Unit-2

2.1.Data Definition:

SQL stands for Structured Query Language use for storing, manipulating and retrieving relational database data.
SQL queries to retrieve data from database same as you can adding and manipulating database data.

- SQL is a very powerful and diverse database language use to storing data into databases. SQL is loosely typed language so you can learn easily.
- In this SQL tutorial, we use command line examples to know about executing speed of SQL. It's take very bit of time for executing and retrieving result.
- SQL is a greater tool with web languages such as PHP, Python, Java, ASP et cetera to build dynamic web applications.

Before starting SQL, relational databases have several point that are important to keep in mind.

1. RDBMS stands for Relational Database Management System.
2. Data Integrity : Store data only once and avoiding data duplication.
3. SQL Constraints : Constraints are the rules which are apply to table columns to store valid data and prevents the user to storing/entering invalid data into table columns.
4. Better security : Assign grant or privilege to a individual User. Using this grant user can store confidential data into table by using username or password.
5. Database Normalization : Database normalization is the process to store database data very efficiently. No need to store same data more then one time and reduce the Data redundancy.
6. Different types of relationships : One to one, One to many, Many to many
    1. One to one relationship : merging for two tables.
    2. One to many relationship : create a foreign key from an parent table to the child table.
    3. Many to many relationship : create a new relation table.

Keep in your mind...

- SQL is not case sensitive.
- But SQL Data is a case sensitive.

SQL
SQL stands for Structured Query Language. SQL is used to create, remove, alter the database and database objects in a database management system and to store, retrieve, update the data in a database. SQL is a standard language for creating, accessing, manipulating database management system. SQL works for all modern relational database management systems, like SQL Server, Oracle, MySQL, etc.

Different types of SQL commands:
SQL commands can be categorized into five categories based on their functionality.

DDL

DDL stands for data definition language. DDL commands are used for creating and altering the database and database object in the relational database management system, like CREATE DATABASE, CREATE TABLE, ALTER TABLE, etc. The most used DDL commands are CREATE, DROP, ALTER, and TRUNCATE.

- ***CREATE:***CREATE command is used to create a database and database object like a table, index, view, trigger, stored procedure, etc.
- ***Syntax:****CREATE TABLE Employee (Id INT, Name VARHCAR(50), Address VARCHAR (100));*
- ***ALTER:***ALTER command is used to restructure the database object and the settings in the database.
- ***Syntax:****ALTER TABLE Employee ADD Salary INT;*
- ***TRUNCATE:****The TRUNCATE command is used to remove all the data from the table. TRUNCATE command empties a table.*
- ***Syntax:****TRUNCATE TABLE Employee;*
- ***DROP:***DROP command is used to remove the database and database object.
- ***Syntax:***DROP TABLE Employee;

DML:DML stands for data manipulation language. DML commands are used for manipulating data in a relational database management system. DML commands are used for adding, removing, updating data in the database system, like INSERT INTO TableName, DELETE FROM TableName, UPDATE tableName set data, etc. The most used DML commands are INSERT INTO, DELETE FROM, UPDATE.

- ***INSERT INTO:****INSERT INTO command is used to add data to the database table.*
- ***Syntax:***INSERT INTO Employee (Id, Name, Address, Salary) VALUES (1, 'Arvind Singh', 'Pune', 1000);
- ***UPDATE***:UPDATE command is used to update data in the database table. A condition can be added using the WHERE clause to update a specific row.
- ***Syntax***:UPDATE Employee SET Address = 'Pune India', Salary = 100 WHERE Id =1;
- ***DELETE****:DELETE command is used to remove data from the database table. A condition can be added using the WHERE clause to remove a specific row which meets the condition.*
- ***Syntax***:DELETE FROM Employee WHERE Id =1;

DQL

DQL stands for the data query language. DQL command is used for fetching the data. DQL command is used for selecting data from the table, view, temp table, table variable, etc. There is only one command under DQL which is the SELECT command.

Syntax

SELECT * FROM Employee;

DCL

DCL stands for data control language. DCL commands are used for providing and taking back the access rights on the database and database objects. DCL command used for controlling user's access on the data. Most used DCL commands are GRANT and REVOKE.

GRANT:GRANT is used to provide access right to the user.

Syntax

GRANT INSERT, DELETE ON Employee TO user;
REVOKE:REVOKE command is used to take back access right from the user, it cancels access right of the user from the database object.
Syntax
REVOKE ALL ON Employee FROM user;
TCL
TCL stands for transaction control language. TCL commands are used for handling transactions in the database. Transactions ensure data integrity in the multi-user environment. TCL commands can rollback and commit data modification in the database. The most used TCL commands are COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION.
COMMIT:COMMIT command is used to save or apply the modification in the database.
ROLLBACK:ROLLBACK command is used to undo the modification.
SAVEPOINT:SAVEPOINT command is used to temporarily save a transaction, the transaction can roll back to this point when it's needed.

Syntax:Just write COMMIT or ROLLBACK or SAVEPOINT;

2.2 Basic Structure of SQL queries:

1. Basic structure of an SQL expression consists of **select, from** and **where** clauses.
   - **select** clause lists attributes to be copied - corresponds to relational algebra **project**.
   - **from** clause corresponds to Cartesian product - lists relations to be used.
   - **where** clause corresponds to selection predicate in relational algebra.
2. Typical query has the form
3. **select** $A_1, A_2, \ldots, A_n$
4.
5. **from** $r_1, r_2, \ldots, r_m$
6.
7. **where** *P*
8.

   where each A_i represents an attribute, each r_i a relation, and *P* is a predicate.

9. This is equivalent to the relational algebra expression

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

   - If the where clause is omitted, the predicate *P* is true.
   - The list of attributes can be replaced with a * to select all.
   - SQL forms the Cartesian product of the relations named, performs a selection using the predicate, then projects the result onto the attributes named.
   - The result of an SQL query is a relation.
   - SQL may internally convert into more efficient expressions.

The select Clause

1. An example: Find the names of all branches in the *account* relation.
2. **select** *bname*

3.
4. **from** *account*
5.
6. **distinct** vs. **all**: elimination or not elimination of duplicates.

   Find the names of all branches in the *account* relation.

   select distinct *bname*

   from *account*

   By default, duplicates are not removed. We can state it explicitly using **all**.

   select all *bname*

   from *account*

7. select * means select all the attributes. Arithmetic operations can also be in the selection list.

The where Clause

1. The predicates can be more complicated, and can involve
   - Logical connectives **and, or** and **not**.
   - Arithmetic expressions on constant or tuple values.
   - The **between** operator for ranges of values.
2. Example: Find account number of accounts with balances between $90,000 and $100,000.
   3. **select** *account#*
   4.
   5. **from** *account*
   6.
   7. **where** *balance* **between** *90000* **and** *100000*

The from Clause

1. The **from** class by itself defines a Cartesian product of the relations in the clause.
2. SQL does not have a natural join equivalent. However, natural join can be expressed in terms of a Cartesian product, selection, and projection.
3. For the relational algebra expression

$$\Pi_{cname,loan\#}(borrower\ 1\ loan)$$

we can write in SQL,

select distinct *cname, borrower.loan#*

from *borrower, loan*

where *borrower.loan# = loan.loan#*

4. More selections with join: ``Find the names and loan numbers of all customers who have a loan at the SFU branch,'' we can write in SQL,
5. **select distinct** *cname, borrower.loan#*
6.
7. **from** *borrower, loan*
8.
9. **where** *borrower.loan# = loan.loan#*
10.
11. **and** *bname=``SFU''*

The Rename Operation

1. Rename: a mechanism to rename both relations and attributes.
2. **as**-clause can appear in both the select and from clauses:

   old-name **as** *new-name*.

3. Example.
   4. **select distinct** *cname, borrower.loan#* **as** *loan_id*
   5.
   6. **from** *borrower, loan*
   7.
   8. **where** *borrower.loan# = loan.loan#*
   9. **and** *bname= ``SFU''*

Tuple Variables

1. Tuple variables can be used in SQL, and are defined in the **from** clause:
2. **select distinct** *cname, T.loan#*
3.
4. **from** *borrower* ***as*** *S, loan* ***as*** *T*
5.
6. **where** *S.loan# = T.loan#*
7.

   Note: The keyword **as** is optional here.

8. These variables can then be used throughout the expression. Think of it as being something like the rename operator.

Finds the names of all branches that have assets greater than at least one branch located in Burnaby.

select distinct *T.bname*

from *branch S, branch T*

where *S.bcity=``Burnaby''* **and** *T.assets > S.assets*

String Operations

1. The most commonly used operation on strings is pattern matching using the operator **like**.
2. String matching operators **%** (any substring) and **_** (underscore, matching any character).

   E.g., ``___%" matches any string with at least 3 characters.

3. Patterns are case sensitive, e.g., ``Jim" does not match ``jim".
4. Use the keyword **escape** to define the *escape* character.

   E.g., like ``ab%tely \% \" escape `` \" matches all the strings beginning with ``ab" followed by a sequence of characters and then ``tely" and then ``% \".

   Backslash overrides the special meaning of these symbols.

5. We can use **not like** for string mismatching.
6. Example. Find all customers whose street includes the substring ``Main".
7. **select** *cname*
8.
9. **from** *customer*
10.
11. **where** *street* **like** ``*%Main%''*
12.
13. SQL also permits a variety of functions on character strings, such as concatenating (using `` || "), extracting substrings, finding the length of strings, converting between upper case and lower case, and so on.

Ordering the Display of Tuples

1. SQL allows the user to control the order in which tuples are displayed.
   - **order by** makes tuples appear in sorted order (ascending order by default).
   - **desc** specifies descending order.
   - **asc** specifies ascending order.

2. **select** *
3.
4. **from** *loan*
5.
6. **order by** *amount* **desc**, *loan#* **asc**
7.

Sorting can be costly, and should only be done when needed.

Duplicate Tuples

- Formal query languages are based on mathematical relations. Thus no duplicates appear in relations.
- As duplicate removal is expensive, SQL allows duplicates.
- To remove duplicates, we use the **distinct** keyword.
- To ensure that duplicates are not removed, we use the **all** keyword.
- *Multiset* (bag) versions of relational algebra operators.
    - if there are c_1 copies of tuples t_1 in r_1, and t_1 satisfies selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
    - for each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$.
    - if there are c_1 copies of tuple t_1 in r_1, and c_2 copies of tuple t_2 in r_2, there is $c_1 \times c_2$ copies of tuple $t_1.t_2$ in $r_1 \times r_2$.
- An SQL query of the form
- **select** $A_1, A_2, \ldots, A_n$
-
- **from** $r_1, r_2, \ldots, r_m$
-
- **where** *P*

is equivalent to the algebra expression

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

using the multiset versions of the relational operators σ, Π, and $\times$.

2.3 Set Operations

1. SQL has the set operations **union, intersect** and **except**.
2. Find all customers having an account.
3. **select distinct** *cname*
4.
5. **from** *depositor*
6.
7. **union**: Find all customers having a loan, an account, or both. branch.
8. (**select** *cname*
9.
10. **from** *depositor*)
11.

union

(**select** *cname*

from *borrower*)

intersect: Find customers having a loan **and** an account.

(**select distinct** *cname*

from *depositor*)

intersect

(**select distinct** *cname*

from *borrower*)

except: Find customers having an account, but **not** a loan.

(**select distinct** *cname*

from *depositor*)

except

(**select** *cname*

from *borrower*)

Some additional details:

- **union** eliminates duplicates, being a set operation. If we want to retain duplicates, we may use **union all**, similarly for **intersect** and **except**.
- Not all implementations of SQL have these set operations.
- **except** in SQL-92 is called **minus** in SQL-86.
- It is possible to express these queries using other operations.

2.4 Aggregate Functions

1. In SQL we can compute functions on groups of tuples using the **group by** clause.

   Attributes given are used to form groups with the same values. SQL can then compute

   - average value -- **avg**
   - minimum value -- **min**
   - maximum value -- **max**
   - total sum of values -- **sum**
   - number in group -- **count**

   These are called **aggregate functions**. They return a single value.

2. Some examples:

1. Find the average account balance at each branch.

```
select bname, avg (balance)

        from account

        group by bname
```

8. Find the number of depositors at each branch.

```
select bname, count (distinct cname)

        from account, depositor

        where account.account# = depositor.account#

        group by bname
```

We use **distinct** so that a person having more than one account will not be counted more than once.

17. Find branches and their average balances where the average balance is more than $1200.

```
select bname, avg (balance)

        from account

        group by bname

        having avg (balance) > 1200
```

Predicates in the **having** clause are applied after the formation of groups.

26. Find the average balance of each customer who lives in Vancouver and has at least three accounts:

```
select depositor.cname, avg (balance)

        from depositor, account, customer

        where depositor.cname = customer.cname and
account.account# = depositor.account#

    and ccity=``Vancouver''

        group by depositor.cname

        having count (distinct account#) ≥ 3
```

3. If a **where** clause and a **having** clause appear in the same query, the **where** clause predicate is applied first.
    - Tuples satisfying **where** clause are placed into groups by the **group by** clause.

- The **having** clause is applied to each group.
- Groups satisfying the having clause are used by the **select** clause to generate the result tuples.
- If no **having** clause is present, the tuples satisfying the **where** clause are treated as a single group.

Null Values

1. With insertions, we saw how **null** values might be needed if values were unknown. Queries involving nulls pose problems.
2. If a value is not known, it cannot be compared or be used as part of an aggregate function.
3. All comparisons involving null are **false** by definition. However, we can use the keyword **null** to test for null values:
4. **select distinct** *loan#*
5.
6. **from** *loan*
7.
8. **where** *amount* **is null**
9.
10. All aggregate functions except **count** ignore tuples with null values on the argument attributes.

2.5 Nested Subqueries: Set Membership

1. We use the **in** and **not in** operations for set membership.
2. **select distinct** *cname*
3.
4. **from** *borrower*
5.
6. **where** *cname* **in**
7.
8. (**select** *cname*
9. **from** *account*
10. **where** *bname=``SFU''*)
11.
12. Note that we can write the same query several ways in SQL.
13. We can also test for more than one attribute:
14. **select distinct** *cname*
15.
16. **from** *borrower, loan*
17.
18. **where** *borrower.loan# = loan.loan#*
19. **and** *bname=``SFU''*
20.
21. **and** (*bname,*
22.
23. (**select** *bnam*
24. **from** *account, depositor* **where** *depositor.account# = account.account#*)

25.

This finds all customers who have a loan and an account at the SFU branch in yet another way.

26. Finding all customers who have a loan but not an account, we can use the **not in** operation.

Set Comparison

1. To compare set elements in terms of inequalities, we can write
2. **select distinct** *T.bname*
3.
4. **from** *branch T,branch S*
5.
6. **where** *T.assets* > *S.assets*
7. **and** *S.bcity*=``*Burnaby''*
8.

or we can write

select *bname*

from *branch*

where *assets* > **some**

(**select** *assets*

from *branch*

where *bcity*=``*Burnaby''*)

to find branches whose assets are greater than some branch in Burnaby.

9. We can use any of the equality or inequality operators with **some**. If we change > **some** to > **all**, we find branches whose assets are greater than all branches in Burnaby.
10. Example. Find branches with the highest average balance. We cannot compose aggregate functions in SQL, e.g. we cannot do **max (avg ...))**.

    Instead, we find the branches for which average balance is greater than or equal to all average balances:

select *bname*

from *account*

group by *bname*

having avg (*balance*) $\geq$ **all**

(**select avg** (*balance*)

from *account*
group by *bname*)

Test for Empty Relations

1. The **exists** construct returns **true** if the argument subquery is nonempty.
2. Find all customers who have a loan **and** an account at the bank.
   3. **select** *cname*
   4.
   5. **from** *borrower*
   6.
   7. **where exists** (**select** *
   8.
   9. **from** *depositor*
   10.
   11. **where** *depositor.cname = borrower.cname*)

Test for the Absence of Duplicate Tuples

1. The **unique** construct returns **true** if the argument subquery contains no duplicate tuples.
2. Find all customers who have only one account at the SFU branch.
   3.**select** *T.cname*
   4.
   5. **from** *depositor* **as** *T*
   6.
   7. **where unique** (**select** *R.cname*

from *account, depositor* **as** *R*

where *T.cname = R.cname* **and**

account.bname = ``SFU'')

2.6:complex queries:

Derived Relations

1. SQL-92 allows a subquery expression to be used in the **from** clause.
2. If such an expression is used, the result relation must be given a name, and the attributes can be renamed.
3. Find the average account balance of those branches where the average account balance is greater than $1,000.

```
4. select bname, avg-balance
5.
6.                 from (select bname, avg(balance)
7.
8.                                                 from account
9.
10.                                                group by
bname)
11.
12.                                                as result(bname,
avg-balance)
13.
14.                 where avg-balance > 1000
```

Views

1. A view in SQL is defined using the **create view** command:
2. **create view** *v* **as** ⟨query expression⟩

   where ⟨*query expression*⟩ is any legal query expression.

   The view created is given the name *v*.

3. To create a view *all-customer* of all branches and their customers:

```
4. create view all-customer as
5.
6.                 (select bname, cname
7.
8.                 from depositor, account
9.
10.                where depositor.account# = account.account#)
11.
12.                union
13.
14.                (select bname, cname
15.
16.                from borrower, loan
17.
18.                where borrower.loan# = loan.loan#)
19.
```

20. Having defined a view, we can now use it to refer to the virtual relation it creates. View names can appear anywhere a relation name can.
21. We can now find all customers of the SFU branch by writing

22. **select** *cname*
23.
24. **from** *all-customer*
25.
26. **where** *bname*=``*SFU*''

2.7 Domain Types in SQL:

1. The SQL-92 standard supports a variety of built-in domain types:
   - **char**(n) (or **character**(n)): fixed-length character string, with user-specified length.
   - **varchar**(n) (or **character varying**): variable-length character string, with user-specified maximum length.
   - **int** or **integer**: an integer (length is machine-dependent).
   - **smallint**: a small integer (length is machine-dependent).
   - **numeric**(*p, d*): a fixed-point number with user-specified precision, consists of *p* digits (plus a sign) and *d* of *p* digits are to the right of the decimal point. E.g., **numeric**(*3, 1*) allows 44.5 to be stored exactly but not 444.5.
   - **real** or **double precision**: floating-point or double-precision floating-point numbers, with machine-dependent precision.
   - **float**(n): floating-point, with user-specified precision of at least *n* digits.
   - **date**: a calendar date, containing four digit year, month, and day of the month.
   - **time**: the time of the day in hours, minutes, and seconds.
2. SQL-92 allows arithmetic and comparison operations on various numeric domains, including, **interval** and *cast* (*type coercion*) such as transforming between *smallint* and *int*. It considers strings with different length are compatible types as well.
3. SQL-92 allows **create domain** statement, e.g.,
4. **create domain** *person-name* **char**(20)

2.8 SQL Integrity Constraints:

Integrity Constraints are used to apply business rules for the database tables.

The constraints available in SQL are **Foreign Key**, **Not Null**, **Unique**, **Check**.
Constraints can be defined in two ways
1) The constraints can be specified immediately after the column definition. This is called column-level definition.
2) The constraints can be specified after all the columns are defined. This is called table-level definition.

1) SQL Primary key:

This constraint defines a column or combination of columns which uniquely identifies each row in the table.

Syntax to define a Primary key at column level:

```
column name datatype [CONSTRAINT constraint_name] PRIMARY KEY
```

Syntax to define a Primary key at table level:

```
[CONSTRAINT constraint_name] PRIMARY KEY
(column_name1,column_name2,..)
```

- **column_name1, column_name2** are the names of the columns which define the primary Key.
- The syntax within the bracket i.e. [CONSTRAINT constraint_name] is optional.

For Example: To create an employee table with Primary Key constraint, the query would be like.

Primary Key at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

or

```
CREATE TABLE employee
( id number(5) CONSTRAINT emp_id_pk PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

Primary Key at column level:

```
CREATE TABLE employee
( id number(5),
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
CONSTRAINT emp_id_pk PRIMARY KEY (id)
);
```

Primary Key at table level:

```
CREATE TABLE employee
( id number(5), NOT NULL,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
ALTER TABLE employee ADD CONSTRAINT PK_EMPLOYEE_ID PRIMARY
KEY (id)
);
```

2) SQL Foreign key or Referential Integrity :

This constraint identifies any column referencing the PRIMARY KEY in another table. It establishes a relationship between two columns in the same table or between different tables. For a column to be defined as a Foreign Key, it should be a defined as a Primary Key in the table which it is referring. One or more columns can be defined as Foreign key.

Syntax to define a Foreign key at column level:

```
[CONSTRAINT constraint_name] REFERENCES
Referenced_Table_name(column_name)
```

Syntax to define a Foreign key at table level:

```
[CONSTRAINT constraint_name] FOREIGN KEY(column_name) REFERENCES
referenced_table_name(column_name);
```

For Example:

1) Lets use the "product" table and "order_items".

Foreign Key at column level:

```
CREATE TABLE product
( product_id number(5) CONSTRAINT pd_id_pk PRIMARY KEY,
product_name char(20),
supplier_name char(20),
unit_price number(10)
);
CREATE TABLE order_items
( order_id number(5) CONSTRAINT od_id_pk PRIMARY KEY,
product_id number(5) CONSTRAINT pd_id_fk REFERENCES,
product(product_id),
product_name char(20),
supplier_name char(20),
unit_price number(10)
);
```

Foreign Key at table level:

```
CREATE TABLE order_items
( order_id number(5) ,
product_id number(5),
product_name char(20),
supplier_name char(20),
unit_price number(10)
CONSTRAINT od_id_pk PRIMARY KEY(order_id),
CONSTRAINT pd_id_fk FOREIGN KEY(product_id) REFERENCES
product(product_id)
);
```

2) If the employee table has a 'mgr_id' i.e, manager id as a foreign key which references primary key 'id' within the same table, the query would be like,

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
mgr_id number(5) REFERENCES employee(id),
salary number(10),
location char(10)
);
```

3) SQL Not Null Constraint :

This constraint ensures all rows in the table contain a definite value for the column which is specified as not null. Which means a null value is not allowed.

Syntax to define a Not Null constraint:

```
[CONSTRAINT constraint name] NOT NULL
```

For Example: To create a employee table with Null value, the query would be like

```
CREATE TABLE employee
( id number(5),
name char(20) CONSTRAINT nm_nn NOT NULL,
dept char(10),
age number(2),
salary number(10),
location char(10)
);
```

4) SQL Unique Key:

This constraint ensures that a column or a group of columns in each row have a distinct value. A column(s) can have a null value but the values cannot be duplicated.

Syntax to define a Unique key at column level:

```
[CONSTRAINT constraint_name] UNIQUE
```

Syntax to define a Unique key at table level:

```
[CONSTRAINT constraint_name] UNIQUE(column_name)
```

For Example: To create an employee table with Unique key, the query would be like,
Unique Key at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
```

```
location char(10) UNIQUE
);
```

or

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10) CONSTRAINT loc_un UNIQUE
);
```

Unique Key at table level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
salary number(10),
location char(10),
CONSTRAINT loc_un UNIQUE(location)
);
```

5) SQL Check Constraint :

This constraint defines a business rule on a column. All the rows must satisfy this rule. The constraint can be applied for a single column or a group of columns.

Syntax to define a Check constraint:

```
[CONSTRAINT constraint_name] CHECK (condition)
```

For Example: In the employee table to select the gender of a person, the query would be like

Check Constraint at column level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
gender char(1) CHECK (gender in ('M','F')),
salary number(10),
location char(10)
);
```

Check Constraint at table level:

```
CREATE TABLE employee
( id number(5) PRIMARY KEY,
name char(20),
dept char(10),
age number(2),
gender char(1),
salary number(10),
location char(10),
CONSTRAINT gender_ck CHECK (gender in ('M','F'))
);
```

2.9 Authorization in Sql:

- Authorization is finding out if the person, once identified, is permitted to have the resource.
- Authorization explains that what you can do and is handled through the DBMS unless external security procedures are available.
- Database management system allows DBA to give different access rights to the users as per their requirements.
- Basic Authorization we can use any one form or combination of the following basic forms of authorizations

  i. Resource authorization:-Authorization to access any system resource. e.g. sharing of database, printer etc.

  ii. Alternation Authorization:- Authorization to add attributes or delete attributes from relations

  iii. Drop Authorization:-Authorization to drop a relation.

- **Granting of privileges:**

  i. A system privilege is the right to perform a particular action,or to perform an action on any schema objects of a particular type.

  ii. An authorized user may pass on this authorization to other users.This process is called as ganting of privileges.

  iii. Syntax:

```
GRANT <privilege list>
ON<relation name or view name>
TO<user/role list>
```

  iv. Example:

  The following grant statement grants user U1,U2 and U3 the select privilege on Emp_Salary relation:

```
GRANT  select
ON Emp_Salary
TO U1,U2 and U3.
```

- **Revoking of privileges:**

  i. We can reject the privileges given to particular user with help of revoke statement.

  ii. To revoke an authorization, we use the revoke statement.

  iii. Syntax:

```
REVOKE <privilege list>
ON<relation name or view name>
```

```
FROM <user/role list>[restrict/cascade]
```

iv. Example:

The revocation of privileges from user or role may cause other user or roles also have to loose that privileges.This behavior is called cascading of the revoke.

```
Revoke select
ON  Emp_Salary
FROM U1,U2,U3.
```

- Some other types of Privileges:

  i. **Reference privileges:**

  SQL permits a user to declare foreign keys while creating relations.

  Example: Allow user U1 to create relation that references key 'Eid' of Emp_Salary relation.

```
GRANT REFERENCES(Eid)
ON Emp_Salary
TO U1
```

  ii. **Execute privileges:**

  This privileges authorizes a user to execute a function or procedure.

  Thus,only user who has execute privilege on a function Create_Acc() can call function.

```
GRANT EXECUTE
ON Create_Acc
TO U1.
```

2.10 Embedded SQL:

1. SQL provides a powerful declarative query language. However, access to a database from a general-purpose programming language is required because,
   - SQL is not as powerful as a general-purpose programming language. There are queries that cannot be expressed in SQL, but can be programmed in C, Fortran, Pascal, Cobol, etc.
   - Nondeclarative actions -- such as printing a report, interacting with a user, or sending the result to a GUI -- cannot be done from within SQL.
2. The SQL standard defines embedding of SQL as *embedded SQL* and the language in which SQL queries are embedded is referred as *host language*.
3. The result of the query is made available to the program one tuple (record) at a time.

4. To identify embedded SQL requests to the preprocessor, we use EXEC SQL statement:
5. EXEC SQL embedded SQL statement END-EXEC
6.

Note: A semi-colon is used instead of END-EXEC when SQL is embedded in C or Pascal.

7. Embedded SQL statements: **declare cursor**, **open**, and **fetch** statements.
8. EXEC SQL
9.
10. **declare** *c* **cursor for**
11.
12. **select** *cname, ccity*
13.
14. **from** *deposit, customer*
15.
16. **where** *deposit.cname* = *customer.cname*
17. **and** *deposit.balance > :amount*
18.
19. END-EXEC
20.

where *amount* is a host-language variable.

EXEC SQL **open** *c* END-EXEC

This statement causes the DB system to execute the query and to save the results within a temporary relation.

A series of **fetch** statement are executed to make tuples of the results available to the program.

EXEC SQL **fetch** *c* **into** *:cn, :cc* END-EXEC

The program can then manipulate the variable *cn* and *cc* using the features of the host programming language.

A single **fetch** request returns only one tuple. We need to use a **while** loop (or equivalent) to process each tuple of the result until no further tuples (when a variable in the SQLCA is set).

We need to use **close** statement to tell the DB system to delete the temporary relation that held the result of the query.

EXEC SQL **close** *c* END-EXEC

21. Embedded SQL can execute any valid **update**, **insert**, or **delete** statements.
22. *Dynamic* SQL component allows programs to construct and submit SQL queries at run time

2.11 DYNAMIC SQL:

Dynamic SQL is a programming technique that allows you to construct SQL statements dynamically at runtime. It allows you to create more general purpose and flexible SQL statement because the full text of the SQL statements may be unknown at compilation. For example, you can use the dynamic SQL to create a stored procedure that queries data against a table whose name is not known until runtime.

Creating a dynamic SQL is simple, you just need to make it a string as follows:

```
'SELECT * FROM production.products';
```

To execute a dynamic SQL statement, you call the stored procedure sp_executesql as shown in the following statement:

```
EXEC sp_executesql N'SELECT * FROM production.products';
```

Because the sp_executesql accepts the dynamic SQL as a Unicode string, you need to prefix it with an N.

Though this dynamic SQL is not very useful, it illustrates a dynamic SQL very well.

Using dynamic SQL to query from any table example

First, declare two variables, @table for holding the name of the table from which you want to query and @sql for holding the dynamic SQL.

```
DECLARE
   @table NVARCHAR(128),
   @sql NVARCHAR(MAX);
```

Second, set the value of the @table variable to production.products.

```
SET @table = N'production.products';
```

Third, construct the dynamic SQL by concatenating the SELECT statement with the table name parameter:

```
SET @sql = N'SELECT * FROM ' + @table;
```

Fourth, call the sp_executesql stored procedure by passing the @sql parameter.

```
EXEC sp_executesql @sql;
```

Putting it all together:

```
DECLARE
   @table NVARCHAR(128),
   @sql NVARCHAR(MAX);

SET @table = N'production.products';

SET @sql = N'SELECT * FROM ' + @table;

EXEC sp_executesql @sql;
```

The code block above produces the exact result set as the following statement:

```
SELECT * FROM production.products;
```

To query data from another table, you change the value of the @table variable. However, it's more practical if we wrap the above T-SQL block in a stored procedure.

SQL Server dynamic SQL and stored procedures

This stored procedure accepts any table and returns the result set from a specified table by using the dynamic SQL:

```
CREATE PROC usp_query (
   @table NVARCHAR(128)
)
AS
BEGIN

   DECLARE @sql NVARCHAR(MAX);
   -- construct SQL
   SET @sql = N'SELECT * FROM ' + @table;
   -- execute the SQL
   EXEC sp_executesql @sql;

END;
```

The following statement calls the usp_query stored procedure to return all rows from the production.brands table:

```
EXEC usp_query 'production.brands';
```

This stored procedure returns the top 10 rows from a table by the values of a specified column:

```
CREATE OR ALTER PROC usp_query_topn(
   @table NVARCHAR(128),
   @topN INT,
   @byColumn NVARCHAR(128)
)
AS
BEGIN
   DECLARE
      @sql NVARCHAR(MAX),
      @topNStr NVARCHAR(MAX);

   SET @topNStr  = CAST(@topN as nvarchar(max));

   -- construct SQL
   SET @sql = N'SELECT TOP ' +  @topNStr  +
            ' * FROM ' + @table +
               ' ORDER BY ' + @byColumn + ' DESC';
   -- execute the SQL
```

```
    EXEC sp_executesql @sql;

END;
```

For example, you can get the top 10 most expensive products from the production.products table:

```
EXEC usp_query_topn
        'production.products',
        10,
        'list_price';
```

This statement returns the top 10 products with the highest quantity in stock:

```
EXEC usp_query_topn
        'production.tocks',
        10,
        'quantity';
```

SQL Server Dynamic SQL and SQL Injection

Let's create a new table named sales.tests for the demonstration:

```
CREATE TABLE sales.tests(id INT);
```

This statement returns all rows from the production.brands table:

```
EXEC usp_query 'production.brands';
```

But it does not prevent users from passing the table name as follows:

```
EXEC usp_query 'production.brands;DROP TABLE sales.tests';
```

This technique is called SQL injection. Once the statement is executed, the sales.tests table is dropped, because the stored procedure usp_query executes both statements:

```
SELECT * FROM production.brands;DROP TABLE sales.tests
```

2.12 PL/SQL:

What is PL/SQL?

PL/SQL stands for a procedural language extension of the structured query language (SQL). PL/SQL (Procedural language/structured query language) is a program to executes in Oracle database.

Furthermore, PL/SQL specially designed for database oriented activities. Also, Oracle PL/SQL allows you to perform data manipulation operation that is safe and flexible.

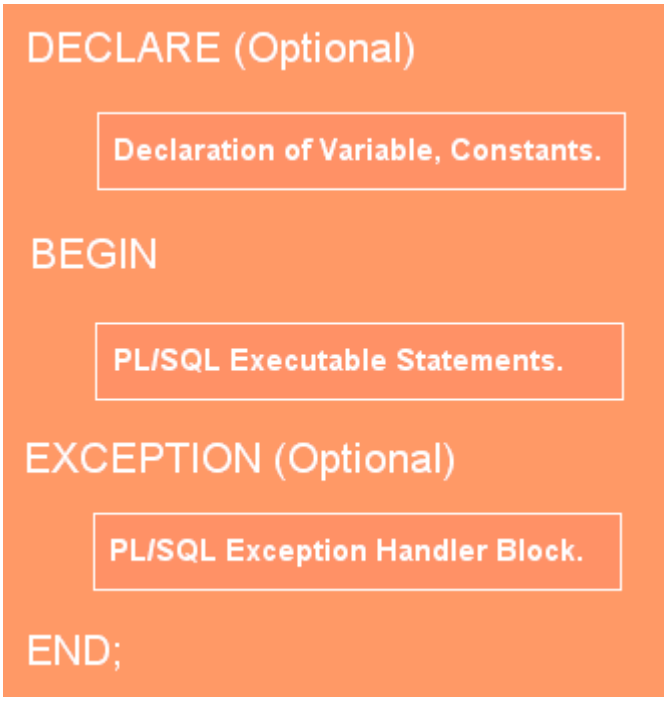
First of all, PL/SQL comes after releasing Oracle database (version 7). The basic structure block of PL/SQL built using DECLARE, BEGIN, EXCEPTION (optional), END keywords. These keyword blocks defined as a declaration block, executable block, exception handling block, and last end block (for end of the structure).

PL/SQL offers the following advantages:

- Reduces network traffic This one is **great advantages** of PL/SQL. Because PL/SQL nature is **entire block** of SQL statements execute into **oracle engine** all at once so it's main benefit is **reducing** the **network traffic**.
- Procedural language support PL/SQL is a **development tools** not only for data manipulation futures but also provide the conditional checking, looping or branching operations same as like **other programming language**.
- Error handling PL/SQL is dealing with **error handling**, It's permits the smart way **handling the errors** and giving **user friendly** error messages, when the errors are encountered.
- Declare variable PL/SQL gives you control to **declare variables** and access them **within the block**. The declared variables can be used at the time of **query processing**.
- Intermediate Calculation Calculations in PL/SQL done quickly and efficiently without using Oracle engines. This **improves** the transaction performance.
- Portable application Applications are written in PL/SQL are **portable** in any **Operating system**. PL/SQL applications are **independence program** to run any computer.

Following are few more PL/SQL advantages:

- PL/SQL is a very secure **functionality tool** for manipulating, controlling, validating, and restricting unauthorized access data from the SQL database.
- Using PL/SQL we can improve **application performance**. It also allows to **deal** with **errors** so we can provide **user friendly error messages**.
- PL/SQL have a **great functionality** to display multiple records from the multiple tables at the same time.
- PL/SQL is **capable** to **send** entire block of statements and execute it in the **Oracle engine** at once.
- What is PL/SQL block? PL/SQL block structure divided into three logical blocks. First, BEGIN block and END; keywords are compulsory. However, the other two blocks DECLARE and EXCEPTION are optional block. Technically, END; is not a block, it is only keyword to end of PL/SQL program.
- PL/SQL code is not executed in single line format like SQL. It is always executed by a grouping of code into a single segment called blocks.
- PL/SQL block structure follows the divide-and-conquer approach to solve the problem stepwise.



- **PL/SQL block Structure**
- PL/SQL Block Structure
- DECLARE
- Variables and constants are declared, initialized within this section.
- Variables and Constants: In this block, declare and initialize variables (and constants). You must have to declare variables and constants in the declarative block before referencing them in a procedural statement.
- Declare Variables and Assigning values: You can define a variable name, data type of a variable, and its size. Date type can be CHAR, VARCHAR2, DATE, NUMBER, INT, or any other.

```
DECLARE       -- DECLARE block, declare and initialize values
   designation VARCHAR2(30);
   eno number(5) := 5;
   id BOOLEAN;
   inter INTERVAL YEAR(2) TO MONTH;
BEGIN         -- BEGIN block, also assign values
   designation := UPPER('Web Developer');
   id := TRUE;
   inter := INTERVAL '45' YEAR;
END;
```

- /

- Declare Constants and Assigning values: Constants are declared the same as a variable, but you have to add the CONSTANT keyword before defining the data type. Once you define, a constant value, you can't change the value.

- designation CONSTANT VARCHAR2(30) := 'Web Developer';

BEGIN

BEGIN block is a procedural statement block which will implement the actual programming logic. This section contains conditional statements (if...else), looping statements (for, while) and Branching Statements (goto), etc.

EXCEPTION

PL/SQL easily detects a user-defined or predefined error condition. PL/SQL is famous for smartly handling errors by giving suitable user-friendly messages. Errors can be rise due to the wrong syntax, bad logical, or not passing validation rules.

You can also define exception in your declarative block, and later you can execute it by RAISE statement.

```
DECLARE
   check_exist EXCEPTION; -- declare exception type
   ...
BEGIN
   ....
   RAISE check_exist;     -- raise exception
   ....
EXCEPTION
  WHEN check_exist THEN   -- execute raise exception
  .....
END;
/
```

Note

1. BEGIN block, and END; keyword are compulsory of any PL/SQL program.
2. Whereas, the DECLARE and EXCEPTION block are optional.

PL/SQL Data Types

In this lesson, you will learn PL/SQL Data Types. PL/SQL variables and constants must have a valid data type. Which specifies storage format. There are six built-in PL/SQL data types

1. Scalar data types - Scalar data types haven't internal components.
2. Composite data types - Composite data types have internal components to manipulate data easily.
3. Reference data types - This data types work like a pointer to hold some value.
4. LOB data types - Stores large objects such as images, graphics, video.
5. Unknown Column types - Identify columns when not know the type of column.
6. User Define data types - Define your own data type that inherited from predefined base data type.

Scalar types

Scalar data type haven't internal components. It is like a linear data type. Scales data type divides into four different types character, numeric, boolean or date/time type.

Numeric Data types

Following are numeric data types in PL/SQL:

<table>
<tr><th>Datatype</th><th>Description, Storage(Maximum)</th></tr>
<tr><td>NUMBER(p,s)</td><td>NUMBER data type used to store numeric data. It contains letters, numbers, and special characters. Storage Range: Precision range(p): 1 to 38 and Scale range(s) : -84 to 127 NUMBER Subtypes: This sub type defines different types storage range.
<table>
<tr><th>Sub Datatype</th><th>Maximum Precision</th><th>Description</th></tr>
<tr><td>INTEGER</td><td>38 digits</td><td rowspan="7">These data types are used to store fixed decimal points. You can use based on your requirements.</td></tr>
<tr><td>INT</td><td>38 digits</td></tr>
<tr><td>SMALLINT</td><td>38 digits</td></tr>
<tr><td>DEC</td><td>38 digits</td></tr>
<tr><td>DECIMAL</td><td>38 digits</td></tr>
<tr><td>NUMERIC</td><td>38 digits</td></tr>
<tr><td>REAL</td><td>63 binary digits</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>DOUBLE PRECISION</td><td>126 binary digits</td></tr>
<tr><td>FLOAT</td><td>126 binary digits</td></tr>
</table>
</td></tr>
<tr><td>BINARY_INTEGER</td><td>BINARY_INTEGER data type store signed integer's value. Note: BINARY_INTEGER values require less storage space compare of NUMBER data type values. Storage Range: from -2147483647 to 2147483647 BINARY_INTEGER Subtypes: This sub type defines constraint to store a value.
<table>
<tr><th>Sub Datatype</th><th>Description</th></tr>
<tr><td>NATURAL</td><td rowspan="2">NATURAL/POSITIVE data type prevent to store negative value, allow only positive values.</td></tr>
<tr><td>POSITIVE</td></tr>
<tr><td>NATURALN</td><td rowspan="2">NATURALN/POSITIVEN data type prevent to assign a NULL value.</td></tr>
<tr><td>POSITIVEN</td></tr>
<tr><td>SIGNTYPE</td><td>SIGNTYPE allow only -1, 0, and 1 values.</td></tr>
</table>
</td></tr>
<tr><td>PLS_INTEGER</td><td>PLS_INTEGER data type used to store signed integers data. Note PLS_INTEGER data type value require less storage space compare of NUMBER data type value. Storage Range: from -2147483647 to 2147483647 Performance: PLS_INTEGER data type gives you better performance on your data. PLS_INTEGER perform arithmetic operation fast than NUMBER/BINARY_INTEGER data type.</td></tr>
</table>

Example

```
DECLARE
  SUBTYPE message IS varchar2(25);
  SUBTYPE age IS INTEGER(2,0);
  description message;
```

```
  ages age;
BEGIN
  description := 'Web Developer';
  ages := 22;
  dbms_output.put_line('I am ' || description || ' and I am ' || ages || ' years Old.');
END;
/
```

Result

I am Web Developer and I am 22 years Old.
PL/SQL procedure successfully completed.

PL/SQL Variables

PL/SQL variable is a relevant name which provide a storage capability during the execution of code. Every PL/SQL variables has a specific data type which defines the size and physical location of the variable memory. Moreover you can specifies initial value of the variable at the time of variable declaration.

Syntax for declaring variable

The general syntax to declaring variable:

```
variable_name Datatype[Size] [NOT NULL] := [ value ];
```

Explanation

- variable_name is the predefined name of the variable.
- Data type is a valid PL/SQL data type.
- Size is an optional specification of data type size to hold the maximum size value.
- NOT NULL is an optional specification of the variable value can't be accept NULL.
- value is also an optional specification, where you can initialize the initial value of variable.
- Each variable declaration is terminated by a semicolon.

Example of Variable Declaration

In this example variable defining employee number (eno) is NOT NULL(compulsory), employee name and initializing initial value to a variable,

Example

```
DECLARE
  eno number(5) NOT NULL := 2 -- NOT NULL (value can't be blank), Assign initial value
  ename varchar2(15) := 'Branson Devs'; -- intialize value at the time of declaration
BEGIN
```

```
   dbms_output.put_line('Declared Value:');
   dbms_output.put_line(' Employee Number: ' || eno || ' Employee Name: ' || ename);
END;
/
```

Backward slash '/' indicates to execute the above PL/SQL Program.

Example Result

Declared Value:
Employee Number: 2 Employee Name: Branson Devs

Variables Scope

PL/SQL variable scope is identified the region range which you can reference the variable. PL/SQL have two type scopes local scope and global scope,

Local variables - Variables declared in inner block and can't be referenced by the outside blocks.
Global variables - Where as variables declared in a outer block and can be referencing by itself in inner blocks.

Variable Scope Example

In this example declaration two variables (num1 and num2) are in the outer block (Global variable) and another one third variable declared (num_addition) into the inner block (local variable). Variable 'num_addition' declared inner block so can't access in the outer block. But num1 and num2 can be accessed anywhere in the block.

Example

```
DECLARE
   num1 number := 10;
   num2 number := 20;
BEGIN
   DECLARE
      num_addition number;
   BEGIN
      num_addition := num1 + num2;
      dbms_output.put_line('Addition is: ' || num_addition);
   END;  -- End of access num_addition variable
END;
/
```

Example Result

Addition is: 30

Variable Scope Identifier (OUTER keyword)

This example is also showing a difference between inner block and outer block variable scope. You can use OUTER keyword to access outer block variable inside the inner block. It's called global qualified name space.

Example

```
DECLARE
   num number := 10;
BEGIN
   DECLARE
      num number := 10;
   BEGIN
      IF num = OUTER.num THEN
        DBMS_OUTPUT.PUT_LINE('Both are same value');
      ELSE
        DBMS_OUTPUT.PUT_LINE('Different value');
      END IF;
   END;  -- End of scope to access num variable
END;
/
```

Example Result

Both are same value

PL/SQL Constants

You can declare PL/SQL constants along with the value and can not change them throughout the program block. The constants declaration functionality is available in almost all programming languages.

Constant Syntax

The general syntax for declaring a constant variable is:

```
Constant_name CONSTANT Datatype[Size] := Value;
```

Explanation

- Constant_name is a predefined name of the constant (similar to a variable name).
- CONSTANT is a reserved keyword.
- Data type is a valid PL/SQL data type.
- Size is an optional specification of data type. It holds maximum capacity value for the particular variable.
- Value must be assigned to a constant when it is declared. You can not assign or change it later.
- Each constant declaration is terminated by a semicolon.

Constant Example

In this example, we will store the employee number which is NOT NULL (compulsory), employee Name and employee department which is constant,

Example

```
DECLARE
   eno number(5) NOT NULL := 2
   ename varchar2(15) := 'Branson Devs';
   edept CONSTANT varchar2(15) := 'Web Developer';
BEGIN
   dbms_output.put_line('Declared Value:');
   dbms_output.put_line(' Employee number: ' || eno || ' Employee Name: ' || ename);
   dbms_output.put_line('Constant Declared:');
   dbms_output.put_line(' Employee Department: ' || edept);
END;
/
```

Backward slash '/' is indicated to execute the above PL/SQL Block Program.

Example Result

```
Declared Value:
Employee number: 2 Employee Name: Branson Devs
Constant Declared:
Employee Department: Web Developer

PL/SQL procedure successfully operation.
```

Variable/Constant Declarations Example

In this example, we will store the pi which is constant real number, radius and area which are real number,

Example

```
DECLARE
```

```
   pi CONSTANT REAL := 3.14159;
   radius REAL := 3;
   area REAL := (pi * radius**2);
BEGIN
   dbms_output.put_line(' PI: ' || pi || ' Radius: ' || radius);
   dbms_output.put_line(' Area: ' || area);
END;
/
```

Example Result

PI: 3.14159 Radius: 1
Area: 28.27431

PL/SQL SET Serveroutput ON

Whenever you start Oracle SQL (PL/SQL) at that time you must have to write the "SET Serveroutput ON" command.

PL/SQL program execution into Oracle engine so we always required to get serveroutput result and display into the screen otherwise result can't be display.

Set PL/SQL serveroutput Result ON

```
SQL> set serveroutput on
```

Set serveroutput on Example

In this example showing you how to turn on serveroutput result, Here first line turn on serveroutput. After define variables and constants to print defined variable value using dbms_output.put_line command.

Example

```
SQL> set serveroutput on
SQL> DECLARE
   eno number(5) NOT NULL := 2
   ename varchar2(15) := 'Branson Devs';
   edept CONSTANT varchar2(15) := 'Web Developer';
BEGIN
   dbms_output.put_line('Declared Value:');
   dbms_output.put_line(' Employee Number: ' || eno || ' Employee Name: ' || ename);
   dbms_output.put_line('Constant Declared:');
```

```
   dbms_output.put_line(' Employee Department: ' || edept);
END;
/
```

Result display only if you execute *"set serveroutput on"* command.

Example Result

```
Declared Value:
Employee number: 2 Employee Name: Branson Devs
Constant Declared:
Employee Department: Web Developer
```

PL/SQL Comment>

PL/SQL Comments you can write single line comments or either multiple line comments,

- Multi-line comments and
- Single line comments

Multi-line comments are delimited with /*..COMMENT TEXT..*/ and Single-line comments starts with two dashes --.

PL/SQL Comments can begin in any column on any line. If you are embedding comments in SQL that will be embedded in PL/SQL you need to be careful for writing a column.

Comment Syntax

```
SQL>-- Single Line Comment
SQL>/*  Multiline Comment line 1
      Multiline Comment line2    */
```

Comment Example

This example is nothing any new same as last lesson only new for showing a how to write single line comment or multi line comment in PL/SQL program,

Example

```
SQL> set serveroutput on
SQL> DECLARE

   eno number(5) NOT NULL := 2; --assign value into eno variable
```

```
   ename varchar2(15) := 'Branson Devs';

   /* Constant value is fixed for edept value is "Web Deloper"
   is fixed all program not required declare all times. */
   edept CONSTANT varchar2(15) := 'Web Developer';
BEGIN
   dbms_output.put_line('Declared Value:');
   dbms_output.put_line(' Employee Number: ' || eno || ' Employee Name: ' || ename);
   dbms_output.put_line('Constant Declared:');
   dbms_output.put_line(' Employee Department: ' || edept);
END;
/
```

Example Result :

```
Declared Value:
Employee number: 2 Employee Name: Branson Devs
Constant Declared:
Employee Department: Web Developer
```

PL/SQL IF THEN ELSE Statements (Conditional Control)

PL/SQL IF THEN ELSE conditional control statements. PL/SQL Conditional Control two type: IF THEN ELSE statement and CASE statement,

PL/SQL IF statement check condition and transfer the execution flow on that matched block depending on a condition. IF statement execute or skip a sequence of one or more statements. PL/SQL IF statement four different type,

1. IF THEN Statement
2. IF THEN ELSE Statement
3. IF THEN ELSIF Statement
4. Nested IF THEN ELSE Statement

IF THEN Statement

IF THEN Statement write in following syntax format:

```
IF ( condition ) THEN
   statement
END IF;
```

Example

We declare one number with initialize 14 value is equal of condition value, Comparing 2 values by using IF THEN statement,

```
DECLARE
  no INTEGER(2) := 14;
BEGIN
  IF ( no = 14 ) THEN
    DBMS_OUTPUT.PUT_LINE('condition true');
  END IF;
END;
/
```

Result

```
condition                                                    true

PL/SQL procedure successfully completed.
```

IF THEN ELSE Statement

IF THEN ELSE Statement write in following syntax format:

```
IF ( condition ) THEN
   statement;
ELSE
   statement;
END IF;
```

Example

Same as above example if condition not true then else part will execute.

```
DECLARE
  no INTEGER(2) := 14;
BEGIN
  IF ( no = 11 ) THEN
    DBMS_OUTPUT.PUT_LINE(no || ' is same');
  ELSE
    DBMS_OUTPUT.PUT_LINE(no || ' is not same');
  END IF;
END;
/
```

Result

14 is not same

PL/SQL procedure successfully completed.

IF THEN ELSIF Statement

IF THEN ELSIF Statement write in following syntax format:

```
IF ( condition-1 ) THEN
   statement-1;
ELSIF ( condition-2 ) THEN
   statement-2;
ELSIF ( condition-3 ) THEN
   statement-3;
ELSE
   statement;
END IF;
```

Above syntax same as below syntax both are logically same

```
IF ( condition-1 ) THEN
   statement-1;
ELSE
   IF ( condition-2 ) THEN
      statement-2;
   ELSE
      IF ( condition-3 ) THEN
         statements-3;
      END IF;
   END IF;
END IF;
```

Example

Here one student result example for archiving grade.

```
DECLARE
  result CHAR(20) := 'second';
BEGIN
  IF ( result = 'distinction' ) THEN
```

```
    DBMS_OUTPUT.PUT_LINE('First Class with Distinction');
  ELSIF ( result = 'first' ) THEN
    DBMS_OUTPUT.PUT_LINE('First Class');
  ELSIF ( result = 'second' ) THEN
    DBMS_OUTPUT.PUT_LINE('Second Class');
  ELSIF ( result = 'third' ) THEN
    DBMS_OUTPUT.PUT_LINE('Third Class');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Fail');
  END IF;
END;
/
```

Result

Second Class

PL/SQL procedure successfully completed.

Nested IF THEN ELSE Statement

Logically IF THEN ELSIF Statement and Nested IF THEN ELSE Statement both are same. Nested IF THEN ELSE Statement write in following syntax format:

```
IF ( condition-1 ) THEN
   statement-1;
ELSE
   IF ( condition-2 ) THEN
      statement-2;
   ELSE
      IF ( condition-3 ) THEN
         statements-3;
      END IF;
   END IF;
END IF;
```

Example

Here check condition students gender male, if not male then finding the result using nested IF THEN ELSE statement.

```
DECLARE
```

```
   gender CHAR(20) := 'female';
   result CHAR(20) := 'second';
BEGIN
  IF ( gender = 'male' ) THEN
    DBMS_OUTPUT.PUT_LINE('Gender Male Record Skip!');
  ELSE
     IF ( result = 'distinction' ) THEN
        DBMS_OUTPUT.PUT_LINE('First Class with Distinction');
     ELSIF ( result = 'first' ) THEN
        DBMS_OUTPUT.PUT_LINE('First Class');
     ELSIF ( result = 'second' ) THEN
        DBMS_OUTPUT.PUT_LINE('Second Class');
     ELSIF ( result = 'third' ) THEN
        DBMS_OUTPUT.PUT_LINE('Third Class');
     ELSE
        DBMS_OUTPUT.PUT_LINE('Fail');
     END IF;
   END IF;
END;
/
```

Result

```
Second                                                    Class

PL/SQL procedure successfully completed.
```

PL/SQL Loop

PL/SQL Loop Basic Loop, FOR Loop, WHILE Loop repeat a number of block statements in your PL/SQL program. Loop use when we have a block of statements for required to repeatedly certain number of times. PL/SQL loop statements 3 different forms:

1. Basic LOOP
2. WHILE LOOP
3. FOR LOOP

Oracle recommended to write a label when use loop statement. It's benefit to improve readability. label is not compulsory for execute loop. compiler does not check to label defined or not. Define label before LOOP keyword and after END LOOP keyword.

Basic LOOP

Basic LOOP write in following syntax format:

```
[ label_name ] LOOP
  statement(s);
END LOOP [ label_name ];
```

Example

```
DECLARE
  no NUMBER := 5;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE ('Inside value:  no = ' || no);
      no := no -1;
      IF no = 0 THEN
         EXIT;
      END IF;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Outside loop end');
END;
/
```

Result

```
Inside value: no = 5
Inside value: no = 4
Inside value: no = 3
Inside value: no = 2
Inside value: no = 1
Outside loop end
```

WHILE LOOP

WHILE LOOP write in following syntax format:

```
[ label_name ] WHILE condition LOOP
  statement(s);
END LOOP [ label_name ];
```

Example

```
DECLARE
```

```
  no NUMBER := 0;
BEGIN
   WHILE no < 10 LOOP
      no := no + 1;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Sum :' || no);
END;
/
```

Result

```
Sum                                    :                                    10

PL/SQL procedure successfully completed.
```

FOR LOOP

FOR LOOP write in following syntax format:

```
[ label_name ] FOR current_value IN [ REVERSE ] lower_value..upper_value LOOP
  statement(s);
END LOOP [ label_name ];
```

Example

```
BEGIN
   FOR no IN 1 .. 5 LOOP
      DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
   END LOOP;
END;
/
```

Result

```
Iteration                              :                                    1
Iteration                              :                                    2
Iteration                              :                                    3
Iteration                              :                                    4
Iteration                              :                                    5

PL/SQL procedure successfully completed.
```

PL/SQL EXIT Statement

PL/SQL EXIT Statement (Sequential Control Statement) are control your iteration loop, PL/SQL EXIT statement to exit the loop.

1. EXIT Statement : This statement to exit the loop.
2. EXIT WHEN Statement : This statement to exit, when WHEN clauses condition true.

EXIT Statement

EXIT statement unconditionally exit the current loop iteration and transfer control to end of current loop. EXIT statement writing syntax,

Syntax

```
[ label_name ] LOOP
  statement(s);
  EXIT;
END LOOP [ label_name ];
```

Example

```
DECLARE
  no NUMBER := 5;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE ('Inside value:  no = ' || no);
      no := no -1;
      IF no = 0 THEN
         EXIT;
      END IF;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Outside loop end');    -- After EXIT control transfer this statement
END;
/
```

Result

```
Inside value: no = 5
Inside value: no = 4
Inside value: no = 3
Inside value: no = 2
Inside value: no = 1
```

Outside loop end

PL/SQL procedure successfully completed.

XIT WHEN Statement

EXIT WHEN statement unconditionally exit the current loop iteration when WHEN clause condition true. EXIT WHEN statement writing syntax,

Syntax

```
[ label_name ] LOOP
  statement(s);
  EXIT WHEN condition;
END LOOP [ label_name ];
```

Example

```
SQL>DECLARE
   i number;
BEGIN
   LOOP
      dbms_output.put_line('Hello');
      i:=i+1;
      EXIT WHEN i>5;
   END LOOP;
END;
/
```

Result

Hello
Hello
Hello
Hello

PL/SQL procedure successfully completed.

PL/SQL CONTINUE Statement

PL/SQL CONTINUE Statement (Sequential Control Statements) are control your iteration loop, PL/SQL CONTINUE Statement to skip the current iteration with in loop.

1. CONTINUE Statement : to skip the current iteration with in loop.

2. CONTINUE WHEN Statement : to skip the current iteration with in loop when WHEN clauses condition true.

CONTINUE Statement

CONTINUE Statement unconditionally skip the current loop iteration and next iteration iterate as normal, only skip matched condition.

Syntax

```
IF condition THEN
   CONTINUE;
END IF;
```

Example

```
DECLARE
  no NUMBER := 0;
BEGIN
   FOR no IN 1 .. 5 LOOP
      IF i = 4 THEN
        CONTINUE;
      END IF;
      DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
   END LOOP;
END;
/
```

Result

```
Iteration                 #                 1
Iteration                 #                 2
Iteration                 #                 3
Iteration                 #                 5

PL/SQL procedure successfully completed.
```

CONTINUE WHEN Statement

CONTINUE WHEN Statement unconditionally skip the current loop iteration when WHEN clauses condition true,

Syntax

```
CONTINUE WHEN condition;
   statement(s);
```

Example

```
DECLARE
  no NUMBER := 0;
BEGIN
   FOR no IN 1 .. 5 LOOP
      DBMS_OUTPUT.PUT_LINE('Iteration : ' || no);
      CONTINUE WHEN no = 4
         DBMS_OUTPUT.PUT_LINE('CONTINUE WHEN EXECUTE Iteration : ' || no);
   END LOOP;
END;
/
```

Result

```
Iteration                    #                    1
Iteration                    #                    2
Iteration                    #                    3
CONTINUE    WHEN    EXECUTE    Iteration    :    4
Iteration                    #                    5

PL/SQL procedure successfully completed.
```

PL/SQL GOTO Statement

PL/SQL GOTO Statements (Sequential Control Statements) are control your iteration loop. PL/SQL GOTO Statement transfers the program execution flow unconditionally.

GOTO Statement

GOTO statement unconditionally transfer program control. GOTO statement writing syntax,

Syntax

```
GOTO code_name
   ----------
   ----------
<<code_name>>
----------
----------
```

Example

```
SQL>BEGIN
```

```
FOR i IN 1..5 LOOP
   dbms_output.put_line(i);
   IF i=4 THEN
      GOTO label1;
   END IF;
END LOOP;
<<label1>>
DBMS_OUTPUT.PUT_LINE('Row Filled');
END;
/
```

Result

```
1
2
3
4
Row                                                    Filled
```

PL/SQL procedure successfully completed.

PL/SQL Case Statement

PL/SQL CASE statement comparing one by one sequencing conditions. CASE statement attempt to match expression that is specified in one or more WHEN condition. CASE statement are following two forms,

1. Simple CASE Statement
2. Searched CASE Statement

Simple CASE Statement

PL/SQL simple CASE statement evaluates selector and attempt to match one or more WHEN condition.

Syntax

```
CASE selector
   WHEN value-1
      THEN statement-1;
   WHEN value-2
      THEN statement-2;
   ELSE
```

```
    statement-3;
END CASE
```

Example

```
SQL>DECLARE
   a number := 7;
BEGIN
   CASE a
      WHEN 1 THEN
         DBMS_OUTPUT.PUT_LINE('value 1');
      WHEN 2 THEN
         DBMS_OUTPUT.PUT_LINE('value 2');
      WHEN 3 THEN
         DBMS_OUTPUT.PUT_LINE('value 3');
      ELSE
         DBMS_OUTPUT.PUT_LINE('no matching CASE found');
   END CASE;
END;
/
```

Result

```
no matching CASE found

PL/SQL procedure successfully operation.
```

Searched CASE Statement

PL/SQL searched CASE statement has not selector and attempt to match one or more WHEN clauses condition.

Syntax

Syntax

```
CASE
   WHEN condition-1 THEN
      statement-1;
   WHEN condition-2 THEN
      statement-2;
   ELSE
```

```
        statement-3;
END CASE;
```

Example

```
SQL>DECLARE
   a number := 3;
BEGIN
   CASE
      WHEN a = 1 THEN
         DBMS_OUTPUT.PUT_LINE('value 1');
      WHEN a = 2 THEN
         DBMS_OUTPUT.PUT_LINE('value 2');
      WHEN a = 3 THEN
         DBMS_OUTPUT.PUT_LINE('value 3');
      ELSE
         DBMS_OUTPUT.PUT_LINE('no matching CASE found');
   END CASE;
END;
/
```

Result

value 3

PL/SQL Exceptions - Built in Exceptions

PL/SQL exceptions are predefined and raised automatically into oracle engine when any error occur during a program.

Each and every error has defined a unique number and message. When warning/error occur in program it's called an exception to contains information about the error.

In PL/SQL built in exceptions or you make user define exception. Examples of built-in type (internally) defined exceptions division by zero, out of memory. Some common built-in exceptions have predefined names such as ZERO_DIVIDE and STORAGE_ERROR.

Normally when exception is fire, execution stops and control transfers to the exception-handling part of your PL/SQL block. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements, which are also raise predefined exceptions.

PL/SQL exceptions consist following three,

1. Exception Type
2. Error Code
3. Error Message

4. Syntax

```
5. DECLARE
6.     declaration statement(s);
7. BEGIN
8.     statement(s);
9. EXCEPTION
10.    WHEN built-in_exception_name_1 THEN
11.        User defined statement (action) will be taken;
12.    WHEN built-in_exception_name_2 THEN
13.        User defined statement (action) will be taken;
14. END;
```

15. Example

```
16. builtin_exp.sql
17. SQL>edit buitin_exp
18. DECLARE
19.    temp enum%rowtype;
20. BEGIN
21.    SELECT * INTO temp FROM enum
22.        WHERE eno=3;
23. EXCEPTION
24.    WHEN no_data_found THEN
25.        dbms_output.put_line("Table haven't data");
26. END;
27. /
```

28. Result
29. SQL>@buitin_exp

    Table haven't data

    PL/SQL procedure successfully operation.

PL/SQL built in exceptions

Following are built in type exception,

Exception	Error Code	Description
ACCESS_INTO_NULL	ORA-06530	Exception raised when assign uninitialized (NULL) object.
CASE_NOT_FOUND	ORA-06592	Exception raised when no any choice case found in CASE statement as well as no ELSE clause in CASE statement.

CURSOR_ALREADY_OPEN	ORA-06511	Exception raised when you open a cursor that is already opened.
DUP_VAL_ON_INDEX	ORA-00001	Exception raised when you store duplicate value in unique constraint column.
INVALID_CURSOR	ORA-01001	Exception raised when you perform operation on cursor and cursor is not really opened.
INVALID_NUMBER	ORA-01722	Exception raised when you try to explicitly conversion from string to a number fail.
LOGIN_DENIED	ORA-01017	Exception raised when log in into oracle with wrong username or password.
NO_DATA_FOUND	ORA-01403	Exception raised when SELECT ... INTO statement doesn't fetch any row from a database table.
NOT_LOGGED_ON	ORA-01012	Exception raised when your program try to get data from database and actually user not connected to Oracle.
PROGRAM_ERROR	ORA-06501	Exception raised when your program is error prone (internal error).
STORAGE_ERROR	ORA-06500	Exception raised when PL/SQL program runs out of memory or may be memory is dumped/corrupted.
SYS_INVALID_ROWID	ORA-01410	Exception raised when you try to explicitly conversion from string character string to a universal rowid (uid) fail.
TIMEOUT_ON_RESOURCE	ORA-00051	Exception raised when database is locked or ORACLE is waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Exception raised when SELECT ... INTO statement returns more than one row.

VALUE_ERROR	ORA-06502	Exception raised when arithmetic, conversion, defined size constraint error occurs.
ZERO_DIVIDE	ORA-01476	Exception raised when you program try to attempt divide by zero number.

PL/SQL User Defined Exception

PL/SQL user defined exception to make your own exception. PL/SQL give you control to make your own exception base on oracle rules. User define exception must be declare yourself and RAISE statement to raise explicitly.

How to Define Exception

1. Declare exception

   You must have to declare user define exception name in DECLARE block.

   ```
   user_define_exception_name EXCEPTION;
   ```

   Exception and Variable both are same way declaring but exception use for store error condition not a storage item.

2. RAISE exception

   RAISE statement to raised defined exception name and control transfer to a EXCEPTION block.

   ```
   RAISE user_define_exception_name;
   ```

3. Implement exception condition

   In PL/SQL EXCEPTION block add WHEN condition to implement user define action.

   ```
   WHEN user_define_exception_name THEN
     User defined statement (action) will be taken;
   ```

Syntax

Check this user defined exception syntax,

```
DECLARE
  user_define_exception_name EXCEPTION;
```

```
BEGIN
   statement(s);
   IF condition THEN
      RAISE user_define_exception_name;
   END IF;
EXCEPTION
   WHEN user_define_exception_name THEN
      User defined statement (action) will be taken;
END;
```

Example

user_exp.sql

```
SQL>edit user_exp
DECLARE
   myex EXCEPTION;
   i NUMBER;
BEGIN
   FOR i IN (SELECT * FROM enum) LOOP
      IF i.eno = 3 THEN
         RAISE myex;
      END IF;
   END LOOP;
EXCEPTION
   WHEN myex THEN
      dbms_output.put.line('Employee number already exist in enum table.');
END;
/
```

Result

```
SQL>@user_exp
Employee number already exist in enum table.

PL/SQL procedure successfully operation.
```

PL/SQL PRAGMA EXCEPTION_INIT

PL/SQL user named exception. you can define your own error message and error number using Pragma EXCEPTION_INIT or RAISE_APPLICATION_ERROR function.

PL/SQL pragma EXCEPTION_INIT

pragma EXCEPTION_INIT: Pragma is a keyword directive to execute proceed at compile time. pragma EXCEPTION_INIT function take this two argument,

1. exception_name
2. error_number

You can define pragrma EXCEPTION_INIT in DECLARE BLOCK on your program.

```
PRAGMA EXCEPTION_INIT(exception_name, -error_number);
```

exception_name and *error_number* define on yourself, where exception_name is character string up to 2048 bytes suppot and error_number is a negative integer range from -20000 to -20999.

Syntax

```
DECLARE
   user_define_exception_name EXCEPTION;
   PRAGMA EXCEPTION_INIT(user_define_exception_name,-error_number);
BEGIN
   statement(s);
   IF condition THEN
      RAISE user_define_exception_name;
   END IF;
EXCEPTION
   WHEN user_define_exception_name THEN
      User defined statement (action) will be taken;
END;
```

Example

user-named_exp.sql

```
SQL>edit user-named_exp
DECLARE
   myex EXCEPTION;
   PRAGMA EXCEPTION_INIT(myex,-20015);
```

```
   n NUMBER := &n;
BEGIN
   FOR i IN 1..n LOOP
      dbms_output.put.line(i);
      IF i=n THEN
         RAISE myex;
      END IF;
   END LOOP;
EXCEPTION
   WHEN myex THEN
      dbms_output.put.line('loop finish');
END;
/
```

Result

```
SQL>@user-named_exp
n                    number              &n=                 5
1
2
3
4
5
loop                                                         finish

PL/SQL procedure successfully operation.
```

PL/SQL RAISE_APPLICATION_ERROR

PL/SQL RAISE_APPLICATION_ERROR

In PL/SQL RAISE_APPLICATION_ERROR function use to assign exception name and exception error code.

Syntax

```
raise_application_error(error_number, error_message);
```

Example

raise_app_error.sql

```
SQL>edit user-named_exp
DECLARE
```

```
   myex EXCEPTION;
   n NUMBER := &n;
BEGIN
   FOR i IN 1..n LOOP
      dbms_output.put.line(i);
      IF i=n THEN
         RAISE myex;
      END IF;
   END LOOP;
EXCEPTION
   WHEN myex THEN
      RAISE_APPLICATION_ERROR(-20015, 'loop finish');
END;
/
```

Result

```
SQL>@raise_app_error
n                    number                    &n=                    5
1
2
3
4
5
ORA-20015:                    loop                    finish
```

PL/SQL procedure successfully operation.

When RAISE_APPLICATION_ERROR execute it's return error message and error code looking same as oracle built-in error.

PL/SQL Functions

PL/SQL functions block create using CREATE FUNCTION statement. The major difference between PL/SQL function or procedure, function return always value where as procedure may or may not return value.

When you create a function or procedure, you have to define IN/OUT/INOUT parameters parameters.

1. IN: IN parameter referring to the procedure or function and allow to overwritten the value of parameter.

2. OUT: OUT parameter referring to the procedure or function and allow to overwritten the value of parameter.
3. IN OUT: Both IN OUT parameter referring to the procedure or function to pass both IN OUT parameter, modify/update by the function or procedure and also get returned.

IN/OUT/INOUT parameters you define in function argument list that get returned back to a result. When you create the function default IN parameter is passed in argument list. It's means value is passed but not returned. Explicitly you have define OUT/IN OUT parameter in argument list.

PL/SQL Functions Syntax

```
CREATE [OR REPLACE] FUNCTION [SCHEMA..] function_name
    [ (parameter [,parameter]) ]
    RETURN return_datatype
  IS | AS
    [declaration_section
       variable declarations;
       constant declarations;
    ]
  BEGIN
    [executable_section
       PL/SQL execute/subprogram body
    ]
  [EXCEPTION]
       [exception_section
       PL/SQL Exception block
       ]
  END [function_name];
  /
```

PL/SQL Function Example

In this example we are creating a function to pass employee number and get that employee name from table. We have emp1 table having employee information,

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45k

2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

Create Function

So lets start passing IN parameter (no). Return datatype set varchar2. Now inside function SELECT ... INTO statement to get the employee name.

fun1.sql

```
SQL>edit fun1
CREATE or REPLACE FUNCTION fun1(no in number)
RETURN varchar2
IS
   name varchar2(20);
BEGIN
   select ename into name from emp1 where eno = no;
   return name;
END;
/
```

Execute Function

After write the PL/SQL function you need to execute the function.

SQL>@fun1
Function created.

PL/SQL procedure successfully completed.

PL/SQL Program to Calling Function

This program call the above define function with pass employee number and get that employee name.

fun.sql

```
SQL>edit fun
DECLARE
   no number :=&no;
```

```
   name varchar2(20);
BEGIN
   name := fun1(no);
   dbms_output.put_line('Name:'||'   '||name);
   end;
/
```

Result

```
SQL>@fun
no                                  number                              &n=2
Name:                               marks                               jems

PL/SQL procedure successfully completed.
```

PL/SQL Drop Function

You can drop PL/SQL function using DROP FUNCTION statements.

Syntax

```
DROP FUNCTION function_name;
```

Example

```
SQL>DROP                            FUNCTION                            fun1;

Function dropped.
```

PL/SQL Functions Syntax

```
CREATE [OR REPLACE] FUNCTION [SCHEMA..] function_name
      [ (parameter [,parameter]) ]
      RETURN return_datatype
   IS | AS
      [declaration_section
         variable declarations;
         constant declarations;
      ]
   BEGIN
      [executable_section
         PL/SQL execute/subprogram body
      ]
```

```
[EXCEPTION]
    [exception_section
    PL/SQL Exception block
    ]
END [function_name];
/
```

PL/SQL Function Example

In this example we are creating a function to pass employee number and get that employee name from table. We have emp1 table having employee information,

EMP_NO	EMP_NAME	EMP_DEPT	EMP_S
1	Forbs ross	Web Developer	45k
2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

Create Function

So lets start passing IN parameter (no). Return datatype set varchar2. Now inside function SELECT ... INTO statement to get the employee name.

fun1.sql

```
SQL>edit fun1
CREATE or REPLACE FUNCTION fun1(no in number)
RETURN varchar2
IS
   name varchar2(20);
BEGIN
   select ename into name from emp1 where eno = no;
   return name;
END;
/
```

Execute Function

After write the PL/SQL function you need to execute the function.

SQL>@fun1
Function created.

PL/SQL procedure successfully completed.

PL/SQL Program to Calling Function

This program call the above define function with pass employee number and get that employee name.

fun.sql

```
SQL>edit fun
DECLARE
   no number :=&no;
   name varchar2(20);
BEGIN
   name := fun1(no);
   dbms_output.put_line('Name:'||'   '||name);
   end;
/
```

Result

SQL>@fun
no number &n=2
Name: marks jems

PL/SQL procedure successfully completed.

PL/SQL Drop Function

You can drop PL/SQL function using DROP FUNCTION statements.

Syntax

```
DROP FUNCTION function_name;
```

Example

SQL>DROP FUNCTION fun1;

Function dropped.

PL/SQL Cursors

In PL/SQL, Cursor area also saying session cursor. because session cursor store information until the session end. Both way you can manage session cursor either implicit cursor or explicit cursor.

Using procedural statement you can get any information using session attribute.

How to Use Cursor

In PL/SQL block SELECT statement can not return more than one row at a time. So Cursor use to some group of rows (more than one row) for implementing certain logic to get all the group of records.

Classification of CURSORS

Cursors can be classified as:

1. Implicit Cursor or Internal Cursor Manage for Oracle itself or internal process itself.
2. Explicit Cursor or User-defined Cursor Manage for user/programmer or external processing.

Implicit Cursor

Oracle uses implicit cursors for its internal processing. Even if we execute a SELECT statement or DML statement Oracle reserves a private SQL area in memory called cursor.

Implicit cursor scope you can get information from cursor by using session attributes until another SELECT statement or DML statement execute.

Explicit Cursor

Explicit Cursor which are construct/manage by user itself call explicit cursor.

User itself to declare the cursor, open cursor to reserve the memory and populate data, fetch the records from the active data set one at a time, apply logic and last close the cursor.

You can not directly assign value to an explicit cursor variable you have to use expression or create subprogram for assign value to explicit cursor variable.

Step for Using Explicit Cursor

1. Declare cursor
2. Open cursor
3. Loop
4. Fetch data from cursor
5. Exit loop
6. Close cursor

```
7. SQL>set serveroutput on
8. SQL>edit implicit_cursor
9. BEGIN
```

```
10.    UPDATE emp_information SET emp_dept='Web Developer'
11.       WHERE emp_name='Saulin';
12.
13.    IF SQL%FOUND THEN
14.       dbms_output.put_line('Updated - If Found');
15.    END IF;
16.
17.    IF SQL%NOTFOUND THEN
18.       dbms_output.put_line('NOT Updated - If NOT Found');
19.    END IF;
20.
21.    IF SQL%ROWCOUNT>0 THEN
22.       dbms_output.put_line(SQL%ROWCOUNT||' Rows Updated');
23.    ELSE
24.       dbms_output.put_line('NO Rows Updated Found');
25.    END;
26. /
```

```
27. Result
28. SQL>@implicit_cursor
    Updated - If Found
    1 Rows Updated

    PL/SQL procedure successfully operation.
```

PL/SQL Explicit Cursor

Explicit Cursor which are construct/manage by user itself call explicit cursor.

User itself to declare the cursor, open cursor to reserve the memory and populate data, fetch the records from the active data set one at a time, apply logic and last close the cursor.

You can not directly assign value to an explicit cursor variable you have to use expression or create subprogram for assign value to explicit cursor variable.

```
29. SQL>set serveroutput on
30. SQL>edit explicit_cursor
31. DECLARE
32.   cursor c is select * from emp_information
33.   where emp_name='bhavesh';
34.   tmp emp_information%rowtype;
35. BEGIN
36.   OPEN c;
37.   Loop exit when c%NOTFOUND;
38.     FETCH c into tmp;
39.     update emp_information set tmp.emp_dept='Web Developer'
40.     where tmp.emp_name='Saulin';
41.   END Loop;
42. IF c%ROWCOUNT>0 THEN
43.   dbms_output.put_line(SQL%ROWCOUNT||' Rows Updated');
44. ELSE
45.   dbms_output.put_line('NO Rows Updated Found');
```

```
46. END IF;
47. CLOSE c;
48. END;
49. /
```

50. Result
51. **SQL>@explicit_cursor**
    1 Rows Updated

PL/SQL procedure successfully completed.

PL/SQL Cursors For Loop

PL/SQL cursor FOR loop has one great advantage of loop continued until row not found. In sometime you require to use explicit cursor with FOR loop instead of use OPEN, FETCH, and CLOSE statement.

FOR loop iterate repeatedly and fetches rows of values from database until row not found.

Explicit Cursor FOR LOOP Example

following one emp_information table:

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45k
2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

Display employee number wise first two employee details emp,

Example

cursor_for_loop.sql

```
SQL>set serveroutput on
SQL>edit cursor_for_loop
DECLARE
   cursor c is select * from emp_information
   where emp_no <=2;
```

```
   tmp emp_information%rowtype;
BEGIN
   OPEN c;
   FOR tmp IN c LOOP
   FETCH c into tmp;
   dbms_output.put_line('EMP_No:    '||tmp.emp_no);
   dbms_output.put_line('EMP_Name:  '||tmp.emp_name);
   dbms_output.put_line('EMP_Dept:  '||tmp.emp_dept);
   dbms_output.put_line('EMP_Salary:'||tmp.emp_salary);
   END Loop;
CLOSE c;
END;
/
```

Result

```
SQL>@cursor_for_loop
EMP_No:    1
EMP_Name:  Forbs                                                ross
EMP_Dept:  Web                                                  Developer
EMP_Salary:45k

EMP_No:    2
EMP_Name:  marks                                                jems
EMP_Dept:  Program                                              Developer
EMP_Salary:38k

PL/SQL procedure successfully completed.
```

PL/SQL Parameterized Cursor

PL/SQL Parameterized cursor pass the parameters into a cursor and use them in to query.

PL/SQL Parameterized cursor define only datatype of parameter and not need to define it's length.

Default values is assigned to the Cursor parameters. and scope of the parameters are locally.

Parameterized cursors are also saying static cursors that can passed parameter value when cursor are opened.

Following example introduce the parameterized cursor. following emp_information table,

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45k
2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

Example

Cursor display employee information from emp_information table whose emp_no four (4).

parameter_cursor_demo.sql

```
SQL>set serveroutput on
SQL>edit parameter_cursor_demo
DECLARE
  cursor c(no number) is select * from emp_information
  where emp_no = no;
  tmp emp_information%rowtype;
BEGIN
  OPEN c(4);
  FOR tmp IN c(4) LOOP
  dbms_output.put_line('EMP_No:    '||tmp.emp_no);
  dbms_output.put_line('EMP_Name:  '||tmp.emp_name);
  dbms_output.put_line('EMP_Dept:  '||tmp.emp_dept);
  dbms_output.put_line('EMP_Salary:'||tmp.emp_salary);
  END Loop;
CLOSE c;
END;
/
```

Result

SQL>@parameter_cursor_demo
EMP_No: 4
EMP_Name: Zenia Sroll
EMP_Dept: Web Developer
EMP_Salary: 42k

PL/SQL procedure successfully completed.

Important key point you must remember

1. Scope of the parameters are locally
2. You can assign default value to a cursor parameter.

PL/SQL Packages

PL/SQL Packages is schema object and collection of related data type (variables, constants), cursors, procedures, functions are defining within a single context. Package are divide into two part,

1. Package Specification
2. Package Body

Package specification block you can define variables, constants, exceptions and package body you can create procedure, function, subprogram.

PL/SQL Package Advantages

1. You can create package to store all related functions and procedures are grouped together into single unit called packages.
2. Package are reliable to granting a privileges.
3. All function and procedure within a package can share variable among them.
4. Package are support overloading to overload functions and procedures.
5. Package are improve the performance to loading the multiple object into memory at once, therefore, subsequent calls to related program doesn't required to calling physically I/O.
6. Package are reduce the traffic because all block execute all at once.

PL/SQL Package Syntax

PL/SQL Specification: This contain the list of variables, constants, functions, procedure names which are the part of the package. PL/SQL specification are public declaration and visible to a program.

Defining Package Specification Syntax

```
CREATE [OR REPLACE] PACKAGE package_name
  IS | AS
  [variable_declaration ...]
```

```
  [constant_declaration ...]
  [exception_declaration ...]
  [cursor_specification ...]
  [PROCEDURE [Schema..] procedure_name
     [ (parameter {IN,OUT,IN OUT} datatype [,parameter]) ]
  ]
  [FUNCTION [Schema..] function_name
     [ (parameter {IN,OUT,IN OUT} datatype [,parameter]) ]
     RETURN return_datatype
  ]
END [package_name];
```

PL/SQL Body: This contains the actual PL/SQL statement code implementing the logics of functions, procedures which are you already before declare in "Package specification".

Creating Package Body Syntax

```
CREATE [OR REPLACE] PACKAGE BODY package_name
  IS | AS
  [private_variable_declaration ...]
  [private_constant_declaration ...]
  BEGIN
     [initialization_statement]
     [PROCEDURE [Schema..] procedure_name
        [ (parameter [,parameter]) ]
        IS | AS
           variable declarations;
           constant declarations;
        BEGIN
           statement(s);
        EXCEPTION
           WHEN ...
        END
     ]
     [FUNCTION  [Schema..] function_name
           [ (parameter [,parameter]) ]
```

```
        RETURN return_datatype
    IS | AS
        variable declarations;
        constant declarations;
    BEGIN
        statement(s);
    EXCEPTION
        WHEN ...
    END
  ]
[EXCEPTION
  WHEN built-in_exception_name_1 THEN
    User defined statement (action) will be taken;
]
END;
/
```

PL/SQL Package Example

PL/SQL Package example step by step explain to you, you are create your own package using this reference example. We have emp1 table having employee information,

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45k
2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

Package Specification

Create Package specification code for defining procedure, function IN or OUT parameter and execute package specification program.

```
CREATE or REPLACE PACKAGE pkg1
   IS | AS
   PROCEDURE pro1
      (no in number, name out varchar2);
   FUNCTION fun1
      (no in number)
      RETURN varchar2;
   END;
/
```

Package Body

Create Package body code for implementing procedure or function that are defined package specification. Once you implement execute this program.

```
CREATE or REPLACE PACKAGE BODY pkg1
IS
   PROCEDURE pro1(no in number,info our varchar2)
      IS
      BEGIN
         SELECT * INTO temp FROM emp1 WHERE eno = no;
      END;

   FUNCTION fun1(no in number) return varchar2
      IS
      name varchar2(20);
      BEGIN
         SELECT ename INTO name FROM emp1 WHERE eno = no;
         RETURN name;
      END;
END;
/
```

Pl/SQL Program calling Package

Now we have a one package pkg1, to call package defined function, procedures also pass the parameter and get the return result.

pkg_prg.sql

```
DECLARE
   no number := &no;
   name varchar2(20);
BEGIN
   pkg1.pro1(no,info);
   dbms_output.put_line('Procedure Result');
   dbms_output.put_line(info.eno||'     '||
                  info.ename||'   '||
                  info.edept||'   '||
                  info.esalary||' '||);
   dbms_output.put_line('Function Result');
   name := pkg1.fun1(no);
   dbms_output.put_line(name);
END;
/
```

Result

Now execute the above created pkg_prg.sql program to asking which user information you want to get, you put user id and give information.

SQL>@pkg_prg

```
no                              number                          &n=2
Procedure                                                       Result
2    marks                  jems    Program                 Developer    38K
Function                                                        Result
marks                                                           jems

PL/SQL procedure successfully completed.
```

PL/SQL Package Alter

You can update package code you just recompile the package body,

Syntax

```
ALTER PACKAGE package_name COMPILE BODY;
```

Recompile the already created/executed package code,

Example

SQL>ALTER PACKAGE pkg1 COMPILE BODY;

Package body Altered.

PL/SQL Package Drop

You can drop package using package DROP statement,

Syntax

```
DROP PACKAGE package_name;
```

Example

SQL>DROP PACKAGE pkg1;

Package dropped.

PL/SQL Triggers

What is PL/SQL Trigger?

Oracle engine invokes automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match.

You can change trigger mode activate/deactivate but you can't explicitly run.

Trigger automatically associated with DML statement, when DML statement execute trigger implicitly execute.

You can create trigger using the CREATE TRIGGER statement. If trigger activated, implicitly fire DML statement and if trigger deactivated can't fire.

Component of Trigger

Triggering SQL statement: SQL DML (INSERT, UPDATE and DELETE) statement that execute and implicitly called trigger to execute.

Trigger Action: When the triggering SQL statement is execute, trigger automatically call and PL/SQL trigger block execute.

Trigger Restriction: We can specify the condition inside trigger to when trigger is fire.

Type of Triggers

1. BEFORE Trigger: BEFORE trigger execute before the triggering DML statement (INSERT, UPDATE, DELETE) execute. Triggering SQL statement is may or may not execute, depending on the BEFORE trigger conditions block.
2. AFTER Trigger: AFTER trigger execute after the triggering DML statement (INSERT, UPDATE, DELETE) executed. Triggering SQL statement is execute as soon as followed by the code of trigger before performing Database operation.

3. ROW Trigger: ROW trigger fire for each and every record which are performing INSERT, UPDATE, DELETE from the database table. If row deleting is define as trigger event, when trigger file, deletes the five rows each times from the table.
4. Statement Trigger: Statement trigger fire only once for each statement. If row deleting is define as trigger event, when trigger file, deletes the five rows at once from the table.
5. Combination Trigger: Combination trigger are combination of two trigger type,
    1. Before Statement Trigger: Trigger fire only once for each statement before the triggering DML statement.
    2. Before Row Trigger : Trigger fire for each and every record before the triggering DML statement.
    3. After Statement Trigger: Trigger fire only once for each statement after the triggering DML statement executing.
    4. After Row Trigger: Trigger fire for each and every record after the triggering DML statement executing.

PL/SQL Triggers Syntax

PL/SQL trigger define using CREATE TRIGGER statement.

```
CREATE [OR REPLACE] TRIGGER trigger_name
  BEFORE | AFTER
  [INSERT, UPDATE, DELETE [COLUMN NAME..]
  ON table_name

  Referencing [ OLD AS OLD | NEW AS NEW ]
  FOR EACH ROW | FOR EACH STATEMENT [ WHEN Condition ]

DECLARE
  [declaration_section
    variable declarations;
    constant declarations;
  ]

BEGIN
  [executable_section
    PL/SQL execute/subprogram body
  ]

EXCEPTION
  [exception_section
```

```
    PL/SQL Exception block
  ]

END;
```

Syntax Description

1. CREATE [OR REPLACE] TRIGGER trigger_name: Create a trigger with the given name. If already have overwrite the existing trigger with defined same name.
2. BEFORE | AFTER : Indicates when the trigger get fire. BEFORE trigger execute before when statement execute before. AFTER trigger execute after the statement execute.
3. [INSERT, UPDATE, DELETE [COLUMN NAME..]: Determines the performing trigger event. You can define more then one triggering event separated by OR keyword.
4. ON table_name: Define the table name to performing trigger event.
5. Referencing [OLD AS OLD | NEW AS NEW]: Give referencing to a old and new values of the data. :old means use existing row to perform event and :new means use executing new row to perform event. You can set referencing names user define name from old (or new).
   You can't referencing old values when inserting a record, or new values when deleting a record, because It's does not exist.
6. FOR EACH ROW | FOR EACH STATEMENT: Trigger must fire when each row gets Affected (ROW Trigger). and fire only once when the entire sql statement is execute (STATEMENT Trigger).
7. WHEN Condition: Optional. Use only for row level trigger. Trigger fire when specified condition is satisfy.

PL/SQL Triggers Example

You can make your own trigger using trigger syntax referencing. Here are fewer trigger example.

Inserting Trigger

This trigger execute BEFORE to convert ename field lowercase to uppercase.

```
CREATE or REPLACE TRIGGER trg1
  BEFORE
  INSERT ON emp1
  FOR EACH ROW
BEGIN
  :new.ename := upper(:new.ename);
END;
/
```

Restriction to Deleting Trigger

This trigger is preventing to deleting row.

Delete Trigger Example

```
CREATE or REPLACE TRIGGER trg1
   AFTER
   DELETE ON emp1
   FOR EACH ROW
BEGIN
   IF :old.eno = 1 THEN
      raise_application_error(-20015, 'You can't delete this row');
   END IF;
END;
/
```

Delete Trigger Result

```
SQL>delete from emp1 where eno = 1;
Error Code: 20015
Error Name: You can't delete this row
```