**Notes on PL/SQL (Procedural Language/Structured Query Language)**

**PL/SQL** is Oracle Corporation's procedural extension for SQL and the Oracle relational database. It allows you to write code that is more structured and robust compared to traditional SQL by incorporating programming constructs such as variables, loops, and conditions.

**Key Features of PL/SQL**

1. **Procedural Language**: Combines the power of SQL with procedural features of programming languages.
2. **Block Structure**: The basic unit of PL/SQL is a block, which can contain declarations, executable statements, and exception handlers.
3. **Error Handling**: Supports exception handling to manage errors effectively.
4. **Variables and Data Types**: You can declare variables and use them in your PL/SQL blocks.
5. **Control Structures**: Supports loops, conditions, and case statements.
6. **Cursors**: For handling query results row-by-row.
7. **Stored Procedures and Functions**: Allows creating reusable code blocks that can be stored in the database and executed later.

**Basic Structure of a PL/SQL Block**

A PL/SQL block has three main sections:

1. **Declaration Section**: Where variables, constants, and cursors are declared.
2. **Execution Section**: Contains the code that performs operations (this part is mandatory).
3. **Exception Handling Section**: Handles errors and exceptions (this part is optional).

Here's the structure:

sql

Copy code

DECLARE

-- Declarations (variables, constants, etc.)

BEGIN

-- Executable statements

EXCEPTION

-- Exception handling

END;

**Example 1: Basic PL/SQL Block**

sql

Copy code

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, PL/SQL!');

END;

* **Explanation**: DBMS\_OUTPUT.PUT\_LINE is used to print output to the console.
* **How to Execute**: Use an Oracle SQL environment like SQL\*Plus or SQL Developer. Make sure SET SERVEROUTPUT ON; is used to enable output.

**Example 2: Declaring and Using Variables**

sql

Copy code

DECLARE

name VARCHAR2(20) := 'John Doe';

age NUMBER := 30;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Name: ' || name);

DBMS\_OUTPUT.PUT\_LINE('Age: ' || age);

END;

* **Explanation**: Variables name and age are declared and used in the BEGIN section. The || operator is used for string concatenation.

**Example 3: Using IF-THEN-ELSE**

sql

Copy code

DECLARE

grade CHAR := 'B';

BEGIN

IF grade = 'A' THEN

DBMS\_OUTPUT.PUT\_LINE('Excellent');

ELSIF grade = 'B' THEN

DBMS\_OUTPUT.PUT\_LINE('Very Good');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Good');

END IF;

END;

* **Explanation**: This block uses an IF-THEN-ELSE structure to evaluate conditions.

**Example 4: Using Loops**

**a. FOR Loop**

sql

Copy code

BEGIN

FOR i IN 1..5 LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || i);

END LOOP;

END;

* **Explanation**: The FOR loop iterates from 1 to 5, printing the iteration number.

**b. WHILE Loop**

sql

Copy code

DECLARE

counter NUMBER := 1;

BEGIN

WHILE counter <= 5 LOOP

DBMS\_OUTPUT.PUT\_LINE('Counter: ' || counter);

counter := counter + 1;

END LOOP;

END;

* **Explanation**: The WHILE loop continues until the counter is greater than 5.

**Example 5: Using Cursors**

A cursor is used to retrieve multiple rows from a query.

sql

Copy code

DECLARE

CURSOR employee\_cursor IS

SELECT first\_name, last\_name FROM employees;

employee\_record employee\_cursor%ROWTYPE;

BEGIN

OPEN employee\_cursor;

LOOP

FETCH employee\_cursor INTO employee\_record;

EXIT WHEN employee\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(employee\_record.first\_name || ' ' || employee\_record.last\_name);

END LOOP;

CLOSE employee\_cursor;

END;

* **Explanation**: This block declares a cursor employee\_cursor to fetch employee names from the employees table and loop through the results.

**Example 6: Exception Handling**

sql

Copy code

DECLARE

num1 NUMBER := 10;

num2 NUMBER := 0;

result NUMBER;

BEGIN

result := num1 / num2;

DBMS\_OUTPUT.PUT\_LINE('Result: ' || result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Division by zero is not allowed.');

END;

* **Explanation**: The ZERO\_DIVIDE exception is used to handle division by zero errors.

**Example 7: Creating and Using Procedures**

sql

Copy code

CREATE OR REPLACE PROCEDURE greet\_user (name IN VARCHAR2) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, ' || name || '!');

END;

-- To execute the procedure

BEGIN

greet\_user('Alice');

END;

* **Explanation**: A procedure greet\_user is created, which takes a name as input and prints a greeting message.

**Example 8: Creating and Using Functions**

sql

Copy code

CREATE OR REPLACE FUNCTION calculate\_area(radius NUMBER) RETURN NUMBER IS

BEGIN

RETURN 3.14159 \* radius \* radius;

END;

-- To call the function

DECLARE

area NUMBER;

BEGIN

area := calculate\_area(5);

DBMS\_OUTPUT.PUT\_LINE('Area: ' || area);

END;

* **Explanation**: A function calculate\_area is created to calculate the area of a circle given the radius.

**Important Concepts in PL/SQL**

1. **Data Types**: NUMBER, VARCHAR2, CHAR, DATE, BOOLEAN, etc.
2. **Control Structures**: IF, CASE, LOOP, WHILE, FOR.
3. **Cursors**: Explicit cursors for handling multiple rows returned by a query.
4. **Error Handling**: Use EXCEPTION to handle errors gracefully.
5. **Procedures and Functions**: Reusable blocks of code that can be called in other PL/SQL blocks or applications.

**Tips for Using PL/SQL**

* Always use SET SERVEROUTPUT ON; to enable output in SQL\*Plus.
* Use %TYPE and %ROWTYPE to declare variables that are dependent on database column types.
* Practice using exception handling to make your programs robust.
* Use cursors when you need to process query results row-by-row.

PL/SQL is powerful for managing data in an Oracle database with added flexibility through programming constructs, making it essential for developers and database administrators working with Oracle databases.

**CURSOR**

**Cursors in PL/SQL**

A **cursor** is a pointer that holds the result set of a query. Cursors are used in PL/SQL to handle query results row-by-row. PL/SQL supports two types of cursors:

1. **Implicit Cursor**: Automatically created by Oracle when a SELECT query returns a single row or when DML statements (like INSERT, UPDATE, DELETE) are executed.
2. **Explicit Cursor**: Defined explicitly by the programmer to process multiple rows returned by a query.

**Implicit Cursor Example**

Oracle automatically manages cursors for SELECT INTO, INSERT, UPDATE, or DELETE operations. Here’s how an implicit cursor works:

sql

Copy code

DECLARE

total\_students NUMBER;

BEGIN

SELECT COUNT(\*) INTO total\_students FROM students;

DBMS\_OUTPUT.PUT\_LINE('Total Students: ' || total\_students);

END;

* **Explanation**: The SELECT INTO statement uses an implicit cursor to fetch the count of students from the students table.

**Explicit Cursor Example**

When dealing with multiple rows, you need to explicitly declare and use a cursor.

**Step-by-Step Guide for Using an Explicit Cursor**

1. **Declare the Cursor**: Define the cursor with a SELECT statement.
2. **Open the Cursor**: Open the cursor to execute the query and populate the result set.
3. **Fetch Data**: Retrieve rows one-by-one from the cursor.
4. **Close the Cursor**: Release the memory occupied by the cursor.

**Example 1: Basic Explicit Cursor**

sql

Copy code

DECLARE

-- Step 1: Declare the cursor

CURSOR emp\_cursor IS

SELECT first\_name, last\_name FROM employees;

-- Variables to store data from the cursor

first\_name employees.first\_name%TYPE;

last\_name employees.last\_name%TYPE;

BEGIN

-- Step 2: Open the cursor

OPEN emp\_cursor;

-- Step 3: Fetch rows from the cursor

LOOP

FETCH emp\_cursor INTO first\_name, last\_name;

-- Exit the loop when no more rows are found

EXIT WHEN emp\_cursor%NOTFOUND;

-- Display the fetched data

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || first\_name || ' ' || last\_name);

END LOOP;

-- Step 4: Close the cursor

CLOSE emp\_cursor;

END;

* **Explanation**: The cursor emp\_cursor selects the first\_name and last\_name columns from the employees table. The loop fetches each row and displays the names until there are no more rows left.

**Example 2: Using Parameters in Explicit Cursor**

Cursors can take parameters to make queries more dynamic.

sql

Copy code

DECLARE

-- Declare the cursor with a parameter for department\_id

CURSOR dept\_cursor(dept\_id NUMBER) IS

SELECT first\_name, last\_name FROM employees WHERE department\_id = dept\_id;

-- Variables to store data from the cursor

first\_name employees.first\_name%TYPE;

last\_name employees.last\_name%TYPE;

BEGIN

-- Open the cursor for department\_id = 10

OPEN dept\_cursor(10);

LOOP

FETCH dept\_cursor INTO first\_name, last\_name;

EXIT WHEN dept\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || first\_name || ' ' || last\_name);

END LOOP;

-- Close the cursor

CLOSE dept\_cursor;

END;

* **Explanation**: The cursor dept\_cursor takes a parameter dept\_id. When opening the cursor, the parameter value 10 is passed, and only employees from that department are fetched.

**Example 3: Cursor Attributes**

PL/SQL provides attributes to manage cursors effectively:

* %FOUND: Returns TRUE if the last fetch returned a row, FALSE otherwise.
* %NOTFOUND: Returns TRUE if the last fetch did not return a row, FALSE otherwise.
* %ROWCOUNT: Returns the number of rows fetched so far.
* %ISOPEN: Returns TRUE if the cursor is open, FALSE otherwise.

sql

Copy code

DECLARE

CURSOR emp\_cursor IS

SELECT first\_name FROM employees;

first\_name employees.first\_name%TYPE;

BEGIN

OPEN emp\_cursor;

LOOP

FETCH emp\_cursor INTO first\_name;

-- Use cursor attributes

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Fetched: ' || first\_name || ' | Row Count: ' || emp\_cursor%ROWCOUNT);

END LOOP;

CLOSE emp\_cursor;

END;

* **Explanation**: %ROWCOUNT is used to display the number of rows fetched.

**Example 4: FOR Loop with Cursor**

You can simplify cursor usage by using a FOR loop, which automatically opens, fetches, and closes the cursor.

sql

Copy code

DECLARE

CURSOR emp\_cursor IS

SELECT first\_name, last\_name FROM employees;

BEGIN

-- Use a FOR loop to iterate over the cursor

FOR emp\_rec IN emp\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || emp\_rec.first\_name || ' ' || emp\_rec.last\_name);

END LOOP;

END;

* **Explanation**: The FOR loop automatically handles the cursor operations, making the code cleaner and easier to read.

**Summary**

* **Implicit Cursor**: Used for single-row queries or DML operations, managed automatically by PL/SQL.
* **Explicit Cursor**: Used when you need to handle multiple rows from a query explicitly.
* **Cursor Attributes**: %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN help manage cursor states and behavior.
* **FOR Loop with Cursor**: Simplifies cursor handling, automatically managing the open, fetch, and close operations.

These examples illustrate the use of cursors in PL/SQL, which provide powerful ways to process query results row-by-row.

**TRIGGER**

**Triggers in PL/SQL**

A **trigger** is a stored procedure in a database that automatically executes or "fires" in response to certain events on a specified table or view. Triggers can be defined to fire before or after an INSERT, UPDATE, or DELETE operation.

**Types of Triggers**

1. **DML Triggers**: Fired by INSERT, UPDATE, or DELETE operations on a table.
2. **Instead-of Triggers**: Specifically used for views to perform the required DML operation.
3. **System/Event Triggers**: Fired in response to database events like LOGON, LOGOFF, STARTUP, etc.

**Syntax for Creating a Trigger**

sql

Copy code

CREATE [OR REPLACE] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF}

{INSERT | UPDATE | DELETE}

ON table\_name

[FOR EACH ROW]

BEGIN

-- Trigger logic goes here

END;

**Examples of Triggers**

**Example 1: Trigger Before Insert**

This trigger checks if an employee's salary is greater than a certain amount before inserting a new record.

sql

Copy code

CREATE OR REPLACE TRIGGER before\_insert\_employee

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

IF :NEW.salary < 3000 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Salary must be at least 3000.');

END IF;

END;

* **Explanation**: :NEW.salary refers to the salary field in the new record being inserted. If the salary is less than 3000, an error is raised, preventing the insertion.

**Example 2: Trigger After Update**

This trigger logs changes made to the salary column of the employees table.

sql

Copy code

CREATE OR REPLACE TRIGGER after\_update\_salary

AFTER UPDATE OF salary ON employees

FOR EACH ROW

BEGIN

INSERT INTO salary\_log (employee\_id, old\_salary, new\_salary, change\_date)

VALUES (:OLD.employee\_id, :OLD.salary, :NEW.salary, SYSDATE);

END;

* **Explanation**:
  + :OLD refers to the data before the update.
  + :NEW refers to the updated data.
  + The trigger inserts a record into the salary\_log table to track changes to the salary column, including the old and new salary values and the date of the change.

**Example 3: Trigger Before Delete**

This trigger prevents the deletion of a department if there are still employees in that department.

sql

Copy code

CREATE OR REPLACE TRIGGER before\_delete\_department

BEFORE DELETE ON departments

FOR EACH ROW

DECLARE

employee\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO employee\_count FROM employees WHERE department\_id = :OLD.department\_id;

IF employee\_count > 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Cannot delete a department with existing employees.');

END IF;

END;

* **Explanation**:
  + The trigger checks the number of employees in the department about to be deleted.
  + If there are employees, an error is raised to prevent the deletion.

**Example 4: Trigger After Insert**

This trigger automatically creates a record in an audit table whenever a new employee is added.

sql

Copy code

CREATE OR REPLACE TRIGGER after\_insert\_employee

AFTER INSERT ON employees

FOR EACH ROW

BEGIN

INSERT INTO audit\_log (action, employee\_id, action\_date)

VALUES ('INSERT', :NEW.employee\_id, SYSDATE);

END;

* **Explanation**:
  + The trigger fires after a new row is inserted into the employees table.
  + It inserts a record into the audit\_log table, logging the action, employee ID, and the date of the action.

**Example 5: Trigger for Updating Inventory**

This example shows a trigger that updates inventory levels after an order is placed.

sql

Copy code

CREATE OR REPLACE TRIGGER update\_inventory

AFTER INSERT ON orders

FOR EACH ROW

BEGIN

UPDATE products

SET quantity = quantity - :NEW.quantity\_ordered

WHERE product\_id = :NEW.product\_id;

END;

* **Explanation**:
  + The trigger updates the quantity of a product in the products table after an order is inserted into the orders table.
  + :NEW.quantity\_ordered and :NEW.product\_id refer to the data from the new order being inserted.

**Trigger Keywords**

* : Refers to the new values of the record being inserted or updated.
* : Refers to the old values of the record being updated or deleted.
* **BEFORE**: The trigger fires before the operation is executed.
* **AFTER**: The trigger fires after the operation is executed.
* **FOR EACH ROW**: Indicates that the trigger should fire for each row affected by the operation.

**Notes on Triggers**

1. **Performance**: Triggers can impact performance, especially if complex logic is executed for every row in large datasets.
2. **Debugging**: Debugging triggers can be challenging because they automatically fire and might not show obvious errors immediately.
3. **Use Cases**: Triggers are useful for enforcing business rules, auditing changes, and maintaining data integrity.

Triggers are a powerful tool in PL/SQL for automating data-related tasks and enforcing rules directly at the database level.