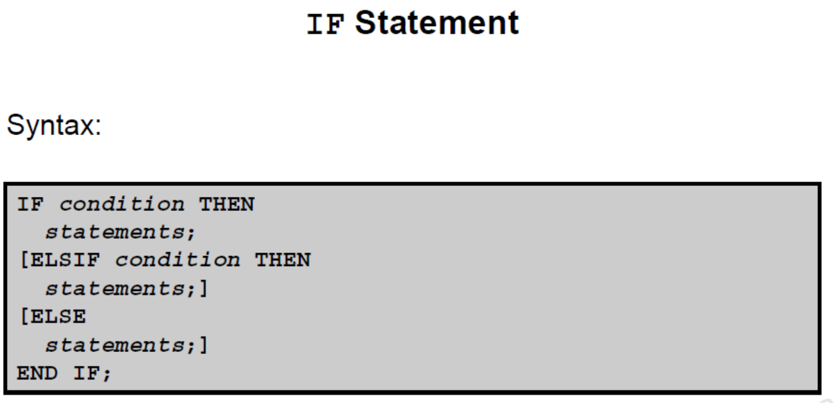
Controlling Flow of Execution:

IF Statement:

ELSIF Statement

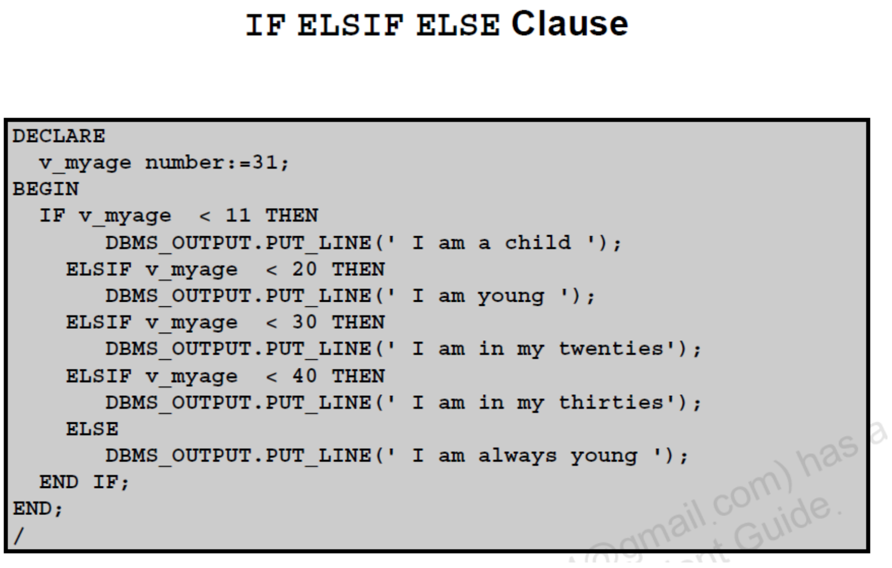


IF Statement

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Example,

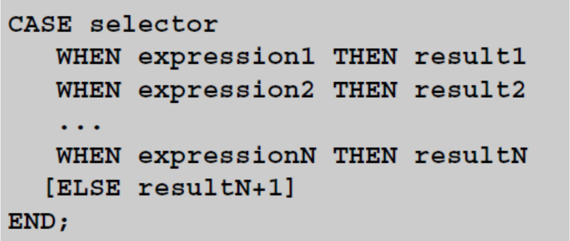


IF ELSIF ELSE Clause

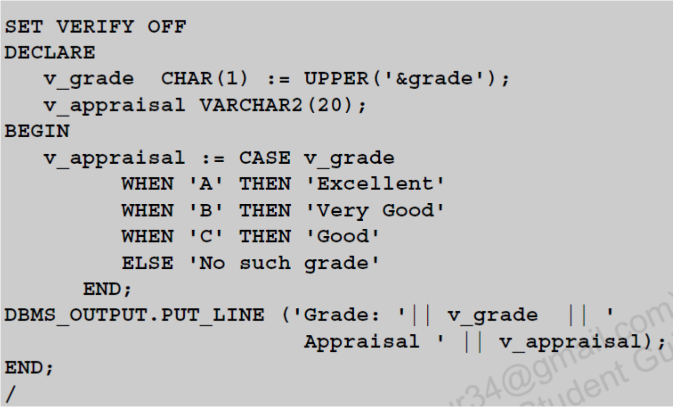
```
DECLARE
  v_myage number:=31;
BEGIN
  IF v_myage  < 11 THEN
       DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSIF v_myage  < 20 THEN
       DBMS_OUTPUT.PUT_LINE(' I am young ');
    ELSIF v_myage  < 30 THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
    ELSIF v_myage  < 40 THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
    ELSE
       DBMS_OUTPUT.PUT_LINE(' I am always young ');
  END IF;
END;
/
```

- IF STATEMENTS
- IF THEN ELSE → ELSE is default Clause
- IF ELSEIF ELSE Clause → ELSE is default Clause

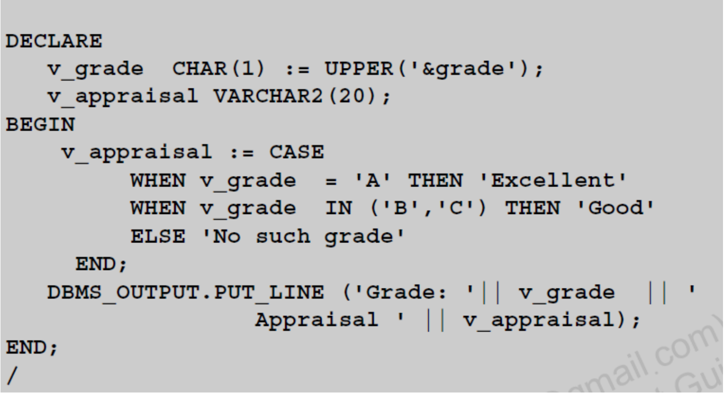
- CASE Expressions selects a result and return it.



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



```
SET VERIFY OFF
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   v_appraisal VARCHAR2(20);
BEGIN
   v_appraisal := CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade  || '
                       Appraisal ' || v_appraisal);
END;
/
```



```
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal := CASE
         WHEN v_grade  = 'A' THEN 'Excellent'
         WHEN v_grade  IN ('B','C') THEN 'Good'
         ELSE 'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade  || '
                  Appraisal ' || v_appraisal);
END;
/
```

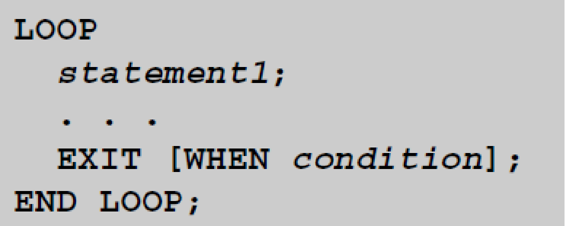
ELSE → Default case

Handling NULLs:

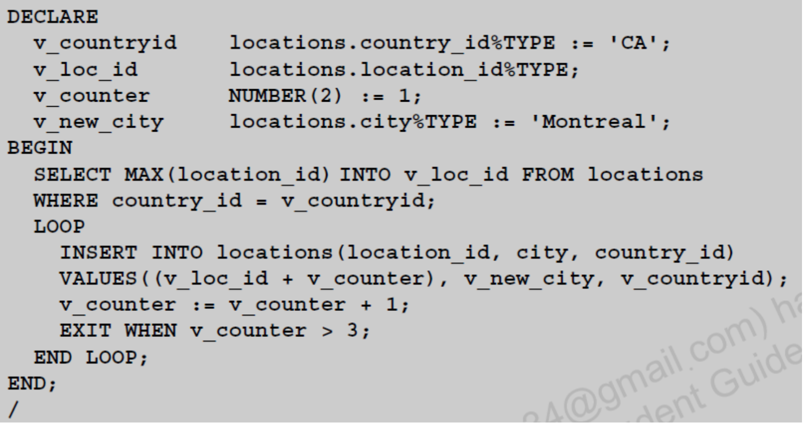
- Comparisons involving null always yields NULL
- Logical Operator NOT to a null always yields NULL
- If condition yields NULL, associated sequence of statements is not executed
- If both the conditions are TRUE → AND Operator gives TRUE
    - NULL AND FALSE = FALSE
- If any condition is TRUE → OR Operator gives TRUE

LOOPS:

1. Basic Loop: performs repetitive actions without conditions
   a. It allows the execution of its statements until the condition is met
   b. When we put condition first it executes at least once if it is true and never execute it again

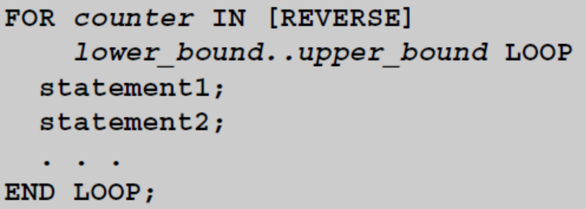


```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

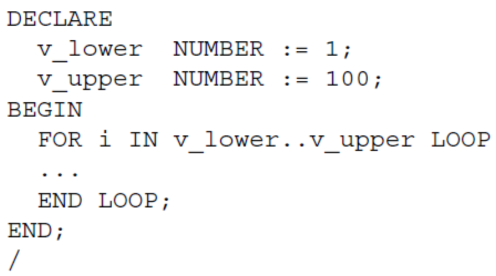


```
DECLARE
  v_countryid    locations.country_id%TYPE := 'CA';
  v_loc_id       locations.location_id%TYPE;
  v_counter      NUMBER(2) := 1;
  v_new_city     locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
  END LOOP;
END;
/
```

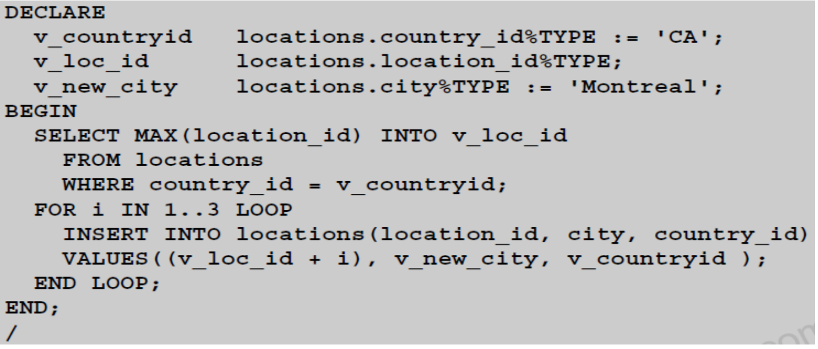
2. For Loop: Performs iterative actions based on counts
   a. To shortcut the test for the number of iterations
   b. Do not declare counter; it is declared implicitly
   c. Lower bound and Upper Bound are inclusive in the range
   d. Lower bound and upper bound of the rang can be literals, variables or expressions
   e. Reference the counter within the loop; it's undefined outside the loop
   f. Loop bound should not be NULL



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

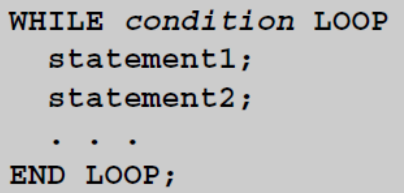


```
DECLARE
  v_lower  NUMBER := 1;
  v_upper  NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
  ...
  END LOOP;
END;
/
```

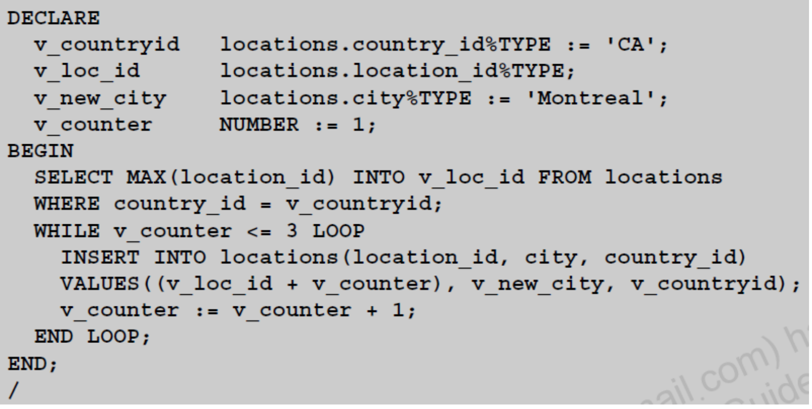


```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id
    FROM locations
    WHERE country_id = v_countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + i), v_new_city, v_countryid );
  END LOOP;
END;
/
```

3. While Loop: Performs iterative actions based on condition
   a. Condition is evaluated at the start of each iteration



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



```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
  v_counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

Usage of Loops:

- Basic loop when statement inside the loop must execute at least once
- While loop when the condition must be evaluated at the start of each iteration
- For loop when the number of iterations is known

Nested Loops:

- Use <<Outer_Loop>> and <<Inner_Loop>> identifiers to identify Outer and Inner Loop respectively
- Loop ending must be like: END LOOP Inner_Loop; and END LOOP Outer_Loop;

Continue Statement:

- What it does? – adds the functionality to begin the next loop iteration
- Provides ability to transfer control to next iteration of Loop
- Eases programming process
- Provides improvement to programming workarounds (Go to Page 175)

Chapter 6

Composite Data Types: (Can hold multiple values)

1. PL/SQL Records → Similar to Structures in third generation languages(C and C++ etc.)
    a. Used to treat related but dissimilar data as a logical unit (logical collective unit)
    b. Can have variable of different type
    c. Use Records when you want to store values of different data types but only one occurrence at a time.
    d. User Defined and Convenient for fetching a row of data from table
    e. Must contain one or more components

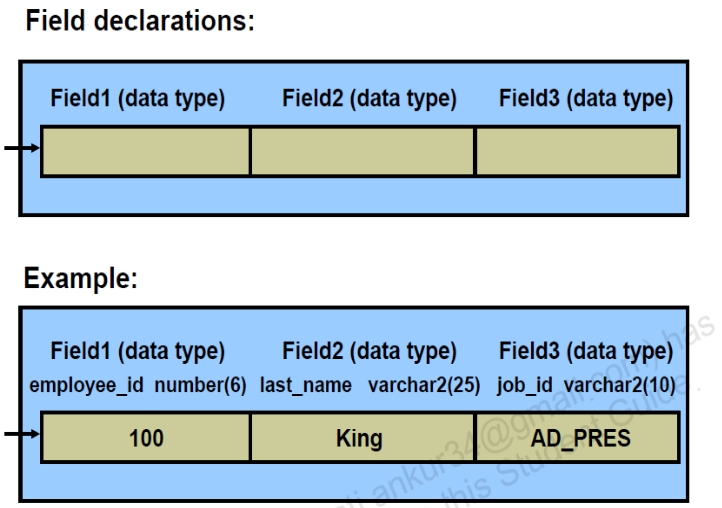
Syntax: TYPE type_name IS RECORD (field_declaration……);

Identifier type_name;

Field_declaration means:

1. Field_type variable%TYPE
2. Table.column%TYPE oe table%ROWTYPE

PL/SQL Record Structure

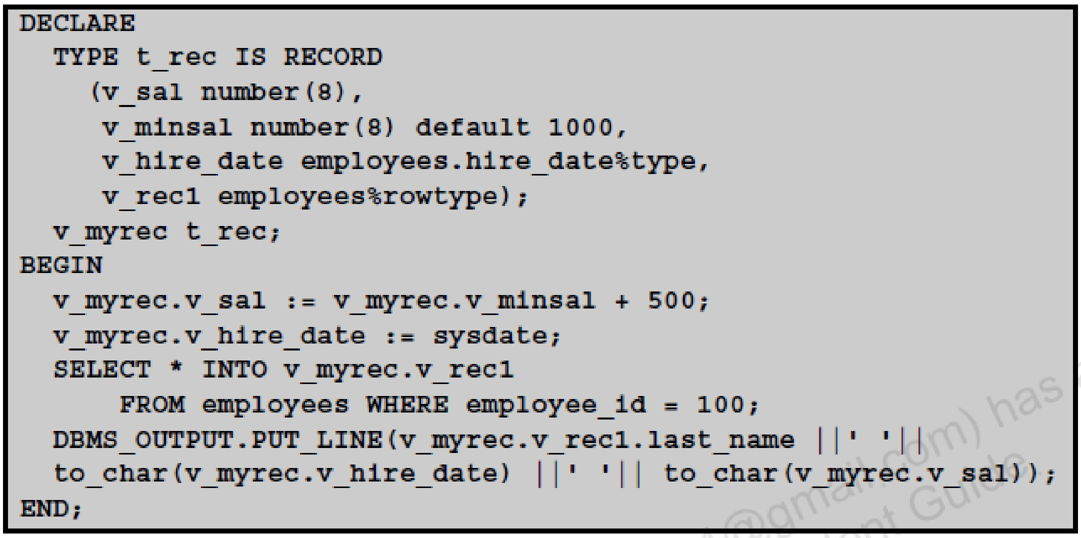


Note: to access the field name you have to user record name also. Example, **record_name.field_name** like **emp_record.job_id**

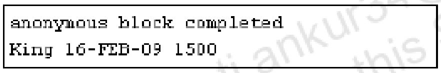
To assign value to the record field Example, **emp_recod.job_id = ‘ST_CLERK’;**

Why %ROWTYPE?

1. Declare a variable according to collection of columns in database table or view
2. Prefix %ROWTYPE with database table or view
3. When you change schema (Using DDL) → Changes are also reflected in DDL directly using
4. Syntax,
   a. identifier reference%ROWTYPE;
      i. identifier = record name
      ii. reference = name of table or view or cursor



```
DECLARE
  TYPE t_rec IS RECORD
    (v_sal number(8),
     v_minsal number(8) default 1000,
     v_hire_date employees.hire_date%type,
     v_rec1 employees%rowtype);
  v_myrec t_rec;
BEGIN
  v_myrec.v_sal := v_myrec.v_minsal + 500;
  v_myrec.v_hire_date := sysdate;
  SELECT * INTO v_myrec.v_rec1
      FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name ||' '||
  to_char(v_myrec.v_hire_date) ||' '|| to_char(v_myrec.v_sal));
END;
```



```
anonymous block completed
King 16-FEB-09 1500
```

Note:

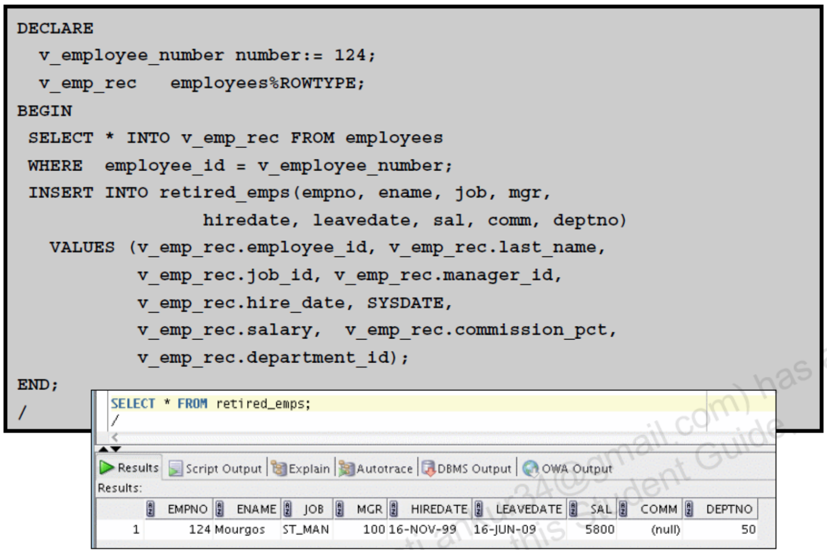
With %TYPE a field inherit the data type of specified column. With %ROWTYPE a field inherits the column name and data types of all columns in the reference table.

Advantages of %ROWTYPE:

1. Number and data types of the underlying database column need not to be known (when you are not sure about the structure of database table)
2. Its easier to retrieve row using SELECT * and you want to do row level INSERT and UPDATE

Example,

Another %ROWTYPE Attribute Example



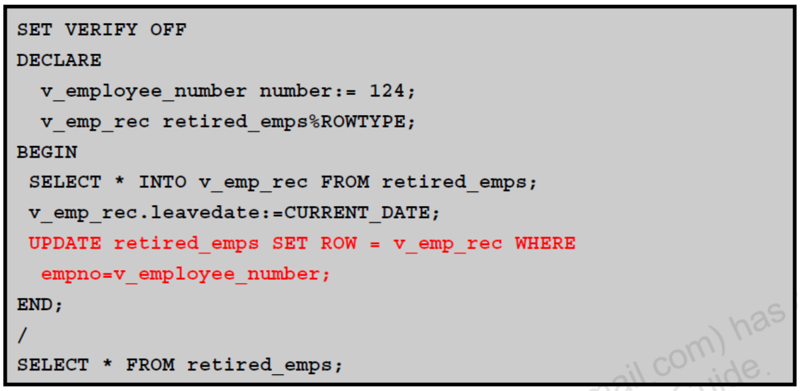
```
DECLARE
  v_employee_number number:= 124;
  v_emp_rec   employees%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps(empno, ename, job, mgr,
                hiredate, leavedate, sal, comm, deptno)
   VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
           v_emp_rec.job_id, v_emp_rec.manager_id,
           v_emp_rec.hire_date, SYSDATE,
           v_emp_rec.salary,  v_emp_rec.commission_pct,
           v_emp_rec.department_id);
END;
/
```

```
SELECT * FROM retired_emps;
/
```

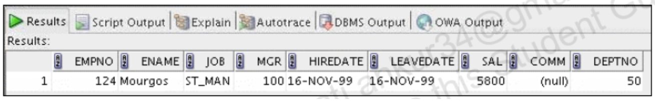
Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-JUN-09	5800	(null)	50

Updating a Row in a Table by Using a Record



```
SET VERIFY OFF
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM retired_emps;
 v_emp_rec.leavedate:=CURRENT_DATE;
 UPDATE retired_emps SET ROW = v_emp_rec WHERE
  empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```



Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

PL/SQL Records: (Page 187):

Run files: code_06_14_n-s → code_06_15_s → code_06_16_s → Learn what's happening.

2. PL/SQL Collection → Arrays
    a. It's a collection with two columns:
        i. PK of integer or string data type
        ii. Column of scalar or record type (data type)

- Used to treat the data as a single unit.
- Use collections when you want to store the values of the same data type
- There are three type of collections:
    a. Associative Array
    b. Nested Table
    c. VArray (Variable Array)

IMP → %TYPE and %ROWTYPE

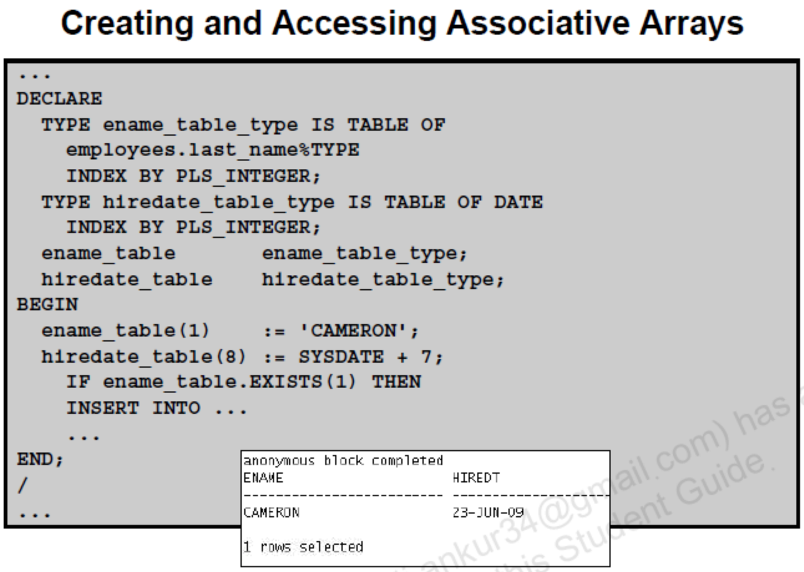
Why Use View? → to reduce the numbers of hits on a disk. Just Create a view (Snapshot of a table) and use that for scanning. Only Materialized views can be edited,

a. Associative Arrays (aka INDEX BY Tables):

- PLS_INTEGER → or BINARY INTEGER → needs less storage space than NUMBER → It's faster in Calculations than Normal INT.
- We can not initialize associative array in declaration. To populate it we have to write explicit executable statement.
- **Storage Structures:**

DB → TABLESPACE → SEGMENT → EXTENTS → ORACLE DATABLOCKS (It's responsible to save data on disk aka hitting the disk)

- There are two type of scans FTS (Full Table Scan) and INDEX BY Scan
- To optimize the quarries, we use index by scan → (Saves money)



Creating and Accessing Associative Arrays

```
...
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY PLS_INTEGER;
  ename_table       ename_table_type;
  hiredate_table    hiredate_table_type;
BEGIN
  ename_table(1)    := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
    ...
END;
/
...
```

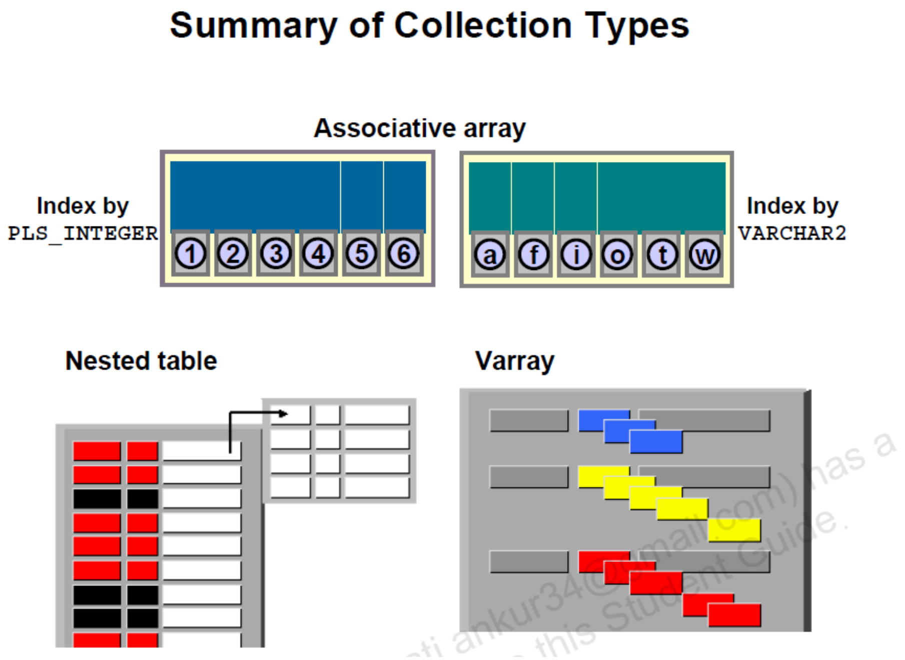
```
anonymous block completed
ENAME                     HIREDT
------------------------- ---------
CAMERON                   23-JUN-09

1 rows selected
```

b. Nested Tables: (Only Definition)

- It can store max of 2 GB of Data (Because it’s variable type)
- Is valid in schema level table
- Nested Table is a valid data type in schema level table but an associative array is not. Therefore, nested tables can be stored in the database where associative array can not be stored in database.
- Size of nested tables can be increased dynamically

c. VARRAY:

- A variable size array is similar to associative array, except that the VARRAY is constrained in a size.
- Valid in a schema level table
- Items inside the Varray is called VARRAYs.
- The maximum size is 2 GB



Summary of Collection Types

Associative Arrays

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be either integer- or character-based. The array value may be of the scalar data type (single value) or the record data type (multiple values).

Because associative arrays are intended for storing temporary data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`.

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, thereby allowing varrays to be stored in database tables.