PLSQL

To copy the whole table into new table with or without data we use **‘create table as’** to create the exact copy of the existing table.

**Syntax**:

**Create table** new\_table\_name **as** **select** column\_names existing\_table\_name **where** 1=2; 1=2 it refers to not copy the data only copy the structure of the table.

**Desc –** this command is used to describe the table like column and their datatypes.

**&** - it is used to get input from the user. It is used before the variable.

**Dbms\_output.put\_line() -** to print the values the output console.

**Plsql block:**

**Declare** -> variables are declared under this statement.

**Begin** -> to state that the sql block has started.

**End** -> to state that the sql block has ended.

**/** -> means the sqldeveloper to compile the statement present inside the begin and end keywords.

Structure of the block:

**Declare**

Variable\_name datatypes(size)

**Begin**

Select statements

**End**

**/**

**%TYPE ->** is used along with the variable name mentioning that the variable resonates the datatypes of the other table column if we are not aware of the datatypes of the returned values.

Eg :

Variable\_name schemaName.tableName.columName%type

**%ROWTYPES** -> it select the single entire row that contains different columns datatype into the declared variable.

Eg:

Variable\_name schemaName.tableName%type

**Complex/composite datatypes or user defined datatypes.**

It is used to create the user defined datatype. It consist of one or more datatypes of different variables.

Structure:

Declare

**Type** userdefined\_variable\_name

**Is record**(varible\_name datatypes,…….);

Variable\_name userdefined\_variable\_name;

**Begin**

Select \* into userdefined\_variable\_name from employees;

**End**

**Collections :**

1. **Associated array or index by table:**

We can store this collection into the database and can be used in plsql program only.

These are useful for storing array of values retrieved from the database.

Size of the collection increase or decrease dynamically.

Data can be added/deleted to any position in the list.

Syntax:

**Type** type\_name

**Is table of** datatype

**Index by** (datatype);

Eg:

**Declare**

**Type** v\_emp\_details **is table of** emp%rowtype

**Index by** simple\_integer;

Var\_emp v\_emp\_details;

Begin

Select \* into var\_emp(2) from emp where empId=102;

Dbms\_output.put\_line(var\_emp(2).name||’,’||var\_emp(2).salary);

End;

/

**2.Nested table:**

We can store the data into the database unlike associate array.

They can be indexed by only integer.

you can perform multiset operation in this collections.

We need to initialize the collection.

This is spare collection and we can be delete the data.

Syntax:

**Type** type\_name

**Is table** of datatype(integer,simple\_integer);

Eg:

**Declare**

**Type** v\_emp\_details **is table of** emp%rowtype;

Var\_emp v\_emp\_details := v\_emp\_details(); ----- initialize the collection by constructor

**Begin**

**Var\_emp.extend(2); ----- extends the collection based on the values to be stored**

Select \* into var\_emp(1) from emp where empId=102;

Dbms\_output.put\_line(var\_emp(1).name||’,’||var\_emp(1).salary);

**End;**

/

**3.Varray:**

We can store the data into the database unlike associate array.

They can be indexed by only integer.

We cannot delete the individual element in this collection so it remains dense.

Need to specify the upper bound

Need to initialize the collection.

Syntax:

**Type** type\_name

**Is varray[size]** **of** datatype(integer,simple\_integer);

**Eg:**

**Declare**

**Type** v\_emp\_details **is varray(10) of** varchar2(20);

Var\_emp v\_emp\_details;

**Begin**

Var\_emp v\_emp\_details := v\_emp\_details();

**Var\_emp.extend(2); ----- extends the collection based on the values to be stored**

Var\_emp(1):=’’

Dbms\_output.put\_line(var\_emp(1).name||’,’||var\_emp(1).salary);

**End;**

/

**Collection methods:**

Count, first, last, prior(arg), next(arg), exists(arg), limit, extend, extend(n), trim, trim(n), delete, Delete(n)

**Multiset operators:**

**Multiset union:**

It return all the records from the collections set

Not sorted by default

Will not remove the duplicate records

**Multiset union distinct:**

Output is sorted by default

Return only unique value.

**Multiset intersect:**

Return common values from the all the collection set.

Remove the duplicate records.

Sorted by default.

**Multiset except:**

Return all the records from the one set excluding the values from the other set.

Remove duplicate.

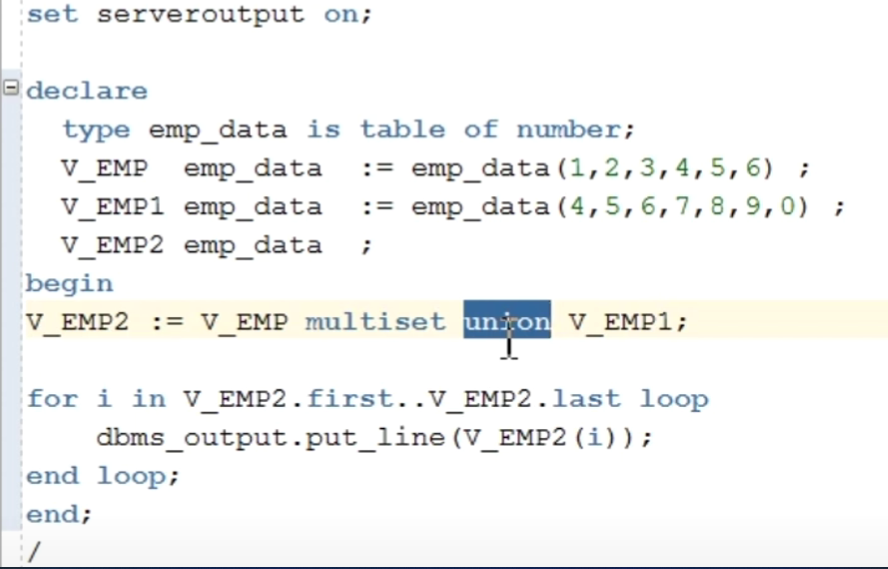
Sorted by default.

**Syntax:**

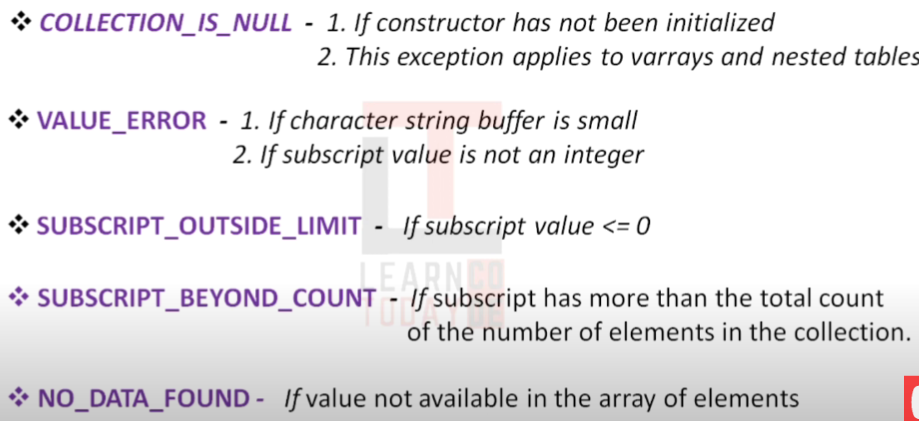
Nested\_table1

Multiset [union|intersect|except]

Nested\_table2



**Exception In collection:**

****

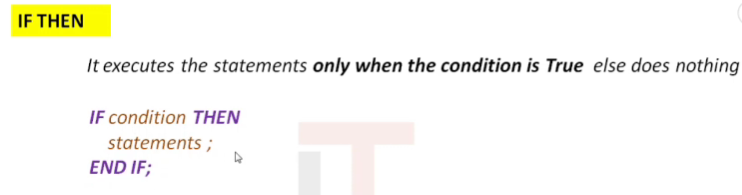
**Control statements:**

A screenshot of a computer

Description automatically generated with low confidence

If statement:

IF THEN



Eg:

A picture containing text, font, line, screenshot

Description automatically generated

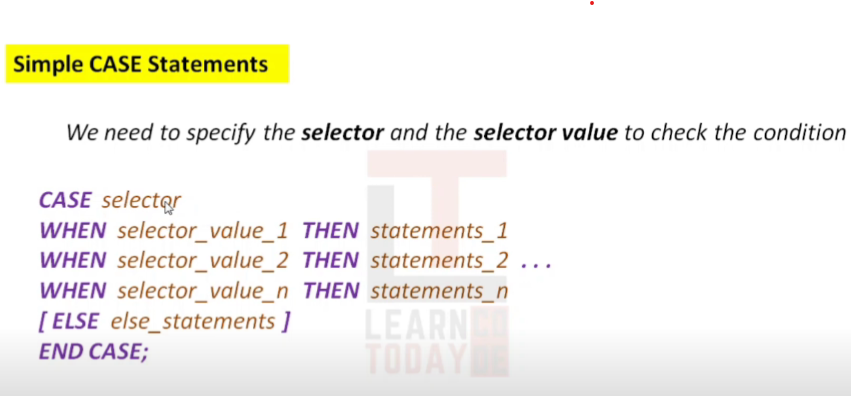
A screenshot of a computer

Description automatically generated with low confidence

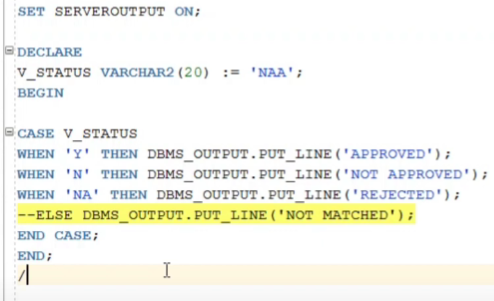
A picture containing text, screenshot, font, design

Description automatically generated

CASE STATEMENT:

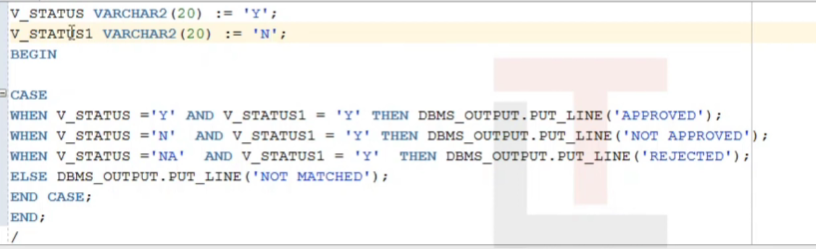


Eg:



SEARCHED CASE STATEMENT:

We can use multiple condition in the case statement unlike simple case statement.

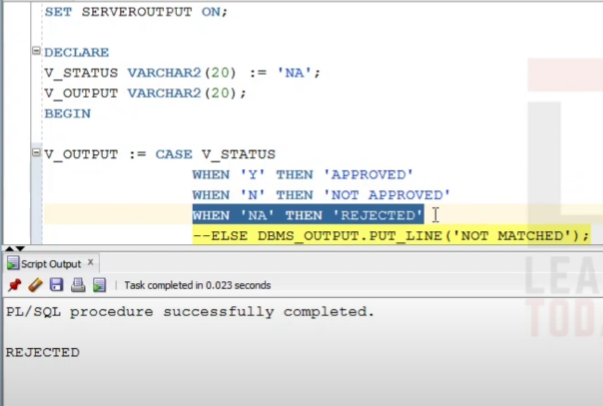


**Expression case statement:**

It is used when the case statement output itself is a input value to the variable.

To do this no semicolon in the case statement, no end case statement at the end only use end with semicolon, do not use predefined packages defined in the oracle.

**Eg:**

****

If no values matches with the case then the oracle assign the value to the variable as null.

**Loop statements:**

Simple loop statements:

Loop without any conditions.

Syntax:

**LOOP**

Statements:

**END LOOP**

**Exit statements:**

* To terminate the loops.
* Must be inside the loop
* Used to exit from the current loop.
* It is used to terminate the loop based on the conditional statements.

Syntax:

**Delclare**

**Begin**

**Loop**

Statements;

**If** condition **then**

**Exit;**

**End if;**

**End loop**

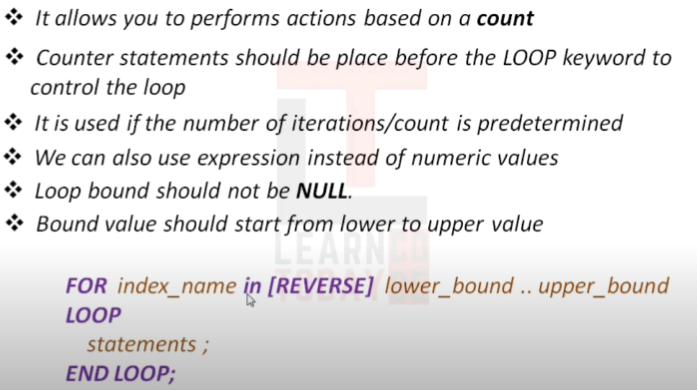
**End:**

**/**

**Exit when statements:**

* It is used to terminate the loop based on the when conditions.
* Statements in the loop should changes every time.

**For loop:**



Eg:

**For I in 1 .. 10** --- counter statements

**Loop**

Dbms\_ouput.put\_line(‘hi’);

**End loop;**

Bound values can be decimal and it take it as nearest interger.

**Continue**:

A picture containing text, screenshot, font

Description automatically generated

**Eg**:

**Declare**

V\_name number:=1;

**Begin**

**For I in 1 .. 5**

**Loop**

V\_name:= V\_name +1;

**If v\_name =3 then**

Continue;

**End if;**

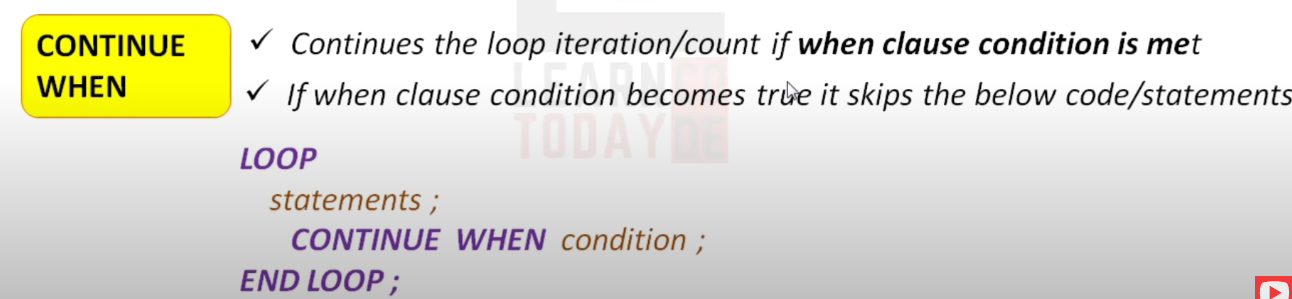
Dbms\_ouput.put\_line(v\_name);

**End loop;**

**End:**

**/**

**Continue when clause:**

****

**Eg:**

**Declare**

V\_name number:=1;

**Begin**

**For I in 1 .. 5**

**Loop**

V\_name:= V\_name +1;

**Continue when** v\_name=3;

Dbms\_ouput.put\_line(v\_name);

**End loop;**

**End:**

**/**

**While loop:**

**Syntax:**

**While** condition

**Loop**

Statements**;**

**End loop;**

**Eg:**

**Declare**

V\_name number:=1;

**Begin**

**While v\_name<10**

**Loop**

V\_name:= V\_name +2;

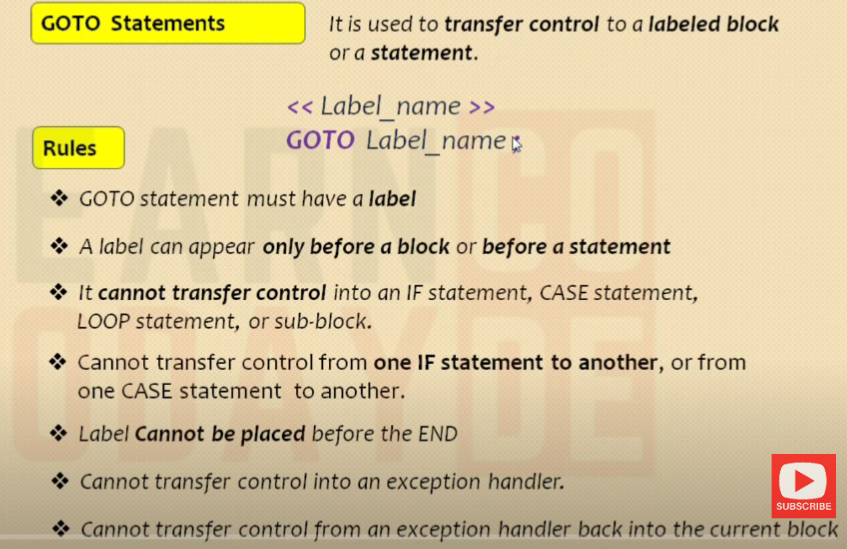
Dbms\_ouput.put\_line(v\_name);

**End loop;**

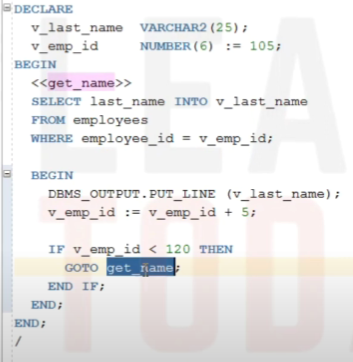
**End:**

**/**

Goto Statements:



Eg:



Null statements pass the controller to the next statements without doing anything.

**Syntax:**

**Null;**

TRIGGER:

Syntax:

CREATE [OR REPLACE] TRIGGER Ttrigger\_name

{BEFORE|AFTER} Triggering\_event ON table\_name

[FOR EACH ROW]

[FOLLOWS another\_trigger\_name]

[ENABLE/DISABLE]

[WHEN condition]

DECLARE

declaration statements

BEGIN

executable statements

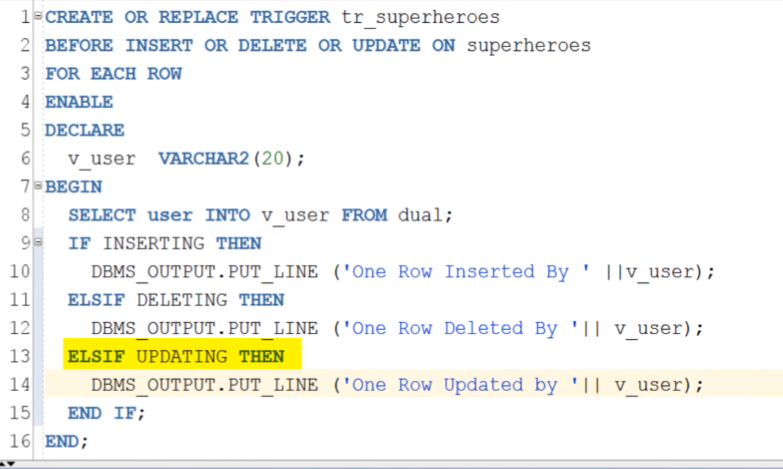
EXCEPTION

exception-handling statements

END;

DML TRIGGERS:

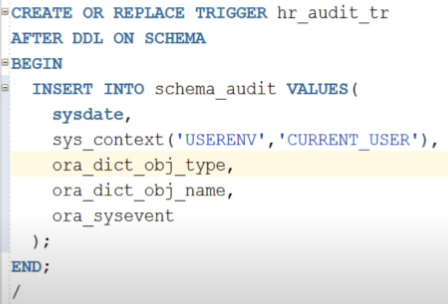
Eg:



DDL trigger:

This trigger will get into the action once the DDL operation(create, alter, truncate,drop,rename) Is performed in the mentioned table.

Eg:



Here ‘**after ddl on schema’** indicates that this trigger will work after any DDL operation has happened in the schema.

**INSTEAD OF INSERT;**

This is helpful when we want to insert the values into the underlying tables of the view.

This trigger is used to make the non-updatable views into updatable.

Eg:

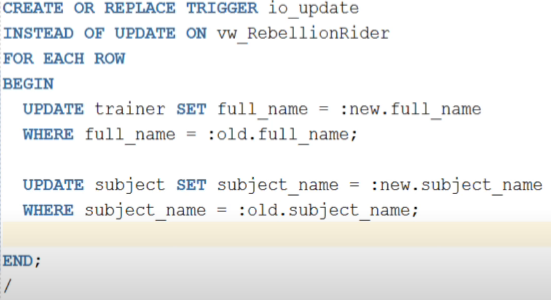
A picture containing text, font, screenshot

Description automatically generated

**INSTEAD OF UPDATE;**

It is similar to instead of insert trigger. We use update command in the trigger to update the underlying table of the views.

Eg:



**INSTEAD OF DELETE;**

Eg:

A screenshot of a computer code

Description automatically generated with low confidence

**Cursor:**

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

**1.Implicit cursor**

|  |  |
| --- | --- |
| **S.No** | **Attribute & Description** |
| 1 | **%FOUND**  Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| 2 | **%NOTFOUND**  The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| 3 | **%ISOPEN**  Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| 4 | **%ROWCOUNT**  Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

**Eg:**

DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 500;

IF sql%notfound THEN

dbms\_output.put\_line('no customers selected');

ELSIF sql%found THEN

total\_rows := sql%rowcount;

dbms\_output.put\_line( total\_rows || ' customers selected ');

END IF;

END;

/

**2.Explicit cursor:**

Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

* Declaring the cursor for initializing the memory
* Opening the cursor for allocating the memory
* Fetching the cursor for retrieving the data
* Closing the cursor to release the allocated memory

Syntax:

DECLARE

CURSOR cursor\_name IS select\_statement;

BEGIN

OPEN cursor\_name;

FETCH cursor\_name INTO PL/SQL variable [PL/SQL record];

CLOSE cursor\_name;

END;

/

Eg:

DECLARE

c\_id customers.id%type;

c\_name customers.name%type;

c\_addr customers.address%type;

CURSOR c\_customers is

SELECT id, name, address FROM customers;

BEGIN

OPEN c\_customers;

LOOP

FETCH c\_customers into c\_id, c\_name, c\_addr;

EXIT WHEN c\_customers%notfound;

dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);

END LOOP;

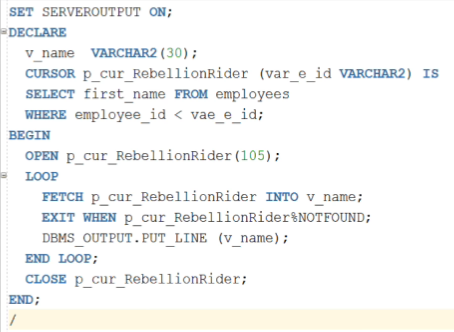
CLOSE c\_customers;

END;

/

**Parameterised cursor:**

We need to specify the parameter variable along with the cursor name if more than one variable separate it by (,).



**Cursor with default parameter value:**

A screenshot of a computer code

Description automatically generated with medium confidence

Even though we have given the default value in the cursor if we gave the any other value during the cursor open it takes this value as main value.

Cursor loop :

A screenshot of a computer code

Description automatically generated with medium confidence

Here **L\_IDX is the %rowtype** record variable. It holds the entire data of the cursor.

Cursor loop with parameter:

A picture containing text, screenshot, font

Description automatically generated