Stanford
nâng tầm tri thức

9

PL/SQL Programming In Oracle

Oracle for Base

ORACLE

Objectives

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

- Cursors
- Records
- Triggers
- Packages

Objectives

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

- **Cursors**
- Records
- Triggers
- Packages

Lesson Stanford

- **Cursors**:
  - Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.
  - A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set

Lesson Stanford

- **Cursors**:
  - You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:
    - **Implicit cursors**
    - **Explicit cursors**

Cursors

- **Implicit Cursors**
  - Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.
  - Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.
  - In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.

Cursors

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Cursors

- Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example:

```
DECLARE
      total_rows number(2);
BEGIN
      UPDATE customers
      SET salary = salary + 500;
      IF sql%notfound THEN
            dbms_output.put_line('no customers selected');
      ELSIF sql%found THEN
            total_rows := sql%rowcount;
            dbms_output.put_line( total_rows || ' customers selected ');
      END IF;
END;
/
```

Cursors

- **Explicit Cursors**

Explicit cursors are programmer defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is:

```
CURSOR cursor_name IS select_statement;
```

Cursors

- **Explicit Cursors**

Working with an explicit cursor involves four steps:

- **Declaring the cursor for initializing in the memory**
- **Opening the cursor for allocating memory**
- **Fetching the cursor for retrieving data**
- **Closing the cursor to release allocated memory**

Cursors

- **Explicit Cursors**
  - **Declaring the Cursor**

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS
SELECT id, name, address FROM customers;
```

Cursors

- **Explicit Cursors**
  - **Opening the Cursor**

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

```
OPEN c_customers;
```

Cursors

- **Explicit Cursors**
  - **Fetching the Cursor**

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

  - **Closing the Cursor**

Closing the cursor means releasing the allocated memory. For example, we will close aboveopened cursor as follows:

```
CLOSE c_customers;
```

Cursors

- **Explicit Cursors**

```
Declare
      empId Employees.Employee_Id%type;
      empName Employees.First_Name%type;
      CURSOR employee_cur is
            Select Employee_id, First_Name from Employees;
Begin
      dbms_output.put_line('EmployeeId      FirstName');
Open employee_cur;
Loop
  FETCH employee_cur into empId, empName;
         dbms_output.put_line(empId || '            ' || empName);
         exit when employee_cur%notfound;
End Loop;
Close employee_cur;
End;
```

Objectives

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

- Cursors
- **Records**
- Triggers
- Packages

Records

- **Records in Oracle**
  - A PL/SQL record is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.
  - PL/SQL can handle the following types of records:
    - **Table-based**
    - **Cursor-based records**
    - User-defined records

Records

- **Table- Based Records**
  - The **%ROWTYPE** attribute enables a programmer to create table-based and cursorbased records. The following example would illustrate the concept of table-based records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
customer_rec customers%rowtype;
BEGIN
SELECT * into customer_rec FROM customers WHERE id = 5;
dbms_output.put_line('Customer ID: ' || customer_rec.id);
dbms_output.put_line('Customer Name: ' || customer_rec.name);
dbms_output.put_line('Customer Address: ' ||
customer_rec.address);
dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
```

Records

- **Cursor- Based Records**

Example:

```
DECLARE
CURSOR employee_cur is
SELECT employee_id, First_Name
FROM Employees;
employee_rec employee_cur%rowtype;
BEGIN
OPEN employee_cur;
LOOP
FETCH employee_cur into employee_rec;
EXIT WHEN employee_cur%notfound;
DBMS_OUTPUT.put_line(employee_rec.Employee_Id || ' ' ||
employee_rec.First_Name);
END LOOP;
END;
```

Objectives

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

- Cursors
- Records
- **Triggers**
- Packages

Triggers

- **Triggers**
  - Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:
    - A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
    - A database definition (DDL) statement (CREATE, ALTER, or DROP).
    - A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

  Triggers could be defined on the table, view, schema, or database with which the event is associated.

Triggers

- **Creating Triggers**

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
      [OF col_name]
ON table_name
      [REFERENCING OLD AS o NEW AS n]
      [FOR EACH ROW]
WHEN (condition)
DECLARE
      Declaration-statements
BEGIN
      Executable-statements
EXCEPTION
      Exception-handling-statements
END;
```

Triggers

- **Creating Triggers**

Example:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
sal_diff number;
BEGIN
sal_diff := :NEW.salary  - :OLD.salary;
dbms_output.put_line('Old salary: ' || :OLD.salary);
dbms_output.put_line('New salary: ' || :NEW.salary);
dbms_output.put_line('Salary difference: ' || sal_diff);
END;
```

Triggers

- **Creating Triggers**

Example:

- **Here following two points are important and should be noted carefully:**
  - OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
  - If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
  - Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

Objectives

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

- Cursors
- Records
- Triggers
- **Packages**

Packages

- **Packages**

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- **Package specification**
- **Package body or definition**

Packages

- **Package Specification**
  - The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.
  - All objects placed in the specification are called public objects. Any subprogram not in the package specification but coded in the package body is called a private object.

Packages

- **Package Specification**
  - The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

Packages

- **Package Body**
  - The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.
  - The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the cust_sal package created above.

Packages

- **Package Body**

Example:

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
PROCEDURE find_sal(c_id customers.id%TYPE) IS
c_sal customers.salary%TYPE;
BEGIN
SELECT salary INTO c_sal
FROM customers
WHERE id = c_id;
dbms_output.put_line('Salary: '|| c_sal);
END find_sal;
END cust_sal;
```

Packages

- **Using the Package Elements**
  - The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

  - Consider, we already have created above package in our database schema, the following program uses the find_sal method of the cust_sal package:

```
DECLARE
code customers.id%type := &cc_id;
BEGIN
cust_sal.find_sal(code);
END;
```

Packages

- **THE PACKAGE SPECIFICATION:**

```
CREATE OR REPLACE PACKAGE c_package AS
-- Adds a customer
PROCEDURE addCustomer(c_id customers.id%type,
c_name customers.name%type,
c_age customers.age%type,
c_addr customers.address%type,
c_sal customers.salary%type);
-- Removes a customer
PROCEDURE delCustomer(c_id customers.id%TYPE);
--Lists all customers
PROCEDURE listCustomer;
END c_package;
```

Packages

- **CREATING THE PACKAGE BODY:**

```
CREATE OR REPLACE PACKAGE BODY c_package AS
PROCEDURE addCustomer(c_id customers.id%type,
c_name customers.name%type,
c_age customers.age%type,
c_addr customers.address%type,
c_sal customers.salary%type)
IS
BEGIN
      INSERT INTO customers (id,name,age,address,salary)
      VALUES(c_id, c_name, c_age, c_addr, c_sal);
END addCustomer;
PROCEDURE delCustomer(c_id customers.id%type) IS
BEGIN
      DELETE FROM customers
      WHERE id = c_id;
END delCustomer;
```

Packages

- **CREATING THE PACKAGE BODY:**

```
PROCEDURE listCustomer IS
CURSOR c_customers is
      SELECT  name FROM customers;
TYPE c_list is TABLE OF customers.name%type;
name_list c_list := c_list();
counter integer :=0;
BEGIN
FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer(' ||counter||
      ')'||name_list(counter));
END LOOP;
END listCustomer;
END c_package;
```

Packages

- **USING THE PACKAGE:**

The following program uses the methods declared and defined in the package c_package.

```
DECLARE
      code customers.id%type:= 8;
BEGIN
      c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
      c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
      c_package.listcustomer;
      c_package.delcustomer(code);
      c_package.listcustomer;
END;
```

THANK YOU !

Stanford – Dạy kinh nghiệm lập trình