Oracle SQL

14 March 2023 11:46

Select Query

```
SELECT col1, col2 FROM tab1 JOIN tab2 ON tab1.col = tab2.col WHERE condition GROUP BY col HAVING condition ORDER BY col AESC|DESC
```

Database Queries

```
CREATE DATABASE databasename;
CREATE DATABASE testDB;
```

```
DROP DATABASE databasename;
DROP DATABASE testDB;
```

```
BACKUP DATABASE databasename TO DISK = 'filepath';
BACKUP DATABASE testDB TO DISK = 'D:\backups\testDB.bak';
```

```
BACKUP DATABASE databasename TO DISK = 'filepath'WITH DIFFERENTIAL;
BACKUP DATABASE testDB TO DISK = 'D:\backups\testDB.bak'WITH DIFFERENTIAL;
```

DIFFERENTIAL : A differential back up only backs up the parts of the database that have changed since the last full database backup.

Data Definition Queries

CREATE TABLE tablename (column_name data_type NOT NULL, CONSTRAINT pkname PRIMARY KEY (col), CONSTRAINT fkname FOREIGN KEY (col) REFERENCES other_table(col_in_other_table), CONSTRAINT ucname UNIQUE (col), CONSTRAINT ckname CHECK (conditions));

```
CREATE TABLE table_name (column1 datatype,column2 datatype,column3 datatype);
CREATE TABLE Persons (PersonID int, LastName varchar(255), FirstName varchar(255), Address varchar(255), City varchar(255));
```

255 represents the length

```
CREATE TABLE new_table_name AS SELECT column1, column2 FROM existing_table_name WHERE ....;
CREATE TABLE Persons2 AS SELECT FirstName, LastName FROM Persons;
```

```
DROP TABLE table_name;
DROP TABLE Persons2;
```

```
TRUNCATE TABLE table_name;
TRUNCATE TABLE Person2;
```

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

```
ALTER TABLE table_name ADD column_name datatype;
ALTER TABLE Persons ADD Email varchar(255);
```

```
ALTER TABLE table_name DROP COLUMN column_name;
ALTER TABLE Persons DROP COLUMN Email;
```

```
ALTER TABLE table_name RENAME COLUMN old_name to new_name;
ALTER TABLE Persons RENAME COLUMN Email to EmailAddress;
```

```
ALTER TABLE table_name MODIFY column_name datatype;
ALTER TABLE persons MODIFY address varchar(80) ;
```

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

```
ALTER TABLE table_name ADD CONSTRAINT constraint_name constraint_type (columns);
ALTER TABLE person ADD CONSTRAINT PK_Person PRIMARY KEY(columns);
```

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
ALTER TABLE person DROP CONSTRAINT PK_Person;
```

```
ALTER TABLE person ADD constraint_type (column);
ALTER TABLE person ADD PRIMARY KEY (ID);
```

```
ALTER TABLE table_name DROP PRIMARY KEY;
ALTER TABLE Person DROP PRIMARY KEY;
```

Data Manipulation Queries

```
INSERT INTO table_name (column1, column2, column3, ... VALUES (value1, value2, value3, ...);
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country) VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
INSERT INTO Customers (CustomerName, City, Country)VALUES ('Cardinal', 'Stavanger', 'Norway');
```
Inserting in specific columns

```
INSERT INTO table_name VALUES (value1, value2, value3, ...);
INSERT INTO Customers VALUES ('Riddhi', 'Nilawar', Hingoli, ...);

INSERT INTO table_name1 SELECT * FROM table_name2;
INSERT INTO TABLEC SELECT * FROM TABLEA;

--INSERT INTO table_name(col1,col2,col3) VALUES (v1,v2,'v3'),((v1,v2,'v3'),(v1,v2,'v3');
--INSERT INTO Info (id,Cost,city) VALUES (1,200,'Pune'),(2,150,'USA'),(3,345,'France');

UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;
UPDATE Customers SET ContactName = 'Alfred Schmidt', City= 'Frankfurt' WHERE CustomerID = 1;
UPDATE Customers SET ContactName='Juan' WHERE Country='Mexico';

DELETE FROM table_name WHERE condition;
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

DELETE FROM table_name;
DELETE FROM Customers;

DESC table_name -> shows the structure of table.
```

Data Control Queries

```
GRANT privilege_name ON objectname TO user;
GRANT SELECT, UPDATE ON employees TO Bhanu;
GRANT CONNECT, RESOURCE, CREATE SYNONYM, CREATE TABLE, CREATE PROCEDURE, CREATE SEQUENCE TO riddhi;

REVOKE privilege_name ON objectname FROM user;
REVOKE SELECT, UPDATE ON employees FROM Bhanu;
```

Data Transaction Queries

```
COMMIT;

ROLLBACK;

ROLLBACK TO savepoint_name;

SAVEPOINT savepoint_name;

RELASE SAVEPOINT savepoint_name;
```

Joins

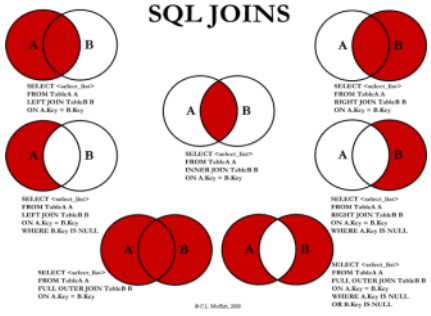
Inner Join

```
SELECT * FROM TableA INNER JOIN TableB ON TableA.PK = TableB.Pk

SELECT * FROM TableA INNER JOIN TableB ON TableA.PK > TableB.Pk
SELECT * FROM TableA INNER JOIN TableB ON TableA.PK < TableB.Pk
SELECT * FROM TableA INNER JOIN TableB ON TableA.PK <> TableB.Pk
```

Equi Join

```
SELECT * FROM TableA INNER JOIN TableB ON TableA.PK = TableB.Pk
```

1. Inner join can have equality (=) and other operators (like <,>,<>) in the join condition.
2. Equi join only have an equality (=) operator in the join condition.
3. Equi join can be an Inner join, Left Outer join, Right Outer join



Left(Outer) Join

```
SELECT * FROM TableA LEFT OUTER JOIN TableB ON TableA.PK = TableB.Pk
```

Right(Outer) Join

```
SELECT * FROM TableA RIGHT OUTER JOIN TableB ON TableA.PK = TableB.Pk
```

Full(Outer) Join

```
SELECT * FROM TableA FULL OUTER JOIN TableB ON TableA.PK = TableB.Pk
```

Left(Outer) Join excluding Inner Join

```
SELECT * FROM TableA LEFT OUTER JOIN TableB ON TableA.PK = TableB.Pk WHERE TableB.PK=NULL
```

Right(Outer) Join excluding Inner Join

```
SELECT * FROM TableA RIGHT OUTER JOIN TableB ON TableA.PK = TableB.Pk WHERE TableA.PK=NULL
```

Full(Outer) Join excluding Inner Join

```
SELECT * FROM TableA FULL OUTER JOIN TableB ON TableA.PK = TableB.Pk WHERE TableA.PK=NULL OR TableB.PK=NULL
```

Cross Join/ Cartesian-Product Join

```
SELECT * FROM TableA, TableB
SELECT * FROM TableA CROSS JOIN TableB
```

Natural Join

```
SELECT column-list FROM left-join-table NATURAL [ INNER | LEFT OUTER | RIGHT OUTER | FULL OUTER ] JOIN right-join-table
```

Natural joins are, by default, natural inner joins; however, there can also be natural left/right/full outer joins. The primary difference between an inner and natural join is that inner joins have an explicit join condition, whereas the natural join's conditions are formed by matching all pairs of columns in the tables that have the same name and compatible data types, making natural joins equi-joins because join condition are equal between common columns. If there are no columns with the same names are found, then the result will be a "cross join"
Ref: https://www.dotnettricks.com/learn/sqlserver/difference-between-inner-join-and-equi-join-and-natural-join

Self Join

```
SELECT column_name(s) FROM table1 T1, table1 T2 WHERE condition;
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City FROM Customers A, Customers B WHERE A.CustomerID <> B.CustomerID AND A.City = B.City ORDER BY A.City;
```

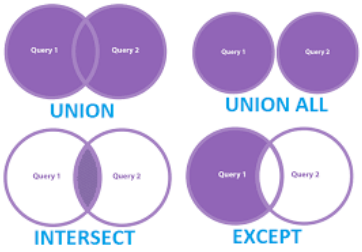
Views

```
CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;
CREATE VIEW Brazil Customers AS SELECT CustomerName, ContactName FROM Customers WHERE Country = 'Brazil';

SELECT * FROM view_name;
SELECT * FROM Brazil Customers;

CREATE OR REPLACE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;
CREATE OR REPLACE VIEW Brazil Customers AS SELECT CustomerName, ContactName, City FROM Customers WHERE Country = 'Brazil';

DROP VIEW view_name;
DROP VIEW Brazil Customers;
```

Set Operations

UNION: Shows unique rows from two result sets.

```
SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;
SELECT 'Customer' AS Type, ContactName, City, Country FROM Customers UNION
SELECT 'Supplier', ContactName, City, Country FROM Suppliers;
```



UNION ALL: Shows all rows from two result sets.

```
SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;
SELECT City, Country FROM Customers WHERE Country='Germany' UNION ALL
SELECT City, Country FROM SuppliersWHERE Country='Germany' ORDER BY City;
```

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL

INTERSECT: Shows rows that exist in both result sets.

```
SELECT column_name(s) FROM table1 INTERSECT SELECT column_name(s) FROM table2;
SELECT 'Customer' AS Type, ContactName, City, Country FROM Customers INTERSECT
SELECT 'Supplier', ContactName, City, Country FROM Suppliers;
```

EXCEPT/MINUS: Shows rows that exist in the first result set but not the second.

```
SELECT column_name(s) FROM table1 MINUS SELECT column_name(s) FROM table2;
SELECT 'Customer' AS Type, ContactName, City, Country FROM Customers MINUS
SELECT 'Supplier', ContactName, City, Country FROM Suppliers;
```

Clauses

HAVING :- The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
```
SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country HAVING COUNT(CustomerID) > 5;
```
Only include countries with more than 5 customers

GROUP BY:- It is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.
```
SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country;
```

ORDER BY:-
```
SELECT * FROM Customers ORDER BY Country, CustomerName;
SELECT * FROM Customers ORDER BY Country ASC, CustomerName DESC;
```
sorted ascending by the "Country" and if any record has same Country then descending by the "CustomerName" column

DISTINCT:- It is used to return only distinct (different) values.
```
SELECT DISTINCT column1, column2, ...FROM table_name;
SELECT DISTINCT Country FROM Customers;
SELECT COUNT(DISTINCT Country) FROM Customers;
```

Operators /Conditions

AND : The AND operator displays a record if all the conditions separated by AND are TRUE.
```
SELECT column1, column2, … FROM table_name WHERE condition1 AND condition2 AND condition3 ...;
SELECT * FROM Customers WHERE Country='Germany' AND City='Berlin';
```

OR: The OR operator displays a record if any of the conditions separated by OR is TRUE.
```
SELECT column1, column2, … FROM table_name WHERE condition1 OR condition2 OR condition3 ...;
SELECT * FROM Customers WHERE City='Berlin' OR City='München';
```

NOT: The NOT operator displays a record if the condition(s) is NOT TRUE.
```
SELECT column1, column2, … FROM table_name WHERE NOT condition;
SELECT * FROM Customers WHERE NOT Country='Germany';
```

COMBINATION :
```
SELECT * FROM Customers WHERE Country='Germany' AND (City='Berlin' OR City='München');
SELECT * FROM Customers WHERE NOT Country='Germany' AND NOT Country='USA';
```

LIKE: The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
The percent sign (%) represents zero, one, or multiple characters
The underscore sign (_) represents one, single character
```
SELECT column1, column2, … FROM table_name WHERE columnN LIKE pattern;
SELECT * FROM Customers WHERE CustomerName LIKE '%a';
```

IN: The IN operator allows you to specify multiple values in a WHERE clause.The IN operator is a shorthand for multiple OR conditions.
```
SELECT column_name(s) FROM table_name WHERE column_name IN (value1, value2, ...);
SELECT column_name(s) FROM table_name WHERE column_name IN (SELECT STATEMENT);
SELECT * FROM Customers WHERE Country IN (SELECT Country FROM Suppliers);
SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'UK');
```

NOT IN
```
SELECT column_name(s) FROM table_name WHERE column_name NOT IN (value1, value2, ...);
SELECT column_name(s) FROM table_name WHERE column_name NOT IN (SELECT STATEMENT);
SELECT * FROM Customers WHERE Country NOT IN (SELECT Country FROM Suppliers);
SELECT * FROM Customers WHERE Country NOT IN ('Germany', 'France', 'UK');
```

IS NULL
```
SELECT column_names FROM table_name WHERE column_name IS NULL;
SELECT CustomerName, ContactName, Address FROM Customers WHERE Address IS NULL;
```

IS NOT NULL
```
SELECT column_names FROM table_name WHERE column_name IS NOT NULL;
SELECT CustomerName, ContactName, Address FROM Customers WHERE Address IS NOT NULL;
```

BETWEEN : The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.
The BETWEEN operator is inclusive: begin and end values are included.
```
SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;
SELECT * FROM Products WHERE Price BETWEEN 10 AND 20;
```

EXISTS: The EXISTS operator is used to test for the existence of any record in a subquery.
The EXISTS operator returns TRUE if the subquery returns one or more records.
```
SELECT SupplierName FROM Suppliers WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =
Suppliers.supplierID AND Price < 20);
```

ANY: ANY means that the condition will be true if the operation is true for any of the values in the range.
```
SELECT column_name(s) FROM table_name WHERE column_name operator ANY (SELECT column_name FROM table_name
WHERE condition);
SELECT ProductName FROM Products WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

ALL:

```
SELECT ProductName FROM Products WHERE ProductID = ALL  (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

Constraints

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly

```
CREATE TABLE table_name ( column1 datatype constraint, column2 datatype constraint, column3 datatype constraint,....);
ALTER TABLE table_name MODIFY column1 datatype constraint, column2 datatype constraint, column3 datatype constraint,....;
```

NOT NULL

```
CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL ,FirstName varchar(255) NOT NULL ,Age int);
ALTER TABLE Persons MODIFY COLUMN Age int NOT NULL;
ALTER TABLE Persons MODIFY COLUMN Age int NULL;
```

UNIQUE

```
CREATE TABLE Persons (ID int NOT NULL UNIQUE);
CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255),Age int, CONSTRAINT UC_Person UNIQUE (ID, LastName));
ALTER TABLE Persons ADD UNIQUE (AGE);
ALTER TABLE Persons ADD CONSTRAINT UC_Person UNIQUE (ID, LastName);
ALTER TABLE Persons DROP CONSTRAINT UC_Person;
```

PRIMARY KEY

```
CREATE TABLE Persons ( ID int NOT NULL PRIMARY KEY, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int);
CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, PRIMARY KEY (ID));
CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, CONSTRAINT PK_Person PRIMARY KEY (ID,
LastName));
In the example above there is only ONE PRIMARY  KEY (PK_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).
ALTER TABLE Persons ADD PRIMARY KEY (ID);
ALTER TABLE Persons ADD CONSTRAINT PK_Person PRIMARY KEY (ID, LastName);
ALTER TABLE Persons DROP CONSTRAINT PK_Person;
```

FOREIGN KEY

```
CREATE TABLE Orders (OrderID int NOT NULL PRIMARY KEY, OrderNumber int NOT NULL, PersonID int FOREIGN KEY REFERENCES Persons(PersonID));
CREATE TABLE Orders (OrderID int NOT NULL, OrderNumber int NOT NULL, PersonID int, PRIMARY KEY (OrderID),
CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID) REFERENCES Persons(PersonID));
ALTER TABLE Orders ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
ALTER TABLE Orders ADD CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
ALTER TABLE Orders DROP CONSTRAINT FK_PersonOrder;
```

CHECK

```
CREATE TABLE Persons ( ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int CHECK (Age>=18) );
CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, City varchar(255),
CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes'));
ALTER TABLE Persons ADD  CHECK  (Age>=18);
ALTER TABLE Persons ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
ALTER TABLE Persons DROP CONSTRAINT CHK_PersonAge;
```

DEFAULT

```
CREATE TABLE Persons ( ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, City varchar(255) DEFAULT 'Sandnes');
ALTER TABLE Persons MODIFY City DEFAULT 'Sandnes';
ALTER TABLE Persons ALTER COLUMN City DROP DEFAULT;
```

INDEX

```
CREATE INDEX index_name  ON table_name (column1, column2, ...);
CREATE UNIQUE INDEX index_name  ON table_name (column1, column2, ...);
CREATE INDEX idx_pname ON Persons (LastName, FirstName);
DROP INDEX index_name;
ALTER  INDEX  old_index  RENAME  TO  new_index;
```

COMMENTS

Single-line

```
-- SELECT * FROM Customers;

SELECT * FROM Customers -- WHERE City='Berlin';
```

Multi-line

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers;

SELECT CustomerName, /*City,*/ Country FROM Customers;

SELECT * FROM Customers WHERE (CustomerName LIKE 'L%'
OR CustomerName LIKE 'R%' /*OR CustomerName LIKE 'S%'
OR CustomerName LIKE 'T%'*/ OR CustomerName LIKE 'W%')
AND Country='USA'
ORDER BY CustomerName;
```

Alias

For Table

```
SELECT column_name(s) FROM table_name AS alias_name;
SELECT o.OrderID, o.OrderDate, c.CustomerName FROM Customers AS c, Orders AS o WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;
```

For Column

```
SELECT column_name AS alias_name FROM table_name;
SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Addres FROM Customers;
```

Wildcards

_ :- represents the single character
% :- represents zer or more character

WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a__%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

Functions

COUNT:

```
SELECT COUNT(column_name) FROM table_name WHERE condition;
SELECT COUNT(ProductID) FROM Products;
```

SUM:

```
SELECT SUM(column_name) FROM table_name WHERE condition;
SELECT SUM(ProductID) FROM Products;
```

MIN:

```
SELECT MIN(column_name) FROM table_name WHERE condition;
SELECT MIN(ProductID) FROM Products;
```

MAX:

```
SELECT MAX(column_name) FROM table_name WHERE condition;
SELECT MAX(ProductID) FROM Products;
```

AVG:

```
SELECT AVG(column_name) FROM table_name WHERE condition;
SELECT AVG(ProductID) FROM Products;
```

Date and Time Functions

- DATE - format YYYY-MM-DD
- DATETIME - format: YYYY-MM-DD HH:MI:SS

- SMALLDATETIME - format: YYYY-MM-DD HH:MI:SS
- TIMESTAMP - format: a unique number

SELECT TO_CHAR(hire_date, 'DD-MON-YYYY') from dual;

SQLCODE

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

From <https://www.ibm.com/docs/en/db2-for-zos/11?topic=applications-sqlcode>

SQLERRM

GREATEST: select the greatest among all values

```
Select greatest(25, 26, 02, 58, 65, 41, 74) from dual;
Select greatest('java', 'phython', 'php', 'Asp.net') from dual;
```

UPPER
TO_CHAR
TO_DATE
SYSDATE:

```
select sysdate from dual;
```

SYSTIMESTAMP
LOWER
TRIM
LTRIM
RTRIM
LENGTH
LPAD
RPAD
CHR
ASCII
INITCAP
INSTR

MOD: This function is used to get the remainder of given values.
n: number to be divided by m
m: number that will divide n
MOD(n, m)

```
Select mod(3,2) from dual;
Select mod(45.2,12) from dual;
```

NVL
REGEXP_COUNT
REGEXP_INSTR
REGEXP_REPLACE
REGEXP_SUBSTR
TRUNC: truncate the given number upto given certain decimal places.

```
Select trunc(145.236) from dual;
Select trunc(145.236,2) from dual;
```

TRANSLATE
DECODE
FLOOR:

```
select floor(11.3) from dual;
select floor(11.5) from dual;
```

CEIL:

```
select ceil(11.5) from dual;
select ceil(11.2)from dual;
select ceil(-12) from dual;
```

SUBSTR:

REPLACE
ROUND:

```
select round(172.41)from dual;
select round(172.411, 2)from dual;
```

ROWNUM : It is used to specify the number of records to return

```
SELECT column_name(s) FROM table_name WHERE ROWNUM <= number;
SELECT * FROM TableA CROSS JOIN TableB WHERE ROWNUM<=5
```

TO_NUMBER

SQRT:

```
select sqrt(25)from dual;
select sqrt(27) from dual;
```

Case Statement

```
SELECT OrderID, Quantity,
CASE
  WHEN Quantity > 30 THEN 'The quantity is greater than 30'
  WHEN Quantity = 30 THEN 'The quantity is 30'
  ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
  WHEN City IS NULL THEN Country
  ELSE City
END);
```

Sequence

```
CREATE SEQUENCE sequence_1
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;
```

```
CREATE SEQUENCE sequence_2
start with 100
increment by -1
minvalue 1
maxvalue 100
cycle;
```

```
CREATE TABLE students(
ID number(10),
NAME char(20)
);
Now insert values into table:
INSERT into students VALUES(sequence_1.nextval,'Ramesh');
INSERT into students VALUES(sequence_1.nextval,'Suresh');
```

Synonym

```
CREATE SYNONYM synonymname
FOR servername.databasename.schemaname.objectname;
```

Oracle PLSQL

15 March 2023 22:38

PLSQL Procedure

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
    exception1-handling-statements
   ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Variable/Datatype/Constants

Variable Declaration and Initialization

```
SET SERVEROUTPUT ON;
DECLARE
   NAME VARCHAR(15);
BEGIN
   NAME := 'Hello';
   DBMS_OUTPUT.PUT_LINE(NAME);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
   NAME VARCHAR(15) := 'HELLO RIDDHI';
BEGIN
   DBMS_OUTPUT.PUT_LINE(NAME);
END;
```

Variable Declaration and Initialization through the query

```
SET SERVEROUTPUT ON;
DECLARE
   NAME VARCHAR(15) := 'HELLO RIDDHI';
BEGIN
   SELECT VALUE INTO NAME FROM TABLEA WHERE PK=1;
   DBMS_OUTPUT.PUT_LINE(NAME);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
   FNAME VARCHAR(15);
   LNAME VARCHAR(15);
BEGIN
   SELECT FIRSTNAME, LASTNAME INTO FNAME,LNAME FROM EMPLOYEE WHERE EMPLOYEEID=1;
   DBMS_OUTPUT.PUT_LINE(FNAME ||' '||LNAME);
END;
```

Anchored Datatype: Data type which you assign to a variable based on a database object.

Syntax: VariableName TableName.ColumnName%TYPE

```
SET SERVEROUTPUT ON;
DECLARE
   FNAME EMPLOYEE.FIRSTNAME%TYPE;
   LNAME EMPLOYEE.LASTNAME%TYPE;

BEGIN
   SELECT FIRSTNAME, LASTNAME INTO FNAME,LNAME FROM EMPLOYEE WHERE EMPLOYEEID=1;
   DBMS_OUTPUT.PUT_LINE(FNAME ||' '||LNAME);
END;
```

CONSTANTS

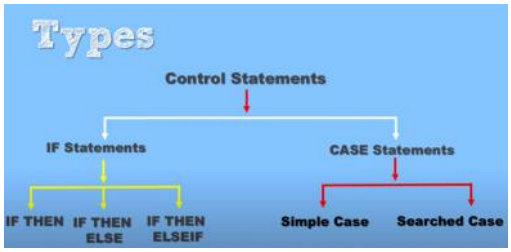
Syntax: Constant-Name CONSTANT Datatype(datatype-width) := value
Initialization of constants should be done in the DECLARE Section only.

```
SET SERVEROUTPUT ON;
DECLARE
   PI CONSTANT NUMBER(7,6) :=3.141592;
BEGIN
   DBMS_OUTPUT.PUT_LINE('PI VALUE: '||PI);
END;
```

DEFAULT & NOT NULL

```
SET SERVEROUTPUT ON;
DECLARE
   PI CONSTANT NUMBER(7,6) NOT NULL DEFAULT 3.141592;
BEGIN
   DBMS_OUTPUT.PUT_LINE('PI VALUE: '||PI);
END;
```

Conditional Statements



IF-THEN

```
SET SERVEROUTPUT ON;
DECLARE
  num NUMBER := 9;
BEGIN
IF num < 10 THEN
 DBMS_OUTPUT.PUT_LINE('Inside The IF');
 END IF;
 DBMS_OUTPUT.PUT_LINE('outside The IF');
END;
```

IF-THEN-ELSE

```
SET SERVEROUTPUT ON;
DECLARE
 v_num    NUMBER := &enter_a_number;
BEGIN
 IF MOD (v_num, 2) = 0 THEN
 DBMS_OUTPUT.PUT_LINE (v_num || ' Is Even');
ELSE
 DBMS_OUTPUT.PUT_LINE (v_num ||' is odd');
END IF;
 DBMS_OUTPUT.PUT_LINE ('IF THEN ELSE Construct complete ');
END;
```

IF-ELSIF-ELSE

```
DECLARE
 v_Place VARCHAR2(30) := '&Place';
BEGIN
 IF v_Place = 'Metropolis' THEN
 DBMS_OUTPUT.PUT_LINE('This City Is Protected By Superman');
ELSIF v_Place = 'Gotham' THEN
 DBMS_OUTPUT.PUT_LINE('This City is Protected By Batman');
ELSIF v_Place = 'Amazon' THEN
DBMS_OUTPUT.PUT_LINE('This City is protected by Wonder Woman');
ELSE
DBMS_OUTPUT.PUT_LINE('Please Call Avengers');
END IF;
DBMS_OUTPUT.PUT_LINE('Thanks For Contacting us');
END;
```

Loops

SIMPLE LOOP

```
SET SERVEROUTPUT ON;
DECLARE
  v_counter   NUMBER :=0;
  v_result  NUMBER;
BEGIN
  LOOP
  v_counter := v_counter+1;
  v_result := 19*v_counter;
  DBMS_OUTPUT.PUT_LINE('19'||' x '||v_counter||' = '|| v_result);
    IF v_counter >=10 THEN
       EXIT;
    END IF;
  END LOOP;
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
  v_counter   NUMBER :=0;
  v_result  NUMBER;
BEGIN
  LOOP
  v_counter := v_counter+1;
  v_result := 19*v_counter;
  DBMS_OUTPUT.PUT_LINE('19'||' x '||v_counter||' = '|| v_result);
  EXIT WHEN v_counter>=10;
  END LOOP;
END;
```

While Loop

```
SET SERVEROUTPUT ON;
DECLARE
  v_counter  NUMBER :=1;
  v_result NUMBER ;
BEGIN
  WHILE  v_counter <= 10
LOOP
  v_result := 9  *v_counter;
  DBMS_OUTPUT.PUT_LINE('9'||' x '||v_counter||' = '||v_result);
  v_counter  := v_counter+1;
END LOOP;
DBMS_OUTPUT.PUT_LINE('out');
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
 v_test BOOLEAN := TRUE;
 v_counter NUMBER  := 0;
BEGIN
 WHILE v_test LOOP
 v_counter := v_counter+1;
 DBMS_OUTPUT.PUT_LINE( v_counter );
  IF v_counter = 10 THEN
    v_test := FALSE;
  END IF;
END LOOP;
DBMS_OUTPUT.PUT_LINE( 'This Statement is outside the loop and will always execute' );
END;
```

For Loop

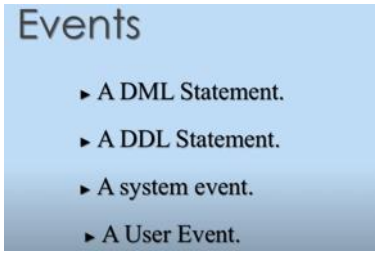
```
SET SERVEROUTPUT ON;
BEGIN
 FOR v_counter IN 1..10 LOOP
   DBMS_OUTPUT.PUT_LINE(v_counter);
 END LOOP;
END;
```

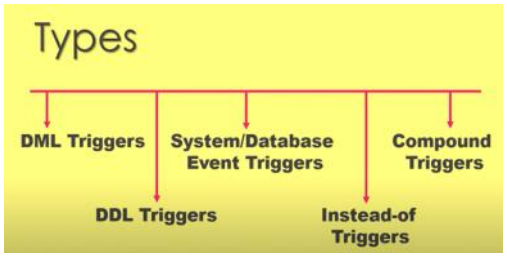
```
SET SERVEROUTPUT ON;
BEGIN
 FOR v_counter IN REVERSE 1..10 LOOP
   DBMS_OUTPUT.PUT_LINE(v_counter);
 END LOOP;
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
 v_result  NUMBER;
BEGIN
 FOR v_counter IN  1..10 LOOP
   v_result:= 19*v_counter;
   DBMS_OUTPUT.PUT_LINE(v_result);
 END LOOP;
END;
```

Triggers

Triggers are used to trigger the events.





Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER} Triggering_event ON table_name
[FOR EACH ROW]
[FOLLOWS another_trigger_name]
[ENABLE/DISABLE]
[WHEN condition]
DECLARE
 declaration statements
BEGIN
 executable statements
EXCEPTION
 exception-handling statements
END;
```

Uses of Database triggers.

1. Using database triggers we can enforce business rules that can't be defined by using integrity constants.
2. Using triggers we can gain strong control over the security.
3. We can also collect statistical information on the table access.
4. We can automatically generate values for derived columns such as auto increment numeric primary key.
5. Using database triggers we can prevent the invalid transactions.

[bctt tweet="We can use #Database Triggers to Prevent Invalid Transactions. Read more" username="Rebellionrider"]

Restriction on The Database Triggers

1. The maximum size of the database trigger body must not exceed 32,760 bytes. This is because triggers' bodies are stored in

1. The maximum size of the database trigger body must not exceed 32,760 bytes. This is because triggers' bodies are stored in LONG datatypes columns.
2. A trigger may not issue transaction control statements or TCL statements such as COMMIT, ROLLBACK or SAVEPOINT. All operations performed when the trigger fires, become part of a transaction. Therefore whenever this transaction is rolled back or committed it leads to the respective rolling back or committing of the operations performed.
3. Any function or procedure called by a database trigger may not issue a transactional control statement. That is unless it contains an autonomous transaction.
4. Declaring LONG or LONG RAW variable is not permissible in the body of the trigger.

Dummy Table

```
CREATE TABLE superheroes (
 sh_name VARCHAR2 (15)
);
```

DROP TRIGGER triger_name; - query to drop the trigger

Trigger will invoke befor inserting rows in table

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TRIGGER bi_Superheroes
BEFORE INSERT ON superheroes
FOR EACH ROW
ENABLE
DECLARE
 v_user  VARCHAR2 (15);
BEGIN
 SELECT user INTO v_user FROM dual;
 DBMS_OUTPUT.PUT_LINE('You Just Inserted a Row Mr./Miss.'|| v_user);
END;
```

INSERT INTO superheroes VALUES ('Ironman');

Trigger will invoke befor modifying rows in table

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TRIGGER bu_Superheroes
BEFORE UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
 v_user  VARCHAR2 (15);
BEGIN
 SELECT user INTO v_user FROM dual;
 DBMS_OUTPUT.PUT_LINE('You Just Modifyed a Row Mr./Miss.'|| v_user);
END;
```

UPDATE superheroes SET SH_NAME = 'Superman' WHERE SH_NAME='Ironman';

Trigger will invoke befor deleting rows in table

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TRIGGER bd_Superheroes
BEFORE DELETE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
 v_user  VARCHAR2 (15);
BEGIN
 SELECT user INTO v_user FROM dual;
 DBMS_OUTPUT.PUT_LINE('You Just Deleted a Row Mr./Miss.'|| v_user);
END;
```

DELETE FROM superheroes WHERE sh_name = 'Superman';

Trigger will invoke befor inserting/updating/deleting rows in table

Predicates : INSERTING, DELETING, UPDATING

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TRIGGER tr_superheroes
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
 v_user  VARCHAR2(15);
BEGIN

 SELECT
   user INTO v_user FROM dual;
 IF INSERTING THEN
   DBMS_OUTPUT.PUT_LINE('one line inserted by '||v_user);
 ELSIF DELETING THEN
```

```
  user INTO v_user FROM dual;
 IF INSERTING THEN
   DBMS_OUTPUT.PUT_LINE('one line inserted by '||v_user);
 ELSIF DELETING THEN
   DBMS_OUTPUT.PUT_LINE('one line Deleted by '||v_user);
 ELSIF UPDATING THEN
   DBMS_OUTPUT.PUT_LINE('one line Updated by '||v_user);
 END IF;
END;
```

```
INSERT INTO superheroes VALUES ('Ironman');
UPDATE superheroes SET SH_NAME = 'Superman' WHERE SH_NAME='Ironman';
DELETE FROM superheroes WHERE sh_name = 'Superman';
```

Table Auditing using DML Triggers

Create Audit Table:

```
CREATE TABLE sh_audit(
 new_name varchar2(30),
 old_name varchar2(30),
 user_name varchar2(30),
 entry_date varchar2(30),
 operation  varchar2(30)
);
```

```
CREATE OR REPLACE trigger superheroes_audit
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
 v_user varchar2 (30);
 v_date  varchar2(30);
BEGIN
 SELECT user, TO_CHAR(sysdate, 'DD/MON/YYYY HH24:MI:SS') INTO v_user, v_date  FROM dual;
 IF INSERTING THEN
   INSERT INTO sh_audit (new_name,old_name, user_name, entry_date, operation)
   VALUES(:NEW.SH_NAME, Null , v_user, v_date, 'Insert');
 ELSIF DELETING THEN
   INSERT INTO sh_audit (new_name,old_name, user_name, entry_date, operation)
   VALUES(NULL,:OLD.SH_NAME, v_user, v_date, 'Delete');
 ELSIF UPDATING THEN
   INSERT INTO sh_audit (new_name,old_name, user_name, entry_date, operation)
   VALUES(:NEW.SH_NAME, :OLD.SH_NAME, v_user, v_date,'Update');
 END IF;
END;
```

Pseudo Records (New/Old)

‘:New’ or ‘:Old’ followed by the name of the column of our source table sh_name.

These Psuedo Records helps us in fetching data from the sh_name column of the underlying source table ‘Superheroes’ and storing it into the audit table sh_audit.

Pseudo Record ‘: NEW’, allows you to access a row currently being processed. In other words, when a row is being inserted or updated into the superheroes table. Whereas Pseudo Record ‘: OLD’ allows you to access a row which is already being either Updated or Deleted from the superheroes table.

```
INSERT INTO superheroes VALUES ('Superman');
UPDATE SUPERHEROES SET SH_NAME = 'Ironman' WHERE SH_NAME='Superman';
DELETE FROM superheroes WHERE SH_NAME = 'Ironman';
```

NEW_NAME	OLD_NAME	USER_NAME	ENTRY_DATE	OPERATION
Superman	(null)	RIDDHI	22/MAR/2023 17:03:53	Insert
Ironman	Superman	RIDDHI	22/MAR/2023 17:03:53	Update
(null)	Ironman	RIDDHI	22/MAR/2023 17:03:54	Delete

Make synchronized backup copy of a table using DML Trigger

Backup table gets automatically populated or updated with the main table simultaneously
Create table without inserting data: **CREATE TABLE superheroes_backup AS SELECT * FROM superheroes WHERE 1=2;**

```
SET SERVEROUTPUT ON;
CREATE or REPLACE trigger Sh_Backup
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
BEGIN
 IF INSERTING THEN
   INSERT INTO superheroes_backup (SH_NAME) VALUES (:NEW.SH_NAME);
 ELSIF DELETING THEN
   DELETE FROM superheroes_backup WHERE SH_NAME =:old.sh_name;
 ELSIF UPDATING THEN
   UPDATE superheroes_backup
   SET SH_NAME =:new.sh_name WHERE SH_NAME =:old.sh_name;
 END IF;
END;
```

```
INSERT INTO superheroes VALUES ('Superman');
UPDATE SUPERHEROES SET SH_NAME = 'Ironman' WHERE SH_NAME='Superman';
DELETE FROM superheroes WHERE SH_NAME = 'Ironman';
```

DDL Trigger with Schema Auditing Example

```
CREATE TABLE schema_audit
 (
   ddl_date      DATE,
   ddl_user      VARCHAR2(15),
   object_created VARCHAR2(15),
   object_name    VARCHAR2(15),
   ddl_operation  VARCHAR2(15)
 );
```

```
CREATE OR REPLACE TRIGGER hr_audit_tr
AFTER DDL ON SCHEMA
BEGIN
INSERT INTO schema_audit VALUES (
sysdate,
sys_context('USERENV','CURRENT_USER'),
ora_dict_obj_type,
ora_dict_obj_name,
ora_sysevent);
END;
```

```
CREATE TABLE RIDDHI(NOTES VARCHAR(100));
```

	DDL_DATE	DDL_USER	OBJECT_CREATED	OBJECT_NAME	DDL_OPERATION
1	22-03-23	RIDDHI	TABLE	RIDDHI	CREATE

System Events

System event triggers come into action when some system event occurs such as database log on, log off, start up or shut down

```
CREATE TABLE hr_evnt_audit
 (
   event_type VARCHAR2(30),
   logon_date DATE,
   logon_time VARCHAR2(15),
   logof_date DATE,
   logof_time VARCHAR2(15)
 );
```

```
CREATE OR REPLACE TRIGGER hr_lgon_audit
AFTER LOGON ON SCHEMA
BEGIN
 INSERT INTO hr_evnt_audit VALUES(
   ora_sysevent,
   sysdate,
   TO_CHAR(sysdate, 'hh24:mi:ss'),
   NULL,
   NULL
 );
 COMMIT;
END;
```

Logoff and Logon in database to see the record

	EVENT_TYPE	LOGON_DATE	LOGON_TIME	LOGOF_DATE	LOGOF_TIME
1	LOGON	22-03-23	17:55:04	(null)	(null)

```
CREATE OR REPLACE TRIGGER hr_lgof_audit
BEFORE LOGOFF ON SCHEMA
BEGIN
 INSERT INTO hr_evnt_audit VALUES(
  ora_sysevent,
  NULL,
  NULL,
  sysdate,
  TO_CHAR(sysdate, 'hh24:mi:ss')
 );
 COMMIT;
END;
```

Logoff and Logon in database to see the record

	EVENT_TYPE	LOGON_DATE	LOGON_TIME	LOGOF_DATE	LOGOF_TIME
1	LOGON	22-03-23	17:55:04	(null)	(null)
2	LOGOFF	(null)	(null)	22-03-23	18:02:12
3	LOGON	22-03-23	18:02:17	(null)	(null)

Instead-Of Triggers

Can be used only on views

Instead-of triggers in oracle database provide a way of modifying views that cannot be modified directly through the DML statements. By using Instead-of triggers, you can perform Insert, Update, Delete and Merge operations on a view in oracle database.

Cursors

Cursor is a pointer to a memory area called context area. This context area is a memory region inside the Process Global Area or PGA assigned to hold the information about the processing of a SELECT statement or DML Statement such as INSERT, DELETE, UPDATE or MERGE.

There are two types of cursors in oracle database:

- Implicit cursor
- Explicit cursor

Steps for creating an Explicit Cursor.To create an explicit cursor you need to follow 4 steps. These 4 steps are:

- Declare
- Open
- Fetch
- Close

```
DECLARE
        CURSOR cursor_name IS select_statement;
BEGIN
         OPEN cursor_name;
         FETCH cursor_name INTO PL/SQL variable [PL/SQL record];
CLOSE cursor_name;
END;
```

How To Create An Explicit Cursor In Oracle Database

```
SET SERVEROUTPUT ON;
DECLARE
 v_name VARCHAR2(30);
 --Declare Cursor
 CURSOR cur_RebellionRider IS
 SELECT firstname FROM EMPLOYEE
 WHERE EMPLOYEEID < 105;
BEGIN
 OPEN cur_RebellionRider;
 LOOP
  FETCH cur_RebellionRider INTO v_name;
  DBMS_OUTPUT.PUT_LINE (v_name);
  EXIT WHEN cur_RebellionRider%NOTFOUND;
 END LOOP;--Simple Loop End
 CLOSE cur_RebellionRider;
END;
```

Cursor Parameter In Oracle Database

```
SET SERVEROUTPUT ON;
DECLARE
```

Cursor Parameter In Oracle Database

```
SET SERVEROUTPUT ON;
DECLARE
 v_name VARCHAR2 (30);
 --Declare Cursor
 CURSOR p_cur_RebellionRider (var_e_id VARCHAR2) IS
 SELECT firstname FROM EMPLOYEE
 WHERE employeeid < var_e_id;
BEGIN
 OPEN p_cur_RebellionRider (105);
LOOP
   FETCH p_cur_RebellionRider INTO v_name;
   EXIT WHEN p_cur_RebellionRider%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE(v_name );
 END LOOP;
 CLOSE p_cur_RebellionRider;
END;
```

How To Create Cursor Parameter With Default Value

```
SET SERVEROUTPUT ON;
DECLARE
 v_name VARCHAR2 (30);
 --Declare Cursor
 CURSOR p_cur_RebellionRider (var_e_id VARCHAR2 :=190) IS
 SELECT firstname FROM EMPLOYEE
 WHERE employeeid < var_e_id;
BEGIN
 OPEN p_cur_RebellionRider ;
LOOP
   FETCH p_cur_RebellionRider INTO v_name;
   EXIT WHEN p_cur_RebellionRider%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE(v_name );
 END LOOP;
 CLOSE p_cur_RebellionRider;
END;
```

Cursor FOR Loop In Oracle Database

```
FOR loop_index IN cursor_name
        LOOP
                Statements…
        END LOOP;
```

```
SET SERVEROUTPUT ON;
DECLARE
 CURSOR cur_RebellionRider IS
 SELECT firstname, lastname FROM employee
 WHERE employeeid < 200;
BEGIN
 FOR L_IDX IN cur_RebellionRider
 LOOP
  DBMS_OUTPUT.PUT_LINE(L_IDX.firstname||' '||L_IDX.lastname);
 END LOOP;
END;
```

Cursor For Loop With Parameterized Cursor

```
SET SERVEROUTPUT ON;
DECLARE
 CURSOR cur_RebellionRider( var_e_id NUMBER) IS
 SELECT firstname, employeeid FROM employee
 WHERE employeeid < var_e_id;
BEGIN
 FOR l_idx IN cur_RebellionRider(200)
 LOOP
  DBMS_OUTPUT.PUT_LINE(l_idx.employeeid||' '||l_idx.firstname);
 END LOOP;
END;
```

Record Datatype

Records are composite data structures made up of different components called *fields*. These fields can have different data types. This means that you can store data of different data types in a single record variable. Similar to the way we define columns in a row of a table.

Types of Record datatype in Oracle database
In Oracle PL/SQL we have three types of Record datatype.

- Table Based Record
- Cursor Based Record, and

User Defined Record.

```
Variable_ name   table_name%ROWTYPE;
Variable_ name   cursor_name%ROWTYPE;
```

Table Based Record Datatype

```
SET SERVEROUTPUT ON;
DECLARE
 v_emp employee%ROWTYPE;
BEGIN
 SELECT * INTO v_emp FROM employee WHERE employeeid = 1;
 DBMS_OUTPUT.PUT_LINE (v_emp.firstname ||' '||v_emp.city);
 DBMS_OUTPUT.PUT_LINE(v_emp.email);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
 v_emp employee%ROWTYPE;
BEGIN
 SELECT firstname INTO v_emp.firstname FROM employee
 WHERE employeeid = 1;
 DBMS_OUTPUT.PUT_LINE (v_emp.firstname);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
 v_emp employee%ROWTYPE;
BEGIN
 SELECT firstname,
  city
 INTO v_emp.firstname,
  v_emp.city
 FROM employee
 WHERE employeeid = 1;
 DBMS_OUTPUT.PUT_LINE (v_emp.firstname);
 DBMS_OUTPUT.PUT_LINE (v_emp.city);
END;
```

Cursor Based Record Datatype Variable

```
SET SERVEROUTPUT ON;
DECLARE
 CURSOR cur_RebellionRider
 IS
 SELECT firstname, city FROM employee
 WHERE employeeid = 1;
 --Cursor Based Record Variable Declare
 var_emp cur_RebellionRider%ROWTYPE;
BEGIN
 OPEN cur_RebellionRider;
 FETCH cur_RebellionRider INTO var_emp;
 DBMS_OUTPUT.PUT_LINE (var_emp.firstname);
 DBMS_OUTPUT.PUT_LINE (var_emp.city);
 CLOSE cur_RebellionRider;
END;
```

```
SET SERVEROUTPUT ON;
BEGIN
 FOR var_emp IN (SELECT firstname, city FROM employee
WHERE employeeid < 200)
 LOOP
  DBMS_OUTPUT.PUT_LINE(var_emp.firstname||' '||var_emp.city);
 END LOOP;
END;
```

User Defined Record Datatype variable

```
SET SERVEROUTPUT ON;
DECLARE
 TYPE rv_dept IS RECORD(
  f_name  VARCHAR2(20),
  d_name  departments.department_name%type
 );
```

```
 TYPE rv_dept IS RECORD(
   f_name  VARCHAR2(20),
   d_name  departments.department_name%type
 );
 var1 rv_dept;
BEGIN
 SELECT first_name , department_name
 INTO var1.f_name, var1.d_name
 FROM employees  join departments
 Using (department_id) WHERE employee_id = 100;

 DBMS_OUTPUT.PUT_LINE(var1.f_name||' '||var1.d_name);
END;
```

Functions

There are two types of PL/SQL functions in Oracle Database, these are

1. Pass-by-Value Functions and
2. Pass-by-Reference functions

```
CREATE [OR REPLACE] FUNCTION function_name
(Parameter 1, Parameter 2…)
RETURN datatype
IS
        Declare variable, constant etc.
BEGIN
        Executable Statements
        Return (Return Value);
END;
```

PL/SQL function for calculating "Area of the Circle".

```
--Function Header
CREATE OR REPLACE FUNCTION circle_area (radius NUMBER)
RETURN NUMBER IS
--Declare a constant and a variable
pi      CONSTANT NUMBER(7,2) :=      3.141;
area    NUMBER(7,2);
BEGIN
 --Area of Circle pi*r*r;
 area := pi * (radius * radius);
 RETURN area;
END;
```

Function call - Type 1

```
SET SERVEROUTPUT ON;
BEGIN
DBMS_OUTPUT.PUT_LINE(circle_area(25));
END;
```

Function call - Type 2

```
SET SERVEROUTPUT ON;
DECLARE
AREA NUMBER(7,2);
BEGIN
AREA:= circle_area(25);
DBMS_OUTPUT.PUT_LINE(AREA);
END;
```

Stored Procedures

```
CREATE [OR REPLACE] PROCEDURE pro_name (Parameter – List)
IS [AUTHID      DEFINER | CURRENT_USER]
        Declare statements
BEGIN
        Executable statements
END procedure name;
/
```

The AUTHID clause is used for setting the authority model for the PL/SQL Procedures. This clause has two flags. DEFINER and CURRENT_USER

As this clause is optional thus in case if you do not use AUTHID clause then Oracle Engine will set the authority (AUTHID) to the DEFINER by default for you. Now, you must be wondering what these DEFINER and CURRENT_USER rights are?

DEFINER right: Definer right is the default right assigned to the procedure by oracle engine. This right means anyone with Execution Privilege on the procedure acts as if they are the owner of the schema in which the privilege is created.

It does not return any value. PL/SQL function return same value

```
        Executable statements
END procedure name;
/
```

It does not return any value. PLSQL function return some value.

DEFINER right: Definer right is the default right assigned to the procedure by oracle engine. This right means anyone with Execution Privilege on the procedure acts as if they are the owner of the schema in which the privilege is created.

CURRENT_USER right: Setting the authority level of a stored procedure to the current_user right overrides the default right which is definer and change it to the invoker rights.

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE Pr_Riddhi IS
 var_name VARCHAR2 (30):= 'Nilawar';
 var_web VARCHAR2 (30) := 'Hitachi.com';
BEGIN
 DBMS_OUTPUT.PUT_LINE('Whats Up Internet? I am '||var_name||' from '||var_web);
END Pr_Riddhi ;
```

Calling stored procedures -Type 1

```
EXECUTE Hita;
```

Calling stored procedures -Type 2

```
EXEC Hita;
```

Calling stored procedures -Type 3

```
BEGIN
   Hita;
END;
```

Stored Proceduress with parameter

```
CREATE OR REPLACE PROCEDURE emp_sal( dep_id NUMBER, sal_raise NUMBER)
IS
BEGIN
 UPDATE emp SET salary = salary * sal_raise WHERE department_id = dep_id;
END;
```

Packages

A package can hold multiple database objects such as

- Stored Procedures
- PL/SQL Functions
- Database Cursors
- Type declarations as well as
- Variables

PL/SQL package is divided into two parts:

1. The Package Specification, also known as the Header and(required)
2. The Package Body(optional)

```
CREATE OR REPALCE PACKAGE BODY pkg_name IS
        Variable declaration;
        Type Declaration;
BEGIN
        Implementation of the package elements…
END [pkg_name];
```

```
--Package Header
CREATE OR REPLACE PACKAGE pkg_test IS
 FUNCTION prnt_strng RETURN VARCHAR2;
 PROCEDURE proc_superhero(name VARCHAR2);
END pkg_test;


--Package Body
CREATE OR REPLACE PACKAGE BODY pkg_test IS
 --Function Implimentation
 FUNCTION prnt_strng RETURN VARCHAR2 IS
  BEGIN
   RETURN 'hitachi.com';
  END prnt_strng;

 --Procedure Implimentation
  PROCEDURE proc_superhero(name VARCHAR2) IS
   BEGIN
    INSERT INTO superheroes (sh_name) VALUES(name);
   END;
```

```
  PROCEDURE proc_superhero(name VARCHAR2) IS
   BEGIN
    INSERT INTO superheroes (sh_name) VALUES(name);
   END;

END pkg_test;
```

```
--Package Calling Function
BEGIN
 DBMS_OUTPUT.PUT_LINE (pkg_test.PRNT_STRNG);
END;
```

Exception Handling

There are two types of PL/SQL exceptions in Oracle database.

1. System-defined exceptions and
2. User-defined exceptions

There are three ways of declaring user-define exceptions in Oracle Database.

- By declaring a variable of EXCEPTION type in declaration section.

You can declare a user defined exception by declaring a variable of EXCEPTION datatype in your code and raise it explicitly in your program using RAISE statement and handle them in the Exception Section.

- Declare user-defined exception using PRAGMA EXCEPTION_INIT function.

Using PRAGMA EXCEPTION_INIT function you can map a non-predefined error number with the variable of EXCEPTION datatype. Means using the same function you can associate a variable of EXCEPTION datatype with a standard error.

- RAISE_APPLICATION_ERROR method.

Using this method you can declare a user defined exception with your own customized error number and message.

Declaring a user-define exception using Exception variable is a three step process. These three steps are –

1. **Declare a variable of exception datatype –** This variable is going to take the entire burden on its shoulders.
2. **Raise the Exception –** This is the part where you tell the compiler about the condition which will trigger the exception.
3. **Handle the exception –** This is the last section where you specify what will happen when the error which you raised will trigger.

Using Exception method

```
SET SERVEROUTPUT ON;
DECLARE
 var_dividend NUMBER := 24;
 var_divisor NUMBER := 0;
 var_result NUMBER;
 ex_DivZero EXCEPTION;
BEGIN
 IF var_divisor = 0 THEN
   RAISE ex_DivZero;
 END IF;
 var_result := var_dividend/var_divisor;
 DBMS_OUTPUT.PUT_LINE('Result = ' ||var_result);
 EXCEPTION WHEN ex_DivZero THEN
    DBMS_OUTPUT.PUT_LINE('Error Error - Your Divisor is Zero');
END;
```

Using Raise Application Error method

```
SET SERVEROUTPUT ON;
ACCEPT var_age NUMBER PROMPT 'What is yor age';
DECLARE
 age   NUMBER := &var_age;
BEGIN
 IF age < 18 THEN
   RAISE_APPLICATION_ERROR (-20008, 'you should be 18 or above for the DRINK!');
 END IF;
 DBMS_OUTPUT.PUT_LINE ('Sure, What would you like to have?');
 EXCEPTION WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE (SQLERRM);
END;
```

Using PRAGMA EXCEPTION_INIT function

```
SET SERVEROUTPUT ON;
DECLARE
 ex_age    EXCEPTION;
 age       NUMBER    := 17;
 PRAGMA EXCEPTION_INIT(ex_age, -20008);
BEGIN
 IF age<18 THEN
   RAISE_APPLICATION_ERROR(-20008, 'You should be 18 or above for the drinks!');
 END IF;

 DBMS_OUTPUT.PUT_LINE('Sure! What would you like to have?');

 EXCEPTION WHEN ex_age THEN
   DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

Notes-misc

21 March 2023 15:30

/ After every block give '/' to end the block

Doubts:
How to terminate the code in oracle sql developer?

```
CREATE TABLE FileDetails
(FileName varchar(255), FilePath varchar(255));

INSERT INTO FILEDETAILS(FILENAME,FILEPATH,MODIFIEDDATE)VALUES('a','c','28-10-2000');

select * from FILEDETAILS ;

CREATE OR REPLACE PROCEDURE get_data( order_a_d VARCHAR2)
IS
BEGIN
select * from FILEDETAILS order by order_a_d;
END;

SELECT * FROM FILEDETAILS ORDER BY filename asc;
```

```
CREATE PROCEDURE GetOrders (filepath VARCHAR2, OrderBy VARCHAR2, SortOrder varchar2)
AS
SELECT filename, filepath, Modifieddate
FROM Sales.SalesOrderHeader
WHERE SalesPersonID = @SalesPersonID
ORDER BY
  CASE WHEN @OrderBy = 'Due Date' AND @SortOrder = 'DESC' THEN DueDate END DESC,
  CASE WHEN @OrderBy = 'Due Date' THEN DueDate END,
  CASE WHEN @OrderBy = 'Order Date' AND @SortOrder = 'DESC' THEN OrderDate END DESC,
  CASE WHEN @OrderBy = 'Order Date' THEN OrderDate END,
  CASE WHEN @OrderBy = 'Total Due' AND @SortOrder = 'DESC' THEN TotalDue END DESC,
  CASE WHEN @OrderBy = 'Total Due' THEN TotalDue END
```