**Minutes of Session – Day 4: PL/SQL Procedures & Functions**

Location: Hybrid (Online + Classroom)  
Facilitator: Niti Dwivedi

Attendees:

**Session Objectives**

* Understand the concepts of Procedures and Functions in PL/SQL.
* Learn to create and execute stored procedures and functions.
* Understand parameter types: IN, OUT, and IN OUT.
* Distinguish between Procedures and Functions.
* Discuss real-time applications, best practices, and common pitfalls.

**1. PL/SQL Procedures**

**Definition:** A Procedure is a named PL/SQL block that performs a specific task and does not mandatorily return a value. It is invoked using the EXECUTE command or from another PL/SQL block.

**Syntax Example:**

CREATE OR REPLACE PROCEDURE update\_salary(p\_emp\_id IN NUMBER, p\_new\_sal IN NUMBER) IS

BEGIN

    UPDATE employees SET salary = p\_new\_sal WHERE employee\_id = p\_emp\_id;

END;

Create a procedure to accepts a customer id and display the customer’s contact information , including first name , last name and email;

Create or replace procedure proc\_display\_contact(p\_customer\_id Number)

Is

R\_contact contacts%ROWTYPE;

Begin

Select \* into r\_contact

From contacts where customer\_id = p\_custoemr\_id;

Dbms\_output.put\_line(r\_contact.first\_name || ‘ ‘ ||r\_contact.last\_name || r.contact.email || ‘==’);

Exception

When other then

Dbms\_output.put\_line(SQLERRM);

End;

**Application Areas:**

* Modularizing business logic
* Performing batch updates
* Encapsulating DML operations

**2. PL/SQL Functions**

**Definition:** A Function is a named PL/SQL block that must return a value using the RETURN keyword. It can be called from SQL statements, procedures, or PL/SQL blocks.

**Syntax Example:**

CREATE OR REPLACE FUNCTION get\_bonus(p\_salary IN NUMBER) RETURN NUMBER IS

BEGIN

    RETURN p\_salary \* 0.10;

END;

Create or replace function factorial(x number)

Return number

Is

F number;

Begin

If x=0 then

F:=1;

Else

F:= x\* factorial(x-1);

End if ;

Return f;

End;

Calling the function:

Declare

New number;

Result number;

Begin

Result := factorial(5);

DBMS\_outour.put\_line(result); End;

Create a function with the name as calculate\_emi with exception handling where the input variables are p\_principal , p\_rate ,p\_term which will return a loan emi

Create or replace function fn\_calculate\_emi(

P\_principal in number,

P\_rate in number,

P\_term in number

) return number is f\_l\_emi number;

Begin

If p\_principal <=0 or p\_rate <=0 or p\_term <=0 then

Raise\_application\_error(-20001,’All input values must be positive.’);

End if;

F\_l\_emi := ((p\_principal \* p\_rate \*p\_term)/100);

Return round(f\_l\_emi,2);

Exception

When zero\_divide then

Raise\_application\_error(-2002, ‘ Division by zer error’);

When value\_error then

Raise\_application\_error(-2003, ‘ Input value error’);

When others then

Raise\_application\_error(-20099, ‘ unexpected error : ’ || SQLERRM);

**Application Areas:**

* Calculations in reports
* Validation logic
* Reusable return value logic

**3. Parameters in Procedures and Functions**

|  |  |  |
| --- | --- | --- |
| **Parameter Mode** | **Description** | **Use Case** |
| IN | Passes value into the procedure/function | Input parameters |
| OUT | Returns value from procedure/function | Output results like status, ID |
| IN OUT | Passes value in and returns updated value | Counters, cumulative values |

**4. Key Differences: Procedure vs Function**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Procedure** | **Function** |
| Return Value | Not mandatory | Must return a value |
| Usage in SQL | Cannot be used directly in SQL | Can be called from SQL |
| Purpose | Perform an action | Compute and return a value |
| Invocation | EXEC or from another block | As part of expression or block |

**Best Practices – Do’s and Don’ts**

**Do’s**

* Use IN, OUT, and IN OUT parameters appropriately.
* Include exception handling (WHEN OTHERS) in both procedures and functions.
* Follow naming conventions (e.g., proc\_, fn\_ prefixes).
* Use functions when a value needs to be returned in SQL.
* Use AUTHID CURRENT\_USER if access rights depend on session user.

**Don’ts**

* Don’t use DML statements inside functions if you plan to call them from SQL.
* Avoid using too many OUT parameters — prefer using records or objects.
* Don’t ignore return values from functions; it defeats the purpose.
* Avoid writing large, monolithic procedures — break into modular blocks.

**Important events for current and upcoming week :**

**Sprint 2 presentation : Thursday/Monday**

**Last Day of training/ Sprint 3: 10/11th of july 2025**

**Week 2 Graded Assessment is on 4th Of July/ Friday**

**No Week 3 Graded Assessment**

**Action items for all participants :**

1. Prepare a Document with Following
   1. List of FAQs based on PL/SQL topics.
   2. Sprint wise Progress of your Project. Ex. Sprint retrospective.
   3. Case Study based on FAQs topics Cursors, Exceptional handling, Collections, Userdefine exceptions, Procedures and Functions etc.

**Scenario-Based Use Case: FinTech – Loan EMI Processing**

**Business Scenario:** A fintech company needs to automate the EMI calculation and payment status update for its loan customers every month.

**Objective:** Use a function to calculate EMI and a procedure to update the EMI status.

**Step-by-Step Implementation:**

**Step 1: Create the EMI Calculation Function**

CREATE OR REPLACE FUNCTION calculate\_emi (

    p\_principal IN NUMBER,

    p\_rate      IN NUMBER,

    p\_term      IN NUMBER

) RETURN NUMBER IS

    l\_emi NUMBER;

BEGIN

    l\_emi := (p\_principal \* p\_rate/12/100) /

             (1 - POWER(1 + p\_rate/12/100, -p\_term));

    RETURN ROUND(l\_emi, 2);

END;

CREATE OR REPLACE FUNCTION calculate\_emi (

    p\_principal IN NUMBER,

    p\_rate      IN NUMBER,

    p\_term      IN NUMBER

) RETURN NUMBER IS

    l\_emi NUMBER;

BEGIN

    l\_emi := (p\_principal \* p\_rate/12/100) /

             (1 - POWER(1 + p\_rate/12/100, -p\_term));

    RETURN ROUND(l\_emi, 2);

END;

 Step-by-Step Explanation of calculate\_emi

 Purpose of the Function

This function is used to calculate EMI (Equated Monthly Installment) for a loan, based on:

Principal amount (p\_principal)

Annual interest rate (p\_rate)

Loan term in months (p\_term)

🔹 Step 1: Function Header

CREATE OR REPLACE FUNCTION calculate\_emi (

    p\_principal IN NUMBER,   -- Loan amount

    p\_rate      IN NUMBER,   -- Annual interest rate (in %)

    p\_term      IN NUMBER    -- Loan duration in months

)

p\_principal: The original loan amount (e.g., 100000)

p\_rate: Annual interest rate as a percentage (e.g., 7.5)

p\_term: Number of months (e.g., 24)

This function returns a NUMBER (the calculated EMI).

🔹 Step 2: Declare Local Variable

l\_emi NUMBER;

A local variable l\_emi is declared to store the intermediate EMI result.

🔹 Step 3: EMI Formula Logic

l\_emi := (p\_principal \* p\_rate/12/100) /

         (1 - POWER(1 + p\_rate/12/100, -p\_term));

​

It calculates on the basis of formula where

P = principal

r = monthly interest rate = p\_rate / 12 / 100

n = number of months (p\_term)

Breakdown:

p\_rate/12/100: Converts annual interest % to monthly decimal

POWER(1 + monthly\_rate, -p\_term): Calculates denominator

Entire expression calculates monthly EMI

🔹 Step 4: Return the Result

RETURN ROUND(l\_emi, 2);

Returns the EMI rounded to 2 decimal places.

 Example Usage

SELECT calculate\_emi(100000, 7.5, 24) AS emi FROM dual;

This would calculate EMI for:

₹100,000 principal

7.5% annual interest

24 months duration

 Output

Assuming inputs:

Principal: 100000

Interest Rate: 7.5%

Term: 24 months

EMI = ~₹4,513.51

—----------------------------------------

Using Exception

**1. Enhanced Function: calculate\_emi (with Exception Handling)**

CREATE OR REPLACE FUNCTION calculate\_emi (

    p\_principal IN NUMBER,

    p\_rate      IN NUMBER,

    p\_term      IN NUMBER

) RETURN NUMBER IS

    l\_emi NUMBER;

BEGIN

    IF p\_principal <= 0 OR p\_rate <= 0 OR p\_term <= 0 THEN

        RAISE\_APPLICATION\_ERROR(-20001, 'All input values must be positive.');

    END IF;

    l\_emi := (p\_principal \* p\_rate/12/100) /

             (1 - POWER(1 + p\_rate/12/100, -p\_term));

    RETURN ROUND(l\_emi, 2);

EXCEPTION

    WHEN ZERO\_DIVIDE THEN

        RAISE\_APPLICATION\_ERROR(-20002, 'Mathematical error: division by zero');

    WHEN VALUE\_ERROR THEN

        RAISE\_APPLICATION\_ERROR(-20003, 'Input value error');

    WHEN OTHERS THEN

        RAISE\_APPLICATION\_ERROR(-20099, 'Unexpected error: ' || SQLERRM);

END;

/

**2. Sample Loans Table**

DROP TABLE Loans PURGE;

CREATE TABLE Loans (

    loan\_id        NUMBER PRIMARY KEY,

    customer\_name  VARCHAR2(100),

    loan\_amount    NUMBER,

    interest\_rate  NUMBER,

    tenure\_months  NUMBER,

    emi\_amount     NUMBER

);

**3. Stored Procedure to Insert Loan & Calculate EMI**

CREATE OR REPLACE PROCEDURE insert\_loan (

    p\_loan\_id       IN NUMBER,

    p\_customer\_name IN VARCHAR2,

    p\_amount        IN NUMBER,

    p\_rate          IN NUMBER,

    p\_tenure        IN NUMBER

) IS

    v\_emi NUMBER;

BEGIN

    -- Calculate EMI using the function

    v\_emi := calculate\_emi(p\_amount, p\_rate, p\_tenure);

    -- Insert into Loans table

    INSERT INTO Loans (

        loan\_id, customer\_name, loan\_amount, interest\_rate, tenure\_months, emi\_amount

    ) VALUES (

        p\_loan\_id, p\_customer\_name, p\_amount, p\_rate, p\_tenure, v\_emi

    );

    DBMS\_OUTPUT.PUT\_LINE('Loan inserted with EMI: ₹' || v\_emi);

EXCEPTION

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

/

**4. Run the Procedure**

**Example:**

BEGIN

    insert\_loan(1001, 'Amit Kumar', 200000, 8.5, 36);

END;

/

 Output

Loan inserted with EMI: ₹6315.11

**5. Query the Table**

SELECT \* FROM Loans;

**Step 2: Create the Procedure to Update EMI Status**

CREATE OR REPLACE PROCEDURE update\_emi\_status (

    p\_loan\_id IN NUMBER,

    p\_emi\_paid IN NUMBER

) IS

    v\_expected\_emi NUMBER;

    v\_principal NUMBER;

    v\_rate NUMBER;

    v\_term NUMBER;

BEGIN

    SELECT principal, interest\_rate, term\_months

    INTO v\_principal, v\_rate, v\_term

    FROM loan\_details

    WHERE loan\_id = p\_loan\_id;

    v\_expected\_emi := calculate\_emi(v\_principal, v\_rate, v\_term);

    IF p\_emi\_paid = v\_expected\_emi THEN

        UPDATE loan\_payment\_status

        SET status = 'PAID'

        WHERE loan\_id = p\_loan\_id;

    ELSE

        UPDATE loan\_payment\_status

        SET status = 'PARTIALLY PAID'

        WHERE loan\_id = p\_loan\_id;

    END IF;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        DBMS\_OUTPUT.PUT\_LINE('Loan not found.');

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

**Step 3: Call the Procedure**

EXEC update\_emi\_status(10001, 5423.50);

**Outcome:**

* Function calculate\_emi modularizes the EMI logic.
* Procedure update\_emi\_status automates the payment tracking.
* Business rules are encapsulated and reusable across systems.

**Key Takeaways**

* Procedures are ideal for performing actions or DML operations.
* Functions are best suited for logic that needs to return values.
* In fintech systems, combining both ensures modular, reusable, and efficient transaction handling.

**Assignment: PL/SQL-Based Library Book Management System**

**Objective**

Implement a PL/SQL system that tracks books, allows issuing and returning, and maintains accurate book stock and issue logs.

**Task List**

**Step 1: Create Required Tables**

* Create students, books, and issue\_log tables
* Apply primary and foreign key constraints
* Include appropriate data types and default values

**Step 2: Insert Sample Data**

* Add at least 2 student records
* Add at least 3 book records with varied stock (0 and above)

**Step 3: Create Sequence**

* Create a sequence seq\_issue\_id to auto-generate issue\_id for issue\_log

**Step 4: Create Function**

* Create a function get\_issued\_count(student\_id) that returns the count of books currently issued to a student (i.e., return\_date IS NULL)

**Step 5: Create Procedure – Issue Book**

* Create procedure issue\_book(student\_id, book\_id, OUT status)  
    
  + Checks if the book is available
  + Inserts into issue\_log
  + Updates stock
  + Returns a success or failure message

**Step 6: Create Procedure – Return Book**

* Create procedure return\_book(issue\_id, OUT status)  
    
  + Updates return\_date in issue\_log
  + Increments book stock

**Step 7: Write Test Blocks**

* Use anonymous PL/SQL blocks to test:  
    
  + Issuing a book
  + Returning a book
  + Getting issued count for a student

**FAQs – Common Questions from Learners**

**Q1. What happens if I try to issue a book with stock = 0?** A: The procedure will catch this and return “Book not available.”

**Q2. Can I issue the same book to multiple students?** A: Yes, as long as stock > 0.

**Q3. What if I try to return a book that’s already returned?** A: The return\_book procedure will show a message like “Invalid issue ID or book already returned.”

**Q4. Why use a function for issued count instead of a procedure?** A: Functions return values and can be directly used in SELECT statements. Procedures perform actions but don't return values directly.

**Q5. What does the sequence seq\_issue\_id do?** A: It auto-generates unique issue\_id values every time a book is issued.

**Future Enhancements / Additional Tasks**

**Enhancement 1: Add Fine Calculation**

* Add a column due\_date in issue\_log
* Write a function to calculate fines for late return

**Enhancement 2: Add librarian Table and Track Issued By**

* Add librarian\_id to issue\_log
* Update procedures to record which librarian issued/returned the book

**Enhancement 3: Use Triggers**

* Create a trigger to prevent issuing more than 3 books to the same student at a time

**Enhancement 4: Create Views**

* vw\_books\_available → Lists books with stock > 0
* vw\_student\_issues → Shows student-wise issue history

**Enhancement 5: Audit Table for Returns**

* Maintain a log table that records return activity with timestamp

**Case Study: Event Ticket Booking & Management System**

**Domain: Event/Ticketing (Concerts, Sports, Seminars)**

**Scenario**

You are designing a backend system for a national-level event management platform. Your system must support:

1. Ticket booking for multiple events.
2. Pricing logic based on event type and seat category.
3. Customer booking history.
4. Seat availability check and updates.
5. Optional: Ticket cancellation and refund logic.

**Assignment Tasks**

**Task 1: Table Design**

Create normalized tables to store the following data:

* Event details (event name, type, venue, date)
* Ticket categories (VIP, Regular, Economy), each with base price and available seats
* Customer master data
* Booking records (linking customers to events and categories)

**Hint:** Use appropriate PRIMARY KEY and FOREIGN KEY constraints. Ensure category-seat mapping is flexible per event if scaling later.

**Task 2: Sequence Creation**

Create a sequence that auto-generates **booking IDs** for each ticket booked.

**Hint:** Use CREATE SEQUENCE for tracking unique booking transactions.

**Task 3: Function – Calculate Final Ticket Price**

Develop a PL/SQL **function** that:

* Accepts ticket\_category\_id and event\_type
* Applies a **pricing multiplier** based on event type:  
    
  + Concert = 1.2
  + Sports = 1.1
  + Conference = 1.0
* Returns the **final price** (base price × multiplier)

**Hint:** Use CASE or IF-ELSIF to apply business logic.

**Task 4: Procedure – Book Ticket**

Write a **procedure** that:

* Accepts event ID, customer ID, and ticket category ID as input
* Checks if seats are available
* Calls your **price calculation function**
* Inserts booking into a bookings table
* Updates seat count
* Returns booking status as an OUT parameter

**Hint:** Handle cases like:

* No seats available
* Invalid category or event
* Booking confirmation with amount paid

**Task 5: Test Your Booking Logic**

Use anonymous PL/SQL blocks to test:

* Booking with available seats
* Booking when seats are full
* Booking for different categories and event types

**Hint:** Use DBMS\_OUTPUT.PUT\_LINE to display status messages.

**Task 6 (Optional – Stretch Goals)**

1. **Cancellation Procedure**
   * Accepts booking ID and restores seat count
   * Updates status and optionally logs refund request
2. **Create Views**
   * View for customer-wise booking history
   * View for category-wise sales summary per event
3. **Promo Code Functionality**
   * Create a function to apply discount using a promo code table

**Expected Learning Outcomes**

|  |  |
| --- | --- |
| **Skill** | **Outcome** |
| Function | Create dynamic business logic returning values |
| Procedure | Implement validation, DML, and OUT parameters |
| Relational Design | Work with normalized structure & joins |
| PL/SQL Control Structures | Apply logic, exception handling |
| Real-World Simulation | Model ticketing platform use case |