**Minutes of Session – Week 2 Day 5** **Topic:** PL/SQL – Packages & Triggers  
 **Date:** 03 July 2025 **Duration:** Insert Duration –7Hrs   
 **Trainer:** Niti  
 **Mode:** Hybrid  
 **Batch:** ORACLE PL/SQL Cohort 2

### **Topics Covered**

#### **1. PL/SQL Packages**

* **Definition & Importance:**
  + A PL/SQL Package is a schema object that groups logically related PL/SQL types, items, and subprograms.
* **Package Components:**
  + **Specification:** Declares public procedures, functions, variables, constants, cursors.
  + **Body:** Contains implementation for procedures and functions; can define private elements.
* **Public vs Private Elements:**
  + **Public:** Declared in specification; accessible outside the package.
  + **Private:** Declared in the body only; not accessible externally.
* **Benefits of Using Packages:**
  + Code modularity and reuse.
  + Performance improvement (loaded into memory once).
  + Encapsulation of business logic.

### What is a Package?

A **package** in PL/SQL is a way to group related procedures, functions, variables, cursors, and types into a **single unit**.

### A. **Package Specification**

* Acts like a **public interface** or **header**.
* Declares **what is available for use** (procedures, functions, variables).
* Does **not contain implementation**.

**Syntax:**

CREATE OR REPLACE PACKAGE emp\_pkg IS

PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2);

FUNCTION get\_total\_employees RETURN NUMBER;

END emp\_pkg;

### B. **Package Body**

* Contains the **actual implementation** of the procedures/functions declared in the spec.
* Can also contain **private** methods or variables that are **not declared** in the specification.

**Syntax:**

CREATE OR REPLACE PACKAGE BODY emp\_pkg IS

-- Private variable (not in spec)

v\_count NUMBER := 0;

-- Public procedure implementation

PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2) IS

BEGIN

INSERT INTO employee VALUES(p\_id, p\_name);

v\_count := v\_count + 1;

END;

-- Public function implementation

FUNCTION get\_total\_employees RETURN NUMBER IS

BEGIN

RETURN v\_count;

END;

END emp\_pkg;

### 🔸 C. Public vs Private in Packages

| **Feature** | **Public** | **Private** |
| --- | --- | --- |
| Declared in | Package Specification | Package Body only |
| Accessible by | External PL/SQL blocks, apps | Only inside the package body |
| Use Case | Business logic to be reused externally | Internal helpers, counters, logs, etc. |

**Rule:**  
If it is declared in the **spec**, it is **public**.  
If it exists **only in the body**, it is **private**.

drop table employees;

CREATE TABLE employees (

    emp\_id    NUMBER PRIMARY KEY,

    emp\_name  VARCHAR2(100),

    join\_date DATE

);

ALTER TABLE employees ADD salary NUMBER;

-- Create the Package Specification First

CREATE OR REPLACE PACKAGE emp\_pkg IS

    PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2, salary number);

    FUNCTION get\_total\_employees RETURN NUMBER;

END emp\_pkg;

/

-- Then Create the Package Body

CREATE OR REPLACE PACKAGE BODY emp\_pkg IS

    -- Private variable (not in spec)

    v\_count NUMBER := 0;

    -- Public procedure implementation

    PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2,salary number) IS

    BEGIN

        INSERT INTO employees (emp\_id, emp\_name,SALARY)

        VALUES (p\_id, p\_name,salary);

        v\_count := v\_count + 1;

    END;

    -- Public function implementation

    FUNCTION get\_total\_employees RETURN NUMBER IS

    BEGIN

        RETURN v\_count;

    END;

END emp\_pkg;

/

-- To execute

BEGIN

    emp\_pkg.add\_employee(1002, 'Manju Bedi',50000);

    DBMS\_OUTPUT.PUT\_LINE('Total Added: ' || emp\_pkg.get\_total\_employees);

END;

/

select \* from employees;

| **Step** | **Description** |
| --- | --- |
| Package Spec | Declares add\_employee, get\_total\_employees |
| Package Body | Implements the logic + uses private state |

#### **2. PL/SQL Triggers**

* **Definition:**
  + A trigger is a stored PL/SQL block that automatically executes in response to specific events on a table or view.
* **Types Covered:**
  + **BEFORE INSERT/UPDATE/DELETE**
  + **AFTER INSERT/UPDATE/DELETE**
* **Use Cases & Application Areas:**
  + Enforcing business rules.
  + Logging/auditing changes.
  + Preventing invalid transactions.
* **Do’s and Don’ts:**
  + Use for enforcing integrity rules not captured by constraints.
  + Keep triggers simple and efficient.
  + Avoid recursive triggers.
  + Don’t include complex business logic that can be done in application layer.
* **Best Practices:**
  + Name triggers meaningfully (e.g., BIU\_EMPLOYEE\_LOG)
  + Minimize the use of row-level triggers when not needed.
  + Document the logic for better maintainability.
  + Avoid using triggers for tasks that can be handled by constraints or packages.

### 

### **Hands-on Demonstration & Practice**

* Created a package with both public and private procedures.
* Implemented **BEFORE INSERT** trigger for validating data.
* Created an **AFTER DELETE** trigger for maintaining audit log.
* Practiced multiple trigger scenarios using test tables and sample data.

### **Key Takeaways**

* Packages improve modularization and encapsulation.
* Triggers are powerful but should be used judiciously to avoid performance issues.
* Proper naming, structure, and documentation are essential for maintainability.

**Assignments / Action Items**

* Create a package to manage employee operations with one private and one public procedure.
* Implement two triggers:  
  + BEFORE UPDATE trigger on Salary column to prevent unauthorized changes.
  + AFTER DELETE trigger to log deleted records into an audit table.

## 

## **2. Triggers: BEFORE / AFTER INSERT, UPDATE, DELETE**

### 🔹 What is a Trigger?

A **trigger** is a stored block that **automatically executes** when a specified DML (INSERT/UPDATE/DELETE) operation occurs on a table.

### A. Timing: BEFORE vs AFTER

| **Type** | **Execution Time** | **Use Case** |
| --- | --- | --- |
| BEFORE | Before the triggering DML executes | Validate or modify data before insert/update |
| AFTER | After the DML executes successfully | Auditing, logging, notifications |

#### 1. BEFORE INSERT Trigger (to auto-set a join date)

CREATE OR REPLACE TRIGGER trg\_set\_join\_date

BEFORE INSERT ON employee

FOR EACH ROW

BEGIN

:NEW.join\_date := SYSDATE;

END;

#### 2. AFTER UPDATE Trigger (to log changes)

CREATE OR REPLACE TRIGGER trg\_log\_salary\_change

AFTER UPDATE ON employee

FOR EACH ROW

BEGIN

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

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

END;

CREATE TABLE emp\_salary\_log (

    log\_id        NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

    emp\_id        NUMBER NOT NULL,

    old\_salary    NUMBER,

    new\_salary    NUMBER,

    change\_date   DATE DEFAULT SYSDATE

);

select \* from employees;

-- AFTER UPDATE Trigger (to log changes)

CREATE OR REPLACE TRIGGER trgg\_log\_salary\_change

AFTER UPDATE ON employees

FOR EACH ROW

BEGIN

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

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

END;

update employees set salary=10000 where emp\_id=1002;

select \* from employees;

select \* from  emp\_salary\_log;

delete from employees where emp\_id=1002;

-- BEFORE DELETE Trigger (to prevent deletion)

CREATE OR REPLACE TRIGGER trgg\_prevent\_delete

BEFORE DELETE ON employees

FOR EACH ROW

BEGIN

    RAISE\_APPLICATION\_ERROR(-20001, 'Deletion not allowed!');

END;

Encapsulation: calling of private procedure within the package itself not by other developers

## 🧪 Simulation Use Case

### Scenario: You want to log internal messages whenever an employee is added, but you don’t want developers outside the package to call the logging logic.

So:

* Only add\_employee is exposed.
* But log\_employee is still executed **behind the scenes**.

### Step-by-Step Setup

#### 1. Create the Table

CREATE TABLE employees (

emp\_id NUMBER PRIMARY KEY,

emp\_name VARCHAR2(100),

salary NUMBER,

join\_date DATE

);

#### 2. Create a Package Specification (Public Interface)

CREATE OR REPLACE PACKAGE emp\_pkg IS

PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2, p\_salary NUMBER);

FUNCTION get\_total\_employees RETURN NUMBER;

END emp\_pkg;

/

Everything declared here (add\_employee, get\_total\_employees) is **public** and accessible **outside the package**.

#### 3. Create the Package Body (Implementation + Private Members)plsql

CREATE OR REPLACE PACKAGE BODY emp\_pkg IS

-- Private variable (Not declared in the spec)

v\_counter NUMBER := 0;

-- Private procedure (Not visible outside the package)

PROCEDURE log\_employee(p\_id NUMBER) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_id || ' added.');

END;

-- Public procedure implementation

PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2, p\_salary NUMBER) IS

BEGIN

INSERT INTO employees(emp\_id, emp\_name, salary, join\_date)

VALUES (p\_id, p\_name, p\_salary, SYSDATE);

v\_counter := v\_counter + 1;

log\_employee(p\_id); -- calling private proc internally

END;

-- Public function implementation

FUNCTION get\_total\_employees RETURN NUMBER IS

BEGIN

RETURN v\_counter;

END;

END emp\_pkg;

/

### How to Identify Public vs Private:

|  |  |  |
| --- | --- | --- |
| add\_employee | Package Spec & Body | Public |
| get\_total\_employees | Package Spec & Body | Public |
| v\_counter | Package Body only | Private |
| log\_employee | Package Body only | Private |

### Use Case (How Private Procedure Is Used Internally)

#### Execute from an Anonymous Block

BEGIN

emp\_pkg.add\_employee(2001, 'Ritika Roy', 52000);

emp\_pkg.add\_employee(2002, 'Manav Gupta', 61000);

DBMS\_OUTPUT.PUT\_LINE('Total Added: ' || emp\_pkg.get\_total\_employees);

END;

/

**Output:**

Employee 2001 added.

Employee 2002 added.

Total Added: 2

### Trying to Call the Private Procedure from Outside?

BEGIN

emp\_pkg.log\_employee(2003); -- This will give error

END;

/

Error:

PLS-00302: component 'LOG\_EMPLOYEE' must be declared

## Real-World Use Case for Private Procedure

| **Use Case** | **Public** | **Private** |
| --- | --- | --- |
| Insert employee record | Yes |  |
| Logging the insert operation |  | Yes |
| Internal auditing |  | Yes |
| Validation helper methods |  | Yes |
| Send notifications |  | Yes |

## Conclusion

**Private procedures** are like **internal tools** in your package. They support your public features, but they’re **invisible to outside users**. This gives:

* Clean public API
* Better encapsulation
* Maintainability and security

Perfect! Let's extend your existing package to add a  **private validation function** that ensures:

Salary must be **greater than 0**, or else the insert will be **rejected** with an error.

## Final Package Structure (Updated)

### 1. Add this logic:

* validate\_salary(p\_salary) → a **private function**
* Raise an exception if salary ≤ 0

### Updated emp\_pkg Package Body

CREATE OR REPLACE PACKAGE BODY emp\_pkg IS

-- Private variable

v\_counter NUMBER := 0;

-- Private function to validate salary

FUNCTION validate\_salary(p\_salary NUMBER) RETURN BOOLEAN IS

BEGIN

IF p\_salary <= 0 THEN

RETURN FALSE;

ELSE

RETURN TRUE;

END IF;

END;

-- Private logger

PROCEDURE log\_employee(p\_id NUMBER) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_id || ' added.');

END;

-- Public procedure with validation

PROCEDURE add\_employee(p\_id NUMBER, p\_name VARCHAR2, p\_salary NUMBER) IS

BEGIN

IF NOT validate\_salary(p\_salary) THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Salary must be greater than 0.');

END IF;

INSERT INTO employees(emp\_id, emp\_name, salary, join\_date)

VALUES (p\_id, p\_name, p\_salary, SYSDATE);

v\_counter := v\_counter + 1;

log\_employee(p\_id);

END;

-- Public function

FUNCTION get\_total\_employees RETURN NUMBER IS

BEGIN

RETURN v\_counter;

END;

END emp\_pkg;

/

### Test It

#### Valid Salary

BEGIN

emp\_pkg.add\_employee(2003, 'Anika Sharma', 75000);

END;

/

Output:

Employee 2003 added.

#### Invalid Salary

BEGIN

emp\_pkg.add\_employee(2004, 'Raj Malhotra', 0); -- invalid salary

END;

/

Output:

ORA-20001: Salary must be greater than 0.

### Why This Matters

| **Benefit** | **Of creating Private validate\_salary** |
| --- | --- |
| Reusable check across procedures |  |
| Hidden from external users |  |
| Keeps add\_employee clean |  |
| Secure and maintainable logic |  |

## 1. What is **Dynamic SQL**?

### 🔸Definition:

Dynamic SQL means you **construct and execute SQL statements at runtime**, instead of hardcoding them.

Used when:

* Table name or column name is dynamic
* Conditional filtering is needed
* DDL statements (CREATE, DROP) must be executed

## ✅ 2. What is EXECUTE IMMEDIATE?

EXECUTE IMMEDIATE is the simplest way in PL/SQL to run dynamic SQL.

### 🔍 Simulation 1: Dynamic INSERT using EXECUTE IMMEDIATE

sql

CopyEdit

DECLARE

v\_sql VARCHAR2(500);

BEGIN

v\_sql := 'INSERT INTO audit\_log (ticket\_id, user\_id, action) VALUES (1002, 101, ''INSERT-DEMO'')';

EXECUTE IMMEDIATE v\_sql;

DBMS\_OUTPUT.PUT\_LINE('✅ Dynamic INSERT executed!');

END;

🧠 **Explanation**:

* You build the SQL as a string.
* Oracle compiles and executes it on-the-fly.

### 🔍 Simulation 2: Dynamic SELECT using bind variables

sql

CopyEdit

DECLARE

v\_sql VARCHAR2(200);

v\_total NUMBER;

v\_dept VARCHAR2(50) := 'IT';

BEGIN

v\_sql := 'SELECT COUNT(\*) FROM user\_credentials WHERE department = :dept';

EXECUTE IMMEDIATE v\_sql INTO v\_total USING v\_dept;

DBMS\_OUTPUT.PUT\_LINE('✅ Number of users in ' || v\_dept || ': ' || v\_total);

END;

🔎 :dept is a **bind variable**, and we supply its value via USING.

## ✅ 3. When to Use Bind Variables

| **Use Case** | **Why Use Bind Variables?** |
| --- | --- |
| Conditional queries | Prevent SQL injection, reuse parsed SQL |
| Variable WHERE conditions | Based on user input (username, ID, etc.) |
| Performance optimization | Reduces parsing overhead |

### 🔍 Simulation 3: DYNAMIC UPDATE with Bind Variables

sql

CopyEdit

DECLARE

v\_sql VARCHAR2(300);

v\_ticket\_id NUMBER := 1001;

v\_status VARCHAR2(20) := 'CLOSED';

BEGIN

v\_sql := 'UPDATE tickets SET status = :1 WHERE ticket\_id = :2';

EXECUTE IMMEDIATE v\_sql USING v\_status, v\_ticket\_id;

DBMS\_OUTPUT.PUT\_LINE('✅ Ticket ' || v\_ticket\_id || ' closed dynamically.');

END;

### 🔍 Simulation 4: DDL Statement Using Dynamic SQL

sql

CopyEdit

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE temp\_log (id NUMBER, info VARCHAR2(100))';

DBMS\_OUTPUT.PUT\_LINE('✅ Table temp\_log created dynamically.');

END;

🔥 DDL cannot be run using static SQL in PL/SQL — it **must** be dynamic!

## ✅ 4. Advanced: Dynamic Table Name with EXECUTE IMMEDIATE

sql

CopyEdit

DECLARE

v\_table\_name VARCHAR2(30) := 'TICKETS';

v\_sql VARCHAR2(100);

v\_count NUMBER;

BEGIN

v\_sql := 'SELECT COUNT(\*) FROM ' || v\_table\_name;

EXECUTE IMMEDIATE v\_sql INTO v\_count;

DBMS\_OUTPUT.PUT\_LINE('🔢 Row count in ' || v\_table\_name || ': ' || v\_count);

END;

📌 Here, table name is decided **at runtime** — that's the power of dynamic SQL.

## ✅ 5. Optional: Use of DBMS\_SQL (more advanced)

This is used when:

* You don’t know the number of columns in advance
* You need generic SQL engines

## **Key Difference Between Procedure and Dynamic SQL**

| **Feature** | **Stored Procedure** | **Dynamic SQL (EXECUTE IMMEDIATE)** |
| --- | --- | --- |
| **Definition** | A named, precompiled PL/SQL block stored in the DB | SQL built and executed at runtime |
| **Usage** | Called using procedure name with parameters | SQL built as a string and executed dynamically |
| **Flexibility** | Less flexible (fixed structure) | Highly flexible (you build SQL dynamically) |
| **Compilation** | Compiled once and reused | Parsed & compiled **every time** it's run |
| **Security** | Safe and manageable (with grants/roles) | May expose SQL injection risk if not handled properly |
| **Best For** | Standard, repeatable logic (e.g. inserting orders, calculating grades) | When table name, columns, or conditions are **not known in advance** |

### 🧠 Easy Analogy for Learners:

* **Procedure** = like a **function in your calculator** (e.g., square(x)) → you reuse it with inputs.
* **Dynamic SQL** = like writing a **new formula every time** based on user input (e.g., (x+5)^2 if x < 10 else (x\*2)).

### 🧪 Example to Compare

#### 🟩 **Stored Procedure Example**

sql

CopyEdit

CREATE OR REPLACE PROCEDURE get\_salary(p\_emp\_id IN NUMBER) IS

v\_salary NUMBER;

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE emp\_id = p\_emp\_id;

DBMS\_OUTPUT.PUT\_LINE('Salary: ' || v\_salary);

END;

✅ Usage:

sql

CopyEdit

BEGIN

get\_salary(101);

END;

#### 🟦 **Dynamic SQL Example**

sql

CopyEdit

DECLARE

v\_sql VARCHAR2(200);

v\_salary NUMBER;

v\_id NUMBER := 101;

BEGIN

v\_sql := 'SELECT salary FROM employees WHERE emp\_id = :1';

EXECUTE IMMEDIATE v\_sql INTO v\_salary USING v\_id;

DBMS\_OUTPUT.PUT\_LINE('Salary: ' || v\_salary);

END;

✅ Usage:  
You write your SQL at runtime — helpful when you don’t know the exact structure (e.g., table name changes dynamically).

## 🔍 When to Use What?

| **Situation** | **Use** |
| --- | --- |
| You know the structure and it won’t change → Fixed logic | ✅ Use **Procedure** |
| Table name, column list, or filters change dynamically | ✅ Use **Dynamic SQL** |

## 🚫 What NOT to do

Avoid using dynamic SQL **just to make a SELECT** if everything is known in advance — it's slower and less secure.

Would you like a simple flowchart to decide between Procedure vs Dynamic SQL?

##### You said:

so stored procedure compiled only once is it

###### ChatGPT said:

Yes, exactly! ✅

### 🔁 **Stored Procedure is Compiled Once**

When you create a stored procedure:

sql

CopyEdit

CREATE OR REPLACE PROCEDURE sample\_proc IS

BEGIN

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

END;

✅ Oracle **compiles and stores** it **once** in the database as a named PL/SQL object.

### 📌 Then what happens when you call it?

When you execute:

sql

CopyEdit

BEGIN

sample\_proc;

END;

✅ Oracle simply **runs the already compiled version** from the shared SQL area — no re-parsing or recompilation (unless it’s invalidated).

### 🔄 In contrast — Dynamic SQL:

sql

CopyEdit

EXECUTE IMMEDIATE 'SELECT \* FROM employees';

❌ This query is:

* Parsed
* Bound
* Compiled
* Executed

**each time it runs** — which adds **overhead**.

### 🧠 Summary:

| **Feature** | **Stored Procedure** | **Dynamic SQL** |
| --- | --- | --- |
| **Compilation** | ✅ Once (at creation) | ❌ Every execution |
| **Stored in DB** | ✅ Yes | ❌ No |
| **Performance** | ✅ Faster on reuse | ❌ Slower if repeated often |
| **Security** | ✅ More secure (grants) | ❌ More vulnerable |

Let me know if you'd like a memory trick or visual to remember this!

##### You said:

Highly flexible (you build SQL dynamically) can you brief this with example compare to procedure

###### ChatGPT said:

Absolutely! Let’s break it down simply:

## 🔄 **Why Dynamic SQL is Highly Flexible (Compared to Stored Procedures)**

Stored procedures are great when you know the exact **structure** of SQL beforehand.

But what if:

* Table name changes?
* Column list varies?
* Filters are passed at runtime?

👉 **That’s when Dynamic SQL is useful!**

### 🔧 Scenario Example – Dynamic SQL vs Stored Procedure

### ✅ **Stored Procedure** (Fixed logic)

sql

CopyEdit

CREATE OR REPLACE PROCEDURE get\_employees\_by\_dept(p\_dept VARCHAR2) IS

BEGIN

FOR r IN (SELECT \* FROM employees WHERE department = p\_dept) LOOP

DBMS\_OUTPUT.PUT\_LINE(r.employee\_id || ' - ' || r.name);

END LOOP;

END;

#### 📌 Limitation:

* Only works with the employees table.
* Columns and table are fixed.
* Can't handle flexible column selection or table switching.

### 🚀 **Dynamic SQL Version** – Highly Flexible

sql

CopyEdit

DECLARE

v\_sql VARCHAR2(1000);

v\_table VARCHAR2(30) := 'employees';

v\_column VARCHAR2(30) := 'department';

v\_value VARCHAR2(30) := 'IT';

BEGIN

v\_sql := 'SELECT \* FROM ' || v\_table || ' WHERE ' || v\_column || ' = :val';

FOR r IN (EXECUTE IMMEDIATE v\_sql USING v\_value) LOOP

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || r.employee\_id);

END LOOP;

END;

#### ✅ Flexibility:

* You can pass table names, columns, and filters at runtime.
* You can dynamically build queries for **reporting**, **admin tools**, **search screens**, etc.

### 💡 Summary Table

| **Feature** | **Stored Procedure** | **Dynamic SQL** |
| --- | --- | --- |
| **Structure** | Fixed (hardcoded SQL) | Flexible (built at runtime) |
| **Best for** | Business logic with known SQL | Admin panels, reporting tools |
| **Runtime table/column choice** | ❌ Not possible | ✅ Fully dynamic |
| **Performance** | ✅ Faster (compiled once) | ❌ Slight overhead |

### 👨‍💼 Real-World Helpdesk Use Cases

| **Scenario** | **Dynamic SQL Use** |
| --- | --- |
| Admin selects department dynamically | Dynamic WHERE |
| Engineer updates ticket via app | Dynamic UPDATE with bind |
| Auto-generate reports table weekly | Dynamic CREATE |
| Role-based filtering | EXECUTE IMMEDIATE with bind |

## 🧠 Final Recap Table

| **Concept** | **Syntax Example** | **When to Use** |
| --- | --- | --- |
| EXECUTE IMMEDIATE | `'SELECT COUNT(\*) FROM ' |  |
| Bind Variables | ...WHERE username = :1 | Prevent SQL injection |
| Dynamic INSERT | `'INSERT INTO ' |  |
| Dynamic DDL | `'CREATE TABLE ' |  |
| DBMS\_SQL | DEFINE\_COLUMN, FETCH\_ROWS | Advanced dynamic structure needs |

Would you like me to now:

* Add a **menu option** in your Helpdesk project for dynamic SQL report?
* Or show a **full packaged procedure** that uses these features together?

Let me know how you'd like to apply it next.

##### You said:

show one more example of row level and statement level of triggers

###### ChatGPT said:

Sure! Let's break down the **two types of triggers** with clear examples and real-world analogies:

## 🔸 1. ROW-LEVEL Trigger

### ✅ What is it?

A **row-level trigger** fires **once for every row** affected by an INSERT, UPDATE, or DELETE.

### 🧠 Real-Life Analogy:

If 5 users log in, **each user's login** gets individually logged.

### 🔍 Example: Audit Log on tickets Table (Row-Level)

sql

CopyEdit

CREATE OR REPLACE TRIGGER trg\_ticket\_audit

AFTER INSERT OR UPDATE OR DELETE ON tickets

FOR EACH ROW

BEGIN

IF INSERTING THEN

INSERT INTO audit\_log (ticket\_id, action, action\_time)

VALUES (:NEW.ticket\_id, 'INSERTED', SYSDATE);

ELSIF UPDATING THEN

INSERT INTO audit\_log (ticket\_id, action, action\_time)

VALUES (:OLD.ticket\_id, 'UPDATED', SYSDATE);

ELSIF DELETING THEN

INSERT INTO audit\_log (ticket\_id, action, action\_time)

VALUES (:OLD.ticket\_id, 'DELETED', SYSDATE);

END IF;

END;

/

### 🔎 Key:

* FOR EACH ROW → mandatory for row-level triggers
* :NEW and :OLD → refer to values before and after DML

## 🔸 2. STATEMENT-LEVEL Trigger

### ✅ What is it?

A **statement-level trigger** fires **only once** for the triggering event, regardless of how many rows are affected.

### 🧠 Real-Life Analogy:

If a bulk update affects 100 records, this trigger logs only **one event**.

### 🔍 Example: Track DML Operations on user\_credentials

sql

CopyEdit

CREATE OR REPLACE TRIGGER trg\_user\_stmt

AFTER INSERT OR DELETE ON user\_credentials

BEGIN

INSERT INTO admin\_log (message, action\_time)

VALUES ('A bulk INSERT or DELETE occurred on user\_credentials', SYSDATE);

END;

/

### 🔎 Key:

* No FOR EACH ROW → it runs **once per operation**
* Cannot use :NEW or :OLD here

## 🔁 Summary: Row vs. Statement Trigger

| **Feature** | **ROW-LEVEL** | **STATEMENT-LEVEL** |
| --- | --- | --- |
| Fires per row? | ✅ Yes | ❌ No, only once per statement |
| :NEW / :OLD? | ✅ Available | ❌ Not allowed |
| Use Case | Audit logs, change tracking | DDL operations, summary logs |
| Syntax | FOR EACH ROW required | No FOR EACH ROW |

## 

## **Case Study: Employee Management System using PL/SQL Packages & Triggers**

### **Objective:**

Design and implement a modular PL/SQL solution using **Packages** and **Triggers** to manage employee records and enforce business rules on data changes.

### **Scenario:**

You are working in the IT department of a company named **SmartTech Solutions**. The company maintains an EMPLOYEE table to store employee records. As part of performance tuning and business logic enforcement, you have been asked to:

1. Modularize business logic using **Packages**.
2. Track changes and prevent invalid operations using **Triggers**.

## **Step-by-Step Implementation**

### **Step 1: Create the Base Table**

CREATE TABLE EMPLOYEE (

EMP\_ID NUMBER PRIMARY KEY,

EMP\_NAME VARCHAR2(100),

DEPARTMENT VARCHAR2(50),

SALARY NUMBER,

CREATED\_AT DATE DEFAULT SYSDATE

);

### **Step 2: Create an Audit Table**

CREATE TABLE EMPLOYEE\_AUDIT (

EMP\_ID NUMBER,

ACTION\_TAKEN VARCHAR2(20),

ACTION\_DATE DATE

);

### **Step 3: Create a Package Specification**

CREATE OR REPLACE PACKAGE EMP\_PKG AS

PROCEDURE ADD\_EMPLOYEE(p\_id NUMBER, p\_name VARCHAR2, p\_dept VARCHAR2, p\_salary NUMBER);

PROCEDURE GET\_EMPLOYEE(p\_id NUMBER);

END EMP\_PKG;

### **Step 4: Create a Package Body**

CREATE OR REPLACE PACKAGE BODY EMP\_PKG AS

-- Private procedure (not exposed outside)

PROCEDURE VALIDATE\_SALARY(p\_salary NUMBER) IS

BEGIN

IF p\_salary < 10000 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Salary must be at least 10,000');

END IF;

END;

-- Public procedure to add employee

PROCEDURE ADD\_EMPLOYEE(p\_id NUMBER, p\_name VARCHAR2, p\_dept VARCHAR2, p\_salary NUMBER) IS

BEGIN

VALIDATE\_SALARY(p\_salary);

INSERT INTO EMPLOYEE (EMP\_ID, EMP\_NAME, DEPARTMENT, SALARY)

VALUES (p\_id, p\_name, p\_dept, p\_salary);

END;

-- Public procedure to get employee data

PROCEDURE GET\_EMPLOYEE(p\_id NUMBER) IS

v\_name EMPLOYEE.EMP\_NAME%TYPE;

v\_dept EMPLOYEE.DEPARTMENT%TYPE;

v\_salary EMPLOYEE.SALARY%TYPE;

BEGIN

SELECT EMP\_NAME, DEPARTMENT, SALARY INTO v\_name, v\_dept, v\_salary

FROM EMPLOYEE WHERE EMP\_ID = p\_id;

DBMS\_OUTPUT.PUT\_LINE('Name: ' || v\_name || ', Department: ' || v\_dept || ', Salary: ' || v\_salary);

END;

END EMP\_PKG;

### **Step 5: Create a BEFORE UPDATE Trigger**

Purpose: Prevent salary decrease beyond a threshold

CREATE OR REPLACE TRIGGER BEFORE\_SALARY\_UPDATE

BEFORE UPDATE OF SALARY ON EMPLOYEE

FOR EACH ROW

BEGIN

IF :NEW.SALARY < :OLD.SALARY THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Salary reduction is not allowed');

END IF;

END;

### **Step 6: Create an AFTER DELETE Trigger**

Purpose: Log deleted employee records in an audit table

CREATE OR REPLACE TRIGGER AFTER\_EMP\_DELETE

AFTER DELETE ON EMPLOYEE

FOR EACH ROW

BEGIN

INSERT INTO EMPLOYEE\_AUDIT (EMP\_ID, ACTION\_TAKEN, ACTION\_DATE)

VALUES (:OLD.EMP\_ID, 'DELETE', SYSDATE);

END;

### **Step 7: Test the Implementation**

#### **Insert Valid Record**

BEGIN

EMP\_PKG.ADD\_EMPLOYEE(101, 'John Doe', 'IT', 15000);

END;

#### **Fetch Employee Info**

BEGIN

EMP\_PKG.GET\_EMPLOYEE(101);

END;

#### **Attempt Invalid Salary Update**

UPDATE EMPLOYEE SET SALARY = 9000 WHERE EMP\_ID = 101; -- Should raise error

#### **Delete Record**

DELETE FROM EMPLOYEE WHERE EMP\_ID = 101;

SELECT \* FROM EMPLOYEE\_AUDIT; -- Should show logged delete entry

### **Learning Outcomes:**

* Understand the use of **PL/SQL Packages** for modularization.
* Learn the difference between **public and private** procedures.
* Apply **triggers** to enforce business rules and perform post-action logging.
* Implement **best practices** like meaningful naming and validation checks.

## **Case Study Title:**

**"Role-Based Order Management with Audit and Exception Handling in an E-Commerce System"**

## **Problem Statement:**

You are working as a database developer for an e-commerce company. The company sells products online, and users (buyers, sellers, admins) interact with the system. Your task is to implement order creation, cancellation, and tracking features using PL/SQL with the following constraints:

1. Only users with specific roles can perform actions like cancel orders or update product status.
2. All critical operations must be logged in an audit table.
3. Exception handling must be included to prevent data corruption or inconsistent state.

## **Step-by-Step Case Study Breakdown**

### **Step 1: Create Base Tables**

#### **A. Users Table**

CREATE TABLE USERS (

USER\_ID NUMBER PRIMARY KEY,

USER\_NAME VARCHAR2(100),

ROLE\_ID NUMBER, -- 1 = Admin, 2 = Seller, 3 = Buyer

CREATED\_AT DATE DEFAULT SYSDATE

);

#### **B. Products Table**

CREATE TABLE PRODUCTS (

PRODUCT\_ID NUMBER PRIMARY KEY,

PRODUCT\_NAME VARCHAR2(100),

SELLER\_ID NUMBER REFERENCES USERS(USER\_ID),

PRICE NUMBER,

STATUS VARCHAR2(20) DEFAULT 'AVAILABLE'

);

#### **C. Orders Table**

CREATE TABLE ORDERS (

ORDER\_ID NUMBER PRIMARY KEY,

BUYER\_ID NUMBER REFERENCES USERS(USER\_ID),

PRODUCT\_ID NUMBER REFERENCES PRODUCTS(PRODUCT\_ID),

ORDER\_DATE DATE DEFAULT SYSDATE,

STATUS VARCHAR2(20) DEFAULT 'PLACED'

);

### **Step 2: Create the Role-Based Access Simulation Table**

CREATE TABLE ROLE\_ACCESS (

ROLE\_ID NUMBER,

ACTION VARCHAR2(50) -- 'CANCEL\_ORDER', 'ADD\_PRODUCT', 'UPDATE\_PRODUCT'

);

Insert sample role permissions:

INSERT INTO ROLE\_ACCESS VALUES (1, 'CANCEL\_ORDER'); -- Admin

INSERT INTO ROLE\_ACCESS VALUES (2, 'UPDATE\_PRODUCT'); -- Seller

INSERT INTO ROLE\_ACCESS VALUES (3, 'CANCEL\_ORDER'); -- Buyer (allowed for their own orders only)

### **Step 3: Create the Audit Table**

CREATE TABLE ORDER\_AUDIT (

AUDIT\_ID NUMBER GENERATED ALWAYS AS IDENTITY,

USER\_ID NUMBER,

ACTION VARCHAR2(50),

ORDER\_ID NUMBER,

TIMESTAMP DATE DEFAULT SYSDATE,

OLD\_STATUS VARCHAR2(20),

NEW\_STATUS VARCHAR2(20)

);

### **Step 4: Create Stored Procedure for Order Cancellation**

CREATE OR REPLACE PROCEDURE CANCEL\_ORDER (

p\_user\_id NUMBER,

p\_order\_id NUMBER

) IS

v\_role\_id NUMBER;

v\_buyer\_id NUMBER;

v\_status VARCHAR2(20);

permission\_missing EXCEPTION;

BEGIN

SELECT ROLE\_ID INTO v\_role\_id FROM USERS WHERE USER\_ID = p\_user\_id;

SELECT BUYER\_ID, STATUS INTO v\_buyer\_id, v\_status FROM ORDERS WHERE ORDER\_ID = p\_order\_id;

-- Role-based access check

IF v\_role\_id = 3 AND p\_user\_id != v\_buyer\_id THEN

RAISE permission\_missing;

END IF;

-- Proceed with cancellation

UPDATE ORDERS

SET STATUS = 'CANCELLED'

WHERE ORDER\_ID = p\_order\_id;

-- Audit

INSERT INTO ORDER\_AUDIT (USER\_ID, ACTION, ORDER\_ID, OLD\_STATUS, NEW\_STATUS)

VALUES (p\_user\_id, 'CANCEL\_ORDER', p\_order\_id, v\_status, 'CANCELLED');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Invalid Order ID or User ID.');

WHEN permission\_missing THEN

DBMS\_OUTPUT.PUT\_LINE('You are not authorized to cancel this order.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unexpected error occurred: ' || SQLERRM);

END;

/

### **Step 5: Create Trigger to Track Product Status Change**

CREATE OR REPLACE TRIGGER PRODUCT\_STATUS\_AUDIT

AFTER UPDATE OF STATUS ON PRODUCTS

FOR EACH ROW

BEGIN

INSERT INTO ORDER\_AUDIT (USER\_ID, ACTION, ORDER\_ID, OLD\_STATUS, NEW\_STATUS)

VALUES (:NEW.SELLER\_ID, 'UPDATE\_PRODUCT', NULL, :OLD.STATUS, :NEW.STATUS);

END;

/

### **Step 6: Create Cursor-Based Report**

DECLARE

CURSOR c\_orders IS

SELECT O.ORDER\_ID, U.USER\_NAME, P.PRODUCT\_NAME, O.STATUS

FROM ORDERS O

JOIN USERS U ON O.BUYER\_ID = U.USER\_ID

JOIN PRODUCTS P ON O.PRODUCT\_ID = P.PRODUCT\_ID

WHERE O.STATUS = 'PLACED';

BEGIN

FOR rec IN c\_orders LOOP

DBMS\_OUTPUT.PUT\_LINE('Order ID: ' || rec.ORDER\_ID || ', Buyer: ' || rec.USER\_NAME || ', Product: ' || rec.PRODUCT\_NAME || ', Status: ' || rec.STATUS);

END LOOP;

END;

/

### **Step 7: Test the Workflow**

1. Create sample users with roles.
2. Add products using the seller role.
3. Place orders using buyers.
4. Attempt to cancel orders using unauthorized users (test exception handling).
5. Update product status and check the audit log.
6. Generate order reports using the cursor.

## **Key Learning Outcomes**

|  |  |
| --- | --- |
| **Topic** | **Implemented Through** |
| **Joins** | Cursor, Views, Reports |
| **Procedures** | CANCEL\_ORDER with access validation and transaction |
| **Triggers** | Product status audit logging |
| **Cursors** | Iterating and displaying placed orders |
| **Exception Handling** | Custom and predefined exceptions in CANCEL\_ORDER |

# **Assignment Case Study: Advanced Trigger Implementation in E-Commerce Inventory System**

## **Case Study Title:**

**Automated Inventory & Audit Control Using PL/SQL Triggers**

## **Problem Statement:**

You are part of the database team for an e-commerce platform that manages products, inventory, and orders. The system must ensure that inventory counts are automatically adjusted when new orders are placed or cancelled. Additionally, any change in inventory must be logged in an audit table for compliance and debugging.

Your task is to implement **BEFORE and AFTER triggers** to handle business logic such as:

* Real-time stock level validation before order insertion
* Auto stock update on order insert/delete
* Change tracking for product stock levels in an audit table

This assignment focuses on designing **robust trigger-based automation with validations, exception handling, and audit logging**.

## **Assignment Objectives:**

* Implement BEFORE INSERT trigger to validate product availability
* Implement AFTER INSERT and AFTER DELETE triggers to adjust stock
* Capture all inventory changes in an INVENTORY\_AUDIT table
* Test edge cases such as ordering with insufficient stock

## **Deliverables**

* SQL scripts for table creation, triggers, and test cases
* Screenshot or output of:  
  + Trigger executions
  + Audit table entries
  + Stock before/after operations
* Explanation of how each trigger works and its purpose
* Edge case test results (e.g., insufficient stock, multiple orders, bulk cancellation)

## **Evaluation Criteria (Total 20 Marks)**

|  |  |
| --- | --- |
| **Category** | **Max Marks** |
| Table & Trigger Implementation | 8 |
| Business Logic Validations | 4 |
| Inventory Audit Accuracy | 4 |
| Exception Handling & Testing | 2 |
| Script Structure & Comments | 2 |