21

Writing Explicit Cursors

Schedule:	Timing	Topic
	45 minutes	Lecture
	25 minutes	Practice
	70 minutes	Total

Objectives

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

- **Distinguish between an implicit and an explicit cursor**
- **Use a PL/SQL record variable**
- **Write a Cursor FOR loop**

Aim

In this lesson you will learn the differences between implicit and explicit cursors. You will also learn when and why to use an explicit cursor.

You may need to use a multiple row SELECT statement in PL/SQL to process many rows. To accomplish this, you declare and control explicit cursors, which are used in loops, including the cursor FOR loop.

About Cursors

Every SQL statement executed by the Oracle8 Server has an individual cursor associated with it:

- **Implicit cursors: Declared for all DML and PL/SQL SELECT statements.**
- **Explicit cursors: Declared and named by the programmer.**

Implicit and Explicit Cursors

The Oracle8 Server uses work areas called "private SQL areas" to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. The cursor directs all phases of processing.

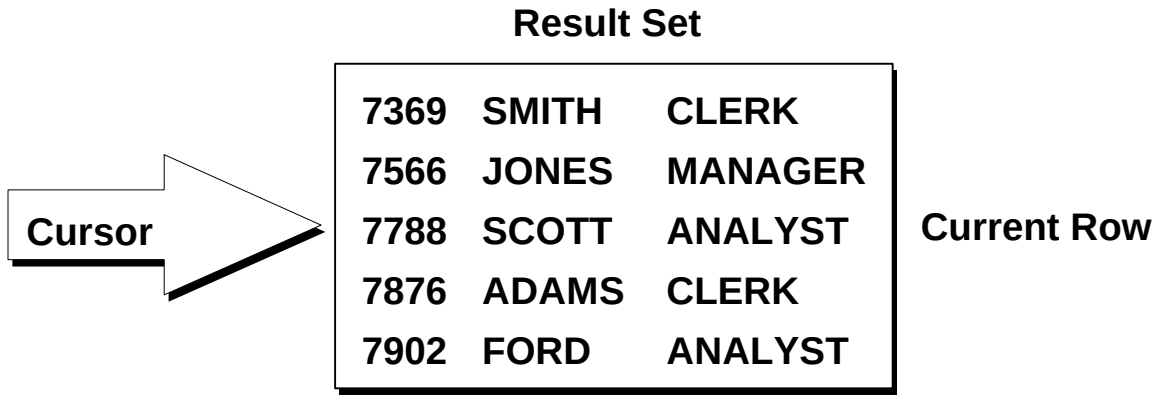
Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.
Explicit	For queries that return more than one row. Explicit cursors are declared and named by the programmer and manipulated through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

Implicit Cursors

The Oracle8 Server implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the "SQL" cursor.

You cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor, but you can use cursor attributes to get information about the most recently executed SQL statement.

Explicit Cursor Functions



Explicit Cursors

Use explicit cursors to individually process each row returned by a multi-row SELECT statement.

The set of rows returned by a multi-row query is called the *result 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 result set. This allows your program to process the rows one at a time.

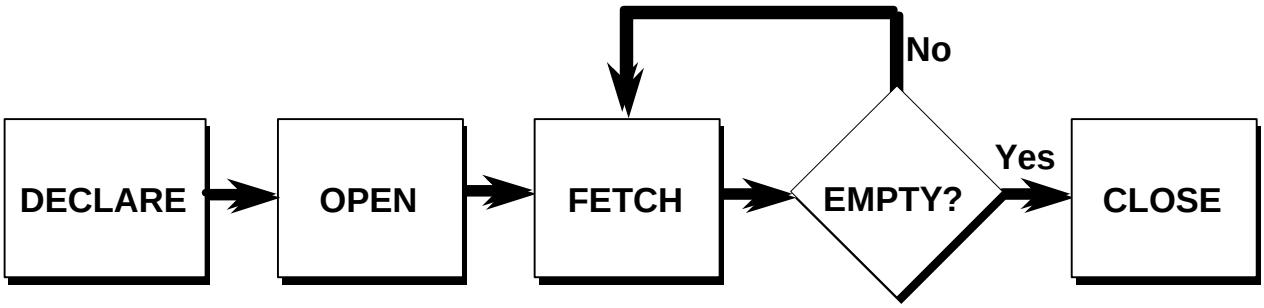
A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in a result set.

Explicit Cursor Functions

- Can process beyond the first row returned by the query, row by row.
- Keep track of which row is currently being processed.
- Allow the programmer to manually control them in the PL/SQL block.

Note: The fetch for an implicit cursor is an array fetch, and the existence of a second row still raises the TOO_MANY_ROWS exception. Furthermore, you can use explicit cursors to perform multiple fetches and to re-execute parsed queries in the work area.

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 found
- Release the active set

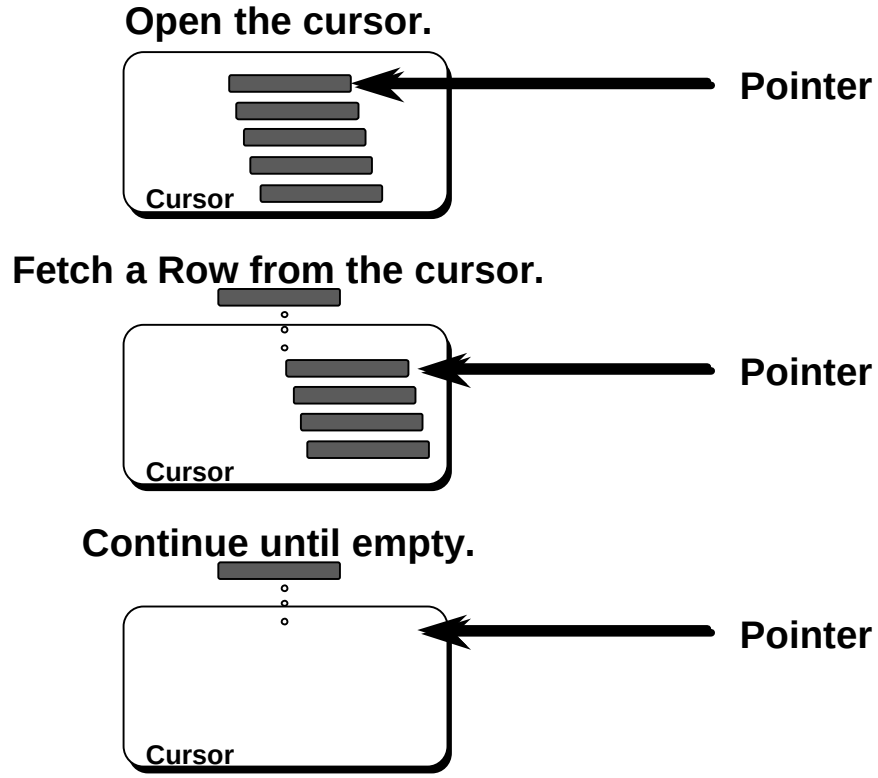
Explicit Cursors (continued)

Now that you have a conceptual understanding of cursors, review the steps to use them. The syntax for each step can be found on the following pages.

Controlling Explicit Cursors Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within 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. The FETCH statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore each fetch accesses a different row returned by the query. In the flow diagram shown in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise it closes 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



Explicit Cursors (continued)

You use the OPEN, FETCH, and CLOSE statements to control a cursor. The OPEN statement executes the query associated with the cursor, identifies the result set, and positions the cursor (pointer) before the first row. The FETCH statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the CLOSE statement disables the cursor.

Declaring the Cursor

Syntax

```
CURSOR cursor_name IS
     select_statement;
```

- **Do not include the INTO clause in the cursor declaration.**
- **If processing rows in a specific sequence is required use the ORDER BY clause in the query.**

Explicit Cursor Declaration

Use the CURSOR statement to declare an explicit cursor. You can reference variables within the query, but you must declare them before the CURSOR statement.

In the syntax,

cursor_name	is a PL/SQL identifier.
select_statement	is a SELECT statement without an INTO clause.

Note: Do not include the INTO clause in the cursor declaration because it appears later in the FETCH statement.

Class Management Note

The parameter and return elements of the cursor definition and syntax will be covered in Lesson 22, page 22-3.

Declaring the Cursor

Example

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename, job, sal
    FROM   emp
    WHERE  sal > 2000;
  CURSOR c2 RETURN dept%ROWTYPE IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```


Explicit Cursor Declaration (continued)

You can reference variables in the query, but you must declare them before the CURSOR statement.

Retrieve the line items of an order one by one.

```
DECLARE
...
  v_ordid             item.ordid%TYPE;
  v_prodid            item.prodid%TYPE;
  v_item_total        NUMBER(11,2);
  CURSOR item_cursor IS
    SELECT prodid, price * quantity
    FROM   item
    WHERE  ordid = v_ordid;
BEGIN
...
```

Opening the Cursor

Syntax

```
OPEN  cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**

OPEN Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax,

cursor_name is the name of the previously declared cursor.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information.
2. Parses the SELECT statement.
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses.
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
5. Positions the pointer just before 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 cursor's status after a fetch.

For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows.

Fetching Data from the Cursor

Syntax

```
FETCH cursor_name INTO [variable1, variable2, ...]
                       | record_name];
```

- **Retrieve the current row values into output variables.**
- **Include the same number of variables.**
- **Match each variable to correspond to the columns positionally.**
- **Test to see if the cursor contains rows.**

FETCH Statement

The FETCH statement retrieves the rows in the result set one at a time. After each fetch, the cursor advances to the next row in the result set.

In the syntax,

cursor_name	is the name of the previously declared cursor.
variable	is an output variable to store the results.
record_name	is the name of the record in which the retrieved data is stored. The record variable can be declared using the %ROWTYPE attribute.

Guidelines

- Include the same number of variables in the INTO clause of the FETCH statement as output columns in the SELECT statement, and be sure that the datatypes 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 if 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.

Note: The FETCH statement performs the following operations:

1. Advances the pointer to the next row in the active set.
2. Reads the data for the current row into the output PL/SQL variables.
3. Exits the cursor FOR loop (see Lesson 22) if the pointer is positioned at the end of the active set.

Fetching Data from the Cursor

Examples

```
FETCH c1 INTO v_deptno, v_dname, v_location;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_record
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```


FETCH Statement (continued)

You use the FETCH statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their datatypes must be compatible.

Retrieve the line items of an order one by one.

```
FETCH item_cursor
INTO v_prodid, v_itemtotal;
```

In the slide example, the value from PRODID will go into V_PRODID, and the value from PRICE*QTY will go into V_ITEMTOTAL.

Closing the Cursor

Syntax

```
CLOSE    cursor_name;
```

- **Close the cursor after completing the processing of the rows.**
- **Reopen the cursor, if required.**
- **Do not attempt to fetch data from a cursor once it has been closed.**

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore you can establish an active set several times.

In the syntax,

cursor_name is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor once it has been closed, or the INVALID_CURSOR exception will be raised.

Note: The CLOSE statement releases the context area.
Although it is possible to terminate the PL/SQL block without closing cursors, you should get into the habit of closing any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter field. OPEN_CURSORS = 50 by default.

```
BEGIN
  OPEN c1;
  FETCH c1 INTO v_deptno;
  ...
  CLOSE c1;
END;
```

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.

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

Note: Do not reference cursor attributes directly in a SQL statement.

Controlling Multiple Fetches

- **Process several rows from an explicit cursor using a loop.**
- **Fetch a row with each iteration.**
- **Use the %NOTFOUND attribute to write a test for an unsuccessful fetch.**
- **Use explicit cursor attributes to test the success of each fetch.**

Controlling Multiple Fetches from Explicit Cursors

To process several rows from an explicit cursor, you typically define a loop to perform a fetch on each iteration. Eventually all rows in the active set are processed, and an unsuccessful fetch sets the %NOTFOUND attribute to TRUE. Use the explicit cursor attributes to test the success of each fetch before any further references are made to the cursor. If you omit an exit criterion, an infinite loop results.

For more information, see
PL/SQL User's Guide and Reference, Release 8 "Interaction With Oracle."

Class Management Note

The reference manual contains a table of cursor values, which is useful for evaluating the attribute value before and after each phase of the cursor management.

The %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 ename_cursor%ISOPEN THEN
   OPEN ename_cursor;
END IF;
LOOP
FETCH ename_cursor...
```


Explicit Cursor Attributes

- You can fetch rows only when the cursor is open. Use the %ISOPEN cursor attribute to determine whether the cursor is open, if necessary.
- Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.
- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows, fetch the rows in a numeric FOR loop, or fetch the rows in a simple loop and determine when to exit the loop.

Note: %ISOPEN returns the status of the cursor; TRUE if open and FALSE if not. It is not usually necessary to inspect %ISOPEN.

The %NOTFOUND and %ROWCOUNT Attributes

- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.
- Use the %NOTFOUND cursor attribute to determine when to exit the loop.

Example

Retrieve the first five line items for an order one by one.

```
...
  v_prodid       item.prodid%TYPE;
  v_itemtotal    NUMBER (11,2);
  v_ordertotal   NUMBER (11,2) := 0;
  CURSOR item_cursor IS    SELECT     prodid, price * quantity
                           FROM       item
                           WHERE      ordid = v_ordid;
BEGIN
  OPEN item_cursor;
  LOOP
    FETCH item_cursor INTO v_prodid, v_itemtotal;
    EXIT WHEN item_cursor%ROWCOUNT > 5
      OR item_cursor%NOTFOUND;
    v_ordertotal := v_ordertotal + v_itemtotal;
  END LOOP;
  CLOSE item_cursor;
END;
```

%NOTFOUND and %ROWCOUNT

Example

```
LOOP
  FETCH ename_cursor INTO v_ename, v_deptno;
  IF ename_cursor%ROWCOUNT > 20 THEN
    ...
  EXIT WHEN ename_cursor%NOTFOUND;
  ...
END LOOP;
```


Note: Before the first fetch, %NOTFOUND evaluates to NULL. So if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If using %ROWCOUNT, add a test for no rows in the cursor by using the %NOTFOUND attribute, because the row count is not incremented if the fetch does not retrieve any rows.

Cursors and Records

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

Example

```
...
  CURSOR emp_cursor IS
    SELECT  empno, sal, hiredate, rowid
    FROM    emp
    WHERE   deptno = 20;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
. . .
  FETCH emp_cursor INTO emp_record;
```


Cursors and Records

You have already seen that you can define records to use 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.

In the example, you can select the ROWID pseudo-column and it will have a corresponding field in the emp_record PL/SQL record.

Cursor FOR Loops

Syntax

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

- **Shortcut to process explicit cursors.**
- **Implicit open, fetch, and close occur.**
- **Do not declare the record; it is implicitly declared.**

A cursor FOR loop processes rows in an explicit cursor. It is a shortcut because the cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was 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. Its scope is only in the loop.
- 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. See Lesson 22 for more information on cursor parameters.
- Do not use a cursor FOR loop when the cursor operations must be handled manually.

Note: You can define a query at the start of the loop itself. The query expression is called a SELECT substatement, and the cursor is internal to the FOR loop. Because the cursor is not declared with a name, you cannot test its attributes.

Cursor FOR Loops

Retrieve line items for an order one by one until there are no more line items left.

Example

```
FOR item_record IN item_cursor LOOP
         -- implicit open and implicit fetch occur
  v_ordertotal := v_ordertotal +
                    (item_record.price *
                    item_record.quantity);
  i := i + 1;
  prodid_table (i) := item_record.prodid;
  ordertotal_table (i) := v_ordertotal;
END LOOP; -- implicit close occurs
```


Class Management Note

Point out the similarities between a cursor FOR loop and a numeric FOR loop; a numeric FOR loop specifies a range of numeric values and then takes on each value within that range; a cursor FOR loop specifies a range of rows from a database table and then retrieves each row within that range.

Process the Rows in a Loop

Example

Retrieve the line items for an order one by one until there are no more line items left. Store the cumulative total for each product in a PL/SQL table.

```
ACCEPT p_ordid   PROMPT 'Please enter the order ID: '
DECLARE
  TYPE id_table_type IS TABLE OF NUMBER(9)
    INDEX BY BINARY_INTEGER;
  TYPE money_table_type IS TABLE OF NUMBER(11,2)
    INDEX BY BINARY_INTEGER;
  v_ordid                    item.ordid%TYPE := &p_ordid
  v_order_total              NUMBER(11,2);
  prodid_table               id_table_type;
  cumulative_total_table     money_table_type;
  i                          BINARY_INTEGER :=0;
  CURSOR item_cursor IS
    SELECT prodid, actualprice, qty
    FROM   item
    WHERE  ordid = v_ordid;
BEGIN
  ...
  FOR item_record IN item_cursor LOOP
    v_order_total := v_order_total + (item_record.price *
                                      item_record.qty);
    i := i + 1;
    prodid_table(i) := item_record.prodid;
    cumulative_total_table(i) := v_order_total;
  END LOOP;
  ...
END;
/
```

Summary

- **Cursor types:**
  - **Implicit cursors: Used for all DML statements and single-row queries.**
  - **Explicit cursors: Used for queries of zero, one, or more rows.**
- **Manipulate explicit cursor with commands:**
  - **Declare, open, fetch, and close.**

Summary

- **Evaluate the cursor status by using cursor attributes.**
- **Use cursor FOR loops as a shortcut to:**
  - **open the cursor.**
  - **fetch rows once for each loop iteration.**
  - **automatically close the cursor after all rows are processed.**

Practice Overview

- **Declaring and using explicit cursors to query rows of a table**
- **Using a cursor FOR loop**
- **Applying cursor attributes to test the cursor status**

Practice Overview

This practice applies 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.

Practice 21

1. Create a PL/SQL block that determines the top employees with respect to salaries.
   a. Accept a number *n* as user input with a SQL*Plus substitution parameter.
   b. In a loop, get the last names and salaries of the top *n* people with respect to salary in the EMP table.
   c. Store the names and salaries in the TOP_DOGS table.
   d. Assume that no two employees have the same salary.
   e. Test a variety of special cases, such a *n* = 0, where *n* is greater than the number of employees in the EMP table. Empty the TOP_DOGS table after each test.

```
Please enter the number of top money makers: 5
NAME        SALARY
----------- ------
KING          5000
FORD          3000
SCOTT         3000
JONES         2975
BLAKE         2850
```

2. Consider the case where several employees have the same salary. If one person is listed, then all people who have the same salary should also be listed.
   a. For example, if the user enters a value of 2 for *n*, then King, Ford and Scott should be displayed. (These employees are tied for second highest salary.)
   b. If the user enters a value of 4, then King, Ford, Scott, and Jones should be displayed.
   c. Delete all rows from TOP_DOGS and test the practice.

```
Please enter the number of top money makers: 2
NAME         SALARY
------------ ------
KING           5000
FORD           3000
SCOTT          3000
```

Practice 21 (continued)

```
Please enter the number of top money makers: 2
NAME         SALARY
------------ ------
KING           5000
FORD           3000
SCOTT          3000
JONES          2975
```

21

Writing Explicit Cursors

Schedule:	Timing	Topic
	45 minutes	Lecture
	25 minutes	Practice
	70 minutes	Total

Objectives

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

- **Distinguish between an implicit and an explicit cursor**
- **Use a PL/SQL record variable**
- **Write a Cursor FOR loop**

Aim

In this lesson you will learn the differences between implicit and explicit cursors. You will also learn when and why to use an explicit cursor.

You may need to use a multiple row SELECT statement in PL/SQL to process many rows. To accomplish this, you declare and control explicit cursors, which are used in loops, including the cursor FOR loop.

About Cursors

Every SQL statement executed by the Oracle8 Server has an individual cursor associated with it:

- **Implicit cursors: Declared for all DML and PL/SQL SELECT statements.**
- **Explicit cursors: Declared and named by the programmer.**

Implicit and Explicit Cursors

The Oracle8 Server uses work areas called "private SQL areas" to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. The cursor directs all phases of processing.

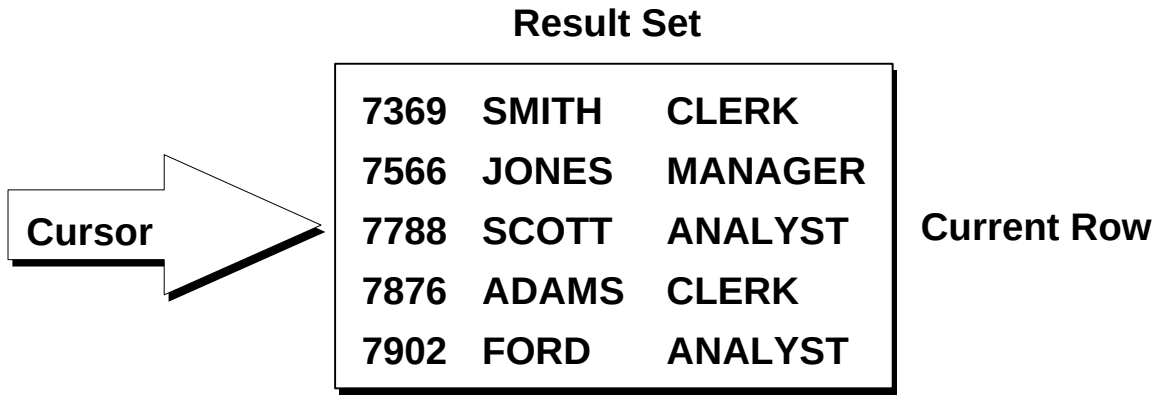
Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.
Explicit	For queries that return more than one row. Explicit cursors are declared and named by the programmer and manipulated through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

Implicit Cursors

The Oracle8 Server implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the "SQL" cursor.

You cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor, but you can use cursor attributes to get information about the most recently executed SQL statement.

Explicit Cursor Functions



Explicit Cursors

Use explicit cursors to individually process each row returned by a multi-row SELECT statement.

The set of rows returned by a multi-row query is called the *result 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 result set. This allows your program to process the rows one at a time.

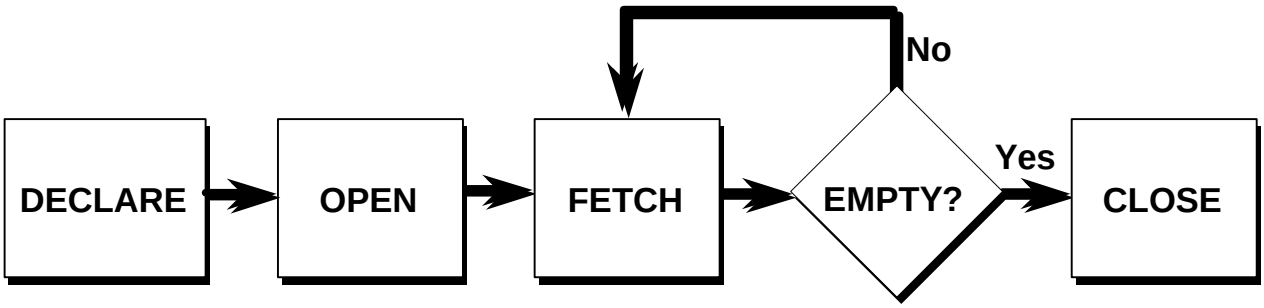
A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in a result set.

Explicit Cursor Functions

- Can process beyond the first row returned by the query, row by row.
- Keep track of which row is currently being processed.
- Allow the programmer to manually control them in the PL/SQL block.

Note: The fetch for an implicit cursor is an array fetch, and the existence of a second row still raises the TOO_MANY_ROWS exception. Furthermore, you can use explicit cursors to perform multiple fetches and to re-execute parsed queries in the work area.

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 found
- Release the active set

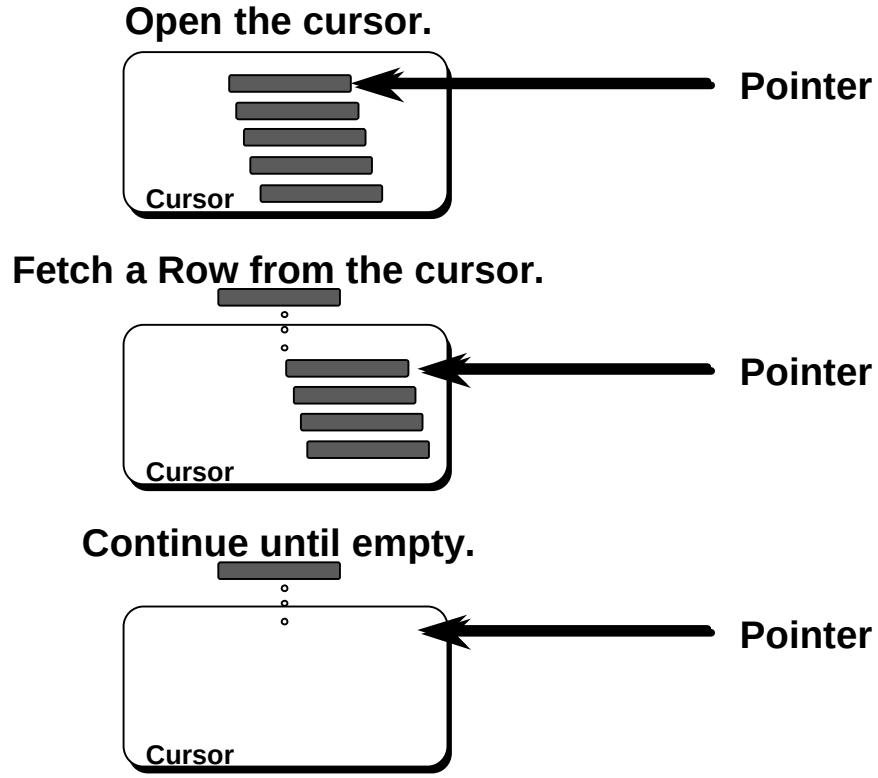
Explicit Cursors (continued)

Now that you have a conceptual understanding of cursors, review the steps to use them. The syntax for each step can be found on the following pages.

Controlling Explicit Cursors Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within 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. The FETCH statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore each fetch accesses a different row returned by the query. In the flow diagram shown in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise it closes 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



Explicit Cursors (continued)

You use the OPEN, FETCH, and CLOSE statements to control a cursor. The OPEN statement executes the query associated with the cursor, identifies the result set, and positions the cursor (pointer) before the first row. The FETCH statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the CLOSE statement disables the cursor.

Declaring the Cursor

Syntax

```
CURSOR cursor_name IS
     select_statement;
```

- **Do not include the INTO clause in the cursor declaration.**
- **If processing rows in a specific sequence is required use the ORDER BY clause in the query.**

Explicit Cursor Declaration

Use the CURSOR statement to declare an explicit cursor. You can reference variables within the query, but you must declare them before the CURSOR statement.

In the syntax,

cursor_name	is a PL/SQL identifier.
select_statement	is a SELECT statement without an INTO clause.

Note: Do not include the INTO clause in the cursor declaration because it appears later in the FETCH statement.

Class Management Note

The parameter and return elements of the cursor definition and syntax will be covered in Lesson 22, page 22-3.

Declaring the Cursor

Example

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename, job, sal
    FROM   emp
    WHERE  sal > 2000;
  CURSOR c2 RETURN dept%ROWTYPE IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```


Explicit Cursor Declaration (continued)

You can reference variables in the query, but you must declare them before the CURSOR statement.

Retrieve the line items of an order one by one.

```
DECLARE
...
  v_ordid             item.ordid%TYPE;
  v_prodid            item.prodid%TYPE;
  v_item_total        NUMBER(11,2);
  CURSOR item_cursor IS
    SELECT prodid, price * quantity
    FROM   item
    WHERE  ordid = v_ordid;
BEGIN
...
```

Opening the Cursor

Syntax

```
OPEN  cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**

OPEN Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax,

cursor_name is the name of the previously declared cursor.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information.
2. Parses the SELECT statement.
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses.
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
5. Positions the pointer just before 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 cursor's status after a fetch.

For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows.

Fetching Data from the Cursor

Syntax

```
FETCH cursor_name INTO [variable1, variable2, ...]
                       | record_name];
```

- **Retrieve the current row values into output variables.**
- **Include the same number of variables.**
- **Match each variable to correspond to the columns positionally.**
- **Test to see if the cursor contains rows.**

FETCH Statement

The FETCH statement retrieves the rows in the result set one at a time. After each fetch, the cursor advances to the next row in the result set.

In the syntax,

cursor_name	is the name of the previously declared cursor.
variable	is an output variable to store the results.
record_name	is the name of the record in which the retrieved data is stored. The record variable can be declared using the %ROWTYPE attribute.

Guidelines

- Include the same number of variables in the INTO clause of the FETCH statement as output columns in the SELECT statement, and be sure that the datatypes 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 if 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.

Note: The FETCH statement performs the following operations:

1. Advances the pointer to the next row in the active set.
2. Reads the data for the current row into the output PL/SQL variables.
3. Exits the cursor FOR loop (see Lesson 22) if the pointer is positioned at the end of the active set.

Fetching Data from the Cursor

Examples

```
FETCH c1 INTO v_deptno, v_dname, v_location;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_record
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```


FETCH Statement (continued)

You use the FETCH statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their datatypes must be compatible.

Retrieve the line items of an order one by one.

```
FETCH item_cursor
INTO v_prodid, v_itemtotal;
```

In the slide example, the value from PRODID will go into V_PRODID, and the value from PRICE*QTY will go into V_ITEMTOTAL.

Closing the Cursor

Syntax

```
CLOSE    cursor_name;
```

- **Close the cursor after completing the processing of the rows.**
- **Reopen the cursor, if required.**
- **Do not attempt to fetch data from a cursor once it has been closed.**

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore you can establish an active set several times.

In the syntax,

cursor_name is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor once it has been closed, or the INVALID_CURSOR exception will be raised.

Note: The CLOSE statement releases the context area.
Although it is possible to terminate the PL/SQL block without closing cursors, you should get into the habit of closing any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter field. OPEN_CURSORS = 50 by default.

```
BEGIN
  OPEN c1;
  FETCH c1 INTO v_deptno;
  ...
  CLOSE c1;
END;
```

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.

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

Note: Do not reference cursor attributes directly in a SQL statement.

Controlling Multiple Fetches

- **Process several rows from an explicit cursor using a loop.**
- **Fetch a row with each iteration.**
- **Use the %NOTFOUND attribute to write a test for an unsuccessful fetch.**
- **Use explicit cursor attributes to test the success of each fetch.**

Controlling Multiple Fetches from Explicit Cursors

To process several rows from an explicit cursor, you typically define a loop to perform a fetch on each iteration. Eventually all rows in the active set are processed, and an unsuccessful fetch sets the %NOTFOUND attribute to TRUE. Use the explicit cursor attributes to test the success of each fetch before any further references are made to the cursor. If you omit an exit criterion, an infinite loop results.

For more information, see
PL/SQL User's Guide and Reference, Release 8 "Interaction With Oracle."

Class Management Note

The reference manual contains a table of cursor values, which is useful for evaluating the attribute value before and after each phase of the cursor management.

The %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 ename_cursor%ISOPEN THEN
   OPEN ename_cursor;
END IF;
LOOP
FETCH ename_cursor...
```


Explicit Cursor Attributes

- You can fetch rows only when the cursor is open. Use the %ISOPEN cursor attribute to determine whether the cursor is open, if necessary.
- Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.
- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows, fetch the rows in a numeric FOR loop, or fetch the rows in a simple loop and determine when to exit the loop.

Note: %ISOPEN returns the status of the cursor; TRUE if open and FALSE if not. It is not usually necessary to inspect %ISOPEN.

The %NOTFOUND and %ROWCOUNT Attributes

- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.
- Use the %NOTFOUND cursor attribute to determine when to exit the loop.

Example

Retrieve the first five line items for an order one by one.

```
...
  v_prodid       item.prodid%TYPE;
  v_itemtotal    NUMBER (11,2);
  v_ordertotal   NUMBER (11,2) := 0;
  CURSOR item_cursor IS    SELECT     prodid, price * quantity
                           FROM       item
                           WHERE      ordid = v_ordid;
BEGIN
  OPEN item_cursor;
  LOOP
    FETCH item_cursor INTO v_prodid, v_itemtotal;
    EXIT WHEN item_cursor%ROWCOUNT > 5
      OR item_cursor%NOTFOUND;
    v_ordertotal := v_ordertotal + v_itemtotal;
  END LOOP;
  CLOSE item_cursor;
END;
```

%NOTFOUND and %ROWCOUNT

Example

```
LOOP
  FETCH ename_cursor INTO v_ename, v_deptno;
  IF ename_cursor%ROWCOUNT > 20 THEN
    ...
  EXIT WHEN ename_cursor%NOTFOUND;
  ...
END LOOP;
```


Note: Before the first fetch, %NOTFOUND evaluates to NULL. So if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If using %ROWCOUNT, add a test for no rows in the cursor by using the %NOTFOUND attribute, because the row count is not incremented if the fetch does not retrieve any rows.

Cursors and Records

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

Example

```
...
  CURSOR emp_cursor IS
    SELECT  empno, sal, hiredate, rowid
    FROM    emp
    WHERE   deptno = 20;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
. . .
  FETCH emp_cursor INTO emp_record;
```


Cursors and Records

You have already seen that you can define records to use 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.

In the example, you can select the ROWID pseudo-column and it will have a corresponding field in the emp_record PL/SQL record.

Cursor FOR Loops

Syntax

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

- **Shortcut to process explicit cursors.**
- **Implicit open, fetch, and close occur.**
- **Do not declare the record; it is implicitly declared.**

A cursor FOR loop processes rows in an explicit cursor. It is a shortcut because the cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was 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. Its scope is only in the loop.
- 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. See Lesson 22 for more information on cursor parameters.
- Do not use a cursor FOR loop when the cursor operations must be handled manually.

Note: You can define a query at the start of the loop itself. The query expression is called a SELECT substatement, and the cursor is internal to the FOR loop. Because the cursor is not declared with a name, you cannot test its attributes.

Cursor FOR Loops

Retrieve line items for an order one by one until there are no more line items left.

Example

```
FOR item_record IN item_cursor LOOP
         -- implicit open and implicit fetch occur
  v_ordertotal := v_ordertotal +
                    (item_record.price *
                    item_record.quantity);
  i := i + 1;
  prodid_table (i) := item_record.prodid;
  ordertotal_table (i) := v_ordertotal;
END LOOP; -- implicit close occurs
```


Class Management Note

Point out the similarities between a cursor FOR loop and a numeric FOR loop; a numeric FOR loop specifies a range of numeric values and then takes on each value within that range; a cursor FOR loop specifies a range of rows from a database table and then retrieves each row within that range.

Process the Rows in a Loop

Example

Retrieve the line items for an order one by one until there are no more line items left. Store the cumulative total for each product in a PL/SQL table.

```
ACCEPT p_ordid   PROMPT 'Please enter the order ID: '
DECLARE
  TYPE id_table_type IS TABLE OF NUMBER(9)
    INDEX BY BINARY_INTEGER;
  TYPE money_table_type IS TABLE OF NUMBER(11,2)
    INDEX BY BINARY_INTEGER;
  v_ordid                    item.ordid%TYPE := &p_ordid
  v_order_total              NUMBER(11,2);
  prodid_table               id_table_type;
  cumulative_total_table     money_table_type;
  i                          BINARY_INTEGER :=0;
  CURSOR item_cursor IS
    SELECT prodid, actualprice, qty
    FROM   item
    WHERE  ordid = v_ordid;
BEGIN
  ...
  FOR item_record IN item_cursor LOOP
    v_order_total := v_order_total + (item_record.price *
                                      item_record.qty);
    i := i + 1;
    prodid_table(i) := item_record.prodid;
    cumulative_total_table(i) := v_order_total;
  END LOOP;
  ...
END;
/
```

Summary

- **Cursor types:**
  - **Implicit cursors: Used for all DML statements and single-row queries.**
  - **Explicit cursors: Used for queries of zero, one, or more rows.**
- **Manipulate explicit cursor with commands:**
  - **Declare, open, fetch, and close.**

Summary

- Evaluate the cursor status by using cursor attributes.
- Use cursor FOR loops as a shortcut to:
  - open the cursor.
  - fetch rows once for each loop iteration.
  - automatically close the cursor after all rows are processed.

Practice Overview

- **Declaring and using explicit cursors to query rows of a table**
- **Using a cursor FOR loop**
- **Applying cursor attributes to test the cursor status**

Practice Overview

This practice applies 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.

Practice 21

1. Create a PL/SQL block that determines the top employees with respect to salaries.
   a. Accept a number *n* as user input with a SQL*Plus substitution parameter.
   b. In a loop, get the last names and salaries of the top *n* people with respect to salary in the EMP table.
   c. Store the names and salaries in the TOP_DOGS table.
   d. Assume that no two employees have the same salary.
   e. Test a variety of special cases, such a *n* = 0, where *n* is greater than the number of employees in the EMP table. Empty the TOP_DOGS table after each test.

```
Please enter the number of top money makers: 5
NAME        SALARY
----------- ------
KING          5000
FORD          3000
SCOTT         3000
JONES         2975
BLAKE         2850
```

2. Consider the case where several employees have the same salary. If one person is listed, then all people who have the same salary should also be listed.
   a. For example, if the user enters a value of 2 for *n*, then King, Ford and Scott should be displayed. (These employees are tied for second highest salary.)
   b. If the user enters a value of 4, then King, Ford, Scott, and Jones should be displayed.
   c. Delete all rows from TOP_DOGS and test the practice.

```
Please enter the number of top money makers: 2
NAME         SALARY
------------ ------
KING           5000
FORD           3000
SCOTT          3000
```

Practice 21 (continued)

```
Please enter the number of top money makers: 2
NAME         SALARY
------------ ------
KING           5000
FORD           3000
SCOTT          3000
JONES          2975
```

21

Handling Exceptions

Schedule:	Timing	Topic
	45 minutes	Lecture
	20 minutes	Practice
	65 minutes	Total

Objectives

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

- **Define PL/SQL exceptions**
- **Recognize unhandled exceptions**
- **List and use different types of PL/SQL exception handlers**
- **Trap unanticipated errors**
- **Describe the effect of exception propagation in nested blocks**
- **Customize PL/SQL exception messages**

Aim

In this lesson you will learn what PL/SQL exceptions are and how to deal with them using predefined, non-predefined, and user-defined exception handlers.

Handling Exceptions with PL/SQL

- What is an exception?
  - Identifier in PL/SQL that is raised during execution.
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

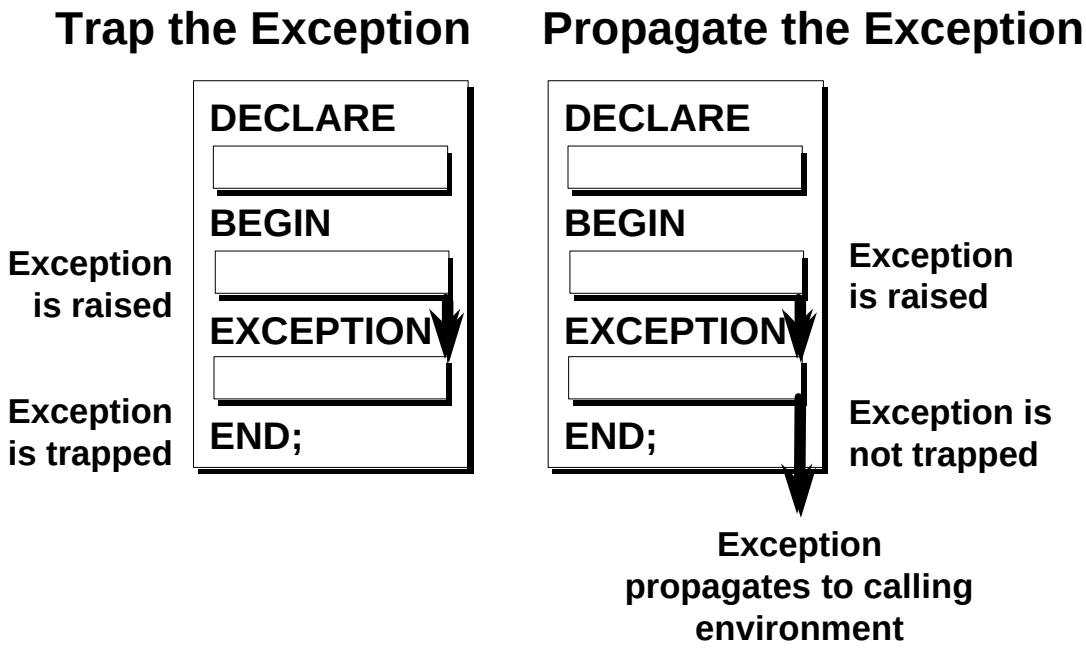
Overview

An exception is an identifier in PL/SQL, raised during the execution of a block that terminates its main body of actions. A block always terminates when PL/SQL raises an exception, but you specify an exception handler to perform final actions.

Two Methods for Raising an Exception

- An Oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a select statement, then PL/SQL raises the exception NO_DATA_FOUND.
- You raise an exception explicitly by issuing the RAISE statement within the block. The exception being raised may be either user defined or predefined.

Handling Exceptions



Trapping an Exception

If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or environment. The PL/SQL block terminates successfully.

Propagating an Exception

You can handle an exception by propagating it to the calling environment. If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure.

Exception Types

- **Predefined Oracle8 Server**
- **Non-predefined Oracle8 Server**

} **Implicitly raised**

- **User-defined** — **Explicitly raised**

You can program for exceptions to avoid disruption at runtime. There are three types of exceptions.

Exception	Description	Directions for Handling
Predefined Oracle8 Server error	One of approximately 20 errors that occur most often in PL/SQL code.	Do not declare, and allow the Oracle8 Server to raise them implicitly.
Non-predefined Oracle8 Server error	Any other standard Oracle8 Server error.	Declare within the declarative section, and allow the Oracle8 Server to raise them implicitly
User-defined error	A condition that the developer determines is abnormal.	Declare within the declarative section, *and* raise explicitly.

Note: Some application tools with client-side PL/SQL, such as Developer/2000 Forms, have their own exceptions.

Trapping Exceptions

Syntax

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```


You can trap any error by including a corresponding routine within the exception handling section of the PL/SQL block. Each handler consists of a WHERE clause, which specifies and exception, followed by a sequence of statements to be executed when that exception is raised.

In the syntax,

exception is the standard name of a predefined exception or the name of a user-defined exception declared within the declarative section.

statement is one or more PL/SQL or SQL statements.

OTHERS is an optional exception-handling clause that traps unspecified exceptions.

WHEN OTHERS Exception Handler

The exception-handling section traps only those exceptions specified; any other exceptions are not trapped unless you use the OTHERS exception handler. This traps any exception not yet handled. For this reason, OTHERS is the last exception handler defined.

The OTHERS handler traps *all* exceptions not already trapped. Some Oracle tools have their own predefined exceptions that you can raise to cause events in the application. OTHERS also traps these exceptions.

Trapping Exceptions Guidelines

- **WHEN OTHERS is the last clause.**
- **EXCEPTION keyword starts exception-handling section.**
- **Several exception handlers are allowed.**
- **Only one handler is processed before leaving the block.**

Guidelines

- Begin the exception-handling section of the block with the keyword EXCEPTION.
- Define several exception handlers, each with its own set of actions, for the block.
- When an exception occurs, PL/SQL processes *only one* handler before leaving the block.
- Place the OTHERS clause after all other exception-handling clauses.
- You can have at most one OTHERS clause.
- Exceptions cannot appear in assignment statements or SQL statements.

.

Trapping Predefined Oracle8 Server Errors

- **Reference the standard name in the exception-handling routine.**
- **Sample predefined exceptions:**
  - **NO_DATA_FOUND**
  - **TOO_MANY_ROWS**
  - **INVALID_CURSOR**
  - **ZERO_DIVIDE**
  - **DUP_VAL_ON_INDEX**

Trap a predefined Oracle8 Server error by referencing its standard name within the corresponding exception-handling routine.

Sample Predefined Exceptions

Exception Name	Oracle Server Error Number	Description
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data.
TOO_MANY_ROWS	ORA-01422	Single row SELECT returned more than one row.
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero.
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value.
INVALID_NUMBER	ORA-01722	Convertion of character string to number fails.
LOGIN_DENIED	ORA-01017	Logging on to Oracle with an invalid username and/or password.

For a complete list of predefined exceptions, see *PL/SQL User's Guide and Reference*, Release 8.0, "Error Handling."

Note: PL/SQL declares predefined exceptions in the STANDARD package.

It is a good idea to always consider the NO_DATA_FOUND and TOO_MANY_ROWS exceptions, which are the most common.

Predefined Exception

Syntax

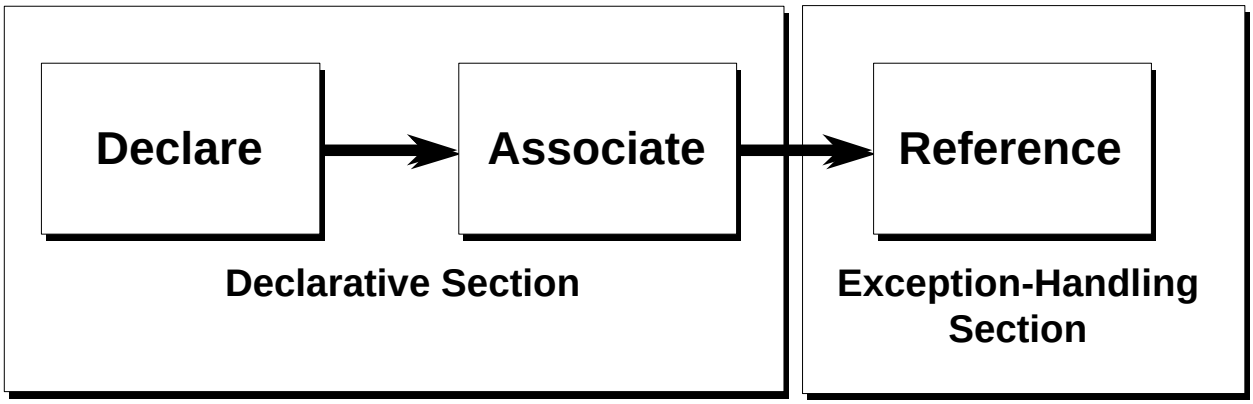
```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1; statement2;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_prodid)||
                         ' is invalid.');
  WHEN TOO_MANY_ROWS THEN
    statement1;
    DBMS_OUTPUT.PUT_LINE('Invalid Data');
  WHEN OTHERS THEN
    statement1; statement2; statement3;
    DBMS_OUTPUT.PUT_LINE('Other error');
END;
```


Trapping Predefined Oracle8 Server Exceptions

In the slide example for each exception, a message is printed out to the user.

Only one exception is raised and handled at any time.

Trapping Non-Predefined Oracle8 Server Errors



- Name the exception
- Code the PRAGMA EXCEPTION_INIT
- Handle the raised exception

You trap a non-predefined Oracle8 Server error by declaring it first, or by using the OTHERS handler. The declared exception is raised implicitly. In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

Note: PRAGMA (also called pseudoinstructions) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle8 Server error number.

Non-Predefined Error

Trap for Oracle8 Server error number - 2292 an integrity constraint violation

```
[DECLARE]
  e_products_remaining  EXCEPTION;
  PRAGMA EXCEPTION_INIT (
            e_products_remaining, -2292);
. . .
BEGIN
. . .
EXCEPTION
  WHEN e_products_remaining THEN
    DBMS_OUTPUT.PUT_LINE ('Product code
            specified is not valid.');
. . .
END;
```

1

2

3

Trapping a Non-Predefined Oracle8 Server Exception

1. Declare the name for the exception within the declarative section.

   Syntax

   ```
   exception  EXCEPTION;
   ```

   where: *exception* is the name of the exception.

2. Associate the declared exception with the standard Oracle8 Server error number using the PRAGMA EXCEPTION_INIT statement.

   Syntax

   ```
   PRAGMA EXCEPTION_INIT(exception, error_number);
   ```

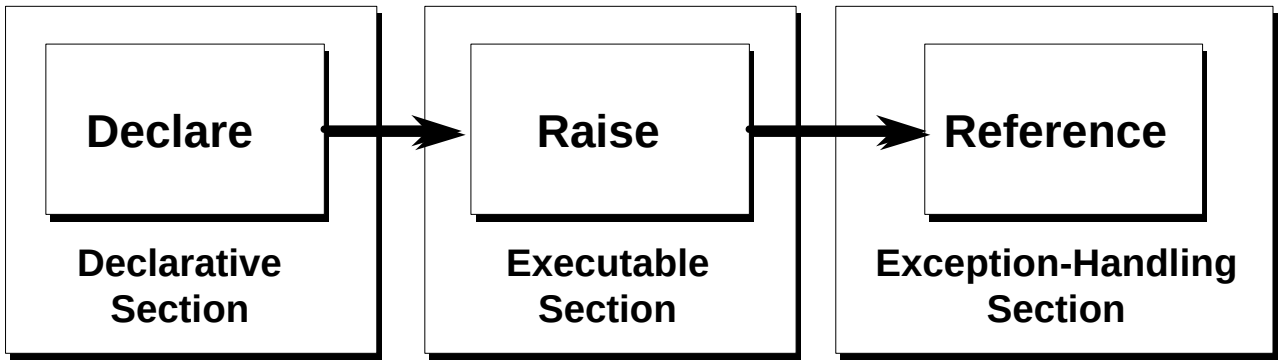
   where: *exception* is the previously declared exception.
   error_number is a standard Oracle8 Server error number.

3. Reference the declared exception within the corresponding exception-handling routine.

   In the slide example: If there is product in stock, halt processing and print a message to the user.

For more information, see *Oracle8 Server Messages, Release 8.*

Trapping User-Defined Exceptions



- **Name the exception**
- **Explicitly raise the exception by using the RAISE statement**
- **Handle the raised exception**

PL/SQL lets you define your own exceptions. User-defined PL/SQL exceptions must be:

- Declared in the declare section of a PL/SQL block.
- Raised explicitly with RAISE statements.

User-Defined Exception

Example

```
[DECLARE]
  e_amount_remaining EXCEPTION;                          1
. . .
BEGIN
. . .
  RAISE e_amount_remaining;                              2
. . .
EXCEPTION
  WHEN e_amount_remaining THEN                           3
    DBMS_OUTPUT.PUT_LINE ('There is still an
amount
      in stock.');
. . .
END;
```


Trapping User-Defined Exceptions

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.

   Syntax

   ```
   exception EXCEPTION;
   ```

   where: *exception* is the name of the exception.

2. Use the RAISE statement to raise the exception explicitly within the executable section.

   Syntax

   ```
   RAISE exception;
   ```

   where: *exception* is the previously declared exception.

3. Reference the declared exception within the corresponding exception handling routine.

In the slide example, the developer has a business rule that states a that a product can not be removed from its database if there is any inventory left in-stock for this product. As there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a DELETE on the PRODUCT table, the block queries the INVENTORY table to see if there is any stock for the product in question. If so, raise an exception.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception back to the calling environment.

Functions for Trapping Exceptions

- **SQLCODE**
  - **Returns the numeric value for the error code.**
- **SQLERRM**
  - **Returns the message associated with the error number.**

Error Trapping Functions

When an exception occurs, you can identify the associated error code or error message by using two functions. Based on the values of the code or message, you can decide what subsequent action to take based on the error.

SQLCODE returns the number of the Oracle error for internal exceptions. You can pass an error number to SQLERRM, which then returns the message associated with the error number.

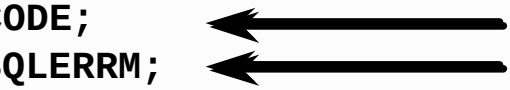
Function	Description
SQLCODE Example SQLCODE Values	Returns the numeric value for the error code. You can assign it to a NUMBER variable.
SQLERRM	Returns character data containing the message associated with the error number.

SQLCODE Value	Description
0	No exception encountered
1	User-defined exception
+100	NO_DATA_FOUND exception
negative number	Another Oracle8 Server error number

Functions for Trapping Exceptions

Example

```
...
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE;
    v_error_message := SQLERRM;
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```



Error Trapping Functions

When an exception is trapped in the WHEN OTHERS section, you can use a set of generic functions to identify those errors.

The example in the slide illustrates the values of SQLCODE and SQLERRM being assigned to variables and then those variables being used in a SQL statement.

Truncate the value of SQLERRM to a known length before attempting to write it to a variable.

Class Management Note

Do not reference the error reporting functions directly in a SQL statement. Instead, first assign their values to a variable. You can pass an error number to SQLERRM to return the associated message; for example, "**SQLERRM(-979)**" returns "ORA-0979: not a GROUP BY expression." These functions can be used as an actual parameter–for example "**error_procedure(SQLERRM)**."

Calling Environments

SQL*Plus	Displays error number and message to screen.
Procedure Builder	Displays error number and message to screen.
Developer/2000 Forms	Accesses error number and message in a trigger by means of the ERROR_CODE and ERROR_TEXT packaged functions.
Precompiler application	Accesses exception number through the SQLCA data structure.
An enclosing PL/SQL block	Traps exception in exception-handling routine of enclosing block.

Propagating Exceptions

Instead of trapping an exception within the PL/SQL block, propagate the exception to allow the calling environment to handle it. Each calling environment has its own way of displaying and accessing errors.

Propagating Exceptions

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
   . . .
   e_no_rows          exception;
   e_integrity        exception;
   PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
   FOR c_record IN emp_cursor LOOP
      BEGIN
        SELECT ...
        UPDATE ...
        IF SQL%NOTFOUND THEN
          RAISE e_no_rows;
        END IF;
      EXCEPTION
        WHEN e_integrity THEN ...
        WHEN e_no_rows THEN ...
      END;
END LOOP;
EXCEPTION
   WHEN NO_DATA_FOUND THEN . . .
   WHEN TOO_MANY_ROWS THEN . . .
END;
```


Propagating an Exception in a Subblock

When a subblock handles an exception, it terminates normally, and control resumes in the enclosing block immediately after the subblock END statement.

However, if PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates in successive enclosing blocks until it finds a handler. If none of these blocks handle the exception, an unhandled exception in the host environment results.

When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

RAISE_APPLICATION_ERROR

Syntax

```
raise_application_error (error_number,
            message[, {TRUE | FALSE}]);
```

- **A procedure that lets you issue user-defined error messages from stored subprograms.**
- **Called only from an executing stored subprogram.**

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR you can report errors to your application and avoid returning unhandled exceptions.

In the syntax,

error_number is a user specified number for the exception between -20000 and -20999.

message is the user-specified message for the exception. It is a character string up to 2048 bytes long.

TRUE | FALSE is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example

```
...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20201,
            'Manager is not a valid employee.');
END;
```

RAISE_APPLICATION_ERROR

- **Used in two different places:**
  - **Executable section.**
  - **Exception section.**
- **Returns error conditions to the user in a manner consistent with other Oracle8 Server errors.**

Example

```
...
DELETE FROM emp
WHERE  mgr = v_mgr;
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20202,'This is not a valid manager');
END IF;
...
```

Summary

- **Program for exceptions that may arise during the execution of the PL/SQL block.**
- **Exception types:**
  - **Predefined Oracle8 Server error.**
  - **Non-predefined Oracle8 Server error.**
  - **User-defined error.**

Summary

- **Handling exceptions:**
  - **Trap the exception within the PL/SQL block in an exception handling routine.**
  - **Allow the exception to propagate to the calling environment.**

Summary

- **Propagate exceptions:**
  - **Propagate an exception through a series of nested blocks.**
  - **Terminate PL/SQL processing with success by handling the exception in an enclosing block.**
  - **Terminate the PL/SQL processing with failure by passing the unhandled exception to the calling environment.**

Practice Overview

- **Handling named exceptions**
- **Creating and invoking user-defined exceptions**

Practice Overview

In this practice, you create exception handlers for specific situations.

Practice 23

1. Write a PL/SQL block to select the name of the employee with a given salary value.
   a. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the MESSAGES table More than one employee with a salary of <*salary*>.
   b. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the MESSAGES table No employee with a salary of <*salary*>.
   c. If the salary entered returns only one row, insert into the MESSAGES table the employees name and the salary amount.
   d. Handle any other exception with an appropriate exception handler and insert into the MESSAGES table Some other error occurred.
   e. Test the block for a variety of test cases.

```
RESULTS
-------------------------------------------
SMITH - 800
More than one employee with a salary of 3000
No employee with a salary of 6000
```

2. Modify *p18q3.sql* to add an exception handler.
   a. Write an exception handler for the error to pass a message to the user that the specified department does not exist.
   b. Execute the PL/SQL block by entering a department that does not exist.

```
Please enter the department number: 50
Please enter the department location: HOUSTON
PL/SQL procedure successfully completed.

G_MESSAGE
-------------------------------------
Department 50 is an invalid department
```

3. Write a PL/SQL block that prints the names of the employees who make plus or minus $100 of the salary value entered.
   a. If there is no employee within that salary range, print a message to the user indicating that is the case. Use an exception for this case.
   b. If there are one or more employees within that range, the message should indicate how many employees are in that salary range.
   c. Handle any other exception with an appropriate exception handler, the message should indicate that some other error occurred.

Practice 23 (continued)

```
Please enter the salary: 800
PL/SQL procedure successfully completed.

G_MESSAGE
------------------------------------------------------------
There is 1 employee(s) with a salary between 700 and 900

Please enter the salary: 3000
PL/SQL procedure successfully completed.

G_MESSAGE
------------------------------------------------------------
There are 3 employee(s) with a salary between 2900 and 3100

Please enter the salary: 6000
PL/SQL procedure successfully completed.


G_MESSAGE
-------------------------------------------------
There are no employee salary between 5900 and 6100
```