to fetch the same records.

The following is the complete code for the example we have been discussing.

```
-- emp.sql
SET SERVEROUTPUT ON
DECLARE
   CURSOR mycur IS SELECT emp_ssn, emp_last_name FROM employee;
   thisemp_ssn number(10);
   thisemp_name varchar2(20);
BEGIN
  OPEN mycur;
      FETCH mycur INTO thisemp_ssn, thisemp_name;
      DBMS_OUTPUT.PUT_LINE(thisemp_ssn || ':' || thisemp_name);
  CLOSE mycur;
END;
/
```

The DBMS_OUTPUT.PUT_LINE displays the first result of the SELECT statement, as shown below.

```
SQL> @ emp.sql

9996666666:Bordoloi
```

To process all the rows within a cursor, we simply need to place the FETCH statement inside a loop, as illustrated in the following example. The loop constantly fetch a record into the declared variables, and check the cursor NOTFOUND attribute to see if it has successfully fetched a row or not. In other word, its purpose is to retrieve all the results from the SELECT statement, and exits when no more rows are returned. Unlike the example shown earlier, which only outputs the first record, the following code displays every row retrieved from the SELECT statement.

```
-- emp2.sql
SET SERVEROUTPUT ON
DECLARE
   CURSOR mycur IS SELECT emp_ssn, emp_last_name FROM employee;
   thisemp_ssn number(10);
   thisemp_name varchar2(20);
BEGIN
   OPEN mycur;
   loop
      FETCH mycur INTO thisemp_ssn, thisemp_name;
      exit when mycur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(thisemp_ssn || ':' || thisemp_name);
```

```
    END LOOP;
    CLOSE mycur;
END;
/
```

The output of the above example is:

```
SQL> @ emp2.sql

9996666666:Bordoloi
9995555555:Joyner
9994444444:Zhu
9998877777:Markis
9992222222:Amin
9991111111:Bock
9993333333:Joshi
9998888888:Prescott
```

An entire PL/SQL record may also be fetched into a ROWTYPE variable. Doing so reduces the number of variables needed. To access a specific field of a record, simply do the following:

<record>.<column name>
in which <record> is a variable of type <cursor name>%ROWTYPE.
An example follows: -

```
-- salary.sql
DECLARE
    CURSOR mycur IS SELECT emp_ssn, emp_salary FROM
employee;
    emprec          mycur%ROWTYPE; -- type
BEGIN
    OPEN mycur;
    LOOP
        FETCH mycur INTO emprec;
        EXIT WHEN mycur%NOTFOUND;
        -- use emprec.emp_ssn to get the emp_ssn of current record.
        IF emprec.emp_ssn = '9996666666' THEN
            DBMS_OUTPUT.PUT_LINE(emprec.emp_ssn      ||    ':'    ||
emprec.emp_salary);
        END IF;
    END LOOP;
    CLOSE mycur;
END;
/
```

The output of the above example is:

```
SQL> @ salary

9996666666:55000
```

You can use the WHERE CURRENT OF clause to execute DML commands against the current row of a cursor. This feature makes it easier to update rows. An example follows:

```
DECLARE
    CURSOR mycur IS SELECT emp_ssn,emp_salary FROM employee;
    emprec          mycur%ROWTYPE;
BEGIN
    OPEN mycur;
    LOOP
        FETCH mycur INTO emprec;
        EXIT WHEN mycur%NOTFOUND;
        IF emprec.emp_ssn = '9996666666' THEN
            DELETE FROM employee WHERE CURRENT OF mycur;
        END IF;
    END LOOP;
   CLOSE mycur;
END;
/
```

Note that it is not necessary to explicitly specify the row that is to be deleted, PL/SQL supplies the required row identifier from the current record in the cursor to ensure that only the correct row is deleted.

It's possible to vary the returned result set by using one or more parameters; parameters allow you to specify the query selection criteria when you open the cursor.

```
-- salary2.sql
SET SERVEROUTPUT ON
DECLARE
    CURSOR mycur (param1 NUMBER) IS SELECT emp_ssn, emp_salary
FROM employee WHERE emp_ssn = param1;
    emprec          mycur%ROWTYPE;
BEGIN
    OPEN mycur('9996666666');
        FETCH mycur INTO emprec;
        DBMS_OUTPUT.PUT_LINE('Salary for ' || emprec.emp_ssn || ':' ||
```

```
emprec.emp_salary);
    CLOSE mycur;
    OPEN mycur('9995555555');
        DBMS_OUTPUT.PUT_LINE('Salary for ' || emprec.emp_ssn || ':' ||
emprec.emp_salary);
        FETCH mycur INTO emprec;
    CLOSE mycur;
END;
/
```

The output of the above example is:

```
SQL> @ salary2

9996666666:55000
9995555555:43000
```

CURSOR FOR LOOPS

There is an alternative method of handling cursors. It is called the cursor FOR loop, in which the processes of opening, fetching, and closing are implicitly handled. This makes the blocks much simpler to code and easier to maintain.

Use the cursor FOR loop to fetch and process each and every record from the cursor.

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR mycur IS SELECT emp_ssn, emp_salary FROM employee WHERE
emp_dpt_number = 7;
BEGIN
    FOR tempcur IN mycur – assign all the values from mycur to tempcur
    LOOP
        DBMS_OUTPUT.PUT_LINE(tempcur.emp_ssn || ':' ||
tempcur.emp_salary);
    END LOOP;
END;
/
```

EXCEPTION (ERROR) HANDLING

In PL/SQL, a warning or error condition is called an **exception**. A block is always terminated when PL/SQL raises an exception, but you can define your own error handler to capture exceptions and perform some final actions before quitting the block.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the runtime system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called **exception handlers**. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

There are two classes of exceptions, these are:

Predefined - Oracle predefined errors which are associated with specific error codes.

User-defined - Declared by the user and raised when specifically requested within a block. You may associate a user-defined exception with an error code if you wish.

The "exception section" usually appears at the end of the PL/SQL- block.

The syntax is:

```
EXCEPTION
        WHEN <exception1-name> THEN
                <Exception1 handling statements>
        WHEN <exception2-name> THEN
                <Exception2 handling statements>
      ....
        WHEN OTHERS THEN
        <Other handling statements>
END;
```

The <exception handling statements> are the code lines that inform the user of the error. The combination of the WHEN <exception name>, the

THEN statement, and the associated exception-handling statements is called the exception handler. The WHEN OHERS statement is a catch-all exception handler that allows you to present general message to describe errors not handles by a specific error handling statement.

If an error occurs within a block, PL/SQL passes the control to the EXCEPTION section of the block. If no EXCEPTION section exists within the program, or the EXCEPTION section doesn't handle the error that has occurred, the block is terminated with an unhandled exception.

Exceptions propagate up through nested blocks until an exception handler that can handle the error is found. If no exception handler is found in any block, the error is passed out to the host environment. Exceptions occur when either an Oracle error occurs (this automatically raises an exception), or you explicitly raise an error using the RAISE statement.

Here are examples of exceptions:

Exception	**Explanation**
NO_DATA_FOUND	If a SELECT statement attempts to retrieve data based on its conditions, this exception is raised when no rows satisfy the SELECT criteria.
TOO_MANY_ROWS	Since each implicit cursor is capable of retrieving only one row, this exception is raised when more than one row are returned.
DUP_VAL_ON_INDEX	This exception detects an attempt to create an entry in an index whose key column values exist. For example, suppose a billing application is keyed on the invoice number. If a program tries to create a duplicate invoice number, this exception would be raised.
VALUE_ERROR	This exception indicates that there has been an assignment operation WHERE the target field is not long enough to hold the value being placed in it. For example, if the text *ABWEFGH* is assigned to a variable defined as "varchar2(6)", then this exception is raised.

“NO_DATA_FOUND” and “TOO_MANY_ROWS” are the two most common errors found when executing a SELECT statement. The example below takes care of these two conditions.

SQLCODE and SQLERRM should be assigned to some variables before you attempt to use them. Notice that the variable *var_err_msg* in the above example is declared as a character of length 512 bytes. It is because the maximum length of an Oracle error message is 512.

A **User-defined** exception should be declared and raised explicitly by a RAISE statement. It can be declared only in the declarative part of the Pl/SQL block.

The syntax is:

In the declarative section,

<exception_name> EXCEPTION;

The syntax for the RAISE statement is:

RAISE <exception_name>;

Practice Exercise on EXCEPTIONS

```
SQL> ED PRE.SQL
DECLARE
 ENAME1 EMP.ENAME%TYPE;
 SALARY1  EMP.SALARY%TYPE;
 BEGIN
 SELECT ENAME,SALARY INTO ENAME1,SALARY1 FROM EMP WHERE
EMPNO=&EMPNO1;
 DBMS_OUTPUT.PUT_LINE('NAME  '||  ENAME1||'  SALARY
'||SALARY1);
EXCEPTION
 WHEN NO_DATA_FOUND THEN
 DBMS_OUTPUT.PUT_LINE('NO SUCH EMPLOYEE EXISTS');
END;
/
```

TESTED DATA:-

```
SQL> @PRE
Enter value for empno1: 7933
NAME SNJ  SALARY  40000
PL/SQL procedure successfully completed.
SQL> /
Enter value for empno1: 7777
NO SUCH EMPLOYEE EXISTS
PL/SQL procedure successfully completed.
```

USERDEFINED EXCEPTIONS
RAISE EXCEPTION

```
SQL> SELECT * FROM DEPT;
      DNO DNAME          LOC
    -------- ----------      ----------
       10    RESEARCH       HYD
       20    MARKETTING     MUMBAI
       30    FINANCE        DELHI
```

```
SQL> ED RAI.SQL
DECLARE
 DNO1 DEPT.DNO%TYPE :=&DNO;
 DNAME1 DEPT.DNAME%TYPE :='&DNAME';
 LOC1 DEPT.LOC%TYPE := '&LOC';
 INVALIDDEPT EXCEPTION;
BEGIN
 UPDATE DEPT SET DNAME =DNAME1,LOC=LOC1 WHERE
DNO=DNO1;
 IF SQL%NOTFOUND THEN
 RAISE INVALIDDEPT;
 END IF;
 COMMIT;
EXCEPTION
  WHEN INVALIDDEPT THEN
  DBMS_OUTPUT.PUT_LINE('  DEPARTMENT'||DNO1||'  DOES  NOT
EXISTS');
END;
/
```

TESTED DATA:-

```
SQL> @RAI
Enter value for dno: 10
Enter value for dname: HR
Enter value for loc: BANGLORE
PL/SQL procedure successfully completed.
SQL> SELECT * FROM DEPT;
      DNO DNAME          LOC
     ------- --- ----------      ----------
       10    HR              BANGLORE
       20    MARKETTING     MUMBAI
       30    FINANCE         DELHI
SQL> @RAI
Enter value for dno: 40
Enter value for dname: HR
Enter value for loc: NEWYARK
DEPARTMENT 40 DOES  NOT EXISTS
PL/SQL procedure successfully completed.
```

RAISE _APPLICATION_ERROR EXCEPTION

```
SQL> SELECT * FROM EMP;
    EMPNO    ENAME      JOB          SALARY       DNO
    ----------  ----------  ----------  ----------  ----------
     7931       NOOR       P.L.          35000         10
     7932       SNH        S.E.           28000         20
     7933       SNJ        C.A.          40000         20
     7934       ISHU       J.L.          25000         40
```

SQL> ED RAE.SQL

```
DECLARE
 DNO1 DEPT.DNO%TYPE := &DNO1;
 TOTEMP NUMBER;
 INVALIDDEPT EXCEPTION;
BEGIN
 IF DNO1 NOT IN (10,20,30,40) THEN
 RAISE INVALIDDEPT;
 ELSE
 SELECT COUNT(*) INTO TOTEMP FROM EMP WHERE DNO=DNO1;
 DBMS_OUTPUT.PUT_LINE('THE TOTAL NUMBER OF EMPLOYEES ARE'|| TOTEMP);
 END IF;
 DBMS_OUTPUT.PUT_LINE('NO EXCEPTION RAISED');
EXCEPTION
 WHEN INVALIDDEPT THEN
 RAISE_APPLICATION_ERROR(-20330,'THERE IS NO SUCH DEPARTMENT');
END;
/
```

TESTED DATA:-

SQL> @RAE

```
Enter value for dno1: 20
old   2:  DNO1 DEPT.DNO%TYPE := &DNO1;
new   2:  DNO1 DEPT.DNO%TYPE := 20;
THE TOTAL NUMBER OF EMPLOYEES ARE 2
NO EXCEPTION RAISED
PL/SQL procedure successfully completed.
SQL> /
Enter value for dno1: 60
old   2:  DNO1 DEPT.DNO%TYPE := &DNO1;
new   2:  DNO1 DEPT.DNO%TYPE := 60;
DECLARE
ERROR at line 1:
ORA-20220: THERE IS NO SUCH DEPARTMENT
ORA-06512: at line 15
```

Triggers

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax for Creating a Trigger

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
  --- sql statements
END;
```

- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE | AFTER | INSTEAD OF }* - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- *{INSERT [OR] | UPDATE [OR] | DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *[ON table_name]* - This clause identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.
We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );

CREATE TABLE product
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

2) Create the price_history_trigger and execute it.

```
CREATE or REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
 :old.product_name,
 :old.supplier_name,
 :old.unit_price);
END;
/
```

3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.
4)If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.
1) Row level trigger - An event is triggered for each row upated, inserted or deleted.
2) Statement level trigger - An event is triggered for each sql statement executed.
PL/SQL Trigger Execution Hierarchy
The following hierarchy is followed when a trigger is fired.
1) BEFORE statement trigger fires first.
2) Next BEFORE row level trigger fires, once for each row affected.
3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
4) Finally the AFTER statement level trigger fires.

For Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product
(Message varchar2(50),
 Current_Date number(32)
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.
1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
Begin
INSERT INTO product_check
Values('Before update, statement level',sysdate);
END;
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is updated.

```
CREATE or REPLACE TRIGGER Before_Upddate_Row_product
BEFORE
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('Before update row level',sysdate);
END;
/
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values('After update, statement level', sysdate);
End;
/
```

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_check' after each row is updated.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
insert On product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('After update, Row level',sysdate);
END;
/
```

Now lets execute a update statement on table product.

```
UPDATE PRODUCT SET unit_price = 800
WHERE product_id in (100,101);
```

Lets check the data in 'product_check' table to see the order in which the trigger is fired.

```
SELECT * FROM product_check;
```

Output:

Mesage	Current_Date
Before update, statement level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
After update, statement level	26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

How To know Information about Triggers.

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger.
The below statement shows the structure of the view 'USER_TRIGGERS'
DESC USER_TRIGGERS;

NAME	Type
TRIGGER_NAME	VARCHAR2(30)
TRIGGER_TYPE	VARCHAR2(16)
TRIGGER_EVENT	VARCHAR2(75)
TABLE_OWNER	VARCHAR2(30)
BASE_OBJECT_TYPE	VARCHAR2(16)
TABLE_NAME	VARCHAR2(30)
COLUMN_NAME	VARCHAR2(4000)
REFERENCING_NAMES	VARCHAR2(128)
WHEN_CLAUSE	VARCHAR2(4000)
STATUS	VARCHAR2(8)
DESCRIPTION	VARCHAR2(4000)
ACTION_TYPE	VARCHAR2(11)
TRIGGER_BODY	LONG

This view stores information about header and body of the trigger.

*SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_Stat_product';*

The above sql query provides the header and body of the trigger 'Before_Update_Stat_product'.

You can drop a trigger using the following command.
DROP TRIGGER trigger_name;

PL/SQL - Packages

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.
A package will have two mandatory parts:

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.
All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a**private** object.
The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

When the above code is executed at SQL prompt, it produces the following result:

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.
The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust_sal*package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in PL/SQL - Variables

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the *find_sal* method of the *cust_sal* package:

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example:

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records:

```
Select * from customers;
+----+----------+-----+-----------+----------+
```

```
| ID | NAME     | AGE | ADDRESS    | SALARY   |
+----+----------+-----+------------+----------+
|  1 | Ramesh   |  32 | Ahmedabad  |  3000.00 |
|  2 | Khilan   |  25 | Delhi      |  3000.00 |
|  3 | kaushik  |  23 | Kota       |  3000.00 |
|  4 | Chaitali |  25 | Mumbai     |  7500.00 |
|  5 | Hardik   |  27 | Bhopal     |  9500.00 |
|  6 | Komal    |  22 | MP         |  5500.00 |
```

THE PACKAGE SPECIFICATION:

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name  customers.name%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);

   -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);
   --Lists all customers
   PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at SQL prompt, it creates the above package and displays the following result:

```
Package created.
```

CREATING THE PACKAGE BODY:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
   PROCEDURE addCustomer(c_id  customers.id%type,
      c_name customers.name%type,
      c_age  customers.age%type,
      c_addr  customers.address%type,
      c_sal   customers.salary%type)
   IS
   BEGIN
      INSERT INTO customers (id,name,age,address,salary)
         VALUES(c_id, c_name, c_age, c_addr, c_sal);
   END addCustomer;

   PROCEDURE delCustomer(c_id   customers.id%type) IS
```

```
   BEGIN
      DELETE FROM customers
        WHERE id = c_id;
   END delCustomer;

   PROCEDURE listCustomer IS
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list is TABLE OF customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
   BEGIN
      FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('  ||counter||
')'||name_list(counter));
      END LOOP;
   END listCustomer;
END c_package;
/
```

Above example makes use of **nested table** which we will discuss in the next chapter. When the above code is executed at SQL prompt, it produces the following result:

```
Package body created.
```

USING THE PACKAGE:

The following program uses the methods declared and defined in the package*c_package*.

```
DECLARE
   code customers.id%type:= 8;
BEGIN
      c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
      c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
      c_package.listcustomer;
      c_package.delcustomer(code);
      c_package.listcustomer;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish

PL/SQL procedure successfully completed
```