ENTRENAMIENTO
EXPERIENCIA

GALAXY
TRAINING

ORACLE DATABASE 19c DEVELOPER: PL/SQL

SESION 02

Ing. Cesar Hijar
Instructor

chijar@galaxy.edu.pe

AGENDA

Programación de objetos I

- Secuencias e interacción con la base de datos
- Estructuras de control.
- Trabajando con composite data types
- Cursores implícitos y explícitos.
- Taller práctico - Quiz

SECUENCIAS E INTERACCION CON LA BD

Inside PL/SQLblock

- **Identifiers:** Identifiers are the names given to PL/SQL objects

Ex: v_empno, v_ename, " first Name "

Not recommended
Case sensitive
You can use space
You can use reserved word

- **Delimiters** : symbols that have special meaning

Ex: ; + *

- **Literals:** Any value that is assigned to a variable is a literal.

Ex: v_ename:='khaled' ; , v_empno:=10; v_flag:=true;

- **Comments:** used to describe you code

```
Ex:  --this code calculate sum of salaries
     /* this code calculate
     sum of salaries
     */
```

SECUENCIAS E INTERACCION CON LA BD

Delimiters are simple or compound symbols that have special meaning in PL/SQL.

Simple symbols

Symbol	Meaning
+	Addition operator
-	Subtraction/negation operator
*	Multiplication operator
/	Division operator
=	Equality operator
@	Remote access indicator
;	Statement terminator

Compound symbols

Symbol	Meaning
<>	Inequality operator
!=	Inequality operator
||	Concatenation operator
--	Single-line comment indicator
/*	Beginning comment delimiter
*/	Ending comment delimiter
:=	Assignment operator

Commenting Code

- Prefix single-line comments with two hyphens (--).
- Place multiple-line comments between the symbols /* and */.

Example:

```
DECLARE
...
v_annual_sal NUMBER (9,2);
BEGIN
/* Compute the annual salary based on the
   monthly salary input from the user */
v_annual_sal := monthly_sal * 12;
--The following line displays the annual salary
DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row functions

```
Ex: v_ename:=substr(ename,1,5 );
    v_lname:= length( first_name );
    v_comm:= nvl( comm,0 );
    v_date:=add_months( hiredate,3 );
```

- Not available in procedural statements:
  - DECODE
  - Group functions

→ But you can use it in SQL statement inside PL/SQL

SECUENCIAS

Is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

```
CREATE SEQUENCE sequence_name
  START WITH value
  INCREMENT BY value
  MAXVALUE value
  MINVALUE value
  CACHE value / NOCACHE
  CYCLE
NOORDER; -- default
```

IDENTITY

Oracle 12c

En esta versión apareció una nueva forma de definir campos auto incrementales, que pudieran ser aplicables a campos PK (clave primaria), pero se debe tomar en cuenta varios factores que se detallan a continuación:

Sintaxis:

```
COLUMN NAME
    GENERATED [ALWAYS | BY DEFAULT [ ON NULL ] ]
        AS IDENTITY [ ( identity_options ) ]
```

Donde:

ALWAY: Define que no es necesario denotar el campo explícitamente en la sentencia INSERT, y si se lo coloca dándole un valor incluso válido, arrojará un error: **"ORA-32795: cannot insert into a generated always identity column"**

BY DEFAULT: Se debe definir esta propiedad si en la sentencia INSERT se lo pretende colocar explícitamente, aceptará el valor otorgado, sin embargo, si se da un valor nulo saldrá un error debido a que todo campo definido como IDENTITY es NOT NULL.

BY DEFAULT ON NULL: Se debe definir esta propiedad si puede darse el caso de otorgar valor nulo, se activará la propiedad específica ON NULL desencadenando la generación por defecto de un nuevo valor auto incrementado a través del SEQUENCE destinado.

identity_options: Es, básicamente, la sintaxis para generador de secuencias - mismas que las opciones CREATE SEQUENCE.

```
CREATE SEQUENCE sequence_name
 MINVALUE value
 MAXVALUE value
 START WITH value
 INCREMENT BY value
 CACHE value;
```

IDENTITY

https://www.oracletutorial.com/oracle-basics/oracle-identity-column/#:~:text=Introduction%20to%20Oracle%20identity%20column,the%20surrogate%20primary%20key%20column.

Using Sequences in PL/SQL Expressions

Starting in 11*g*:

```
DECLARE
  v_new_id NUMBER;
BEGIN
  v_new_id := my_seq.NEXTVAL;
END;
/
```

Before 11*g*:

```
DECLARE
  v_new_id NUMBER;
BEGIN
  SELECT my_seq.NEXTVAL INTO v_new_id FROM Dual;
END;
/
```

Data Type Conversion

- Converts data to comparable data types
- Is of two types:
  - Implicit conversion
  - Explicit conversion
- Functions:
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER
  - TO_TIMESTAMP

Data Type Conversion

1
```
-- implicit data type conversion
v_date_of_joining DATE:= '02-Feb-2000';
```

2
```
-- error in data type conversion
v_date_of_joining DATE:= 'February 02,2000';
```

3
```
-- explicit data type conversion
v_date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```

Data Type Conversion (continued)

Note the three examples of implicit and explicit conversions of the `DATE` data type in the slide:

1. Because the string literal being assigned to `date_of_joining` is in the default format, this example performs implicit conversion and assigns the specified date to `date_of_joining`.
2. The PL/SQL returns an error because the date that is being assigned is not in the default format.
3. The `TO_DATE` function is used to explicitly convert the given date in a particular format and assign it to the `DATE` data type variable `date_of_joining`.

Nested Blocks

PL/SQL blocks can be nested.

- An executable section (BEGIN … END) can contain nested blocks.
- An exception section can contain nested blocks.

```
Declare
...
Begin
    ...
    ...
        declare
        ....
        begin
        ...
       End;
End;
```

Nested Blocks

Example:

```
DECLARE
 v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(v_inner_variable);
   DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Variable Scope and Visibility

```
DECLARE
 v_father_name VARCHAR2(20):='Patrick';
 v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
   v_child_name VARCHAR2(20):='Mike';
   v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
  END;
 DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
END;
/
```

SQL & PL/SQL programming Guidelines

Make code maintenance easier by:

- Documenting code with comments
- Developing a case convention for the code
- Developing naming conventions for identifiers and other objects
- Enhancing readability by indenting

Category	Case Convention	Examples
SQL statements	Uppercase	`SELECT, INSERT`
PL/SQL keywords	Uppercase	`DECLARE, BEGIN, IF`
Data types	Uppercase	`VARCHAR2, BOOLEAN`
Identifiers and parameters	Lowercase	`v_sal, emp_cursor, g_sal, p_empno`
Database tables and columns	Lowercase	`employees, employee_id, department_id`

SQL & PL/SQL programming Guidelines

Indenting Code

For clarity, indent each level of code.

```
BEGIN
  IF x=0 THEN
     y:=1;
  END IF;
END;
/
```

```
DECLARE
  deptno        NUMBER(4);
  location_id   NUMBER(4);
BEGIN
  SELECT   department_id,
           location_id
  INTO     deptno,
           location_id
  FROM     departments
  WHERE    department_name
           = 'Sales';
...
END;
/
```

SQL Statements in PL/SQL

- Retrieve a row from the database by using the `SELECT` command.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the `COMMIT`, `ROLLBACK`, or `SAVEPOINT` command.

- PL/SQL does not directly support data definition language (DDL) statements,

PL/SQL does not directly support data control language (DCL) statements, such as `GRANT` or `REVOKE`. You can use dynamic SQL to execute them.

SELECT Statements in PL/SQL

- The INTO clause is required.
- Queries must return only one row.

Example:

```
DECLARE
 v_fname VARCHAR2(25);
BEGIN
 SELECT first_name INTO v_fname
 FROM employees WHERE employee_id=200;
 DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
END;
/
```

Queries Must Return Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: queries must return only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages these errors by raising standard exceptions, which you can handle in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Include a WHERE condition in the SQL statement so that the statement returns a single row. You learn about exception handling later in the course.

SECUENCIAS E INTERACCION CON LA BD

Naming Conventions

```
DECLARE
   hire_date       employees.hire_date%TYPE;
   sysdate         hire_date%TYPE;
   employee_id     employees.employee_id%TYPE := 176;
BEGIN
   SELECT       hire_date, sysdate
   INTO         hire_date, sysdate
   FROM         employees
   WHERE        employee_id = employee_id;
END;
/
```

ambiguedades

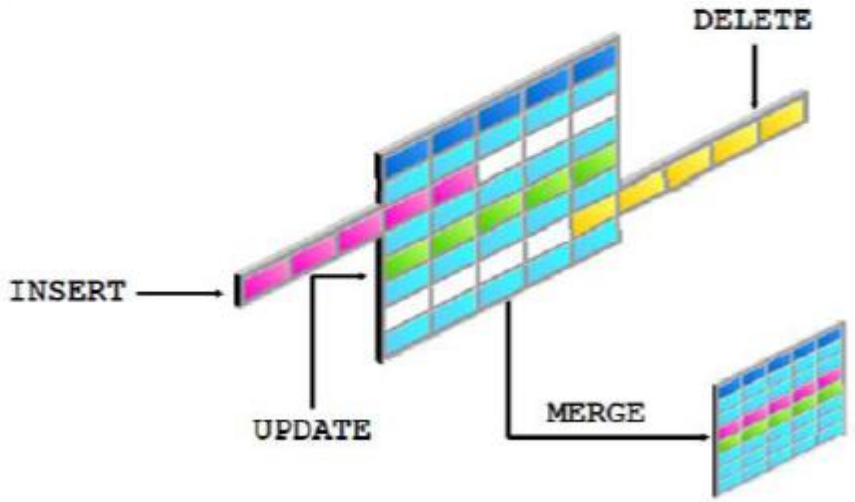
the names of database columns take precedence over the names of local variables.

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

Using PL/SQL to Manipulate Data

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



Inserting Data: Example

Add new employee information to the EMPLOYEES table.

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',CURRENT_DATE, 'AD_ASST', 4000);
END;
/
```

Updating Data: Example

Increase the salary of all employees who are stock clerks.

```
DECLARE
  sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE      employees
  SET         salary = salary + sal_increase
  WHERE       job_id = 'ST_CLERK';
END;
/
```

Deleting Data: Example

Delete rows that belong to department 10 from the `employees` table.

```
DECLARE
  deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   employees
  WHERE  department_id = deptno;
END;
/
```

SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle server.
- A cursor is used to handle the result set of a `SELECT` statement.
- There are two types of cursors:
  - **Implicit:** Created and managed internally by the Oracle server to process SQL statements
  - **Explicit:** Declared explicitly by the programmer

SQL Cursor Attributes for Implicit Cursors

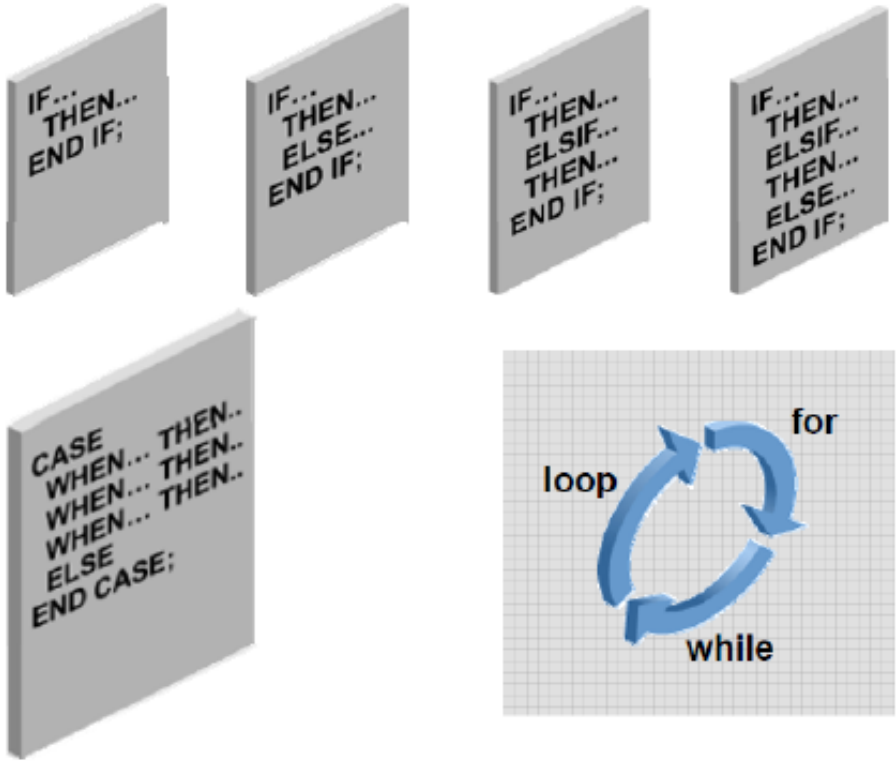
Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement returned at least one row
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement did not return even one row
SQL%ROWCOUNT	An integer value that represents the number of rows affected by the most recent SQL statement

ESTRUCTURAS DE CONTROL

ESTRUCTURAS DE CONTROL

Controlling Flow of Execution



IF STATEMENT

1

```
IF x>10  Then
.....
......
End if;
```

2

```
IF x>10  Then
.....
......
ELSE
.......
End if;
```

3

```
IF x=10  Then
.....
ELSIF X=9
.......
ELSIF X=8
......
End if;
```

4

```
IF x=10  Then
.....
ELSIF X=9
.......
ELSIF X=8
......
ELSE
......
End if;
```

NULL Value in IF Statement

```
DECLARE
  v_myage  number;
BEGIN
  IF v_myage  < 11 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
/
```

```
anonymous block completed
 I am not a child
```

In the example shown in the slide, the variable `v_myage` is declared but not initialized. The condition in the `IF` statement returns `NULL` rather than `TRUE` or `FALSE`. In such a case, the control goes to the `ELSE` statement.

Ojo con el valor de v_myage

CASE Expressions

```
CASE selector
   WHEN expression1 THEN result1
   WHEN expression2 THEN result2
   ...
   WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
/
```

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

CASE Statement

A CASE expression evaluates the condition and returns a value, whereas a CASE statement evaluates the condition and performs an action. A CASE statement can be a complete PL/SQL block.

- CASE statements end with END CASE;
- CASE expressions end with END;

Handling Nulls

Consider the following example:

```
x := 5;
y := NULL;
...
IF x != y THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

You may expect the sequence of statements to execute because x and y seem unequal. But nulls are indeterminate. Whether or not x is equal to y is unknown. Therefore, the IF condition yields NULL and the sequence of statements is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

In the second example, you may expect the sequence of statements to execute because a and b seem equal. But, again, equality is unknown, so the IF condition yields NULL and the sequence of statements is bypassed.

Comparación de NULL y otros valores

ESTRUCTURAS DE CONTROL

Iterative Control: LOOP Statements

- Loops repeat a statement (or sequence of statements) multiple times.
- There are three loop types:
  - Basic loop (Should have exit)
  - FOR loop (based on count)
  - WHILE loop (Based on condition)

The Loop should have exist condition
Otherwise the loop is infinite

Basic Loops

Syntax:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

WHILE Loops

Syntax:

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Use the WHILE loop to repeat statements while a condition is TRUE.

FOR Loops

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

In the syntax:

counter	Is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached
REVERSE	Causes the counter to decrement with each iteration from the upper bound to the lower bound **Note:** The lower bound is still referenced first.
lower_bound	Specifies the lower bound for the range of counter values
upper_bound	Specifies the upper bound for the range of counter values

Nested Loops and Labels

- You can nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the `EXIT` statement that references the label.

Nested Loops and Labels

You can nest `FOR`, `WHILE`, and basic loops within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception was raised. However, you can label loops and exit the outer loop with the `EXIT` statement.

Label names follow the same rules as other identifiers. A label is placed before a statement, either on the same line or on a separate line. White space is insignificant in all PL/SQL parsing except inside literals. Label basic loops by placing the label before the word `LOOP` within label delimiters (*<<label>>*). In `FOR` and `WHILE` loops, place the label before `FOR` or `WHILE`.

If the loop is labeled, the label name can be included (optionally) after the `END LOOP` statement for clarity.

Trabajando con composite data types

Composite Data Types

- Can hold multiple values (unlike scalar types)
- Are of two types:
  - PL/SQL records
  - PL/SQL collections
    - `INDEX BY` tables or associative arrays
    - Nested table
    - `VARRAY`

TRABAJANDO CON COMPOSITE DATA TYPES

What is a PL/SQL Record

A PL/SQL record is a composite data structure that is a group of related data stored in *fields*.

Each field in the PL/SQL record has its own name and data type.

Declaring a PL/SQL Record

1- *programmer-defined records.*
2- *table-based record.* *%Rowtype*
3- *cursor-based record. (will be covered later)*

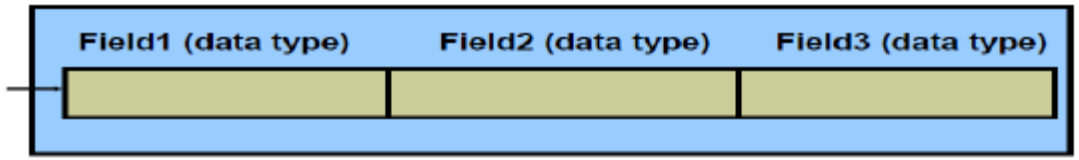
TRABAJANDO CON COMPOSITE DATA TYPES

PL/SQL Records

1- programmer-defined records

To declare programmer-defined record, first you have to define a record type by using TYPE statement with the fields of record explicitly.
Then, you can declare a record based on record type that you've defined.

```
DECLARE

TYPE t_EMP IS RECORD
( V_EMP_id employees.employee_id%type,
  v_first_name employees.first_name%type,
  v_last_name employees.last_name%type
);

v_emp t_EMP;

BEGIN
```



TRABAJANDO CON COMPOSITE DATA TYPES

Continue ....
1- programmer-defined records

PL/SQL Records

A record is a group of related data items stored in fields, each with its own name and data type.

- Each record defined can have as many fields as necessary.
- Records can be assigned initial values and can be defined as NOT NULL.
- Fields without initial values are initialized to NULL.
- The DEFAULT keyword can also be used when defining fields.
- You can define RECORD types and declare user-defined records in the declarative part of any block, subprogram, or package.
- You can declare and reference nested records. One record can be the component of another record.

TRABAJANDO CON COMPOSITE DATA TYPES

1- programmer-defined records

Example

```
DECLARE

TYPE t_EMP IS RECORD
( V_EMP_id employees.employee_id%type,
  v_first_name employees.first_name%type,
  v_last_name employees.last_name%type
);

v_emp t_EMP;

BEGIN
  select employee_id  ,first_name        ,last_name
  into v_emp
  from
  employees
  where employee_id=100;

  dbms_output.put_line(v_emp.V_EMP_id||' '||v_emp.v_first_name||' '||v_emp.v_last_name);

END;
```

TRABAJANDO CON COMPOSITE DATA TYPES

PL/SQL Records

2- table-based record %Rowtype

%ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE
   identifier  reference%ROWTYPE;
```

```
DECLARE
 emp_record   employees%ROWTYPE;
 ...
```

PL/SQL Records

2- table-based record %Rowtype

Advantages of Using %ROWTYPE

- The number and data types of the underlying database columns need not be known—and, in fact, might change at run time.
- The %ROWTYPE attribute is useful when retrieving a row with the SELECT * statement.

TRABAJANDO CON COMPOSITE DATA TYPES

PL/SQL Records

2- table-based record %Rowtype

```
--using the %rowtype
declare

v_dept DEPARTMENTS%rowtype;

begin

select department_id,department_name,manager_id,location_id
into v_dept
from DEPARTMENTS  where department_id=10;

insert into copy_DEPARTMENTS values v_dept;
/*
insert into copy_DEPARTMENTS values (v_dept.department_id,v_dept.department_name,
*/

end;
```

TRABAJANDO CON COMPOSITE DATA TYPES

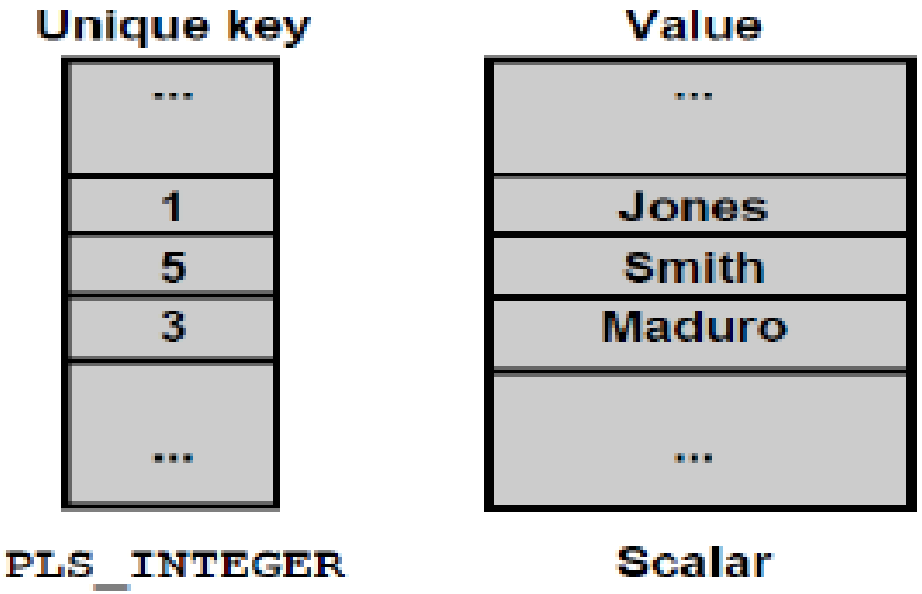
Composite Data Types

- Can hold multiple values (unlike scalar types)
- Are of two types:
  - PL/SQL records
  - PL/SQL collections
    - `INDEX BY` tables or associative arrays
    - Nested table
    - `VARRAY`

TRABAJANDO CON COMPOSITE DATA TYPES

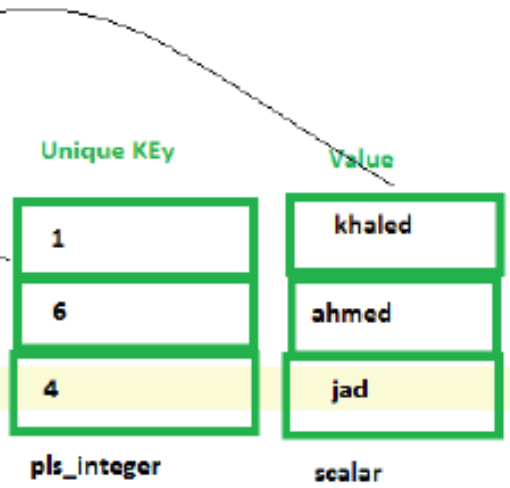
INDEX BY Tables or Associative Arrays

- Are PL/SQL structures with two columns:
  - Primary key of integer or string data type
  - Column of scalar or record data type
- Are unconstrained in size. However, the size depends on the values that the key data type can hold.



TRABAJANDO CON COMPOSITE DATA TYPES

```
declare

type tab_no is table of varchar2(100)
index by pls_integer;

v_tab_no tab_no;

begin

v_tab_no(1):='khaled';
v_tab_no(6):='ahmed';
v_tab_no(4):='jad';

dbms_output.put_line(v_tab_no(1));
dbms_output.put_line(v_tab_no(6));
dbms_output.put_line(v_tab_no(4));
end;
```



Syntax:

```
TYPE type_name IS TABLE OF
     {column_type | variable%TYPE
     | table.column%TYPE} [NOT NULL]
     | table%ROWTYPE
     [INDEX BY PLS_INTEGER | BINARY_INTEGER
      | VARCHAR2(<size>)];
identifier    type_name;
```

TRABAJANDO CON COMPOSITE DATA TYPES

Using INDEX BY Table Methods

The following methods make INDEX BY tables easier to use:

- EXISTS
- COUNT
- FIRST
- LAST
- PRIOR
- NEXT
- DELETE

Syntax: *table_name.method_name[(parameters)]*

Method	Description
EXISTS(*n*)	Returns TRUE if the *n*th element in a PL/SQL table exists
COUNT	Returns the number of elements that a PL/SQL table currently contains
FIRST	• Returns the first (smallest) index number in a PL/SQL table • Returns NULL if the PL/SQL table is empty
LAST	• Returns the last (largest) index number in a PL/SQL table • Returns NULL if the PL/SQL table is empty
PRIOR(*n*)	Returns the index number that precedes index *n* in a PL/SQL table
NEXT(*n*)	Returns the index number that succeeds index *n* in a PL/SQL table
DELETE	• DELETE removes all elements from a PL/SQL table. • DELETE(n) removes the *n*th element from a PL/SQL table. • DELETE(m, n) removes all elements in the range *m* ... *n* from a PL/SQL table.

TRABAJANDO CON COMPOSITE DATA TYPES

INDEX BY Table of Records

```
declare
type tab_no is table of employees%rowtype
index by pls_integer;

v_tab_no tab_no;
v_total number;

begin
v_tab_no(1).employee_id:=1;
v_tab_no(1).first_name:='ahmed';
v_tab_no(1).last_name:='jad';

v_tab_no(2).employee_id:=2;
v_tab_no(2).first_name:='khaled';
v_tab_no(2).last_name:='yaser';

dbms_output.put_line(v_tab_no(1).employee_id||v_tab_no(1).first_name||v_tab_no(1).last_name);
dbms_output.put_line(v_tab_no(2).employee_id||v_tab_no(2).first_name||v_tab_no(2).last_name);

end;
```

TRABAJANDO CON COMPOSITE DATA TYPES

Nested Tables

Example:

```
TYPE location_type IS TABLE OF locations.city%TYPE;
offices location_type;
```

- No index in nested table (unlike index by table)
- It is valid data type in SQL (unlike index by table, only used in PL/SQL)
- Initialization required
- Extend required
- Can be stored in DB

Syntax

```
TYPE type_name IS TABLE OF
 {column_type | variable%TYPE
 | table.column%TYPE} [NOT NULL]
 | table.%ROWTYPE
```

TRABAJANDO CON COMPOSITE DATA TYPES

Nested Tables

```
declare
type t_locations is table of varchar2(100);

loc t_locations;

begin

loc:=t_locations('jordan','uae','Syria');

dbms_output.put_line(loc(1) );
dbms_output.put_line(loc(2) );
dbms_output.put_line(loc(3) );

end;
```

VARRAY

```
declare
type t_locations is varray(3) of varchar2(100);

loc t_locations;

begin

loc:=t_locations('jordan','uae','Syria');

dbms_output.put_line(loc(1) );
dbms_output.put_line(loc(2) );
dbms_output.put_line(loc(3) );

end;
```

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Cursors

Every SQL statement executed by the Oracle server has an associated individual cursor:

- Implicit cursors: Declared and managed by PL/SQL for all DML and PL/SQL `SELECT` statements
- Explicit cursors: Declared and managed by the programmer

Explicit Cursor Operations

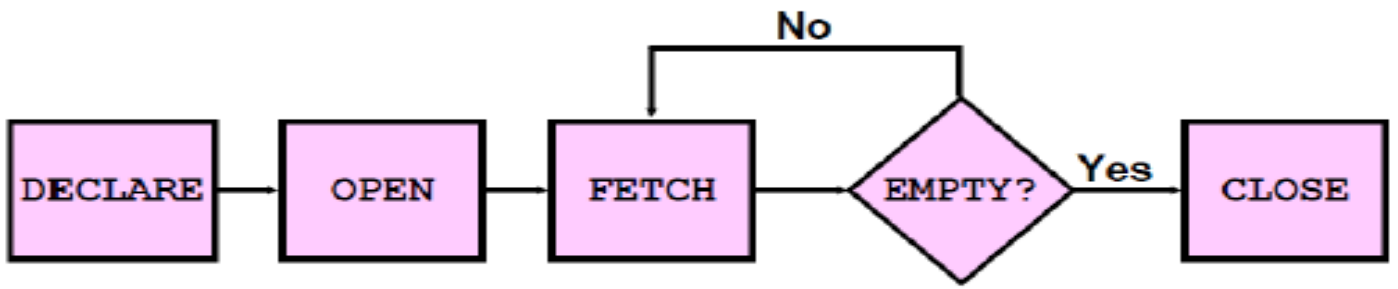
You declare explicit cursors in PL/SQL when you have a `SELECT` statement that returns multiple rows. You can process each row returned by the `SELECT` statement.

Explicit cursor functions:

- Can perform row-by-row processing beyond the first row returned by a query
- Keep track of the row that is currently being processed
- Enable the programmer to manually control explicit cursors in the PL/SQL block

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Controlling Explicit Cursors



- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to FETCH if rows are found.
- Release the active set.

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Explicit Cursor Attributes

Use explicit cursor attributes to obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Hints when declaring a cursor

- Do not include the `INTO` clause in the cursor declaration because it appears later in the `FETCH` statement.
- If processing rows in a specific sequence is required, use the `ORDER BY` clause in the query.
- The cursor can be any valid `SELECT` statement, including joins, subqueries, and so on.

PISTA

```
DECLARE
  CURSOR c_emp_dept20 is
  SELECT employee_id, first_name FROM employees
  where department_id=30
  Order by first_name;
```

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Hints when Opening a cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
...
BEGIN
  OPEN c_emp_cursor;
```

The OPEN statement executes the query associated with the cursor, identifies the active set, and positions the cursor pointer at the first row. The OPEN statement is included in the executable section of the PL/SQL block.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area
2. Parses the SELECT statement
3. Binds the input variables (sets the values for the input variables by obtaining their memory addresses)
4. Identifies the active set (the set of rows that satisfy the search criteria). Rows in the active set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows from the cursor to the variables.
5. Positions the pointer to the first row in the active set

Note: If a query returns no rows when the cursor is opened, PL/SQL does not raise an exception. You can find out the number of rows returned with an explicit cursor by using the <cursor_name>%ROWCOUNT attribute.

Hints when fetching data from a cursor

Fetching Data from the Cursor

The FETCH statement retrieves the rows from the cursor one at a time. After each fetch, the cursor advances to the next row in the active set. You can use the %NOTFOUND attribute to determine whether the entire active set has been retrieved.

The FETCH statement performs the following operations:

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

Hints when closing the Cursor

Closing the Cursor

The CLOSE statement disables the cursor, releases the context area, and "undefines" the active set. Close the cursor after completing the processing of the FETCH statement. You can reopen the cursor if required. A cursor can be reopened only if it is closed. If you attempt to fetch data from a cursor after it has been closed, then an INVALID_CURSOR exception will be raised.

Note: Although it is possible to terminate the PL/SQL block without closing cursors, you should make it a habit to close any cursor that you declare explicitly to free up resources.
There is a maximum limit on the number of open cursors per session, which is determined by the OPEN_CURSORS parameter in the database parameter file. (OPEN_CURSORS = 50 by default.)

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

```
DECLARE
  CURSOR c_emp_dept20 is
  SELECT employee_id, first_name FROM employees
  where department_id=30;

  v_empno employees.employee_id%type;
  v_first_name employees.first_name%type;

BEGIN
OPEN c_emp_dept20;

  loop
      fetch c_emp_dept20 into v_empno, v_first_name;

      exit when c_emp_dept20%notfound;

      dbms_output.put_line(v_empno||' '||v_first_name);
  end loop;

  close c_emp_dept20;

END;
```

Declaring the cursor

Opening the Cursor

fetching Data from cursor

exit the loop using cursor attributes

close the cursor

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Cursor FOR Loops

Syntax:

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
BEGIN
   FOR emp_record IN c_emp_cursor
    LOOP
     DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
     ||' ' ||emp_record.last_name);
    END LOOP;
END;
/
```

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Cursors with Parameters

```
DECLARE
  CURSOR   c_emp_cursor (deptno NUMBER) IS
   SELECT  employee_id, last_name
   FROM    employees
   WHERE   department_id = deptno;
   ...
BEGIN
  OPEN c_emp_cursor (10);
  ...
  CLOSE c_emp_cursor;
  OPEN c_emp_cursor (20);
  ...
```

Parameter data types are the same as those for scalar variables, but you do not give them sizes.

You can pass parameters to the cursor that is used in a cursor FOR loop:

```
DECLARE
CURSOR c_emp_cursor(p_deptno NUMBER, p_job VARCHAR2)IS
   SELECT ...
BEGIN
   FOR emp record IN c emp cursor(10, 'Sales') LOOP ...
```

CURSORES IMPLÍCITOS Y EXPLÍCITOS.

Using FOR UPDATE in Cursor

FOR UPDATE Clause

always last Statement in select

Syntax:

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

- Use explicit locking to deny access to other sessions for the duration of a transaction.
- Lock the rows *before* the update or delete.

WHERE CURRENT OF Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

- Use cursors to update or delete the current row.
- Include the `FOR UPDATE` clause in the cursor query to lock the rows first.
- Use the `WHERE CURRENT OF` clause to reference the current row from an explicit cursor.

```
UPDATE employees
   SET    salary = ...
   WHERE CURRENT OF c_emp_cursor;
```