**Packages**

**Package is an Object** that that contains functions, procedures, cursors, global variables, constants, types into single unit.

Packages are typically used to improve application performance because they automatically determine how many packages to load into memory when we call a packages subprogram for the first time.

In addition, when we call a subsequence subprogram, the Oracle server calls those subprograms directly from memory instead of disk.

This approach reduces disk I/O (input/output) operations automatically. Therefore, packages also improve the functionality of the programs.

Every package in Oracle consists of two pieces, which are...

1. **Package Specification**
2. **Package Body**

**Package body** objects in Oracle are **private** by default, but **package specification** objects are **public** by default. Variables, constants, cursors, types, procedures, and functions are declared in package specifications. On the other hand, we are implementing functions and processes in the package body.

Without modifying the package definition, we can update, debug, or alter a package body.

**Package Specification: -**

Basic syntax of Package Specification

**CREATE [OR REPLACE] PACKAGE** package\_name

**IS/AS**

-- Global Variables, Constant Declarations;

-- Types Declarations;

-- Cursor Declaration;

-- Procedure Declarations;

-- Functions Declarations;

**END** package\_name;

/

**Package Body: -**

Basic syntax of Package body

**CREATE [OR REPLACE] PACKAGE BODY** package\_name

**IS/AS**

-- Procedure Implementation;

-- Function Implementation;

**END** package\_name;

**Calling Packaged subprograms: -**

**1.Calling Packaged Procedures**

**Method 1: -**

Syntax: - **exec** packagename.procedurename (actual parameters)

**Method 2: -**

Syntax: -

**Begin**

packagename.procedurename (actual parameters)

**END;**

**2.Calling Package Functions**

**Method 1: -** Using select statement.

Select packagename.functionname (actual parameters) **from dual;**

**Method 2: -** Using anonymous block

To call a function in anonymous block, we need at least one variable.

DECLARE

Varname datatype (based on function return type)

BEGIN

Varname := packagename.functionname (actual parameters);

END;

**Example: -**

Create a Pl SQL Package to Update Employee salary, Retrieve full name of an Employee and Fetch employee details.

**-- Package Specification**

CREATE OR REPLACE PACKAGE EmployeePackage AS

-- Procedure to update the salary of an employee

PROCEDURE UpdateSalary(p\_employee\_id NUMBER,

p\_new\_salary NUMBER);

-- Function to retrieve the full name of an employee

FUNCTION GetFullName(p\_employee\_id NUMBER)

RETURN VARCHAR2;

-- Cursor to fetch employee details for a specific department

FUNCTION GetEmployeesInDepartment(p\_department\_id NUMBER)

RETURN SYS\_REFCURSOR;

END EmployeePackage;

/

**-- Package Body**

CREATE OR REPLACE PACKAGE BODY EmployeePackage AS

-- Procedure to update the salary of an employee

PROCEDURE UpdateSalary(p\_employee\_id NUMBER,

p\_new\_salary NUMBER)

IS

BEGIN

UPDATE employee

SET salary = p\_new\_salary

WHERE employee\_id = p\_employee\_id;

COMMIT; -- Commit the transaction

END UpdateSalary;

-- Function to retrieve the full name of an employee

FUNCTION GetFullName(p\_employee\_id NUMBER) RETURN VARCHAR2 IS

v\_full\_name VARCHAR2(100);

BEGIN

SELECT first\_name || ' ' || last\_name

INTO v\_full\_name

FROM employee

WHERE employee\_id = p\_employee\_id;

RETURN v\_full\_name;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL; -- Handle the case when employee is not found

END GetFullName;

-- Cursor to fetch employee details for a specific department

FUNCTION GetEmployeesInDepartment(p\_department\_id NUMBER)

RETURN SYS\_REFCURSOR IS

v\_cursor SYS\_REFCURSOR;

BEGIN

OPEN v\_cursor FOR

SELECT employee\_id, first\_name, last\_name, salary

FROM employee

WHERE department\_id = p\_department\_id;

RETURN v\_cursor;

END GetEmployeesInDepartment;

END EmployeePackage;

/

To use this package, you can call the procedures/functions from a PL/SQL block. For example:

DECLARE

v\_employee\_id NUMBER := 101;

v\_new\_salary NUMBER := 60000;

v\_full\_name VARCHAR2(100);

v\_department\_id NUMBER := 20;

v\_employee\_cursor SYS\_REFCURSOR;

v\_employee\_rec employees%ROWTYPE;

BEGIN

-- Call the UpdateSalary procedure

EmployeePackage.UpdateSalary(v\_employee\_id, v\_new\_salary);

-- Call the GetFullName function

v\_full\_name := EmployeePackage.GetFullName(v\_employee\_id);

DBMS\_OUTPUT.PUT\_LINE('Full Name: ' || v\_full\_name);

-- Call the GetEmployeesInDepartment function with Cursor

v\_employee\_cursor := EmployeePackage.GetEmployeesInDepartment(v\_department\_id);

-- Loop through the cursor and display employee details

LOOP

FETCH v\_employee\_cursor INTO v\_employee\_rec;

EXIT WHEN v\_employee\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || v\_employee\_rec.employee\_id || ', ' ||

'Name: ' || v\_employee\_rec.first\_name || ' ' || v\_employee\_rec.last\_name || ', ' ||

'Salary: ' || v\_employee\_rec.salary);

END LOOP;

CLOSE v\_employee\_cursor;

END;

/

**Advantages of Packages: -**

1. **Modularity and Organization:**

Code can be arranged into modular sections with the use of packages, which facilitates management and comprehension.

Code readability and maintainability are improved by logically grouping related processes and functions within a package.

1. **Encapsulation: -**

Code is contained within packages, enabling the separation of private and public components. A degree of abstraction and protection can be added by having private procedures or functions that are not immediately available from outside the package.

1. **Global Variables: -**

Packages allow you to create global variables that can be shared across various procedures and functions within the package. This promotes information sharing and avoids the need for redundant variable declarations.

1. **Improved Performance: -**

Packages can help increase performance by lowering the overhead related to repeatedly processing code. Following compilation, a package's code is kept in the database where it can be utilized by other processes or features in the same package.

1. **Dependency Management: -**

Packages help in the management of dependencies among various PL/SQL application components. Modifications to the package definition minimize the likelihood of errors and expedite the development process by eliminating the need to recompile dependent programs.

1. **Version Control: -**

The ability to make changes to a package independently of other program components makes it a technique for version control. Controlled update and modification management is made easier as a result.

**Disadvantages of Packages: -**

1. **Complexity: -**

If packages are not correctly created and arranged, they can become complex and challenging to handle. Achieving a balance between simplicity and modularity is crucial.

1. **Overhead: -**

The use of packages comes with a minor overhead because package state needs to be maintained. Although this expense is usually insignificant, in applications where performance is crucial, it may be taken into account.

1. **Learning Curve: -**

Packages might be a confusing topic for developers who are new to PL/SQL. There is a learning curve involved in efficiently using packages.

1. **Potential for Overuse: -**

Overuse of packages can lead to the creation of superfluous levels of abstraction. This may result in an unnecessarily complex and challenging to maintain codebase.

**Triggers**

A collection of instructions that are automatically carried out (or "triggered") in reaction to specified occurrences on a given table or view is known as a trigger.

Triggers are used in databases to maintain data integrity, carry out extra tasks, and enforce business rules.

They can be programmed to run before or after particular events, such **INSERT, UPDATE, and DELETE.**

Here is a general definition of triggers in PL/SQL:

**Trigger Definition:**

A trigger is a named PL/SQL block or stored procedure that is automatically executed (or "triggered") in response to a specific event on a specified table or view.

**Important Features of Triggers: -**

**Event:** **-** A trigger is linked to a particular database action, like INSERT, UPDATE, or DELETE.

**Timing: -** Triggers can be categorized according to when they are executed:

**BEFORE Triggers:** Carried out prior to the activation incident. used to amend or verify data before making changes to it.

**AFTER Triggers:** Executed after the triggering event. Used for actions that should occur after the data has been changed.

**Range:** Triggers can be classified on several levels:

**Row-Level Triggers:** One action is taken for every impacted row.

**Statement-Level Triggers:** Regardless of the number of rows impacted, just one execution of the full triggering statement is required.

**Basic Syntax of Triggers: -**

**CREATE OR REPLACE TRIGGER** trigger\_name

**{BEFORE | AFTER | INSTEAD OF}** -- Timing

**{INSERT | UPDATE | DELETE | {INSERT OR UPDATE} | {UPDATE OR INSERT}}** -- Event

[{**OF** column\_name | **ON** table\_name |

{**FOR EACH ROW** | **FOR EACH STATEMENT**}}] -- Range

[**WHEN** (condition)] -- Condition (Optional)

**IS**

**BEGIN**

-- Trigger logic

-- PL/SQL statements

**END** [trigger\_name];

/

**Row level TRIGGER Example: -**

Row Level Triggers require the use of the "for each row" clause in the trigger specification because the trigger body is run for each and every row for each DML statement.

DML transaction values are internally and automatically stored in two rollback segment qualifiers in Oracle whenever we use a row level trigger.

These are

* :OLD
* :NEW

**Example: -**

**CREATE OR REPLACE TRIGGER** delete\_employee\_trigger

**BEFORE DELETE ON** employees

**FOR EACH ROW**

**BEGIN**

-- Insert deleted employee information into the audit log

INSERT INTO employee\_audit\_log (log\_id, employee\_id, employee\_name, deletion\_date)

VALUES (employee\_audit\_log\_seq.NEXTVAL, :OLD.employee\_id, :OLD.employee\_name, SYSDATE);

DBMS\_OUTPUT.PUT\_LINE ('Employee ' || :OLD.employee\_name || ' has been deleted.');

**END**;

**The applications listed below utilize row level triggers in Oracle. These are...**

* Putting business rules into action
* Examining some columns
* Automatic rise

**Statement Level Triggers Example: -**

The trigger body in statement level triggers is only run once for each DML statement. Oracle statement level triggers do not have the ":old" or ":new" qualifiers, and statement level triggers do not contain the "for each row" phrase.

Statement level triggers are typically required for creating time component-based applications through the usage of triggers. Statement level trigger performance is often much higher than row level trigger performance in all relational databases.

**Example: -**

**CREATE OR REPLACE TRIGGER** before\_insert\_statement\_example

**BEFORE INSERT ON** employee

DECLARE

v\_total\_sal NUMBER:= 0;

BEGIN

SELECT NVL (SUM (salary), 0) INTO v\_total\_sal FROM employees;

DBMS\_OUTPUT.PUT\_LINE ('Total Salary before INSERT: ' || v\_total\_sal);

END before\_insert\_statement\_example;

/

**Instead of Triggers: -**

Oracle 8i introduced instead of trigger. **Instead of trigger** are created on **view** by default instead of trigger are row level trigger.

In general, DML processes cannot be completed using complex views on base tables. Oracle 8i introduced instead of trigger in PL/SQL to solve this issue.

When we create an instead of a trigger on complex view, then only we are permitted to do DML operations through the complex view to the base table.

For this reason, instead of trigger convert non-updatable views become updatable views.

**Syntax:**

**CREATE OR REPLACE TRIGGER** TRIGGERNAME

**INSTEAD OF** insert/update/delete on viewname

**FOR EACH ROW**

[declare]

……………….

……………….

Begin

……………….

……………….

end;

**Example 1: -**

**CREATE OR REPLACE TRIGGER** tr2

**INSTEAD OF** insert on v1

**FOR EACH ROW**

begin

insert into test1 (name) values (:new.name);

insert into test2 (sub) values (:new.sub);

end;

**Example 2: -**

**CREATE OR REPLACE TRIGGER** instead\_of\_orders\_trigger

**INSTEAD OF** INSERT OR UPDATE OR DELETE ON orders\_v

**FOR EACH ROW**

DECLARE

v\_action VARCHAR2(10);

BEGIN

-- Determine the action type

IF INSERTING THEN

v\_action := 'INSERT';

ELSIF UPDATING THEN

v\_action := 'UPDATE';

ELSIF DELETING THEN

v\_action := 'DELETE';

END IF;

INSERT INTO order\_audit\_log (audit\_id, action, action\_date, total\_orders)

VALUES (audit\_id\_seq.NEXTVAL, v\_action, SYSDATE, (SELECT COUNT(\*) FROM orders));

INSERT INTO orders values(100, sysdate, 1000, 10000);

-- You can also perform other actions based on the type of operation

-- Example: Allow the original operation to proceed for INSERT and UPDATE

IF INSERTING OR UPDATING THEN

-- You may want to perform additional actions here

NULL;

DBMS\_OUTPUT.put\_line('Hi');

END IF;

-- Example: Prevent DELETE operation

IF DELETING THEN

-- DBMS\_OUTPUT.put\_line('delete');

raise\_application\_error(-20000, 'DELETE operation is not allowed.');

END IF;

END;