Table of Contents

[CHAPTER 7: COMPOSITE DATA TYPES 2](#_Toc416701183)

[Theory 2](#_Toc416701184)

[AIM 5](#_Toc416701185)

[Lab Exercise 7: COMPOSITE DATA TYPES 6](#_Toc416701186)

[1. Record 7](#_Toc416701187)

[2. Query and DML with Record 13](#_Toc416701188)

[3. Associative Array/INDEX-BY Table 20](#_Toc416701189)

[4. VARRAY 24](#_Toc416701190)

[5. Nested Table 28](#_Toc416701191)

[SUMMARY 32](#_Toc416701192)

[REFERENCES 33](#_Toc416701193)

[INDEX 34](#_Toc416701194)

# CHAPTER 7: COMPOSITE DATA TYPES

## Theory

In this chapter, we will explore the second part of data types: Composite data types. In chapter 3: Variable Declaration, Data Type and Expression, you knew that Oracle data type consists mainly of two broad categories: (1) Scalar Data types and (2) Composite Data types. Scalar Data types are those atomic data types which do not contain sub-components. Composite Data Types, in contrast, are those data types which contain sub-components.

There are two main categories of Composite Data Types: (1) Records, (2) Collections. For simplicity, you can perceive Record data type as row and Collection data type as column. Thus, a Record may contain fields of different data types, while a Collection consists of exactly one data type. To access fields inside a record, you should use "." notation as RECORD\_VARIALE.FIELD\_NAME. You are familiar with Record data type because you already saw it when you used %ROWTYPE in a previous chapters. In contrast, you should use an index syntax to reach an element in a Collection as COLLECTION\_VARIABLE (INDEX).

Day(3)

Day(2)

Day(1)

**Day**  
Collection

(3)

(2)

(1)

Emp.FName

Emp.BOD

Emp.Id

**Emp**   
Record

FName (CHAR)

DOB (DATE)

Id (INTEGER)

There are three types of Collections:

1. **Associative Array/ Index-by Table**:   
   It is an Oracle PL/SQL version of Hash-Table. If you are not failure with Hash-Table concept, please visit this link: <http://en.wikipedia.org/wiki/Hash_table>

Associative Array stores data in key-value pair. A key represents an index and it can be either string or PLS\_INTEGER.

Use Associative Array for:

* A relatively small lookup table, because you load the associative array in memory each time you invoke a PL/SQL block.
* Passing collections to and from the database server

1. **VARRAY**:  
   Stands for Variable Array. It corresponds to the array data structure or vector. The main limitation of VARRAY is that you need to specify its size in declaration. Thus, use VARRAY when:

* You know the maximum number of elements.
* You usually access the elements sequentially.

1. **Nested Table**:

Nested Table corresponds to a table column. You don't have to specify a Nested Table size in declaration; it is dynamically allocate elements. However, it does not guarantee the order of the elements. Thus, as you intensively add, delete, and retrieve elements from a Nested Table, you may find that an element "A" which was at index "5", is now at index "7". Therefore, use Nested Table when:

* The number of elements is not set.
* Index values are not consecutive.
* You must delete or update some elements, but not all elements simultaneously.
* You would create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

To use Composite Data Types (Records and Collections) follow the steps shown below:

This step is a required for **VARRAY** and **Nested Table** collection. You must initialize their elements before use them.

Create a variable of the type declared in the previous step. You can define as many variables of the same type as you want.

Initialize Variable

Declare Variable

Declare Type

Use "." notation for Record and (INDEX) notation for collection data type.

You should declare a structure of a Record/Collection as either **local type**, **public item** or **standalone type**.

Use Variable

## AIM

The AIM of the following exercise is to demonstrate the basics of using composite data types (Record and Collection).

The steps involved will include:

* Record
* Query and DML with Record
* Associative Array/INDEX-BY Table
* VARRAY
* Nested Table

In general, lab exercises are done in sequential order. Thus, it is assumed that you successfully completed the previous labs. However, not all previous labs are required. Please be sure to run the following lab before proceeding:

* Installing Oracle Database 12c.

Estimated Completion Time:

30 minutes

# Lab Exercise 7: COMPOSITE DATA TYPES

|  |
| --- |
|  |

## Record

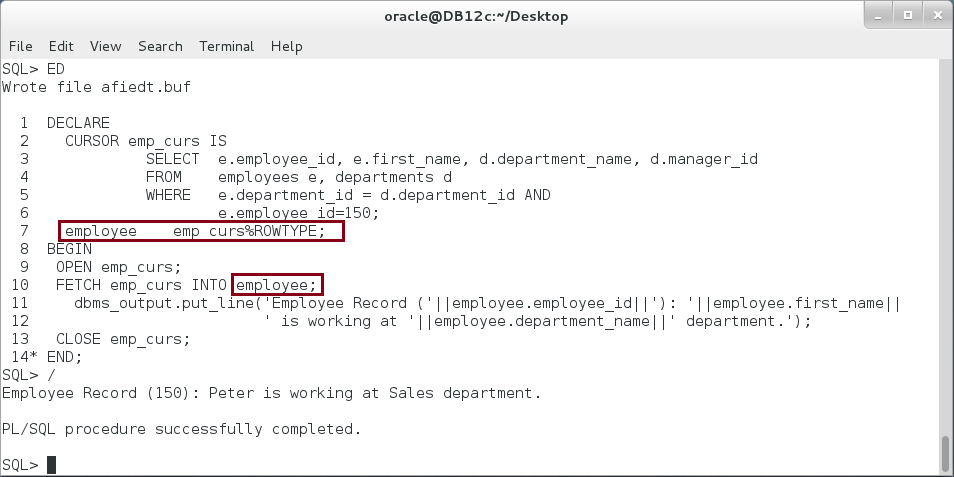
**Step 1:** Open the Terminal, open SQL\*Plus console and connect to hr schema.

|  |  |
| --- | --- |
| Command | Description |
| sqlplus | Open SQL\*Plus console. |
| hr/oracle | connect to **hr** schema. |

****

**Step 2:** Execute the following block:

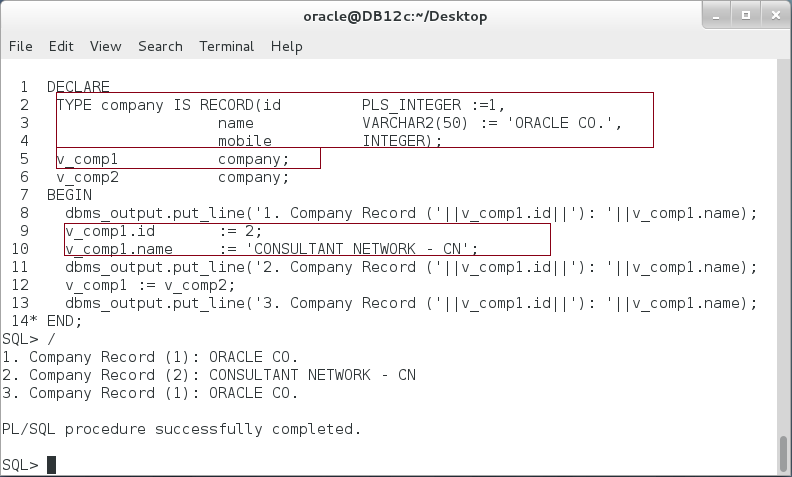
|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| CURSOR emp\_curs IS |  |
| SELECT e.employee\_id, e.first\_name,  d.department\_name, d.manager\_id |  |
| FROM employees e, departments d |  |
| WHERE e.department\_id = d.department\_id AND |  |
| e.employee\_id=150; |  |
| **employee emp\_curs%ROWTYPE;** |  |
| BEGIN |  |
| OPEN emp\_curs; |  |
| FETCH emp\_curs INTO **employee;** |  |
| dbms\_output.put\_line('Employee Record ('||  **employee.**employee\_id||  '): '||**employee.**first\_name||' is working at '||  **employee.**department\_name|| ' department.'); |  |
|  |
| CLOSE emp\_curs; |  |
| END; |  |
| / |  |

****

**Please note**: You already familiar with a cursor-based record. Using "ROWTYPE" cursor attribute, you create "employee" record. Note

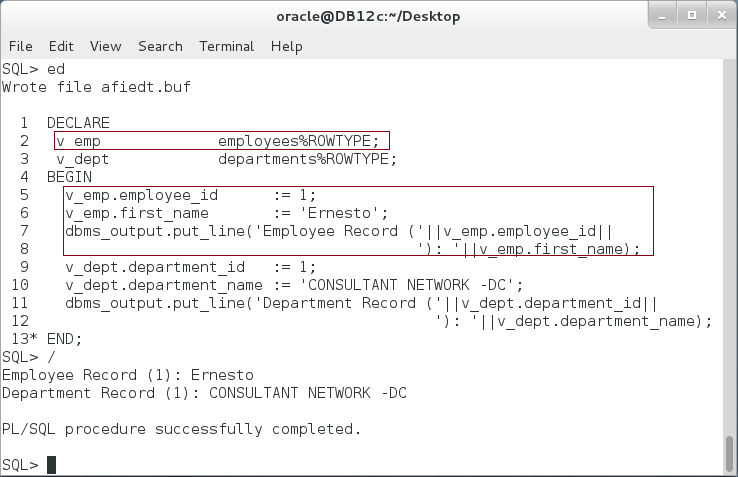
**Step 3:** You can create your own record type as show below:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| TYPE company IS RECORD  (id PLS\_INTEGER :=1,   name VARCHAR2(50) := 'ORACLE CO.',  mobile INTEGER); | **(1) Declare Type**  User-Defined record "**company**" |
| v\_comp1 company; | **(2) Declare Variable** |
| v\_comp2 company; |
| BEGIN |  |
| dbms\_output.put\_line  ('1. Company Record ('||v\_comp1.id||'): '||  v\_comp1.name); | **(3) Use Variable** |
| v\_comp1.id := 2; | Assign value to Record field |
| v\_comp1.name := 'CONSULTANT NETWORK - CN'; |
| dbms\_output.put\_line  ('2. Company Record ('||v\_comp1.id||'): '||  v\_comp1.name); |  |
| v\_comp1 := v\_comp2; | **Assign Record variable to other Record variable of the same type.** |
| dbms\_output.put\_line  ('3. Company Record ('||v\_comp1.id||'): '||  v\_comp1.name); | **Explain the output** |
| END; |  |
| / |  |



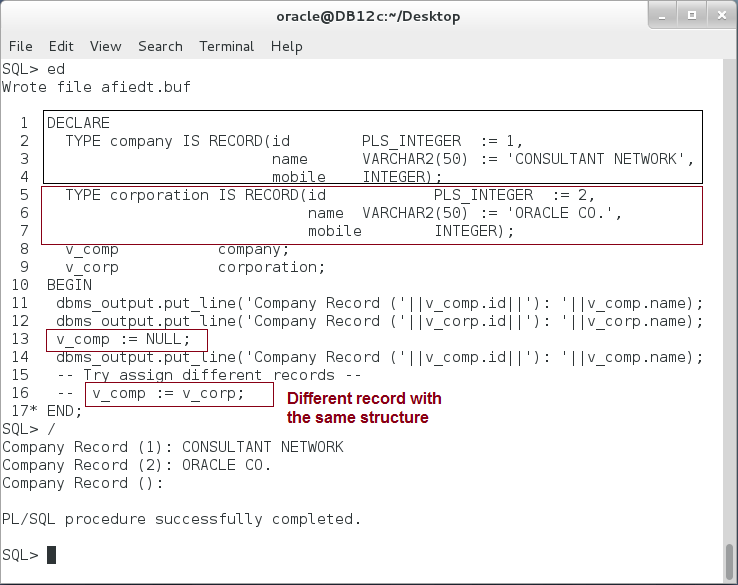
**Step 4:** The third technique used to create a record is by defining a record based on table or view as show below:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| v\_emp employees%ROWTYPE; | **Define Record variable** based on Table |
| v\_dept departments%ROWTYPE; |
| BEGIN |  |
| v\_emp.employee\_id := 1; | **Note a "." notation** |
| v\_emp.first\_name := 'Ernesto'; |
| dbms\_output.put\_line  ('Employee Record ('||v\_emp.employee\_id|| | **Explain the output.** |
| '): '||v\_emp.first\_name); |
| v\_dept.department\_id := 1; |  |
| v\_dept.department\_name := 'CONSULTANT NETWORK -DC'; |  |
| dbms\_output.put\_line  ('Department Record ('||v\_dept.department\_id||'): ' | **Explain the output.** |
| ||v\_dept.department\_name); |
| END; |  |
| / |  |

****

**Step 5:** What if we try to assign NULL to a Record variable? What about assigning different Record variables with the same sub-types? Execute the following block:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| TYPE company IS RECORD  (id PLS\_INTEGER := 1,  name VARCHAR2(50) := 'CONSULTANT NETWORK',  mobile INTEGER); | **First Record** |
| TYPE corporation IS RECORD  (id PLS\_INTEGER := 2,  name VARCHAR2(50) := 'ORACLE CO.',  mobile INTEGER); | **Second Record** |
| v\_comp company; |  |
| v\_corp corporation; |  |
| BEGIN |  |
| dbms\_output.put\_line('Company Record ('||v\_comp.id||'): '||v\_comp.name); |  |
| dbms\_output.put\_line('Company Record ('||v\_corp.id||'): '||v\_corp.name); |  |
| **v\_comp := NULL;** |  |
| dbms\_output.put\_line('Company Record ('||v\_comp.id||'): '||v\_comp.name); |  |
| -- Try assign different records -- | **Try and explain** |
| -- **v\_comp := v\_corp;** |
| END; |  |
| / |  |

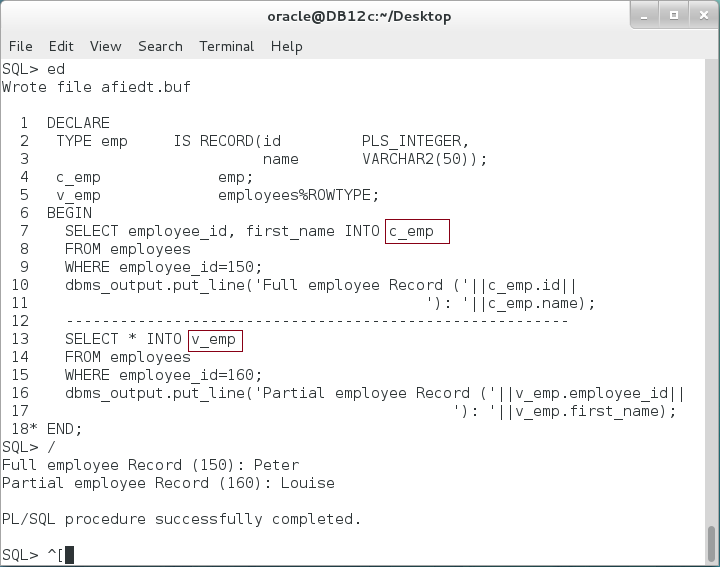


**Please note**: Oracle does not allow the assignment operation between two different types unless there exists an implicit conversion between them.

## Query and DML with Record

**Step 1:** Execute the following PL/SQL block:

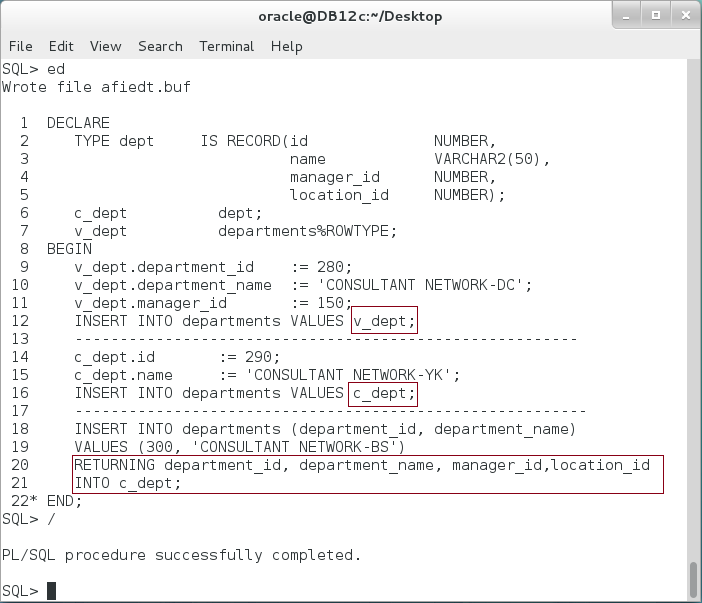
|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| TYPE emp IS RECORD  (id PLS\_INTEGER,   name VARCHAR2(50)); | User-Defined Record (**USR**) |
| c\_emp emp; | UDR variable |
| v\_emp employees%ROWTYPE; | Table-Based Record |
| BEGIN |  |
| SELECT **employee\_id, first\_name** INTO **c\_emp** | **Partial record assignment using UDR** |
| FROM employees |
| WHERE employee\_id=150; |
| dbms\_output.put\_line  ('Full employee Record ('||c\_emp.id||'):  '||c\_emp.name); |  |
| SELECT **\*** INTO **v\_emp** | **Full record assignment using Table-based Record** |
| FROM employees |
| WHERE employee\_id=160; |
| dbms\_output.put\_line  ('Partial employee Record ('||v\_emp.employee\_id  ||'):'  ||v\_emp.first\_name); |  |
| END; |  |
| / |  |



**Please note:** a UDR can be used to handle a partial assignment of a row.

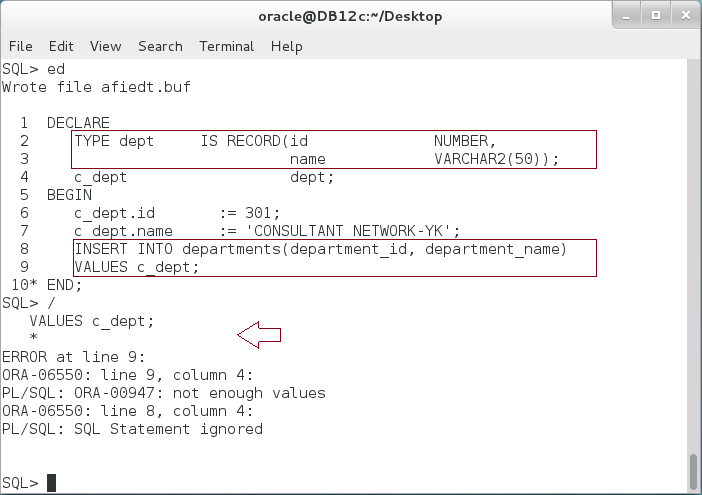
**Step 2:** UDR and Table-Based record can also be used in DML. Execute the following block:

|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| TYPE dept IS RECORD(id NUMBER, | UDR exactly as DEPARTMENTS row. |
| name VARCHAR2(50), |
| manager\_id NUMBER, |
| location\_id NUMBER); |
| c\_dept dept; |  |
| v\_dept departments%ROWTYPE; |  |
| BEGIN |  |
| v\_dept.department\_id := 280; | Use Table-Based Record in INSERT statement. |
| v\_dept.department\_name := 'CONSULTANT NETWORK-DC'; |
| v\_dept.manager\_id := 150; |
| INSERT INTO departments VALUES v\_dept; |
| c\_dept.id := 290; | Use UDR in INSERT statement. |
| c\_dept.name := 'CONSULTANT NETWORK-YK'; |
| INSERT INTO departments VALUES c\_dept; |
| INSERT INTO departments  (department\_id, department\_name)  VALUES (300, 'CONSULTANT NETWORK-BS')  RETURNING department\_id, department\_name,   manager\_id,location\_id  INTO c\_dept; | Use Table-Based Record/UDR in a RETURNING INTO |
| END; |  |
| / |  |



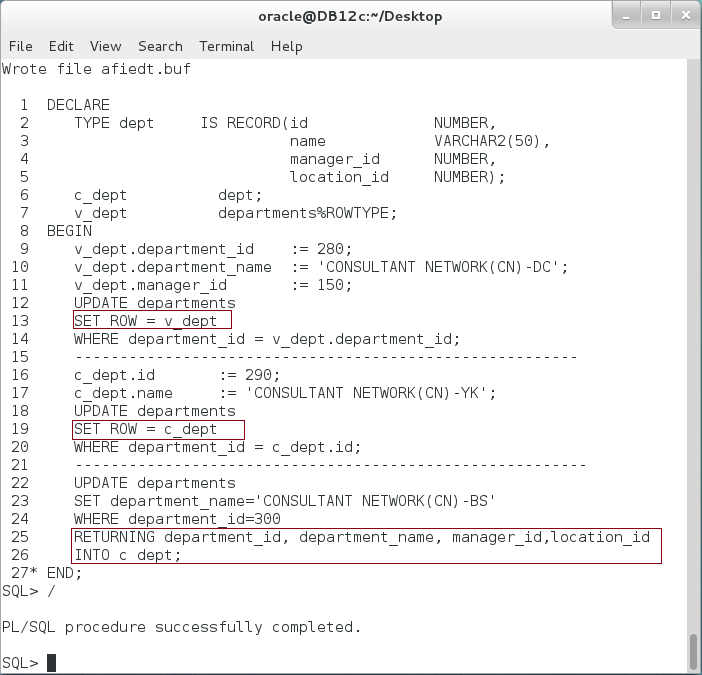
**Please note**: In the previous block, UDR and Table-Based Record represent a full row of DEPARTMENTS table. What about a partial record?

**Step 3:** Modify the previous block, so the UDR represents only two columns of DEPARTMENTS row. Try using UPDATE with partial row as show below:



**Step 4:** Variable cursors can be defined with a text variable as shown below:

|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| TYPE dept IS RECORD(id NUMBER, |  |
| name VARCHAR2(50), |  |
| manager\_id NUMBER, |  |
| location\_id NUMBER); |  |
| c\_dept dept; |  |
| v\_dept departments%ROWTYPE; |  |
| BEGIN |  |
| v\_dept.department\_id := 280; | Use Table-Based Record in UPDATE statement |
| v\_dept.department\_name := 'CONSULTANT NETWORK(CN)-DC'; |
| v\_dept.manager\_id := 150; |
| **UPDATE** departments |
| **SET ROW = v\_dept** |
| WHERE department\_id = v\_dept.department\_id; |
| c\_dept.id := 290; | Use UDR in UPDATE statement |
| c\_dept.name := 'CONSULTANT NETWORK(CN)-YK'; |
| **UPDATE** departments |
| **SET ROW = c\_dept** |
| WHERE department\_id = c\_dept.id; |
| **UPDATE** departments | Use UDR/Table-Based Record in RETURNING INTO |
| SET department\_name='CONSULTANT NETWORK(CN)-BS' |
| WHERE department\_id=300 |
| **RETURNING** department\_id, department\_name,  manager\_id,location\_id |
| **INTO** c\_dept; |
| END; |  |

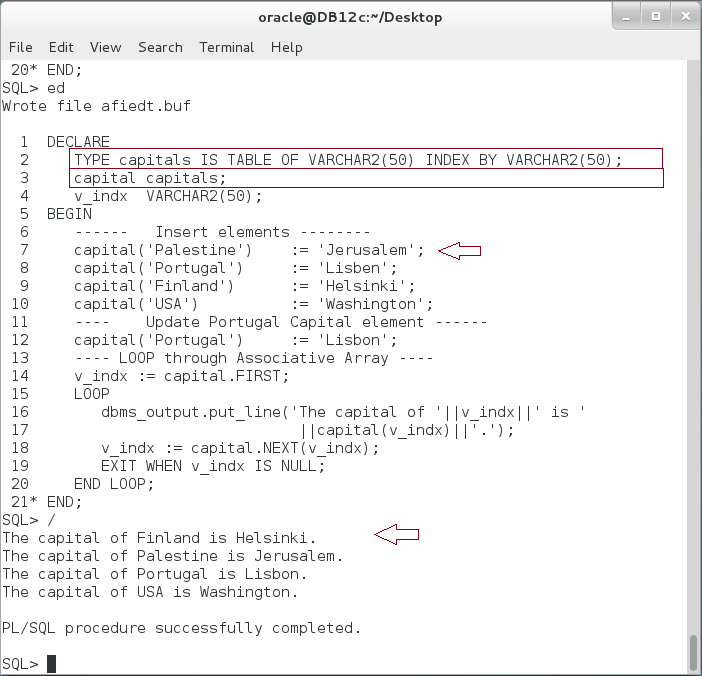
****

**Please Note:** The syntax of UPDATE statement has been modified, so the keyword ROW must appear on the left side of assignment and a Record on the right side. **Can we use a partial row assignment here?**

## Associative Array/INDEX-BY Table

**Step 1:** The last method used is an Explicit FOR LOOP cursor. It is quite similar to an implicit FOR LOOP cursor but its SQL statement defined outside FOR LOOP as show below:

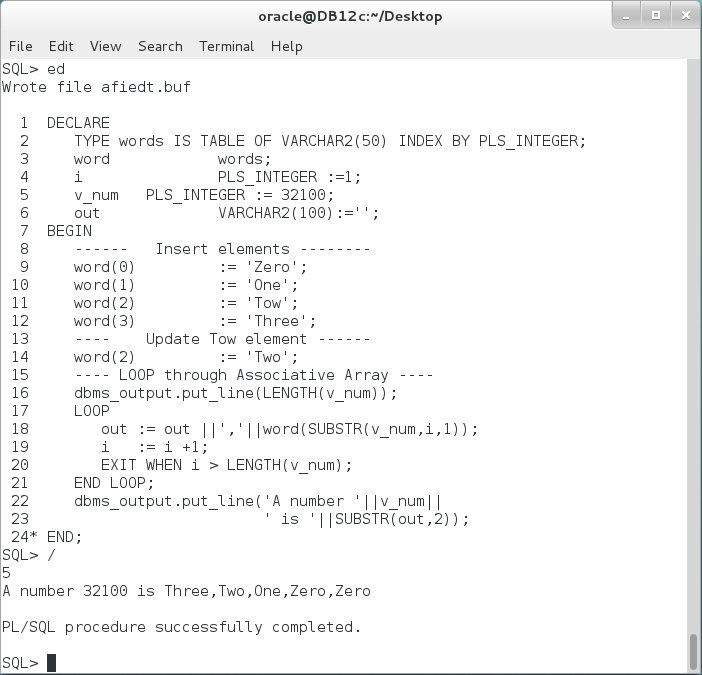
|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| TYPE capitals IS TABLE OF VARCHAR2(50)  INDEX BY VARCHAR2(50); | **(1) Declare Associative Array** |
| capital capitals; | **(2) Declare Variable** |
| v\_indx VARCHAR2(50); |  |
| BEGIN |  |
| ------ Insert elements -------- | **Set Variable** |
| capital('Palestine') := 'Jerusalem'; |
| capital('Portugal') := 'Lisben'; |
| capital('Finland') := 'Helsinki'; |
| capital('USA') := 'Washington'; |
| ---- Update Portugal Capital element ------ | **Update Variable** |
| capital('Portugal') := 'Lisbon'; |
| ---- LOOP through Associative Array ---- |  |
| v\_indx := capital.**FIRST**; | **Loop through Associative Array** |
| LOOP |
| dbms\_output.put\_line('The capital of '||v\_indx||' is '||capital(v\_indx)||'.'); |
| v\_indx := capital.**NEXT**(v\_indx); |
| EXIT WHEN v\_indx IS NULL; |
| END LOOP; |
| END; |  |

****

**Please note:** The order of elements in Associative Array is unpredictable. In a previous example, the index is a string. This feature is applicable only for Associative Array. You may also note a collection attributes FIRST and NEXT. These attributes will be covered in the next chapter.

**Step 2:** Again, Oracle recommends referring variables by parameters not by handwriting them inside the query. Modify the previous block as shown below:

|  |  |
| --- | --- |
| Command | Description |
| DECLARE |  |
| TYPE words IS TABLE OF VARCHAR2(50)  INDEX BY PLS\_INTEGER; | **Associative Array index by INTEGER** |
| word words; |  |
| i PLS\_INTEGER :=1; |  |
| v\_num PLS\_INTEGER := 32100; |  |
| out VARCHAR2(100):=''; |  |
| BEGIN |  |
| ------ Insert elements -------- | **Set Array value** |
| word(0) := 'Zero'; |
| word(1) := 'One'; |
| word(2) := 'Tow'; |
| word(3) := 'Three'; |
| ---- Update Tow element ------ |  |
| word(2) := 'Two'; |  |
| ---- LOOP through Associative Array ---- |  |
| dbms\_output.put\_line(LENGTH(v\_num)); |  |
| LOOP | The value of Associative Array is in its performance to call unordered items. |
| out := out ||','||word(SUBSTR(v\_num,i,1)); |
| i := i +1; |
| EXIT WHEN i > LENGTH(v\_num); |
| END LOOP; |
| dbms\_output.put\_line('A number '||v\_num|| |
| ' is '||SUBSTR(out,2)); |
| END; |  |
| / |  |

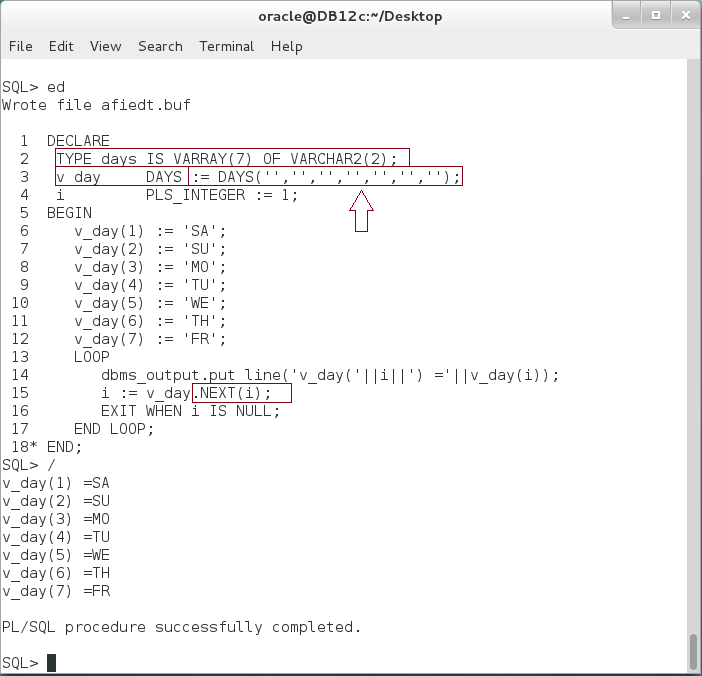


**Please note:** Associative Array is best used to allocate item in an unordered sequence such as in interpreter and syntax checker. .

## VARRAY

**Step 1:** The last

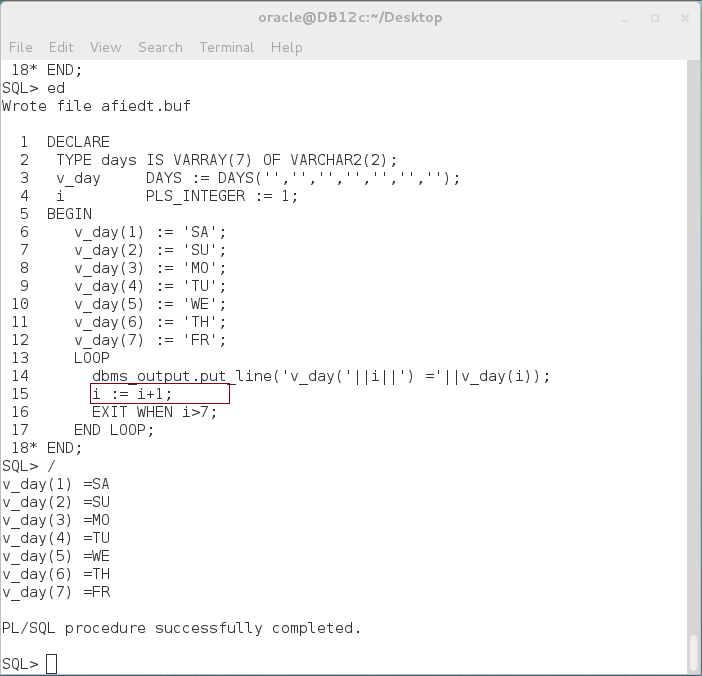
|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| TYPE days IS VARRAY(7) OF VARCHAR2(2); | **(1) Declare VARRAY** |
| v\_day DAYS := DAYS('','','','','','',''); | **(2) Declare variable**  **(3) Initialize variable** |
| i PLS\_INTEGER := 1; |  |
| BEGIN |  |
| v\_day(1) := 'SA'; | **(4) Use variable** |
| v\_day(2) := 'SU'; |
| v\_day(3) := 'MO'; |
| v\_day(4) := 'TU'; |
| v\_day(5) := 'WE'; |
| v\_day(6) := 'TH'; |
| v\_day(7) := 'FR'; |
| LOOP |
| dbms\_output.put\_line('v\_day('||i||') ='||v\_day(i)); |  |
| **i := v\_day.NEXT(i);** |  |
| EXIT WHEN i IS NULL; |  |
| END LOOP; |  |
| END; |  |
| / |  |



**Please Note:** You must initialize VARRAY variable. Try to remove the initialization step from the previous block. Also note how NEXT attribute is used to navigate through array.

**Step 2:** Modify the previous block so NEXT is replaced by i+1 and note the output.

|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| TYPE days IS VARRAY(7) OF VARCHAR2(2); |  |
| v\_day DAYS := DAYS('','','','','','',''); |  |
| i PLS\_INTEGER := 1; |  |
| BEGIN |  |
| v\_day(1) := 'SA'; |  |
| v\_day(2) := 'SU'; |  |
| v\_day(3) := 'MO'; |  |
| v\_day(4) := 'TU'; |  |
| v\_day(5) := 'WE'; |  |
| v\_day(6) := 'TH'; |  |
| v\_day(7) := 'FR'; |  |
| LOOP |  |
| dbms\_output.put\_line('v\_day('||i||') ='||v\_day(i)); |  |
| **i := i+1;** |  |
| EXIT WHEN i>7; |  |
| END LOOP; |  |
| END; |  |
| / |  |

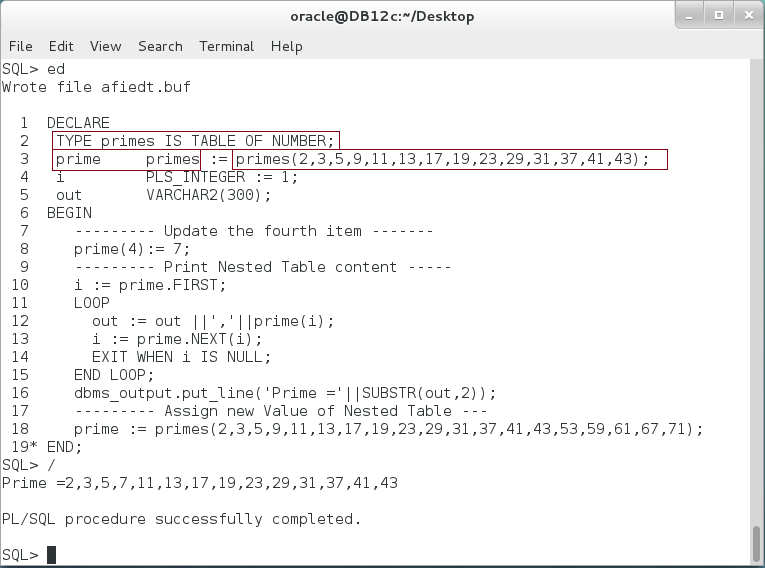
****

**Please Note:** VARRAY guarantees the order of elements to remain as it was. This feature is applicable only for VARRAY. Therefore, use VARRAY when the order of the element has meaning. In addition, VARRAY is always **dense**, so the next item is always one-step away from the previous step: i+1.

## Nested Table

**Step 1:** Execute the following block:

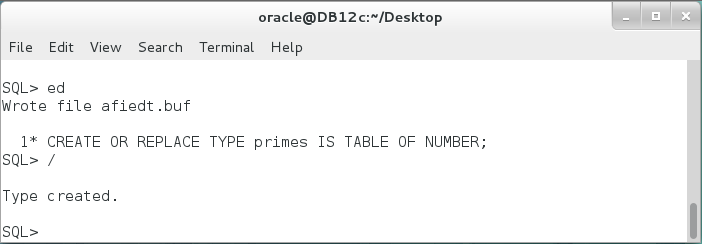
|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| TYPE primes IS TABLE OF NUMBER; | **(1) Declare Type** |
| prime primes := primes(2,3,5,9,11,13,17,19,  23,29,31,37,41,43); | **(2) Declare Variable**  **(3) Initialize Variable** |
| i PLS\_INTEGER := 1; |  |
| out VARCHAR2(300); |  |
| BEGIN | The order of the elements inside a Nested Table is not predictable. |
| --------- Update the fourth item ------- |
| prime(4):= 7; |
| --------- Print Nested Table content ----- |
| i := prime.**FIRST**; |
| LOOP |
| out := out ||','||prime(i); |
| i := prime.**NEXT**(i); |
| EXIT WHEN i IS NULL; |  |
| END LOOP; |  |
| dbms\_output.put\_line('Prime ='||SUBSTR(out,2)); |  |
| --------- Assign new Value of Nested Table --- |  |
| prime := **primes**(2,3,5,9,11,13,17,19,23,  29,31,37,41,43,53,59,61,67,71); | Use type constructor to assign value to Nested Table variable |
| END; |  |
| / |  |

****

**Please note:** The size of the Nested Table is not specified; it is dynamic. The initialization step is must. The order of the elements is not guaranteed in Nested Table and it is not dense as VARRAY. Therefore, you must use NEXT or PRIOR attributes to navigate through Nested Table.

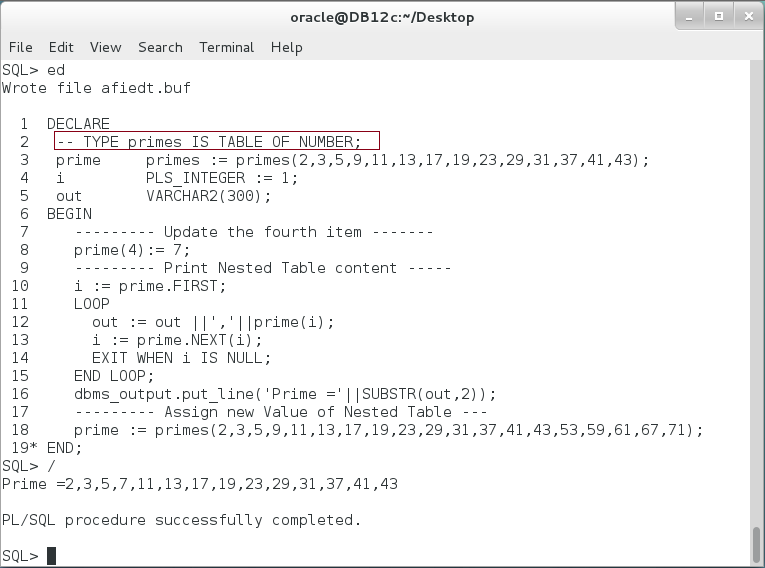
**Step 2:** Nested Table can be created as **standalone** type, so you can use it in any block later without declaration. Create **primes** standalone Nested Table as the following:

|  |  |
| --- | --- |
| Line | Description |
| CREATE OR REPLACE TYPE primes IS TABLE OF NUMBER; | Create a **Standalone** Nested Table |

****

**Step 3:** Re-execute the previous block without declaring **primes** Nested Table as shown below:

|  |  |
| --- | --- |
| Line | Description |
| DECLARE |  |
| **--** TYPE primes IS TABLE OF NUMBER; | **Remove [prime] declaration.** |
| prime primes := primes(2,3,5,9,11,13,17,19,  23,29,31,37,41,43); | **Use a standalone [primes]** |
| i PLS\_INTEGER := 1; |  |
| out VARCHAR2(300); |  |
| BEGIN |  |
| --------- Update the fourth item ------- |  |
| prime(4):= 7; |  |
| --------- Print Nested Table content ----- |  |
| i := prime.FIRST; |  |
| LOOP |  |
| out := out ||','||prime(i); |  |
| i := prime.NEXT(i); |  |
| EXIT WHEN i IS NULL; |  |
| END LOOP; |  |
| dbms\_output.put\_line('Prime ='||SUBSTR(out,2)); |  |
| --------- Assign new Value of Nested Table --- |  |
| prime := primes(2,3,5,9,11,13,17,19,23,  29,31,37,41,43,53,59,61,67,71); |  |
| END; |  |
| / |  |

****

**Please note:** In the previous block use the standalone Nested Table type. **Which Nested Table declaration will be used if a local type with same name is declared in the block?**

# SUMMARY

Composite Data types are those types that contain sub-types. There are two main categories of Composite Data types: Record and Collection. A Record can be seen as a row while a Collection can be seen somehow as a column. A Record may has one or more fields of different data types (including user-defined data type). A Record can be used in SELECT and DML statements to shorten and to organize the code. A Collection, in contrast, has one data type but one and more elements. There are three different types of Collection: VARRAY, Associative Array, and Nested Table. VARRAY is a variable array which guarantees the order and the dense of element. However, its size must be specified in declaration. Associative Array is as a Hash-Table. The elements order is not guaranteed but the performance of allocating any element is independent of the number of elements inside the Associative Array. A Nested Table is like the Associative Array; it is unbounded. However, its index is automatically generated by a number index. Nested Table is best used to join with other tables and views.

After completing this lab exercise, you should be able to identify how and when to use a Record and Collection data types.

# REFERENCES

* http://docs.oracle.com/database/121/LNPLS/composites.htm#LNPLS005

# INDEX

Associative Array 3, 5, 17, 18, 19, 20, 21, 29

Collection 2, 5, 29

Composite data types 2

Nested Table 3, 5, 25, 26, 27, 28, 29

Record 2, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 29

Scalar Data types 2

VARRAY 3, 5, 21, 23, 24, 26, 29