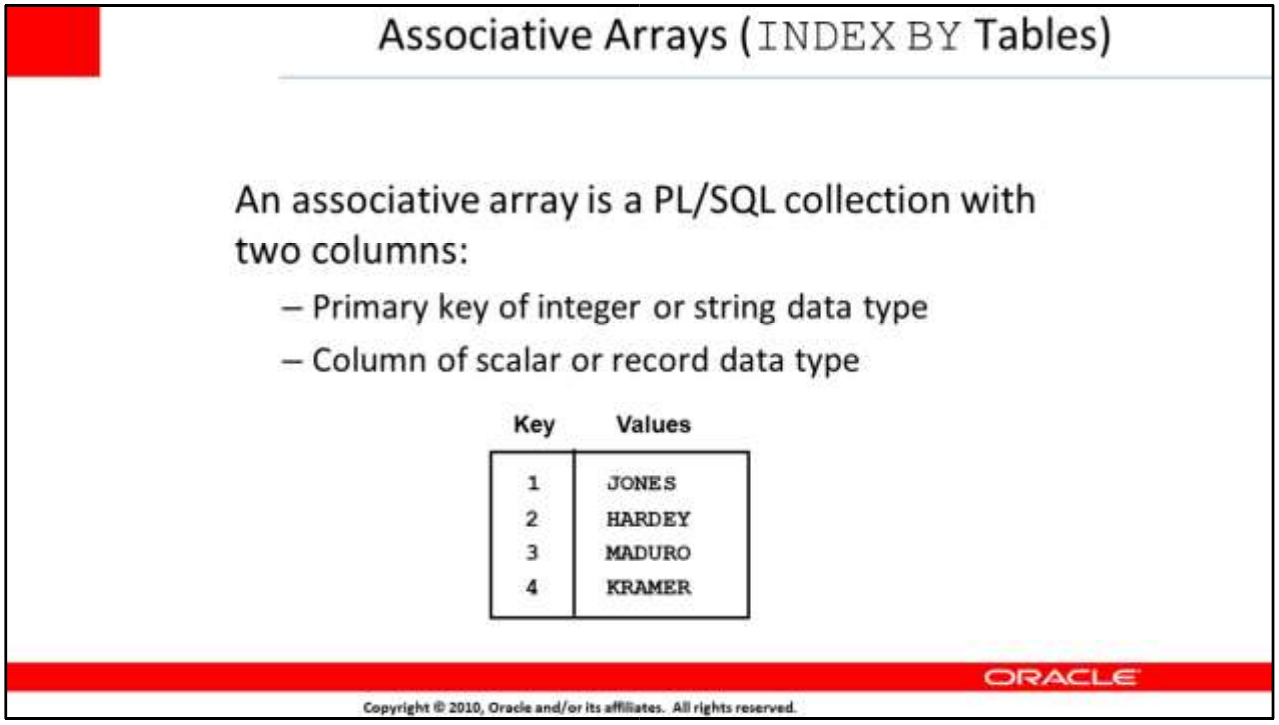


Associative Arrays (INDEX BY Tables)

An associative array is a PL/SQL collection with two columns:

- Primary key of integer or string data type
- Column of scalar or record data type

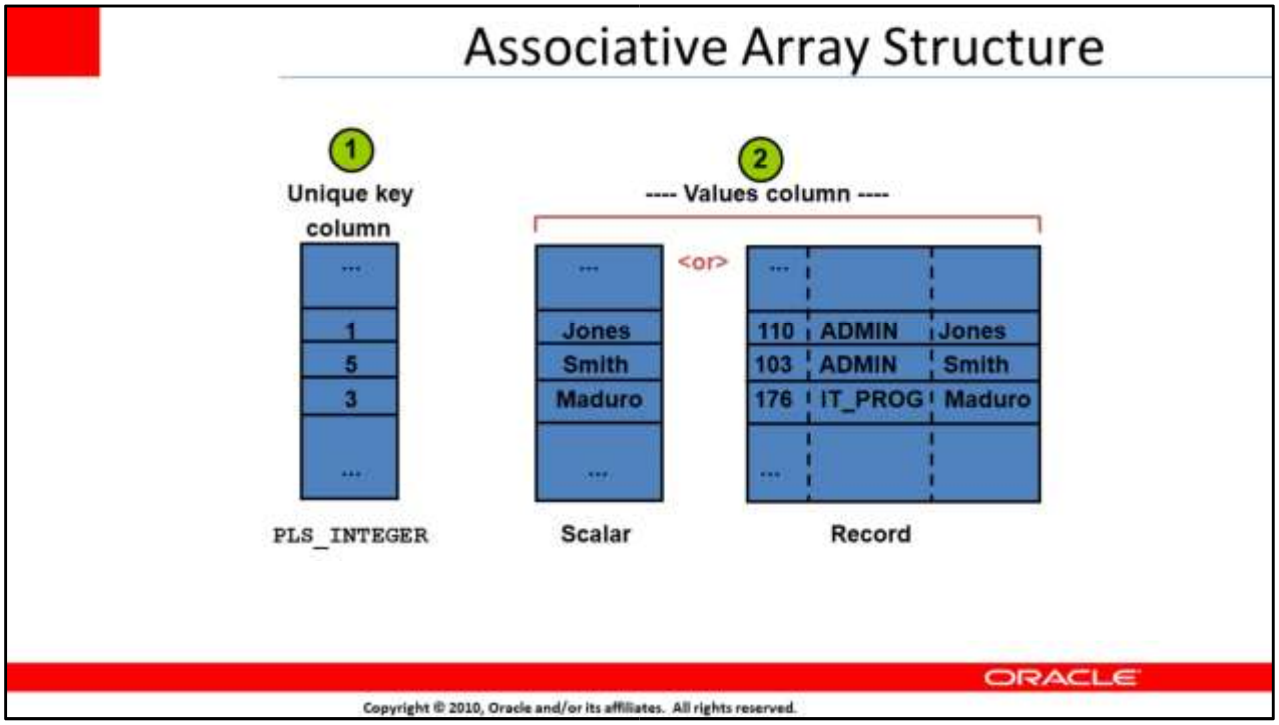
Key	Values
1	JONES
2	HARDEY
3	MADURO
4	KRAMER

Associative Arrays (INDEXBY Tables)

An associative array is a type of PL/SQL collection. It is a composite data type, and is user defined. Associative arrays are sets of key-value pairs. They can store data using a primary key value as the index, where the key values are not necessarily sequential. Associative arrays are also known as INDEXBY tables. Associative arrays have only two columns, neither of which can be named:

The first column, of integer or string type, acts as the primary key.

The second column, of scalar or record data type, holds values.



Associative Array Structure

Associative Array Structure

As previously mentioned, associative arrays have two columns. The second column either holds one value per row, or multiple values. Unique Key Column: The data type of the key column can be:

Numeric, either BINARY_INTEGER or PLS_INTEGER. These two numeric data types require less storage than NUMBER, and arithmetic operations on these data types are faster than the NUMBER arithmetic.

VARCHAR2 or one of its subtypes

“Value” Column: The value column can be either a scalar data type or a record data type. A column with scalar data type can hold only one value per row, whereas a column with record data type can hold multiple values per row.

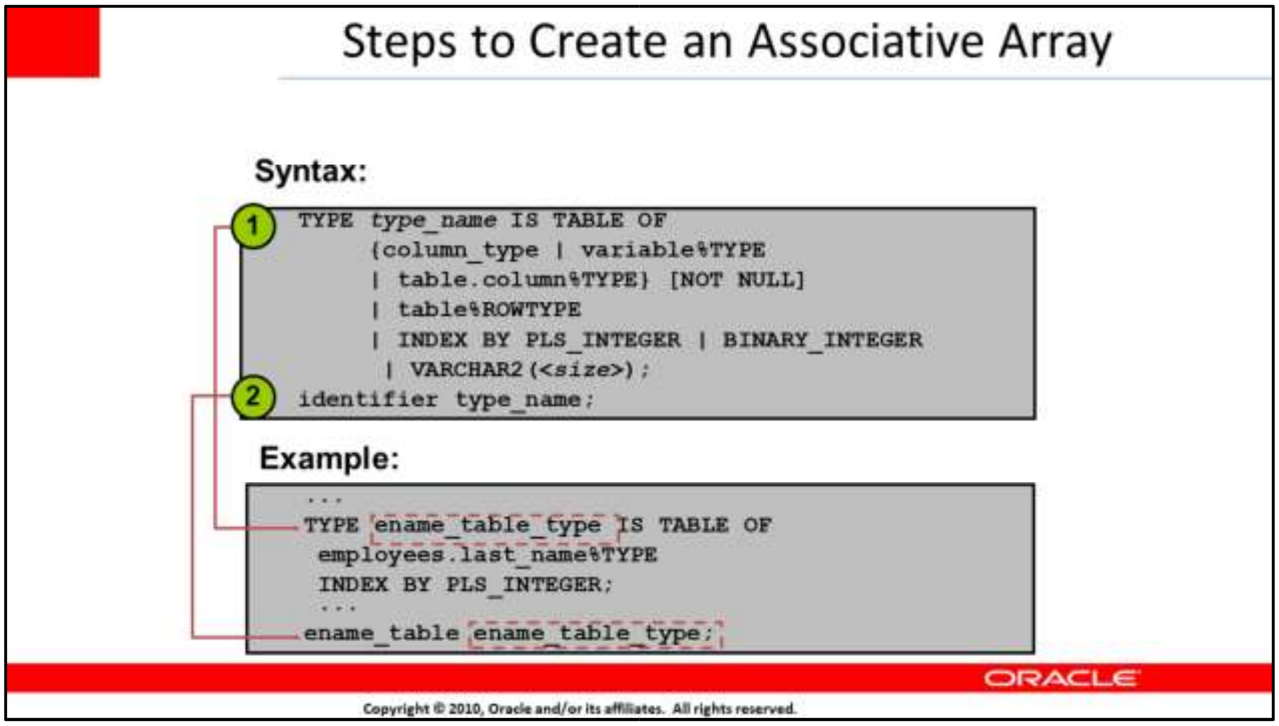
Other Characteristics

An associative array is not populated at the time of declaration. It contains no keys or values, and you cannot initialize an associative array in its declaration.

An explicit executable statement is required to populate the associative array.

6 - 2

Like the size of a database table, the size of an associative array is unconstrained. That is, the number of rows can increase dynamically so that your associative array grows as new rows are added. Note that the keys do not have to be sequential, and can be both positive and negative.



Steps to Create an Associative Array

Syntax:

```
1  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>);
2  identifier type_name;
```

Example:

```
...
TYPE ename_table_type IS TABLE OF
 employees.last_name%TYPE
 INDEX BY PLS_INTEGER;
 ...
ename_table ename_table_type;
```

Steps to Create an Associative Array

There are two steps involved in creating an associative array:

1. Declare a TABLE data type using the INDEXBY option.
2. Declare a variable of that data type. Syntax type_name Is the name of the

TABLE type (This name is used in the subsequent

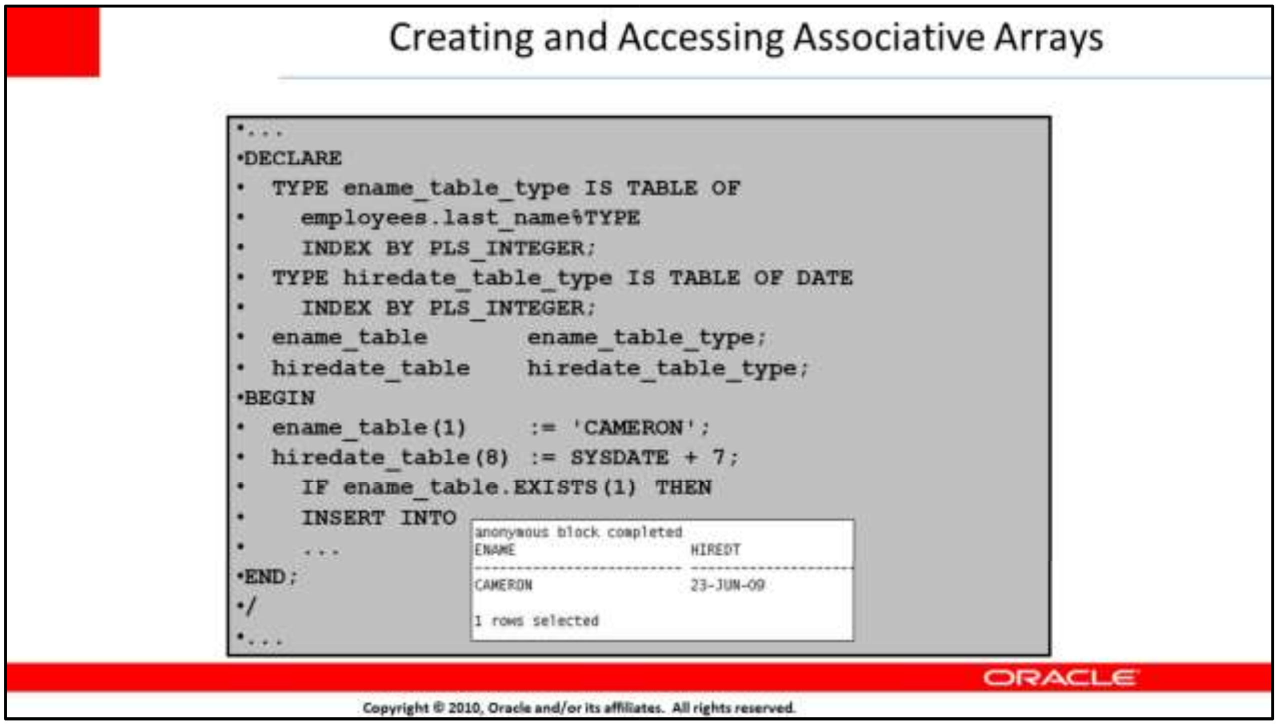
declaration of the array identifier.)

column_type	Is any scalar or composite data type such as VARCHAR2, DATE, NUMBER, or %TYPE (You can use the %TYPE attribute to provide the column data type.)
identifier	Is the name of the identifier that represents an entire associative array

Note: The NOTNULL constraint prevents nulls from being assigned to the associative array.

Example

In the example, an associative array with the variable name ename_table is declared to store the last names of employees.



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
```

Creating and Accessing Associative Arrays

The example in the slide creates two associative arrays, with the identifiers ename_table and hiredate_table.

The key of each associative array is used to access an element in the array, by using the following syntax:

identifier(index)

In both arrays, the index value belongs to the PLS_INTEGER type. To reference the first row in the ename_table associative array, specify: ename_table(1)

To reference the eighth row in the hiredate_table associative array, specify: hiredate_table(8)

Note

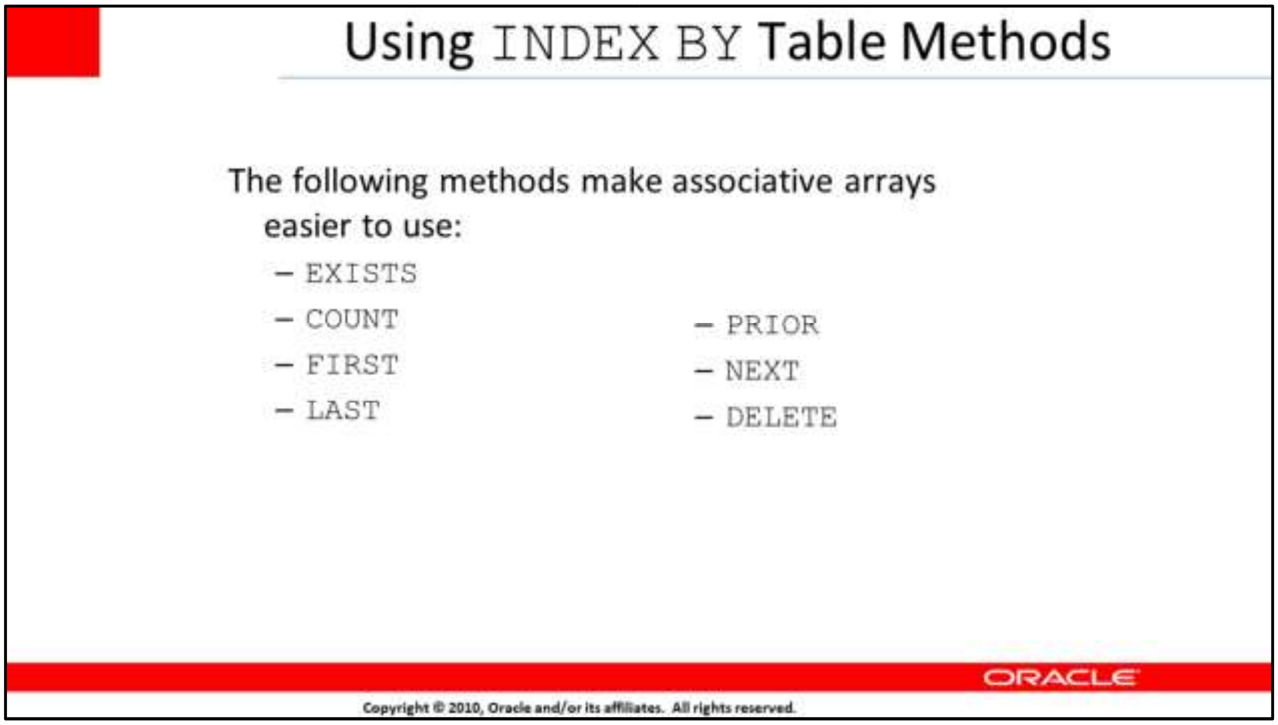
The magnitude range of a PLS_INTEGER is –2,147,483,647 through 2,147,483,647, so the primary key value can be negative. Indexing does not need to start with 1.

The exists(i)method returns TRUE if a row with index i is returned. Use the exists method to prevent an error that is raised in reference to a nonexistent table element.

The complete code example is found in code_6_21_s.sql.

Using INDEXBY Table Methods

An INDEXBY table method is a built-in procedure or function that operates on an associative array and is called by using the dot notation.



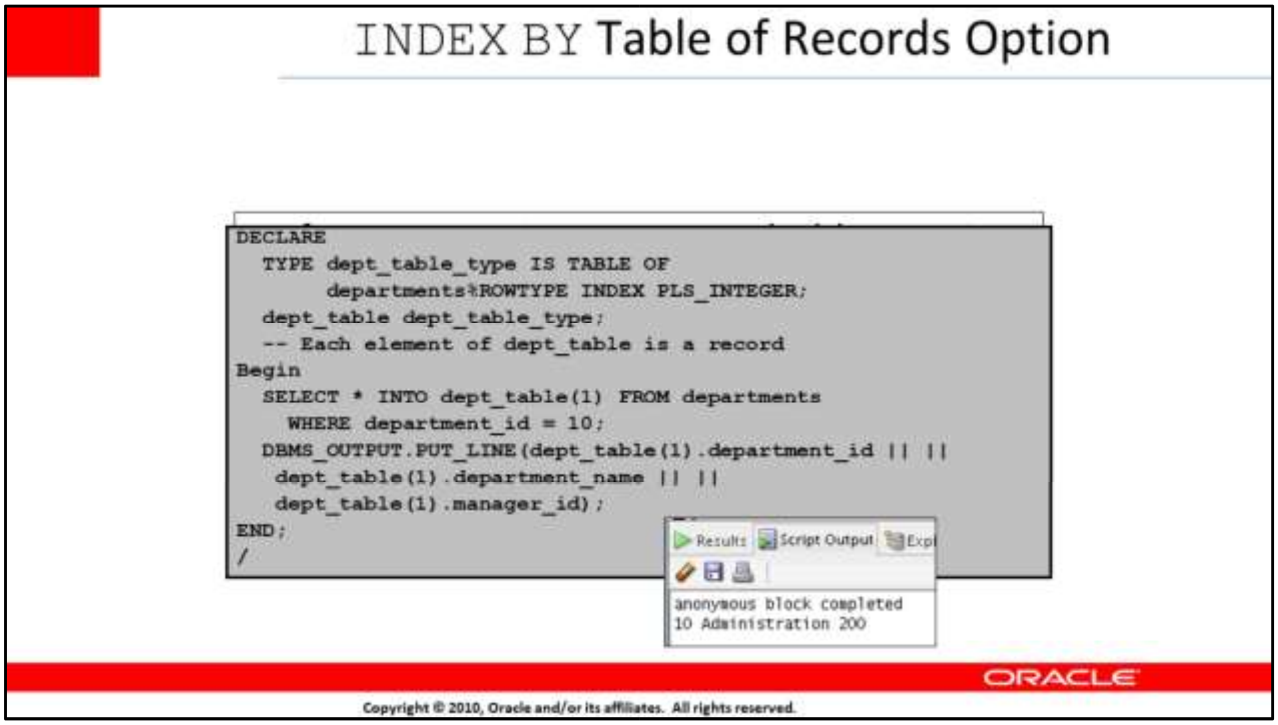
Using INDEX BY Table Methods

The following methods make associative arrays easier to use:

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

Syntax: table_name.method_name[(parameters)]

Method	Description
EXISTS(n)	Returns TRUE if the nth element in an associative array exists
COUNT	Returns the number of elements that an associative array currently contains
FIRST	□□ Returns the first (smallest) index number in an associative array □□Returns NULL if the associative array is empty
LAST	□□ Returns the last (largest) index number in an associative array □□Returns NULL if the associative array is empty
PRIOR(n)	Returns the index number that precedes index n in an associative array
NEXT(n)	Returns the index number that succeeds index n in an associative array
DELETE	□□ DELETE removes all elements from an associative array. □□ DELETE(n) removes the nth element from an associative array. □□ DELETE(m, n) removes all elements in the range m ... n from an associative array.



INDEX BY Table of Records Option

```
DECLARE
  TYPE dept_table_type IS TABLE OF
       departments%ROWTYPE INDEX PLS_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
Begin
  SELECT * INTO dept_table(1) FROM departments
    WHERE department_id = 10;
  DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id || ||
   dept_table(1).department_name || ||
   dept_table(1).manager_id);
END;
/
```

```
anonymous block completed
10 Administration 200
```

INDEXBY Table of Records Option

As previously discussed, an associative array that is declared as a table of scalar data type can store the details of only one column in a database table. However, there is often a need to store all the columns retrieved by a query. The INDEXBY table of records option enables one array definition to hold information about all the fields of a database table.

Creating and Referencing a Table of Records

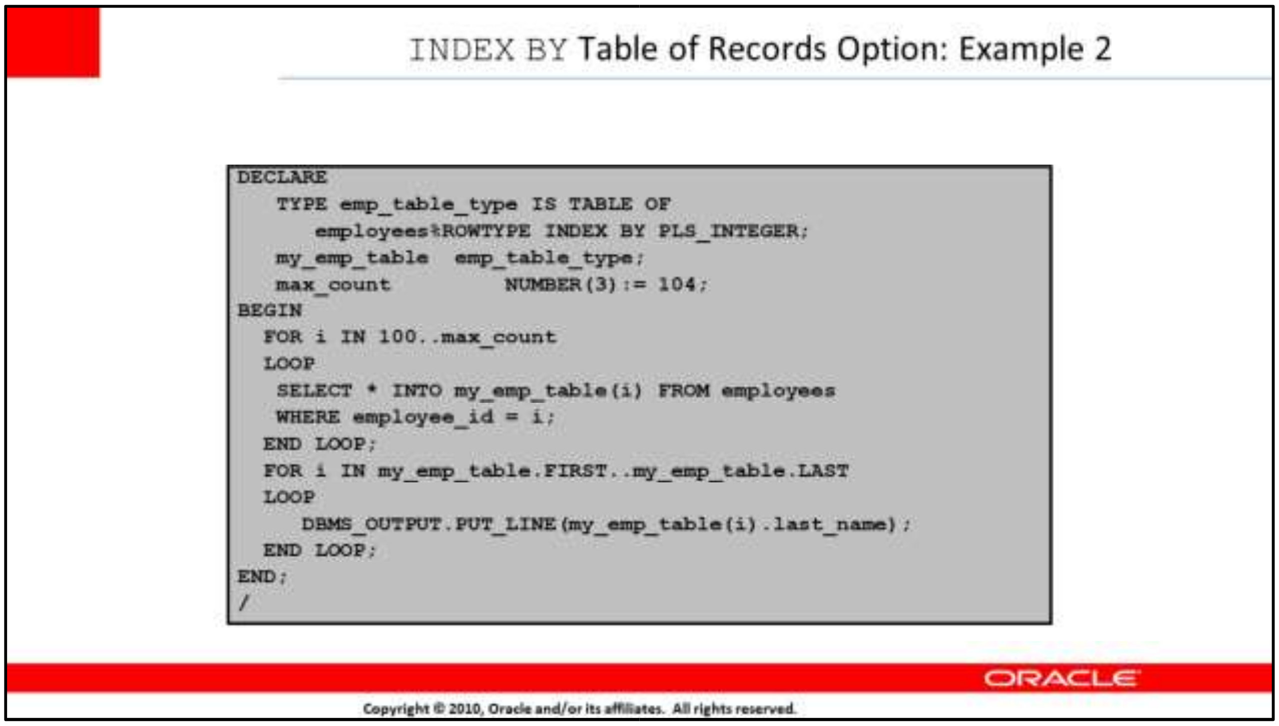
As shown in the associative array example in the slide, you can:

- Use the %ROWTYPE attribute to declare a record that represents a row in a database table
- Refer to fields within the dept_table array because each element of the array is a record

The differences between the %ROWTYPE attribute and the composite data type PL/SQL record are as follows:

- PL/SQL record types can be user-defined, whereas %ROWTYPE implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use %ROWTYPE, you cannot specify the fields. The %ROWTYPE attribute represents a table row with all the

fields based on the definition of that table.
User-defined records are static, but %ROWTYPE records are dynamic— they are based on a table structure. If the table structure changes, the record structure also picks up the change.



INDEX BY Table of Records Option: Example 2

```
DECLARE
   TYPE emp_table_type IS TABLE OF
      employees%ROWTYPE INDEX BY PLS_INTEGER;
   my_emp_table  emp_table_type;
   max_count          NUMBER(3):= 104;
BEGIN
  FOR i IN 100..max_count
  LOOP
   SELECT * INTO my_emp_table(i) FROM employees
   WHERE employee_id = i;
  END LOOP;
  FOR i IN my_emp_table.FIRST..my_emp_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
  END LOOP;
END;
/
```

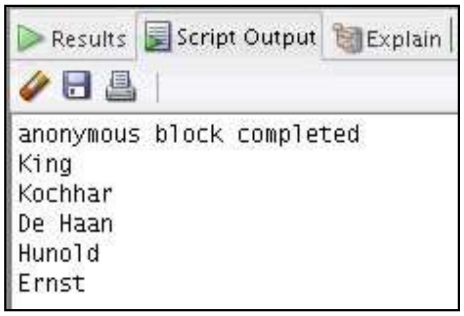
INDEXBY Table of Records: Example 2

The example in the slide declares an associative array, using the INDEXBY table of records option, to temporarily store the details of employees whose employee IDs are between 100 and 104. The variable name for the array is emp_table_type.

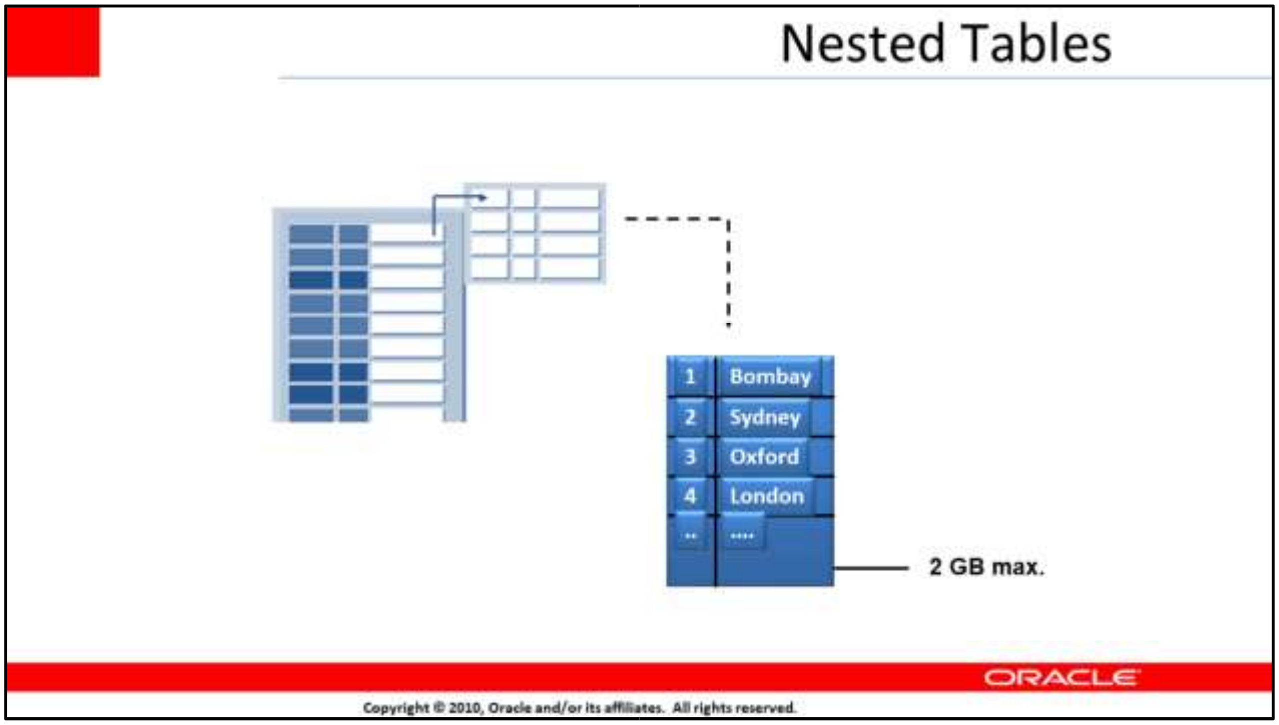
Using a loop, the information of the employees from the EMPLOYEES table is retrieved and stored in the array. Another loop is used to print the last names from the array. Note the use of the first and last methods in the example.

Note: The slide demonstrates one way to work with an associative array that uses the INDEXBY table of records method. However, you can do the same more efficiently using cursors. Cursors are explained in the lesson titled “Using Explicit Cursors.”

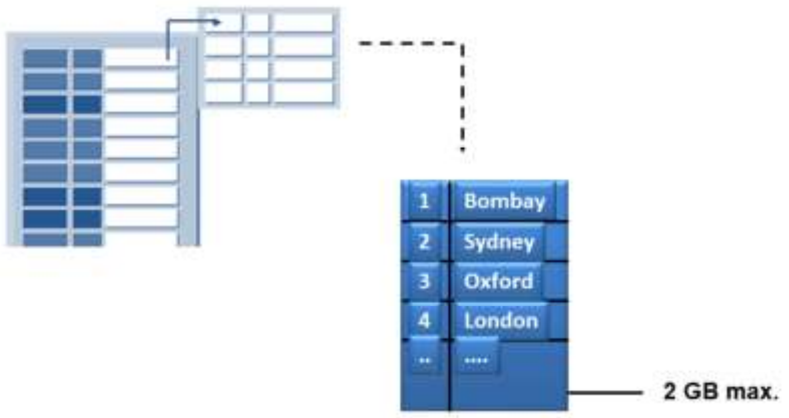
The results of the code example is as follows:



```
anonymous block completed
King
Kochhar
De Haan
Hunold
Ernst
```



Nested Tables



Nested Tables

The functionality of nested tables is similar to that of associative arrays; however, there are differences in the nested table implementation.

The nested table is a valid data type in a schema-level table, but an associative array is not. Therefore, unlike associative arrays, nested tables can be stored in the database.

The size of a nested table can increase dynamically, although the maximum size is 2 GB.

The “key” cannot be a negative value (unlike in the associative array). Though reference is made to the first column as key, there is no key in a nested table. There is a column with numbers.

Elements can be deleted from anywhere in a nested table, leaving a sparse table with nonsequential “keys.” The rows of a nested table are not in any particular order.

When you retrieve values from a nested table, the rows are given consecutive subscripts starting from 1. Syntax

```
TYPE type_name IS TABLE OF
```

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

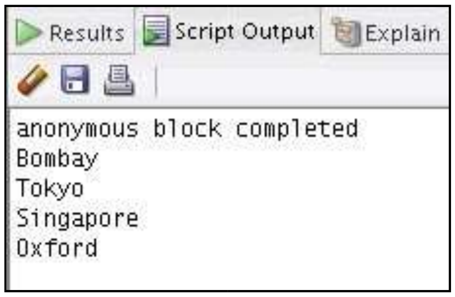
Nested Tables (continued) Example:

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

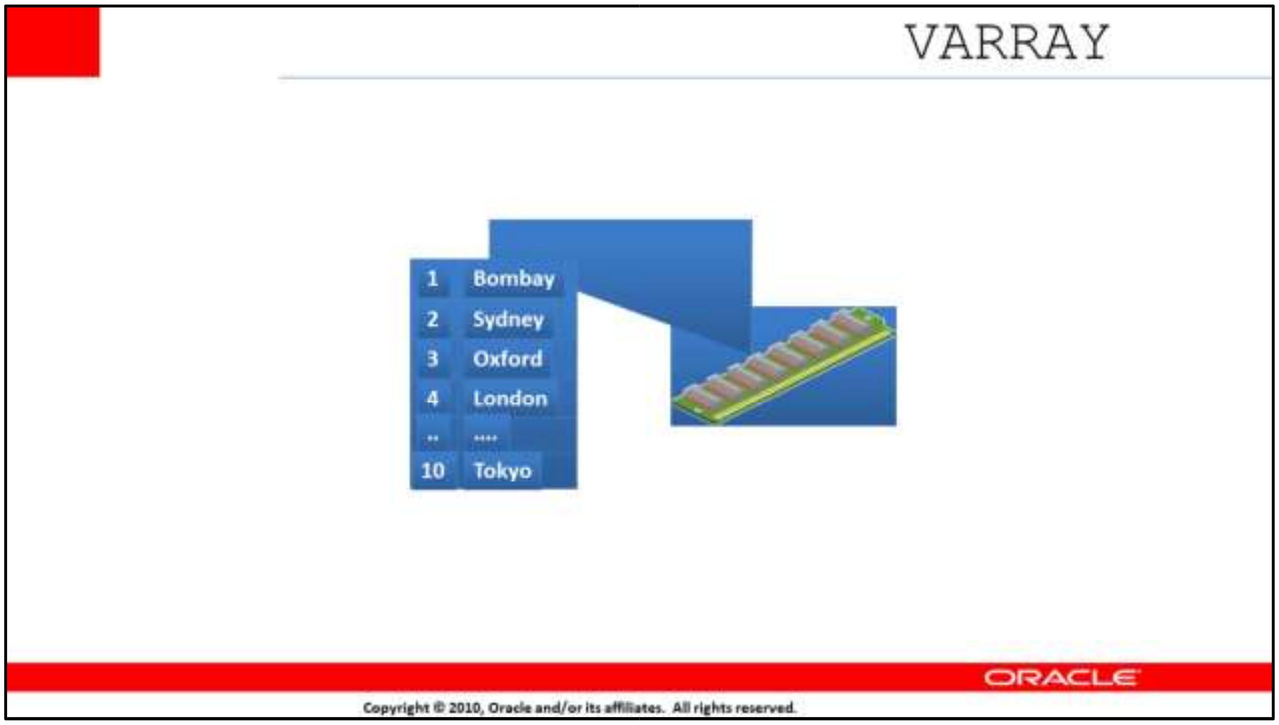
If you do not initialize a nested table, it is automatically initialized to NULL. You can initialize the offices nested table by using a constructor: offices := location_type('Bombay', 'Tokyo','Singapore', 'Oxford');

The complete code example and output is as follows:

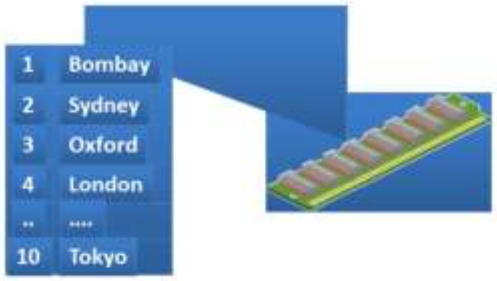
```
SET SERVEROUTPUT ON;

  DECLARE
    TYPE location_type IS TABLE OF
     locations.city%TYPE; offices location_type;
     table_count NUMBER;
  BEGIN
    offices := location_type('Bombay', 'Tokyo','Singapore',
     'Oxford');
 FOR i in 1.. offices.count() LOOP
   DBMS_OUTPUT.PUT_LINE(offices(i));
    END LOOP;
  END;
  /
```



```
anonymous block completed
Bombay
Tokyo
Singapore
Oxford
```



VARRAY



VARRAY

A variable-size array (VARRAY) is similar to an associative array, except that a VARRAY is constrained in size.

A VARRAY is valid in a schema-level table.

Items of VARRAY type are called VARRAYs.

VARRAYs have a fixed upper bound. You have to specify the upper bound when you declare them. This is similar to arrays in C language. The maximum size of a VARRAY is 2 GB, as in nested tables.

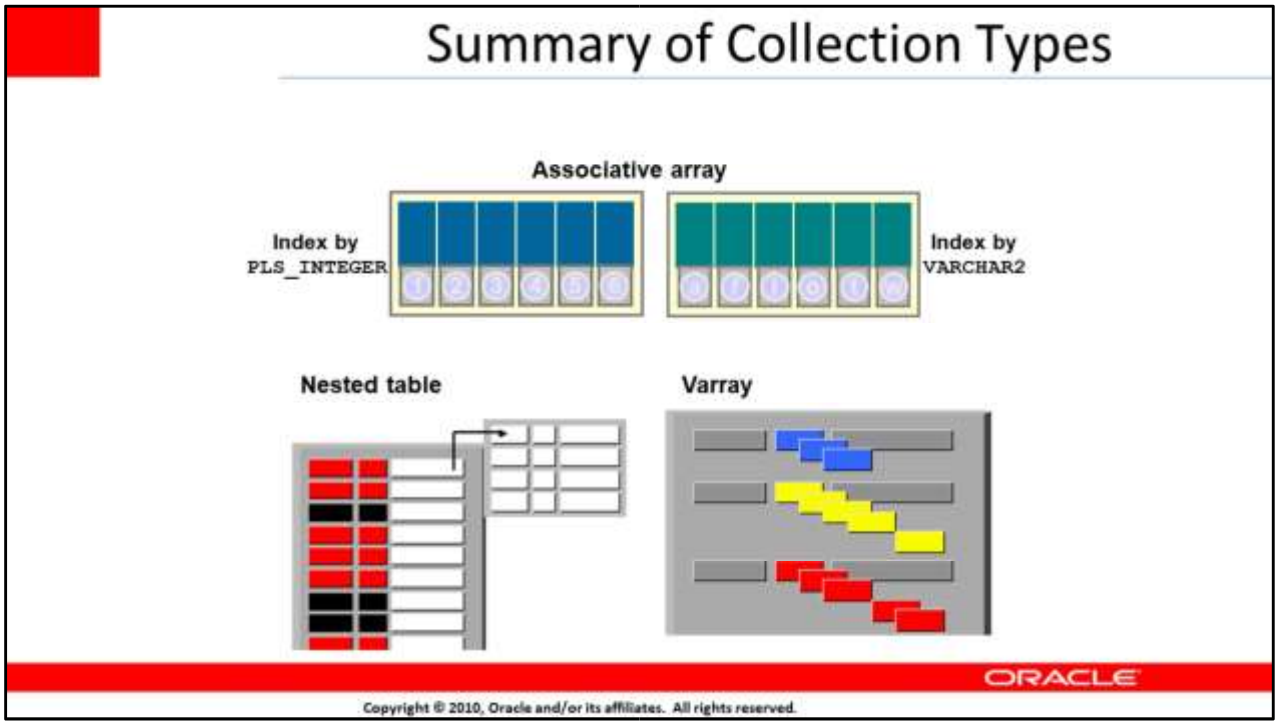
The distinction between a nested table and a VARRAY is the physical storage mode. The elements of a VARRAY are stored inline with the table's data unless the size of the VARRAY is greater than 4 KB. Contrast that with nested tables, which are always stored out-of-line.

You can create a VARRAY type in the database by using SQL.

Example:

```
TYPE location_type IS
VARRAY(3) OF
locations.city%TYPE; offices
location_type;
```

The size of this VARRAY is restricted to 3. You can initialize a VARRAY by using constructors. If you try to initialize the VARRAY with more than three elements, a “Subscript outside of limit” error message is displayed.



Summary of Collection Types

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 SELECTINTO. 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.