7

Using Explicit Cursors

Objectives

After completing this lesson, you should be able to do the following:

- Distinguish between implicit and explicit cursors
- Discuss the reasons for using explicit cursors
- Declare and control explicit cursors
- Use simple loops and cursor `FOR` loops to fetch data
- Declare and use cursors with parameters
- Lock rows with the `FOR UPDATE` clause
- Reference the current row with the `WHERE CURRENT` clause

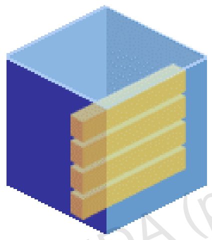
Lesson Aim

You have learned about implicit cursors that are automatically created by PL/SQL when you execute a SQL `SELECT` or DML statement. In this lesson, you learn about explicit cursors. You learn to differentiate between implicit and explicit cursors. You also learn to declare and control simple cursors as well as cursors with parameters.

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**



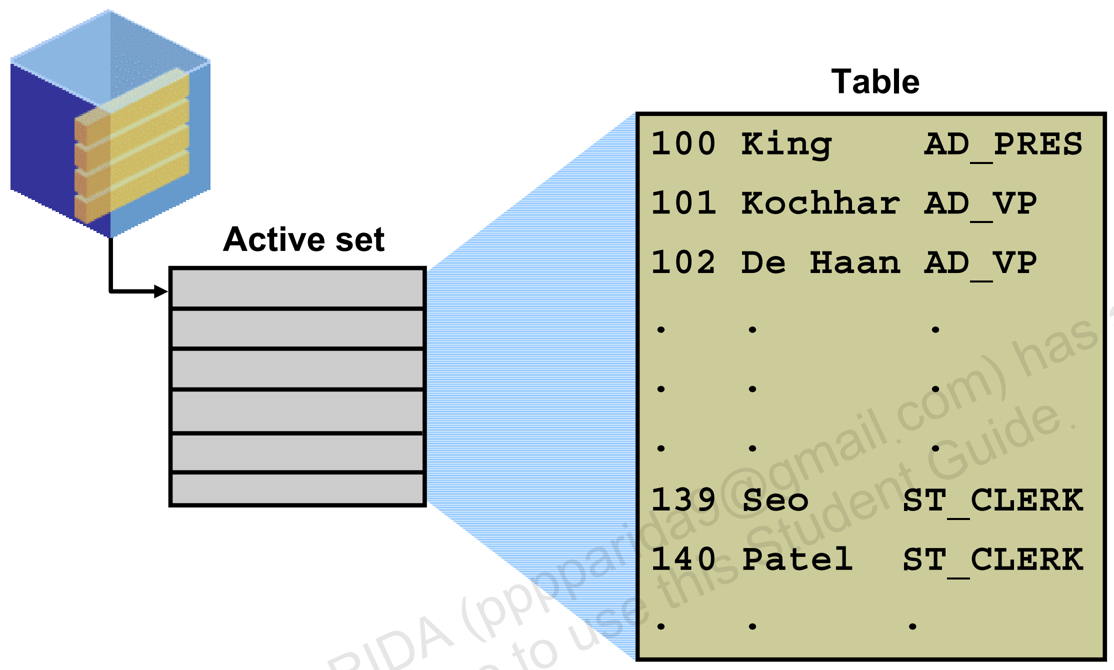
Cursors

The Oracle server uses work areas (called *private SQL areas*) to execute SQL statements and to store processing information. You can use explicit cursors to name a private SQL area and to access its stored information.

Cursor Type	**Description**
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL `SELECT` statements.
Explicit	For queries that return more than one row, explicit cursors are declared and managed by the programmer and manipulated through specific statements in the block's executable actions.

The Oracle server implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor. Using PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor.

Explicit Cursor Operations



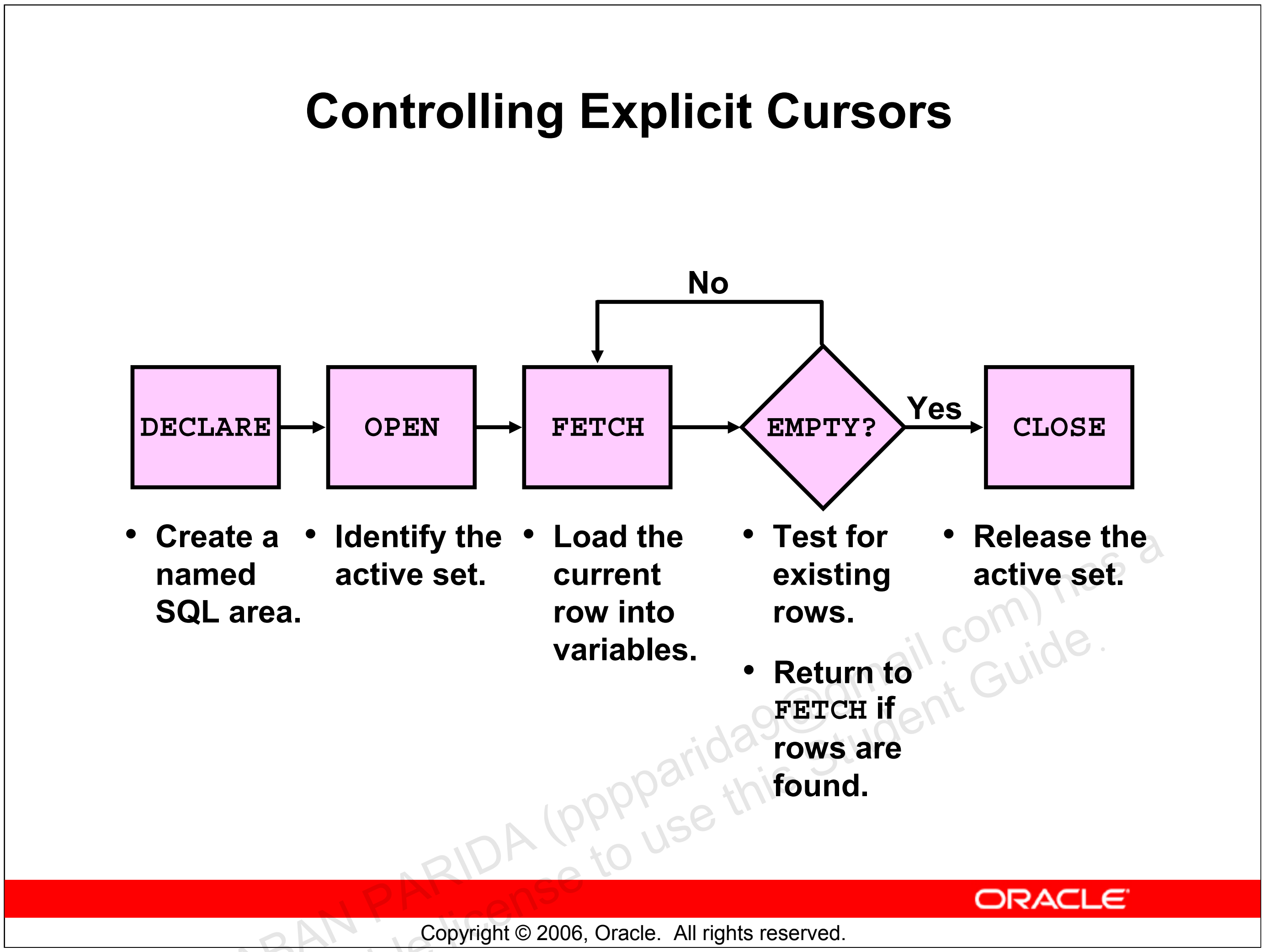
Explicit Cursor Operations

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.

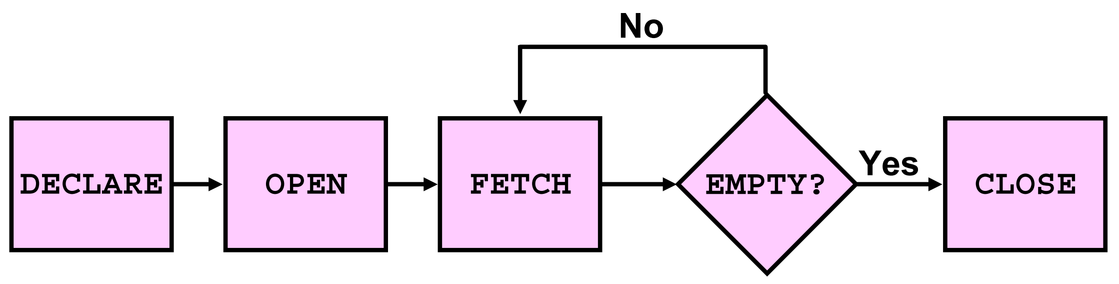
The set of rows returned by a multiple-row query is called the *active set*. Its size is the number of rows that meet your search criteria. The diagram in the slide shows how an explicit cursor "points" to the current row in the active set. This enables your program to process the rows one at a time.

Explicit cursor functions:

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



Controlling Explicit Cursors



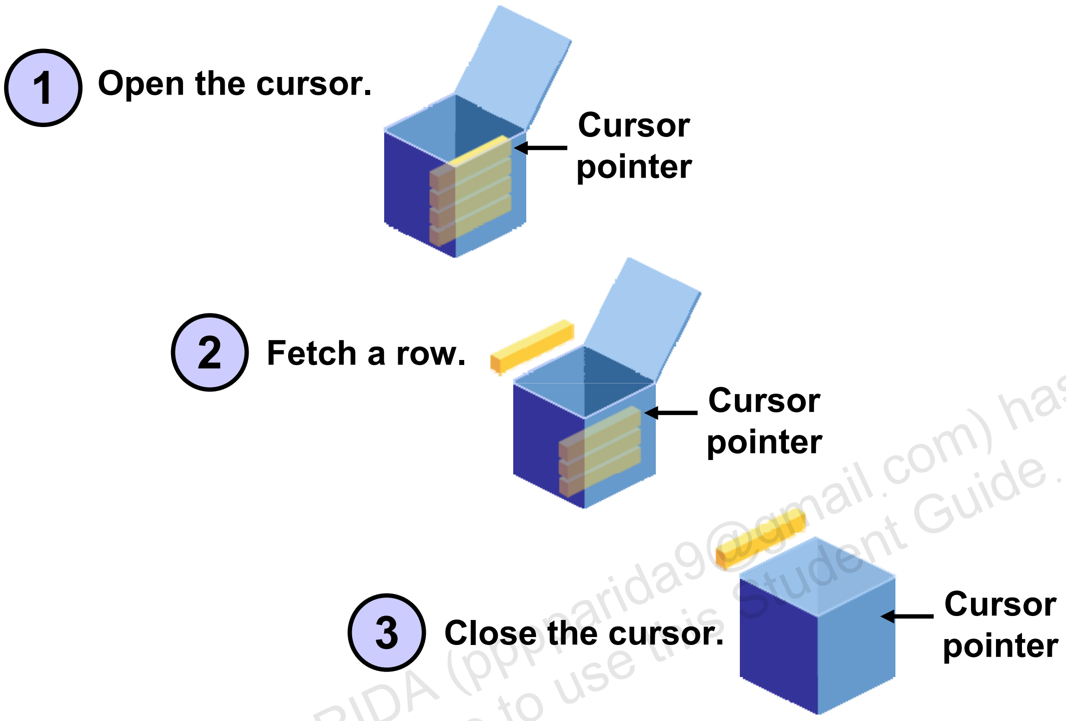
- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to `FETCH` if rows are found.
- Release the active set.

Controlling Explicit Cursors

Now that you have a conceptual understanding of cursors, review the steps to use them.

1. In the declarative section of a PL/SQL block, declare the cursor by naming it and defining the structure of the query to be associated with it.
2. Open the cursor.
   The `OPEN` statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor.
   In the flow diagram shown in the slide, after each fetch you test the cursor for any existing row. If there are no more rows to process, you must close the cursor.
4. Close the cursor.
   The `CLOSE` statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

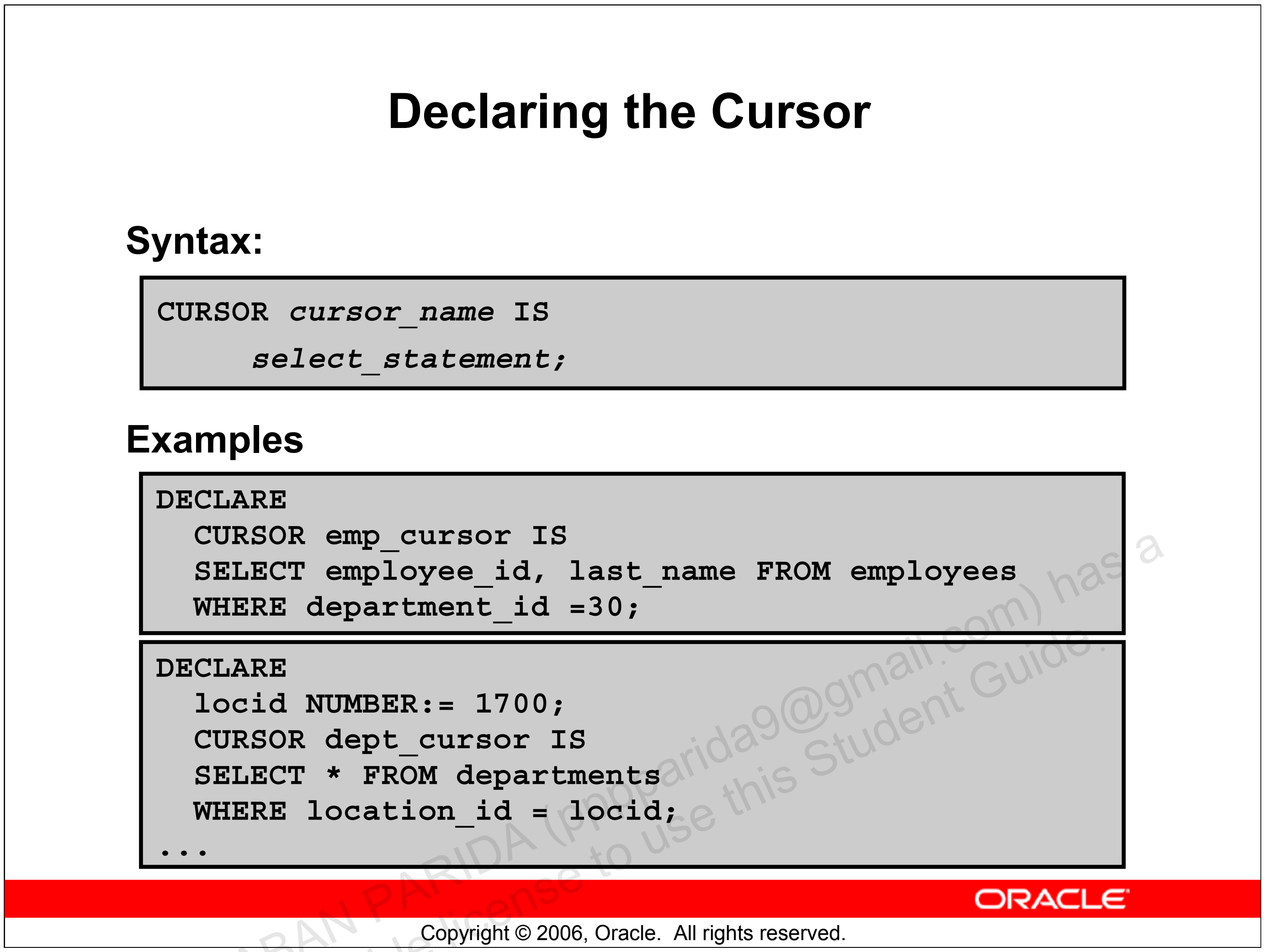
Controlling Explicit Cursors



Controlling Explicit Cursors (continued)

A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in the active set.

1. The `OPEN` statement executes the query associated with the cursor, identifies the active set, and positions the cursor at the first row.
2. The `FETCH` statement retrieves the current row and advances the cursor to the next row until either there are no more rows or until a specified condition is met.
3. The `CLOSE` statement releases the cursor.



Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples

```
DECLARE
  CURSOR emp_cursor IS
  SELECT employee_id, last_name FROM employees
  WHERE department_id =30;
```

```
DECLARE
  locid NUMBER:= 1700;
  CURSOR dept_cursor IS
  SELECT * FROM departments
  WHERE location_id = locid;
...
```

Declaring the Cursor

The syntax to declare a cursor is shown in the slide. In the syntax:

cursor_name	Is a PL/SQL identifier
select_statement	Is a `SELECT` statement without an `INTO` clause

The active set of a cursor is determined by the `SELECT` statement in the cursor declaration. It is mandatory to have an `INTO` clause for a `SELECT` statement in PL/SQL. However, note that the `SELECT` statement in the cursor declaration cannot have an `INTO` clause. That is because you are only defining a cursor in the declarative section and not retrieving any rows into the cursor.

Note

- 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 `ANSI SELECT` statement, including joins, subqueries, and so on.

Declaring the Cursor (continued)

The `emp_cursor` cursor is declared to retrieve the `employee_id` and `last_name` columns for those employees working in the department with a `department_id` of 30.

The `dept_cursor` cursor is declared to retrieve all the details for the department with the `location_id` 1700. Note that a variable is used while declaring the cursor. These variables are considered bind variables, which must be visible when you are declaring the cursor. These variables are examined only once at the time the cursor opens. You have learned that explicit cursors are used when you have to retrieve and operate on multiple rows in PL/SQL. However, this example shows that you can use the explicit cursor even if your `SELECT` statement returns only one row.

Opening the Cursor

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

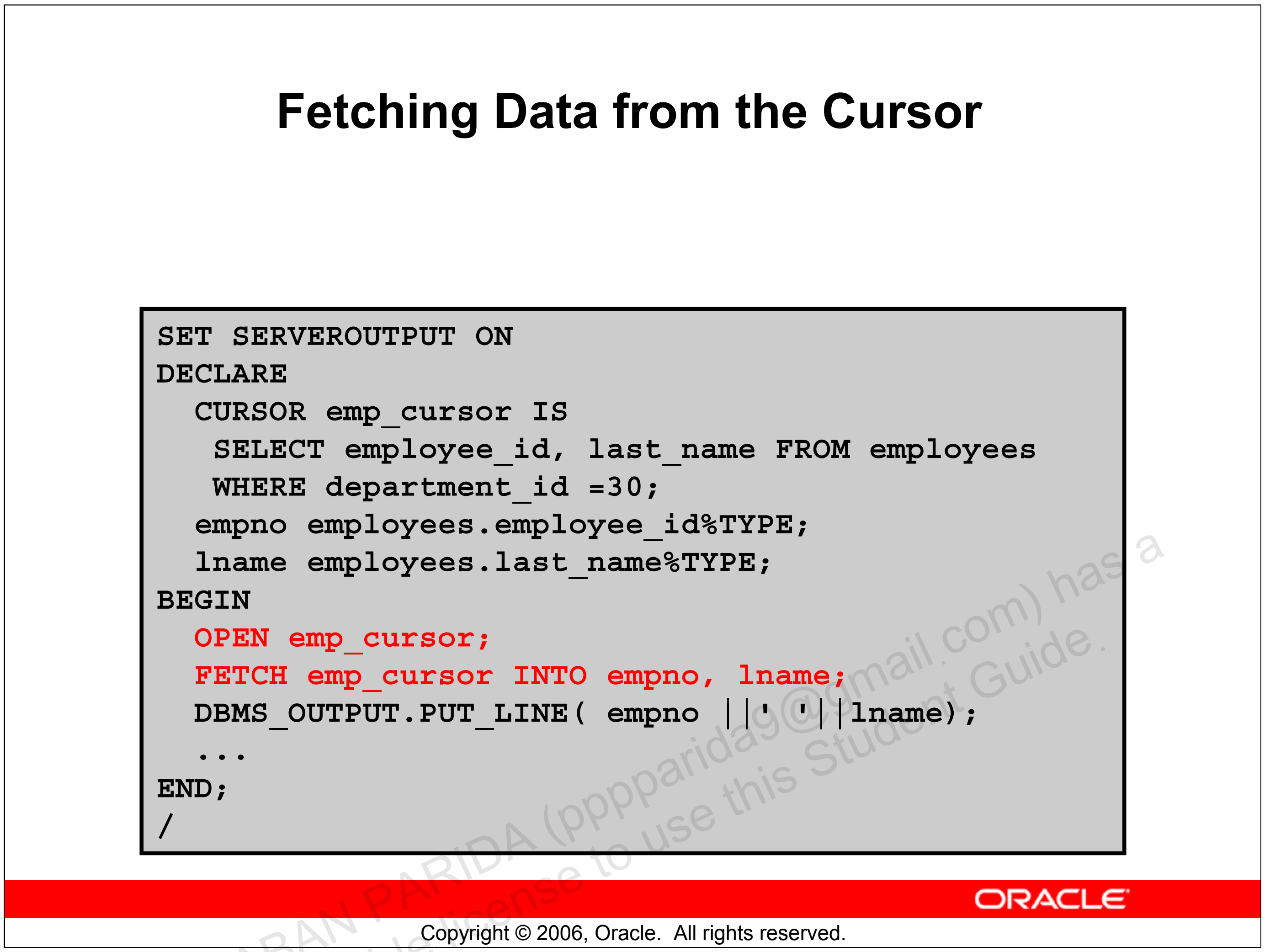
Opening the Cursor

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 the query returns no rows when the cursor is opened, PL/SQL does not raise an exception. However, you can test the status of the implicit cursor after a fetch by using the SQL%ROWCOUNT cursor attribute. For explicit cursors, use <*cursor_name*>%ROWCOUNT.



Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO empno, lname;
  DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  ...
END;
/
```

ORACLE

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.

Consider the example shown in the slide. Two variables, empno and lname, are declared to hold the fetched values from the cursor. Examine the FETCH statement.

The output of the PL/SQL block is as follows:

```
114 Raphaely
PL/SQL procedure successfully completed.
```

You have successfully fetched the values from the cursor to the variables. However, there are six employees in department 30, but only one row was fetched. To fetch all rows, you must use loops. In the next slide, you see how a loop is used to fetch all the rows.

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

Fetching Data from the Cursor (continued)

- Include the same number of variables in the INTO clause of the FETCH statement as there are columns in the SELECT statement, and be sure that the data types are compatible.
- Match each variable to correspond to the columns positionally.
- Alternatively, define a record for the cursor and reference the record in the FETCH INTO clause.
- Test to see whether the cursor contains rows. If a fetch acquires no values, there are no rows left to process in the active set and no error is recorded.

Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE  department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO empno, lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  END LOOP;
  ...
END;
/
```

Fetching Data from the Cursor (continued)

Observe that a simple `LOOP` is used to fetch all the rows. Also, the cursor attribute `%NOTFOUND` is used to test for the exit condition. The output of the PL/SQL block is:

```
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
PL/SQL procedure successfully completed.
```

Closing the Cursor

```
...
  LOOP
    FETCH emp_cursor INTO empno, lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  END LOOP;
 CLOSE emp_cursor;
END;
/
```

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.)

Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL record.

```
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE  department_id =30;
   emp_record  emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

Cursors and Records

You have already seen that you can define records that have the structure of columns in a table. You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set, because you can simply fetch into the record. Therefore, the values of the row are loaded directly into the corresponding fields of the record.

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.**

Cursor FOR Loops

You have learned to fetch data from cursors by using simple loops. You now learn to use a cursor FOR loop, which processes rows in an explicit cursor. It is a shortcut because the cursor is opened, a row is fetched once for each iteration in the loop, the loop exits when the last row is processed, and the cursor is closed automatically. The loop itself is terminated automatically at the end of the iteration where the last row is fetched.

In the syntax:

record_name	Is the name of the implicitly declared record
cursor_name	Is a PL/SQL identifier for the previously declared cursor

Guidelines

- Do not declare the record that controls the loop; it is declared implicitly.
- Test the cursor attributes during the loop, if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement.

Cursor FOR Loops

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

Cursor FOR Loops (continued)

The example that was used to demonstrate the usage of a simple loop to fetch data from cursors is rewritten to use the cursor `FOR` loop.

The `emp_record` is the record that is implicitly declared. You can access the fetched data with this implicit record (as shown in the slide). Observe that no variables are declared to hold the fetched data using the `INTO` clause. The code does not have `OPEN` and `CLOSE` statements to open and close the cursor, respectively.

Explicit Cursor Attributes

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

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor variable name, these attributes return useful information about the execution of a cursor manipulation statement.

Note: You cannot reference cursor attributes directly in a SQL statement.

%ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

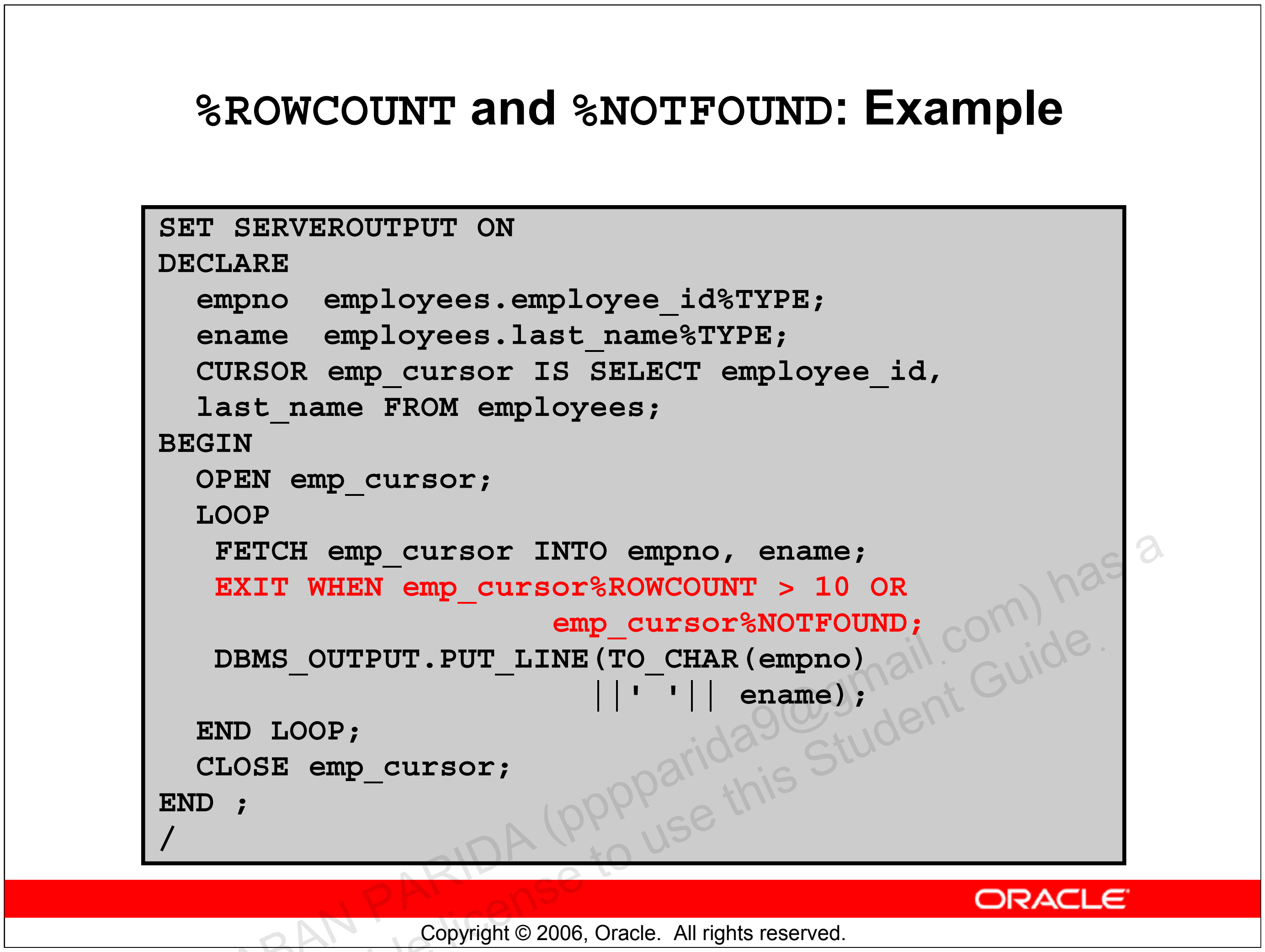
Example

```
IF NOT emp_cursor%ISOPEN THEN
   OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
```

%ISOPEN Attribute

- You can fetch rows only when the cursor is open. Use the `%ISOPEN` cursor attribute to determine whether the cursor is open.
- Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.
- Use the `%ROWCOUNT` cursor attribute to do the following:
  - Process an exact number of rows
  - Fetch the rows in a loop and determine when to exit the loop

Note: `%ISOPEN` returns the status of the cursor: `TRUE` if open and `FALSE` if not.



%ROWCOUNT and %NOTFOUND: Example

```
SET SERVEROUTPUT ON
DECLARE
  empno  employees.employee_id%TYPE;
  ename  employees.last_name%TYPE;
  CURSOR emp_cursor IS SELECT employee_id,
  last_name FROM employees;
BEGIN
  OPEN emp_cursor;
  LOOP
   FETCH emp_cursor INTO empno, ename;
   EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                    emp_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(empno)
                        ||' '|| ename);
  END LOOP;
  CLOSE emp_cursor;
END ;
/
```

%ROWCOUNT and %NOTFOUND: Example

The example in the slide retrieves the first ten employees one by one. This example shows how %ROWCOUNT and %NOTFOUND attributes can be used for exit conditions in a loop.

Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

Example

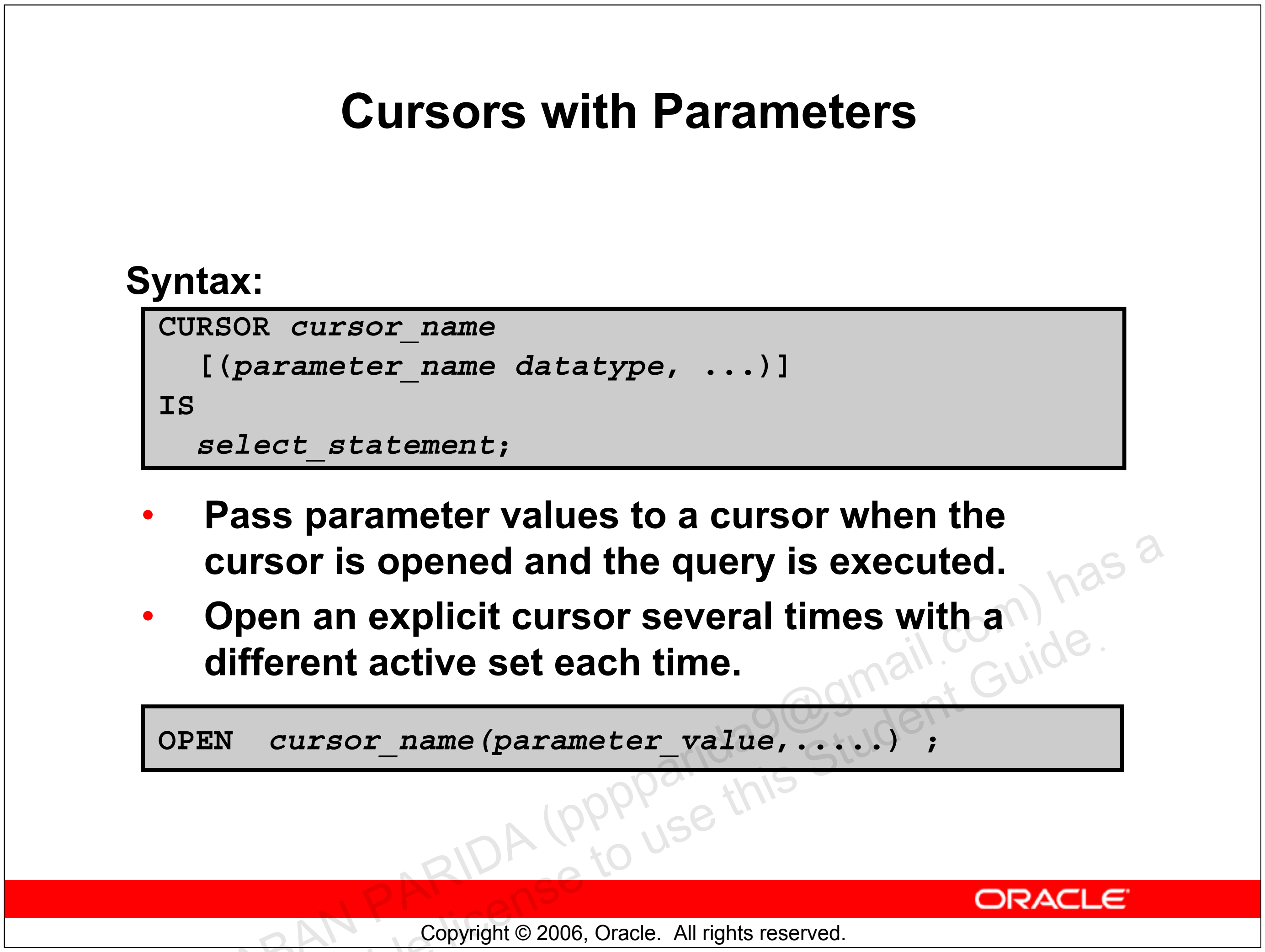
```
SET SERVEROUTPUT ON
BEGIN
  FOR emp_record IN (SELECT employee_id, last_name
   FROM employees WHERE department_id =30)
  LOOP
   DBMS_OUTPUT.PUT_LINE( emp_record.employee_id ||'
   '||emp_record.last_name);
  END LOOP;
END;
/
```

Cursor FOR Loops Using Subqueries

Note that there is no declarative section in this PL/SQL block. The difference between the cursor FOR loops using subqueries and the cursor FOR loop lies in the cursor declaration. If you are writing cursor FOR loops using subqueries, you need not declare the cursor in the declarative section. You have to provide the SELECT statement that determines the active set in the loop itself.

The example that was used to illustrate a cursor FOR loop is rewritten to illustrate a cursor FOR loop using subqueries.

Note: You cannot reference explicit cursor attributes if you use a subquery in a cursor FOR loop because you cannot give the cursor an explicit name.



Cursors with Parameters

Syntax:

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

- **Pass parameter values to a cursor when the cursor is opened and the query is executed.**
- **Open an explicit cursor several times with a different active set each time.**

```
OPEN  cursor_name(parameter_value,.....) ;
```

Cursors with Parameters

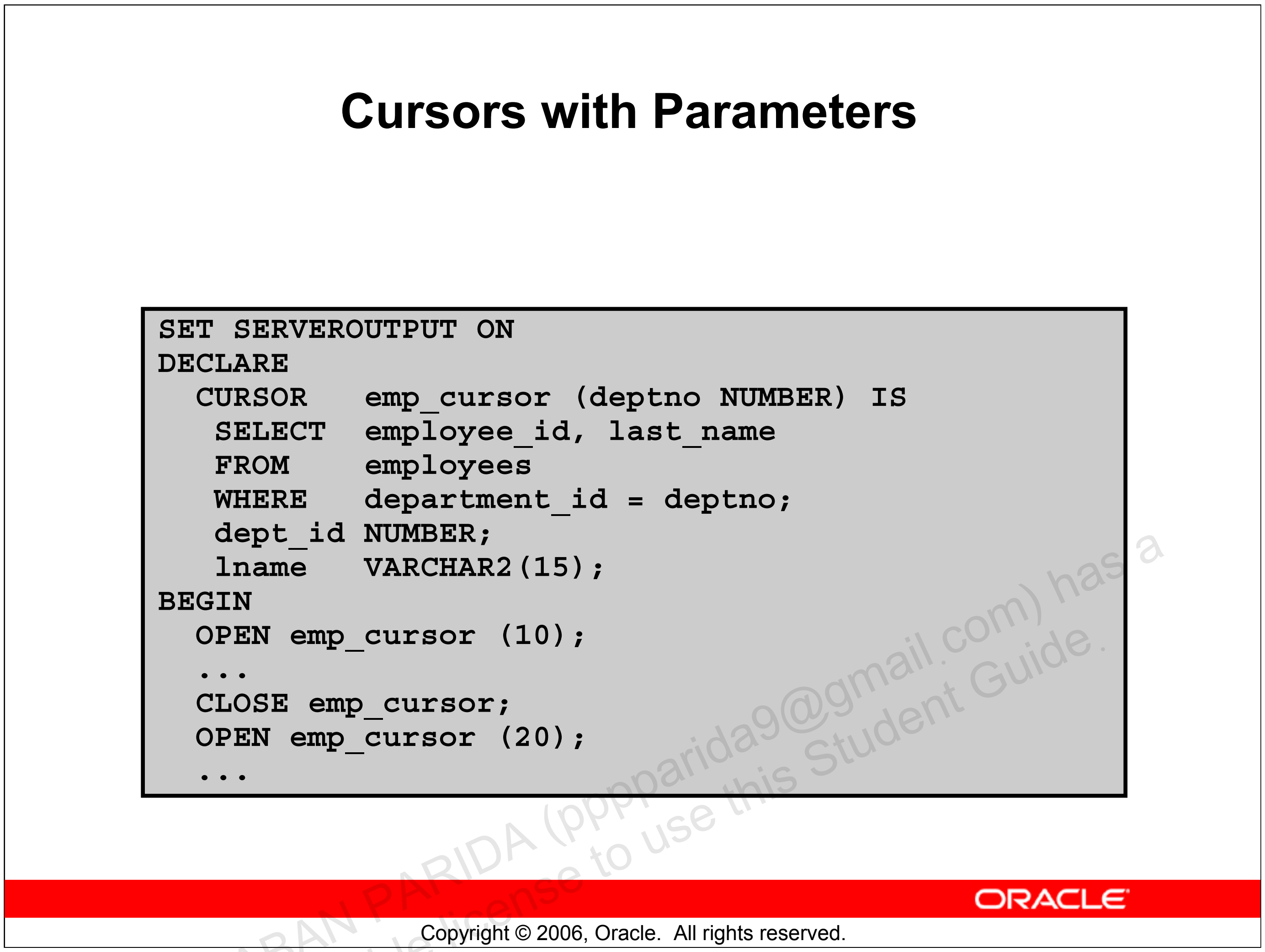
You can pass parameters to a cursor in a cursor `FOR` loop. This means that you can open and close an explicit cursor several times in a block, returning a different active set on each occasion. For each execution, the previous cursor is closed and reopened with a new set of parameters.

Each formal parameter in the cursor declaration must have a corresponding actual parameter in the `OPEN` statement. Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for references in the query expression of the cursor.

In the syntax:

cursor_name	Is a PL/SQL identifier for the declared cursor
parameter_name	Is the name of a parameter
datatype	Is the scalar data type of the parameter
select_statement	Is a `SELECT` statement without the `INTO` clause

The parameter notation does not offer greater functionality; it simply allows you to specify input values easily and clearly. This is particularly useful when the same cursor is referenced repeatedly.



Cursors with Parameters

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR   emp_cursor (deptno NUMBER) IS
   SELECT  employee_id, last_name
   FROM    employees
   WHERE   department_id = deptno;
   dept_id NUMBER;
   lname   VARCHAR2(15);
BEGIN
  OPEN emp_cursor (10);
  ...
  CLOSE emp_cursor;
  OPEN emp_cursor (20);
  ...
```

Cursors with Parameters (continued)

Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for reference in the cursor's query. In the following example, a cursor is declared and is defined with one parameter:

```
DECLARE
   CURSOR emp_cursor(deptno NUMBER) IS SELECT ...
```

The following statements open the cursor and return different active sets:

```
OPEN emp_cursor(10);
OPEN emp_cursor(20);
```

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

```
DECLARE
CURSOR emp_cursor(p_deptno NUMBER, p_job VARCHAR2)IS
   SELECT ...
BEGIN
   FOR emp_record IN emp_cursor(10, 'Sales') LOOP ...
```

FOR UPDATE Clause

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.**

FOR UPDATE Clause

If there are multiple sessions for a single database, there is the possibility that the rows of a particular table were updated after you opened your cursor. You see the updated data only when you reopen the cursor. Therefore, it is better to have locks on the rows before you update or delete rows. You can lock the rows with the `FOR UPDATE` clause in the cursor query.

In the syntax:

column_reference	Is a column in the table against which the query is performed (A list of columns may also be used.)
`NOWAIT`	Returns an Oracle server error if the rows are locked by another session.

The `FOR UPDATE` clause is the last clause in a select statement, even after `ORDER BY` (if it exists). When querying multiple tables, you can use the `FOR UPDATE` clause to confine row locking to particular tables. `FOR UPDATE OF col_name(s)` locks rows only in tables that contain `col_name(s)`.

The FOR UPDATE Clause (continued)

The SELECT ... FOR UPDATE statement identifies the rows that are to be updated or deleted, and then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure that the row is not changed by another session before the update.

The optional NOWAIT keyword tells the Oracle server not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the NOWAIT keyword, the Oracle server waits until the rows are available.

Example:

```
DECLARE
  CURSOR emp_cursor IS
  SELECT employee_id, last_name, FROM employees
  WHERE department_id = 80 FOR UPDATE OF salary NOWAIT;
  ...
```

If the Oracle server cannot acquire the locks on the rows it needs in a SELECT FOR UPDATE, it waits indefinitely. Use NOWAIT to handle such situations. If the rows are locked by another session and you have specified NOWAIT, opening the cursor results in an error. You can try to open the cursor later. You can use WAIT instead of NOWAIT, specify the number of seconds to wait, and determine whether the rows are unlocked. If the rows are still locked after *n* seconds, an error is returned.

It is not mandatory for the FOR UPDATE OF clause to refer to a column, but it is recommended for better readability and maintenance.

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 emp_cursor;
```

WHERE CURRENT OF Clause

The `WHERE CURRENT OF` clause is used in conjunction with the `FOR UPDATE` clause to refer to the current row in an explicit cursor. The `WHERE CURRENT OF` clause is used in the `UPDATE` or `DELETE` statement, whereas the `FOR UPDATE` clause is specified in the cursor declaration. You can use the combination for updating and deleting the current row from the corresponding database table. This enables you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the row ID. You must include the `FOR UPDATE` clause in the cursor query so that the rows are locked on `OPEN`.

In the syntax:

cursor	Is the name of a declared cursor (The cursor must have been declared with the `FOR UPDATE` clause.)

Cursors with Subqueries

Example

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.department_id, t1.department_name,
           t2.staff
    FROM   departments t1, (SELECT department_id,
                                   COUNT(*) AS staff
                            FROM employees
                            GROUP BY department_id) t2
    WHERE t1.department_id = t2.department_id
    AND   t2.staff >= 3;
...
```

Cursors with Subqueries

A subquery is a query (usually enclosed by parentheses) that appears within another SQL statement. When evaluated, the subquery provides a value or set of values to the outer query. Subqueries are often used in the `WHERE` clause of a select statement. They can also be used in the `FROM` clause, creating a temporary data source for that query.

In the example in the slide, the subquery creates a data source consisting of department numbers and the number of employees in each department (known by the alias `STAFF`). A table alias, `t2`, refers to this temporary data source in the `FROM` clause. When this cursor is opened, the active set contains the department number, department name, and total number of employees working for those departments that have three or more employees.

Summary

In this lesson, you should have learned how to:

- **Distinguish cursor types:**
  - **Implicit cursors are used for all `DML` statements and single-row queries.**
  - **Explicit cursors are used for queries of zero, one, or more rows.**
- **Create and handle explicit cursors**
- **Use simple loops and cursor `FOR` loops to handle multiple rows in the cursors**
- **Evaluate the cursor status by using the cursor attributes**
- **Use the `FOR UPDATE` and `WHERE CURRENT OF` clauses to update or delete the current fetched row**

Summary

The Oracle server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: `%FOUND`, `%ISOPEN` `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor `FOR` loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor `FOR` loops do this implicitly. If you are updating or deleting rows, lock the rows by using a `FOR UPDATE` clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a `WHERE CURRENT OF` clause in conjunction with the `FOR UPDATE` clause to reference the current row fetched by the cursor.

Practice 7: Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query rows of a table
- Using a cursor `FOR` loop
- Applying cursor attributes to test the cursor status
- Declaring and using cursors with parameters
- Using the `FOR UPDATE` and `WHERE CURRENT OF` clauses

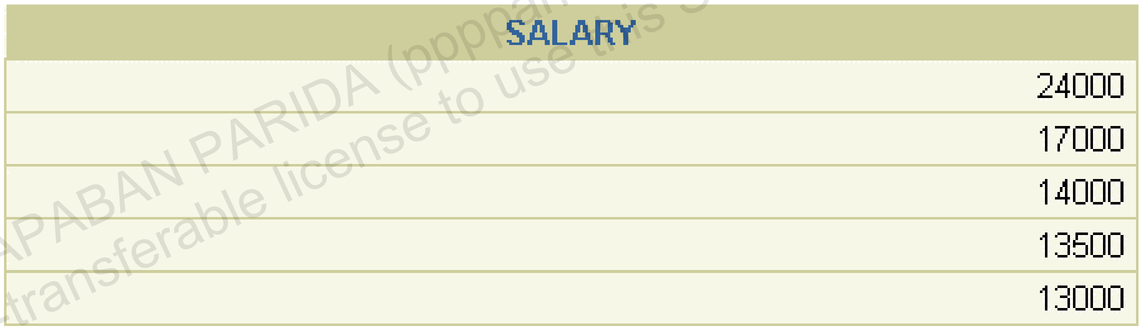
Practice 7: Overview

In this practice, you apply your knowledge of cursors to process a number of rows from a table and populate another table with the results using a cursor `FOR` loop. You also write a cursor with parameters.

Practice 7

1. Create a PL/SQL block that determines the top *n* salaries of the employees.
   a. Execute the script `lab_07_01.sql` to create a new table, `top_salaries`, for storing the salaries of the employees.
   b. Accept a number *n* from the user where *n* represents the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter 5.

      Note: Use the `DEFINE` command to define a variable `p_num` to provide the value for *n*. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.
   c. In the declarative section, declare two variables: `num` of type `NUMBER` to accept the substitution variable `p_num`, `sal` of type `employees.salary`. Declare a cursor, `emp_cursor`, that retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.
   d. In the executable section, open the loop and fetch top *n* salaries and insert them into `top_salaries` table. You can use a simple loop to operate on the data. Also, try and use `%ROWCOUNT` and `%FOUND` attributes for the exit condition.
   e. After inserting into the `top_salaries` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `employees` table.



SALARY
24000
17000
14000
13500
13000

   f. Test a variety of special cases, such as *n* = 0 or where *n* is greater than the number of employees in the `employees` table. Empty the `top_salaries` table after each test.
2. Create a PL/SQL block that does the following:
   a. Use the `DEFINE` command to define a variable `p_deptno` to provide the department ID.
   b. In the declarative section, declare a variable `deptno` of type `NUMBER` and assign the value of `p_deptno`.
   c. Declare a cursor, `emp_cursor`, that retrieves the `last_name`, `salary`, and `manager_id` of the employees working in the department specified in `deptno`.

Practice 7 (continued)

d. In the executable section use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5000 and if the manager ID is either 101 or 124, display the message <<*last_name*>> Due for a raise. Otherwise, display the message <<*last_name*>> Not due for a raise.

e. Test the PL/SQL block for the following cases:

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise Mourgas Not Due for a raise . . . . . . Rajs Due for a raise
80	Russel Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . . . .

Practice 7 (continued)

3. Write a PL/SQL block, which declares and uses cursors with parameters.
   In a loop, use a cursor to retrieve the department number and the department name from the `departments` table for a department whose `department_id` is less than 100. Pass the department number to another cursor as a parameter to retrieve from the `employees` table the details of employee last name, job, hire date, and salary of those employees whose `employee_id` is less than 120 and who work in that department.
   a. In the declarative section, declare a cursor `dept_cursor` to retrieve `department_id`, `department_name` for those departments with `department_id` less than 100. Order by `department_id`.
   b. Declare another cursor `emp_cursor` that takes the department number as parameter and retrieves `last_name`, `job_id`, `hire_date`, and `salary` of those employees with `employee_id` of less than 120 and who work in that department.
   c. Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.
   d. Open the `dept_cursor`, use a simple loop and fetch values into the variables declared. Display the department number and department name.
   e. For each department, open the `emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables and print all the details retrieved from the `employees` table.

      Note: You may want to print a line after you have displayed the details of each department. Use appropriate attributes for the exit condition. Also determine whether a cursor is already open before opening the cursor.
   f. Close all the loops and cursors, and end the executable section. Execute the script.

Practice 7 (continued)

The sample output is shown below.

```
Department Number : 10 Department Name : Administration
--------------------------------------------------------------------------------
Department Number : 20 Department Name : Marketing
--------------------------------------------------------------------------------
Department Number : 30 Department Name : Purchasing
Raphaely PU_MAN 07-DEC-94 11000
Khoo PU_CLERK 18-MAY-95 3100
Baida PU_CLERK 24-DEC-97 2900
Tobias PU_CLERK 24-JUL-97 2800
Himuro PU_CLERK 15-NOV-98 2600
Colmenares PU_CLERK 10-AUG-99 2500
--------------------------------------------------------------------------------
Department Number : 40 Department Name : Human Resources
--------------------------------------------------------------------------------
Department Number : 50 Department Name : Shipping
--------------------------------------------------------------------------------
Department Number : 60 Department Name : IT
Hunold IT_PROG 03-JAN-90 9000
Ernst IT_PROG 21-MAY-91 6000
Austin IT_PROG 25-JUN-97 4800
Pataballa IT_PROG 05-FEB-98 4800
Lorentz IT_PROG 07-FEB-99 4200
--------------------------------------------------------------------------------
Department Number : 70 Department Name : Public Relations
--------------------------------------------------------------------------------
Department Number : 80 Department Name : Sales
--------------------------------------------------------------------------------
Department Number : 90 Department Name : Executive
King AD_PRES 17-JUN-87 24000
Kochhar AD_VP 21-SEP-89 17000
De Haan AD_VP 13-JAN-93 17000
--------------------------------------------------------------------------------
PL/SQL procedure successfully completed.
```

Practice 7 (continued)

4. Load the script `lab_06_04_soln.sql`.
   a. Look for the comment "DECLARE A CURSOR CALLED emp_records TO HOLD salary, first_name, and last_name of employees" and include the declaration. Create the cursor such that it retrieves the salary, first_name, and last_name of employees in the department specified by the user (substitution variable `emp_deptid`). Use the `FOR UPDATE` clause.
   b. Look for the comment "INCLUDE EXECUTABLE SECTION OF INNER BLOCK HERE" and start the executable block.
   c. Only employees working in the departments with `department_id` 20, 60, 80,100, and 110 are eligible for raises this quarter. Check if the user has entered any of these department IDs. If the value does not match, display the message "SORRY, NO SALARY REVISIONS FOR EMPLOYEES IN THIS DEPARTMENT." If the value matches, then, open the cursor `emp_records`.
   d. Start a simple loop and fetch the values into `emp_sal`, `emp_fname`, and `emp_lname`. Use `%NOTFOUND` for the exit condition.
   e. Include a `CASE` expression. Use the following table as reference for the conditions in the `WHEN` clause of the `CASE` expression.
      Note: In your `CASE` expression use the constants such as `c_range1`, `c_hike1` which are already declared.

`salary`	**Hike percentage**
< 6500	20
> 6500 < 9500	15
> 9500 <12000	8
>12000	3

   For example, if the salary of the employee is less than 6500, then increase the salary by 20 percent. In every `WHEN` clause, concatenate the `first_name` and `last_name` of the employee and store it in the `INDEX BY` table. Increment the value in variable `i` so that you can store the string in the next location. Include an `UPDATE` statement with the `WHERE CURRENT OF` clause.
   f. Close the loop. Use the `%ROWCOUNT` attribute and print the number of records that were modified. Close the cursor.
   g. Include a simple loop to print the names of all the employees whose salaries were revised.
      Note: You already have the names of these employees in the `INDEX BY` table.
      Look for the comment "CLOSE THE INNER BLOCK" and include an `END IF` statement and an `END` statement.
   f. Save your script as `lab_07_04_soln.sql`.