# Week 1 ControlStructures Hands-On

## Scenario: The bank wants to apply a discount to loan interest rates for customers above 60 years old.

### Question: Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

### Explanation

This PL/SQL block is designed to automate the process of applying a discount to loan interest rates for elderly customers. It uses a cursor to efficiently iterate through each customer record. For every customer, it calculates their age based on their date of birth. If a customer is found to be over 60 years old, their associated loan interest rate is reduced by 1%. The DBMS\_OUTPUT.PUT\_LINE statement provides real-time feedback, indicating which customer IDs received the discount. The COMMIT statement ensures that all changes are permanently saved to the database, while the EXCEPTION block handles any potential errors, rolling back changes to maintain data integrity.

-- PL/SQL code for Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.  
DECLARE  
 CURSOR c\_customers IS SELECT CustomerID, DOB, InterestRate FROM Customers;  
 v\_age NUMBER;  
BEGIN  
 FOR rec IN c\_customers LOOP  
 v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, rec.DOB) / 12);  
 IF v\_age > 60 THEN  
 UPDATE Loans SET InterestRate = InterestRate \* 0.99 WHERE CustomerID = rec.CustomerID;  
 DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount to loan interest for Customer ID: ' || rec.CustomerID);  
 END IF;  
 END LOOP;  
 COMMIT;  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.  
Applied 1% discount to loan interest for Customer ID: 101  
Applied 1% discount to loan interest for Customer ID: 105  
PL/SQL procedure successfully completed.

### Output Narrative

The simulated output demonstrates the successful application of the interest rate discount. You can see messages confirming that a 1% discount was applied to loan interests for specific customer IDs (101 and 105 in this example). The final line, ‘PL/SQL procedure successfully completed.’, indicates that the entire block ran without unhandled errors, and all eligible customers received their discounts.

# Week 1 ControlStructures Hands-On

## Scenario: A customer can be promoted to VIP status based on their balance.

### Question: Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

### Explanation

This PL/SQL block identifies and promotes customers to VIP status based on their account balance. It iterates through all customer records, checking if their Balance exceeds $10,000. For customers meeting this criterion, a hypothetical IsVIP flag (which would need to be a column in the Customers table) is set to TRUE. The block provides output confirming each customer promoted to VIP status and includes robust error handling to ensure data consistency.

-- PL/SQL code for Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.  
DECLARE  
 CURSOR c\_customers IS SELECT CustomerID, Balance FROM Customers;  
BEGIN  
 FOR rec IN c\_customers LOOP  
 IF rec.Balance > 10000 THEN  
 -- Assuming an IsVIP column exists, or we can add it  
 -- ALTER TABLE Customers ADD IsVIP BOOLEAN DEFAULT FALSE;  
 UPDATE Customers SET IsVIP = TRUE WHERE CustomerID = rec.CustomerID;  
 DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || rec.CustomerID || ' promoted to VIP status.');  
 END IF;  
 END LOOP;  
 COMMIT;  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.  
Customer ID: 203 promoted to VIP status.  
Customer ID: 207 promoted to VIP status.  
PL/SQL procedure successfully completed.

### Output Narrative

The output shows which customers have been successfully upgraded to VIP status. In this example, Customer IDs 203 and 207 were identified as having balances over $10,000 and were subsequently promoted. The ‘PL/SQL procedure successfully completed.’ message confirms the successful execution of the block.

# Week 1 ControlStructures Hands-On

## Scenario: The bank wants to send reminders to customers whose loans are due within the next 30 days.

### Question: Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

### Explanation

This PL/SQL block is designed to help banks send timely reminders to customers about their upcoming loan due dates. It uses a cursor to select all loans that are due within the next 30 days from the current date. For each identified loan, it constructs and prints a clear reminder message, including the loan ID, customer name, and the exact due date. This automated process ensures that customers are well-informed, reducing the likelihood of missed payments.

-- PL/SQL code for Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.  
DECLARE  
 CURSOR c\_loans IS  
 SELECT c.Name, l.LoanID, l.EndDate  
 FROM Loans l JOIN Customers c ON l.CustomerID = c.CustomerID  
 WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;  
BEGIN  
 FOR rec IN c\_loans LOOP  
 DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID || ' for ' || rec.Name || ' is due by ' || TO\_CHAR(rec.EndDate, 'YYYY-MM-DD') || '.');  
 END LOOP;  
EXCEPTION  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.  
Reminder: Loan 123 for John Doe is due by 2025-07-25.  
Reminder: Loan 456 for Jane Smith is due by 2025-07-10.  
PL/SQL procedure successfully completed.

### Output Narrative

The output displays the reminder messages generated for loans approaching their due dates. For instance, Loan 123 for John Doe is due by July 25, 2025, and Loan 456 for Jane Smith is due by July 10, 2025. This output would typically be used to generate automated emails or SMS messages to customers.

# Week 2 ErrorHandling Hands-On

## Scenario: Handle exceptions during fund transfers between accounts.

### Question: Write a stored procedure SafeTransferFunds that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

### Explanation

The SafeTransferFunds stored procedure is designed to handle money transfers between two bank accounts securely, with robust error handling. It first checks if the source account has sufficient funds. If not, it raises a custom exception (e\_insufficient\_funds). If funds are sufficient, it updates the balances of both accounts and records the transactions in the Transactions table. Crucially, it uses COMMIT to finalize successful transfers and ROLLBACK to undo all changes if any error occurs, ensuring atomicity and data integrity. It also includes specific exception handling for NO\_DATA\_FOUND (if an account doesn’t exist) and a general OTHERS handler for unexpected errors.

-- PL/SQL code for Write a stored procedure SafeTransferFunds that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.  
CREATE OR REPLACE PROCEDURE SafeTransferFunds(  
 p\_from\_account\_id IN NUMBER,  
 p\_to\_account\_id IN NUMBER,  
 p\_amount IN NUMBER  
) AS  
 v\_from\_balance NUMBER;  
 e\_insufficient\_funds EXCEPTION;  
 PRAGMA EXCEPTION\_INIT(e\_insufficient\_funds, -20001);  
BEGIN  
 SELECT Balance INTO v\_from\_balance FROM Accounts WHERE AccountID = p\_from\_account\_id;  
  
 IF v\_from\_balance < p\_amount THEN  
 RAISE e\_insufficient\_funds;  
 END IF;  
  
 UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account\_id;  
 UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account\_id;  
  
 INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
 VALUES (SEQ\_TRANSACTION\_ID.NEXTVAL, p\_from\_account\_id, SYSDATE, p\_amount, 'TransferOut');  
 INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
 VALUES (SEQ\_TRANSACTION\_ID.NEXTVAL, p\_to\_account\_id, SYSDATE, p\_amount, 'TransferIn');  
  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');  
EXCEPTION  
 WHEN e\_insufficient\_funds THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in source account.');  
 -- Log error to an error log table  
 WHEN NO\_DATA\_FOUND THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: One or both accounts not found.');  
 -- Log error  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);  
 -- Log error  
END;  
/

### Output

-- Simulated output for Write a stored procedure SafeTransferFunds that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.  
-- Example 1: Successful transfer  
Funds transferred successfully.  
  
-- Example 2: Insufficient funds  
Error: Insufficient funds in source account.  
  
-- Example 3: Account not found  
Error: One or both accounts not found.

### Output Narrative

The simulated output showcases different scenarios for fund transfers. **Example 1** demonstrates a successful transfer, where the message ‘Funds transferred successfully.’ confirms the operation. **Example 2** illustrates an attempt to transfer more money than available, resulting in the ‘Error: Insufficient funds in source account.’ message, and the transaction is rolled back. **Example 3** shows the outcome when one of the specified accounts does not exist, leading to ‘Error: One or both accounts not found.’, preventing any erroneous updates.

# Week 2 ErrorHandling Hands-On

## Scenario: Manage errors when updating employee salaries.

### Question: Write a stored procedure UpdateSalary that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

### Explanation

The UpdateSalary procedure is designed to adjust an employee’s salary by a given percentage. It first retrieves the current salary of the specified employee. Then, it calculates the new salary and updates the Employees table. A COMMIT ensures the change is saved. Crucially, it includes error handling for the NO\_DATA\_FOUND exception, which occurs if the p\_employee\_id does not correspond to an existing employee. In such cases, the transaction is rolled back, and an informative error message is displayed, preventing updates to non-existent records.

-- PL/SQL code for Write a stored procedure UpdateSalary that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.  
CREATE OR REPLACE PROCEDURE UpdateSalary(  
 p\_employee\_id IN NUMBER,  
 p\_percentage\_increase IN NUMBER  
) AS  
 v\_current\_salary NUMBER;  
BEGIN  
 SELECT Salary INTO v\_current\_salary FROM Employees WHERE EmployeeID = p\_employee\_id;  
  
 UPDATE Employees  
 SET Salary = v\_current\_salary \* (1 + p\_percentage\_increase / 100)  
 WHERE EmployeeID = p\_employee\_id;  
  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for Employee ID: ' || p\_employee\_id);  
EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID ' || p\_employee\_id || ' does not exist.');  
 -- Log error message  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);  
 -- Log error  
END;  
/

### Output

-- Simulated output for Write a stored procedure UpdateSalary that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.  
-- Example 1: Successful update  
Salary updated successfully for Employee ID: 101.  
  
-- Example 2: Employee not found  
Error: Employee ID 999 does not exist.

### Output Narrative

The output illustrates two scenarios for salary updates. **Example 1** shows a successful update for Employee ID 101, confirming the salary adjustment. **Example 2** demonstrates the error handling in action: when an attempt is made to update a non-existent employee (ID 999), the procedure catches the NO\_DATA\_FOUND error, rolls back any potential changes, and outputs a clear error message, ensuring data integrity.

# Week 2 ErrorHandling Hands-On

## Scenario: Ensure data integrity when adding a new customer.

### Question: Write a stored procedure AddNewCustomer that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

### Explanation

The AddNewCustomer procedure is designed to safely insert new customer records into the Customers table. It takes customer details as input and attempts to insert them. A critical part of this procedure is its error handling for duplicate CustomerID values. If an attempt is made to insert a customer with an ID that already exists (triggering an ORA-00001 unique constraint violation), the procedure catches this specific error using PRAGMA EXCEPTION\_INIT, rolls back the failed insertion, and informs the user that the customer already exists, thus preventing data duplication.

-- PL/SQL code for Write a stored procedure AddNewCustomer that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.  
CREATE OR REPLACE PROCEDURE AddNewCustomer(  
 p\_customer\_id IN NUMBER,  
 p\_name IN VARCHAR2,  
 p\_dob IN DATE,  
 p\_balance IN NUMBER  
) AS  
 e\_customer\_exists EXCEPTION;  
 PRAGMA EXCEPTION\_INIT(e\_customer\_exists, -00001); -- ORA-00001: unique constraint violated  
BEGIN  
 INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
 VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('New customer added successfully: ' || p\_name);  
EXCEPTION  
 WHEN e\_customer\_exists THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_customer\_id || ' already exists. Insertion prevented.');  
 -- Log error  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);  
 -- Log error  
END;  
/

### Output

-- Simulated output for Write a stored procedure AddNewCustomer that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.  
-- Example 1: Successful insertion  
New customer added successfully: Alice Brown.  
  
-- Example 2: Customer already exists  
Error: Customer with ID 101 already exists. Insertion prevented.

### Output Narrative

The output demonstrates the procedure’s behavior in two key scenarios. **Example 1** shows a successful addition of a new customer, ‘Alice Brown’, with a confirmation message. **Example 2** illustrates the error handling: when an attempt is made to add a customer with an existing ID (101), the procedure gracefully handles the duplicate, preventing the insertion and providing a clear error message, ensuring the uniqueness of customer records.

# Week 3 StoredProcedures Hands-On

## Scenario: The bank needs to process monthly interest for all savings accounts.

### Question: Write a stored procedure ProcessMonthlyInterest that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

### Explanation

The ProcessMonthlyInterest procedure automates the calculation and application of monthly interest to all savings accounts. It updates the Balance of every account where AccountType is ‘Savings’ by increasing it by 1%. The COMMIT statement ensures that these financial updates are permanently recorded. Error handling is included to catch any unforeseen issues during the process, ensuring that if a problem occurs, the transaction is rolled back, and an error message is logged, preventing partial updates.

-- PL/SQL code for Write a stored procedure ProcessMonthlyInterest that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.  
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS  
BEGIN  
 UPDATE Accounts  
 SET Balance = Balance \* 1.01  
 WHERE AccountType = 'Savings';  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for all savings accounts.');  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error processing monthly interest: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a stored procedure ProcessMonthlyInterest that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.  
Monthly interest processed for all savings accounts.  
PL/SQL procedure successfully completed.

### Output Narrative

The output confirms that the monthly interest processing for all savings accounts was successfully completed. This indicates that the balances of all eligible savings accounts have been updated with the applied interest, and the transaction was committed to the database without errors.

# Week 3 StoredProcedures Hands-On

## Scenario: The bank wants to implement a bonus scheme for employees based on their performance.

### Question: Write a stored procedure UpdateEmployeeBonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

### Explanation

The UpdateEmployeeBonus procedure allows for the mass application of a bonus to employees within a specific department. It takes the p\_department name and a p\_bonus\_percentage as input. The procedure then updates the Salary of all employees belonging to that department by the specified percentage. A COMMIT ensures the salary changes are saved, and a general exception handler is in place to roll back the transaction and report any errors, maintaining payroll data integrity.

-- PL/SQL code for Write a stored procedure UpdateEmployeeBonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.  
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(  
 p\_department IN VARCHAR2,  
 p\_bonus\_percentage IN NUMBER  
) AS  
BEGIN  
 UPDATE Employees  
 SET Salary = Salary \* (1 + p\_bonus\_percentage / 100)  
 WHERE Department = p\_department;  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Bonus applied to employees in ' || p\_department || ' department.');  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error applying bonus: ' || SQLERRM);  
END;

### Output

-- Simulated output for Write a stored procedure UpdateEmployeeBonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.  
Bonus applied to employees in IT department.  
PL/SQL procedure successfully completed.

### Output Narrative

The output confirms that the bonus was successfully applied to all employees in the ‘IT’ department. This means their salaries have been updated according to the specified bonus percentage, and the changes are now permanent in the database.

# Week 3 StoredProcedures Hands-On

## Scenario: Customers should be able to transfer funds between their accounts.

### Question: Write a stored procedure TransferFunds that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

### Explanation

The TransferFunds procedure facilitates the transfer of a specified p\_amount from a p\_from\_account\_id to a p\_to\_account\_id. Before initiating the transfer, it performs a crucial check to ensure the source account has sufficient balance. If not, it raises a custom application error, preventing overdrafts. Upon successful validation, it debits the source account, credits the destination account, and records both sides of the transaction in the Transactions table. The COMMIT ensures the changes are permanent, and the EXCEPTION block handles any errors, rolling back the entire operation to maintain financial consistency.

-- PL/SQL code for Write a stored procedure TransferFunds that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.  
CREATE OR REPLACE PROCEDURE TransferFunds(  
 p\_from\_account\_id IN NUMBER,  
 p\_to\_account\_id IN NUMBER,  
 p\_amount IN NUMBER  
) AS  
 v\_from\_balance NUMBER;  
BEGIN  
 SELECT Balance INTO v\_from\_balance FROM Accounts WHERE AccountID = p\_from\_account\_id;  
  
 IF v\_from\_balance < p\_amount THEN  
 RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in source account.');  
 END IF;  
  
 UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account\_id;  
 UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account\_id;  
  
 INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
 VALUES (SEQ\_TRANSACTION\_ID.NEXTVAL, p\_from\_account\_id, SYSDATE, p\_amount, 'TransferOut');  
 INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
 VALUES (SEQ\_TRANSACTION\_ID.NEXTVAL, p\_to\_account\_id, SYSDATE, p\_amount, 'TransferIn');  
  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error transferring funds: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a stored procedure TransferFunds that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.  
Funds transferred successfully.  
PL/SQL procedure successfully completed.

### Output Narrative

The output ‘Funds transferred successfully.’ indicates that the transfer operation was completed without any issues. This means the specified amount was successfully moved from the source account to the destination account, and both the account balances and transaction records have been updated accordingly.

# Week 4 Functions Hands-On

## Scenario: Calculate the age of customers for eligibility checks.

### Question: Write a function CalculateAge that takes a customer’s date of birth as input and returns their age in years.

### Explanation

The CalculateAge function is a simple yet powerful utility for determining a person’s age in years based on their date of birth. It takes a DATE input (p\_dob) representing the birth date. The core logic uses MONTHS\_BETWEEN to find the total number of months between the birth date and the current system date (SYSDATE), then divides by 12 to get years. TRUNC is used to get a whole number, effectively giving the age in completed years. This function is invaluable for eligibility checks or demographic analysis.

-- PL/SQL code for Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.  
CREATE OR REPLACE FUNCTION CalculateAge(  
 p\_dob IN DATE  
) RETURN NUMBER IS  
BEGIN  
 RETURN TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);  
END;  
/

### Output

-- Simulated output for Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.  
SELECT CalculateAge(TO\_DATE('1990-01-15', 'YYYY-MM-DD')) AS CustomerAge FROM DUAL;  
  
CUSTOMERAGE  
-----------  
35

### Output Narrative

The output demonstrates how to call the CalculateAge function and its result. When called with a birth date of ‘1990-01-15’, the function correctly returns ‘35’, indicating the customer’s age in years. This confirms the function’s accuracy in calculating age based on the provided date of birth.

# Week 4 Functions Hands-On

## Scenario: The bank needs to compute the monthly installment for a loan.

### Question: Write a function CalculateMonthlyInstallment that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

### Explanation

The CalculateMonthlyInstallment function is a financial tool designed to compute the fixed monthly payment required for a loan. It takes the total p\_loan\_amount, the annual p\_interest\_rate, and the p\_loan\_duration\_years as inputs. The function first converts the annual interest rate to a monthly rate and calculates the total number of payments. It then applies the standard loan amortization formula to determine the precise monthly installment. A special condition handles zero interest rates to prevent division by zero. This function is essential for loan officers and financial planning.

-- PL/SQL code for Write a function CalculateMonthlyInstallment that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.  
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(  
 p\_loan\_amount IN NUMBER,  
 p\_interest\_rate IN NUMBER,  
 p\_loan\_duration\_years IN NUMBER  
) RETURN NUMBER IS  
 v\_monthly\_interest\_rate NUMBER := (p\_interest\_rate / 100) / 12;  
 v\_number\_of\_payments NUMBER := p\_loan\_duration\_years \* 12;  
BEGIN  
 IF v\_monthly\_interest\_rate = 0 THEN  
 RETURN p\_loan\_amount / v\_number\_of\_payments;  
 ELSE  
 RETURN p\_loan\_amount \* v\_monthly\_interest\_rate / (1 - POWER(1 + v\_monthly\_interest\_rate, -v\_number\_of\_payments));  
 END IF;  
END;  
/

### Output

-- Simulated output for Write a function CalculateMonthlyInstallment that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.  
SELECT CalculateMonthlyInstallment(100000, 5, 30) AS MonthlyPayment FROM DUAL;  
  
MONTHLYPAYMENT  
--------------  
536.82

### Output Narrative

The output shows the calculated monthly installment for a hypothetical loan. For a loan of $100,000 at a 5% annual interest rate over 30 years, the function correctly calculates a monthly payment of $536.82. This demonstrates the function’s ability to accurately determine loan installments, which is crucial for financial planning and loan agreements.

# Week 4 Functions Hands-On

## Scenario: Check if a customer has sufficient balance before making a transaction.

### Question: Write a function HasSufficientBalance that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

### Explanation

The HasSufficientBalance function provides a quick and reliable way to check if a given bank account has enough funds for a transaction. It accepts an p\_account\_id and a p\_amount to be checked. The function queries the Accounts table to retrieve the current Balance for the specified account. It then returns TRUE if the account’s balance is greater than or equal to the requested amount, and FALSE otherwise. Error handling for NO\_DATA\_FOUND ensures that if an invalid account ID is provided, it gracefully returns FALSE.

-- PL/SQL code for Write a function HasSufficientBalance that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.  
CREATE OR REPLACE FUNCTION HasSufficientBalance(  
 p\_account\_id IN NUMBER,  
 p\_amount IN NUMBER  
) RETURN BOOLEAN IS  
 v\_balance NUMBER;  
BEGIN  
 SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_account\_id;  
 RETURN v\_balance >= p\_amount;  
EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 RETURN FALSE;  
END;  
/

### Output

-- Simulated output for Write a function HasSufficientBalance that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.  
DECLARE  
 v\_has\_funds BOOLEAN;  
BEGIN  
 v\_has\_funds := HasSufficientBalance(1, 500);  
 IF v\_has\_funds THEN  
 DBMS\_OUTPUT.PUT\_LINE('Account 1 has sufficient funds.');  
 ELSE  
 DBMS\_OUTPUT.PUT\_LINE('Account 1 does NOT have sufficient funds.');  
 END IF;  
END;  
/  
  
Account 1 has sufficient funds.  
PL/SQL procedure successfully completed.

### Output Narrative

The output demonstrates the usage and result of the HasSufficientBalance function. In this example, when checking if Account 1 has $500, the output confirms ‘Account 1 has sufficient funds.’, indicating that the balance meets the requirement. This function is vital for pre-transaction validation, preventing overdrafts and ensuring smooth financial operations.

# Week 5 Triggers Hands-On

## Scenario: Automatically update the last modified date when a customer’s record is updated.

### Question: Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer’s record is updated.

### Explanation

The UpdateCustomerLastModified trigger is an automated mechanism that ensures the LastModified timestamp of a customer record is always up-to-date. This BEFORE UPDATE trigger fires automatically every time a row in the Customers table is modified. For each row being updated (FOR EACH ROW), it sets the LastModified column of the new row version (:NEW.LastModified) to the current system date (SYSDATE). This is crucial for auditing and tracking changes to customer data without requiring manual intervention in application code.

-- PL/SQL code for Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.  
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified  
BEFORE UPDATE ON Customers  
FOR EACH ROW  
BEGIN  
 :NEW.LastModified := SYSDATE;  
END;  
/

### Output

-- Simulated output for Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.  
Trigger created.  
  
-- Example: Update a customer record  
UPDATE Customers SET Name = 'John Doe Updated' WHERE CustomerID = 1;  
  
-- Verify LastModified column (assuming you can query it)  
SELECT CustomerID, Name, LastModified FROM Customers WHERE CustomerID = 1;  
  
CUSTOMERID NAME LASTMODIFIED  
---------- ---------------- ------------------  
 1 John Doe Updated 29-JUN-25

### Output Narrative

The output first confirms that the trigger has been successfully created. Following an example UPDATE statement that changes a customer’s name, the subsequent SELECT query demonstrates the effect of the trigger. You can observe that the LastModified column for Customer ID 1 has been automatically updated to the current date (e.g., ‘29-JUN-25’), even though the UPDATE statement itself did not explicitly modify this column. This verifies the trigger’s functionality in maintaining accurate modification timestamps.

# Week 5 Triggers Hands-On

## Scenario: Maintain an audit log for all transactions.

### Question: Write a trigger LogTransaction that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

### Explanation

The LogTransaction trigger is designed to create an immutable audit trail for all financial transactions. It’s an AFTER INSERT trigger on the Transactions table, meaning it activates immediately after a new transaction record is successfully added. For each new transaction, it automatically inserts a corresponding record into the AuditLog table, capturing key details like TransactionID, AccountID, Amount, TransactionType, and a timestamp. This ensures that every transaction is logged for compliance, security, and historical analysis, providing a clear record of all financial movements.

-- PL/SQL code for Write a trigger LogTransaction that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.  
CREATE TABLE AuditLog (  
 LogID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
 TransactionID NUMBER,  
 AccountID NUMBER,  
 Amount NUMBER,  
 TransactionType VARCHAR2(10),  
 LogDate TIMESTAMP DEFAULT SYSTIMESTAMP  
);  
  
CREATE OR REPLACE TRIGGER LogTransaction  
AFTER INSERT ON Transactions  
FOR EACH ROW  
BEGIN  
 INSERT INTO AuditLog (TransactionID, AccountID, Amount, TransactionType)  
 VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.Amount, :NEW.TransactionType);  
END;  
/

### Output

-- Simulated output for Write a trigger LogTransaction that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.  
Table created.  
Trigger created.  
  
-- Example: Insert a transaction  
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (1001, 1, SYSDATE, 150, 'Deposit');  
  
-- Verify AuditLog (assuming you can query it)  
SELECT TransactionID, AccountID, Amount, TransactionType FROM AuditLog WHERE TransactionID = 1001;  
  
TRANSACTIONID ACCOUNTID AMOUNT TRANSACTIONTYPE  
------------- ----------- ---------- ---------------  
 1001 1 150 Deposit

### Output Narrative

The output first confirms the successful creation of both the AuditLog table and the LogTransaction trigger. An example INSERT statement for a deposit transaction is then shown. The subsequent SELECT query on the AuditLog table demonstrates that the trigger automatically captured the details of the newly inserted transaction (Transaction ID 1001, Account ID 1, Amount 150, Type ‘Deposit’). This verifies that the audit logging mechanism is functioning correctly, providing a reliable record of all transactions.

# Week 5 Triggers Hands-On

## Scenario: Enforce business rules on deposits and withdrawals.

### Question: Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

### Explanation

The CheckTransactionRules trigger is a powerful mechanism for enforcing critical business rules directly at the database level, ensuring data integrity for financial transactions. This BEFORE INSERT trigger activates before any new record is added to the Transactions table. It contains logic to validate withdrawals (ensuring sufficient balance) and deposits (ensuring positive amounts). If a rule is violated, it raises a custom application error, preventing the invalid transaction from being recorded and providing immediate feedback, thus safeguarding financial data.

-- PL/SQL code for Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.  
CREATE OR REPLACE TRIGGER CheckTransactionRules  
BEFORE INSERT ON Transactions  
FOR EACH ROW  
BEGIN  
 IF :NEW.TransactionType = 'Withdrawal' THEN  
 DECLARE  
 v\_balance NUMBER;  
 BEGIN  
 SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;  
 IF v\_balance < :NEW.Amount THEN  
 RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient balance for withdrawal.');  
 END IF;  
 END;  
 ELSIF :NEW.TransactionType = 'Deposit' THEN  
 IF :NEW.Amount <= 0 THEN  
 RAISE\_APPLICATION\_ERROR(-20003, 'Deposit amount must be positive.');  
 END IF;  
 END IF;  
END;  
/

### Output

-- Simulated output for Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.  
Trigger created.  
  
-- Example 1: Valid withdrawal  
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (1002, 1, SYSDATE, 50, 'Withdrawal');  
  
PL/SQL procedure successfully completed.  
  
-- Example 2: Insufficient balance for withdrawal  
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (1003, 1, SYSDATE, 5000, 'Withdrawal');  
  
ORA-20002: Insufficient balance for withdrawal.  
ORA-06512: at "SCHEMA.CHECKTRANSACTIONRULES", line 9  
ORA-04088: error during execution of trigger "SCHEMA.CHECKTRANSACTIONRULES"  
  
-- Example 3: Invalid deposit amount  
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (1004, 2, SYSDATE, 0, 'Deposit');  
  
ORA-20003: Deposit amount must be positive.  
ORA-06512: at "SCHEMA.CHECKTRANSACTIONRULES", line 16  
ORA-04088: error during execution of trigger "SCHEMA.CHECKTRANSACTIONRULES"

### Output Narrative

The output demonstrates the trigger’s effectiveness in enforcing business rules. **Example 1** shows a successful, valid withdrawal. **Example 2** illustrates a failed withdrawal attempt due to insufficient funds; the trigger intercepts this, prevents the insertion, and raises an ORA-20002 error. **Example 3** demonstrates a failed deposit due to a non-positive amount, resulting in an ORA-20003 error. These examples confirm that the trigger actively prevents invalid transactions, maintaining the integrity of financial data.

# Week 6 Cursors Hands-On

## Scenario: Generate monthly statements for all customers.

### Question: Write a PL/SQL block using an explicit cursor GenerateMonthlyStatements that retrieves all transactions for the current month and prints a statement for each customer.

### Explanation

This PL/SQL block leverages an explicit cursor to generate comprehensive monthly statements for all customers. It retrieves all transactions that occurred in the current month, joining Transactions, Accounts, and Customers tables to gather complete details. The cursor then iterates through these transactions, grouping them by customer and formatting the output to resemble a clear, readable bank statement. This automated process simplifies the generation of periodic financial reports, providing customers with a detailed overview of their account activity.

-- PL/SQL code for Write a PL/SQL block using an explicit cursor GenerateMonthlyStatements that retrieves all transactions for the current month and prints a statement for each customer.  
DECLARE  
 CURSOR c\_transactions IS  
 SELECT t.TransactionID, c.Name AS CustomerName, a.AccountID, t.Amount, t.TransactionType, t.TransactionDate  
 FROM Transactions t  
 JOIN Accounts a ON t.AccountID = a.AccountID  
 JOIN Customers c ON a.CustomerID = c.CustomerID  
 WHERE TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM')  
 ORDER BY c.Name, t.TransactionDate;  
  
 v\_current\_customer\_name VARCHAR2(100);  
BEGIN  
 DBMS\_OUTPUT.PUT\_LINE('--- Monthly Statements ---');  
 v\_current\_customer\_name := NULL;  
  
 FOR rec IN c\_transactions LOOP  
 IF rec.CustomerName != v\_current\_customer\_name THEN  
 IF v\_current\_customer\_name IS NOT NULL THEN  
 DBMS\_OUTPUT.PUT\_LINE('--------------------------');  
 END IF;  
 DBMS\_OUTPUT.PUT\_LINE('\nCustomer: ' || rec.CustomerName);  
 DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || rec.AccountID);  
 DBMS\_OUTPUT.PUT\_LINE('Date Type Amount');  
 DBMS\_OUTPUT.PUT\_LINE('--------------------------');  
 v\_current\_customer\_name := rec.CustomerName;  
 END IF;  
 DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(rec.TransactionDate, 'DD-MON-YY') || ' ' || RPAD(rec.TransactionType, 9) || ' ' || TO\_CHAR(rec.Amount, '999,999.00'));  
 END LOOP;  
 DBMS\_OUTPUT.PUT\_LINE('--------------------------');  
 DBMS\_OUTPUT.PUT\_LINE('--- End of Statements ---');  
EXCEPTION  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error generating statements: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a PL/SQL block using an explicit cursor GenerateMonthlyStatements that retrieves all transactions for the current month and prints a statement for each customer.  
--- Monthly Statements ---  
  
Customer: Jane Smith  
Account ID: 2  
Date Type Amount  
--------------------------  
29-JUN-25 Withdrawal 300.00  
--------------------------  
  
Customer: John Doe  
Account ID: 1  
Date Type Amount  
--------------------------  
29-JUN-25 Deposit 200.00  
--------------------------  
--- End of Statements ---  
PL/SQL procedure successfully completed.

### Output Narrative

The simulated output presents a formatted monthly statement for each customer with transactions in the current month. For example, it shows Jane Smith’s withdrawal and John Doe’s deposit, along with their respective account IDs, transaction dates, types, and amounts. The clear separation by customer and the summary lines make it easy to review financial activity, mimicking a real-world bank statement.

# Week 6 Cursors Hands-On

## Scenario: Apply annual fee to all accounts.

### Question: Write a PL/SQL block using an explicit cursor ApplyAnnualFee that deducts an annual maintenance fee from the balance of all accounts.

### Explanation

This PL/SQL block efficiently applies a fixed annual maintenance fee to all bank accounts. It uses an explicit cursor (FOR UPDATE) to select each account, ensuring that the rows are locked during the update to prevent concurrent modification issues. For every account, it deducts a predefined v\_annual\_fee from its Balance. The DBMS\_OUTPUT.PUT\_LINE statement provides a clear log of which accounts were debited. The COMMIT ensures all fee deductions are permanently recorded, and the EXCEPTION block handles any errors, rolling back the entire operation if a problem occurs.

-- PL/SQL code for Write a PL/SQL block using an explicit cursor ApplyAnnualFee that deducts an annual maintenance fee from the balance of all accounts.  
DECLARE  
 CURSOR c\_accounts IS SELECT AccountID, Balance FROM Accounts FOR UPDATE;  
 v\_annual\_fee CONSTANT NUMBER := 25;  
BEGIN  
 FOR rec IN c\_accounts LOOP  
 UPDATE Accounts  
 SET Balance = Balance - v\_annual\_fee  
 WHERE AccountID = rec.AccountID;  
 DBMS\_OUTPUT.PUT\_LINE('Applied annual fee of ' || v\_annual\_fee || ' to Account ID: ' || rec.AccountID);  
 END LOOP;  
 COMMIT;  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error applying annual fee: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a PL/SQL block using an explicit cursor ApplyAnnualFee that deducts an annual maintenance fee from the balance of all accounts.  
Applied annual fee of 25 to Account ID: 1  
Applied annual fee of 25 to Account ID: 2  
PL/SQL procedure successfully completed.

### Output Narrative

The output clearly indicates that the annual fee of $25 was successfully applied to both Account ID 1 and Account ID 2. This confirms that the PL/SQL block iterated through the accounts and correctly deducted the fee from each, with the final message indicating successful completion of the entire process.

# Week 6 Cursors Hands-On

## Scenario: Update the interest rate for all loans based on a new policy.

### Question: Write a PL/SQL block using an explicit cursor UpdateLoanInterestRates that fetches all loans and updates their interest rates based on the new policy.

### Explanation

This PL/SQL block is designed to systematically update loan interest rates based on a new policy. It uses an explicit cursor (FOR UPDATE) to iterate through each loan record, allowing for safe modification. Inside the loop, it applies a conditional logic: if a loan’s current interest rate is 5%, it’s adjusted to 4.5%; otherwise, it’s increased by 0.1%. The DBMS\_OUTPUT.PUT\_LINE statement provides a clear audit trail of each rate change. The COMMIT ensures all updates are saved, and the EXCEPTION block handles any errors, rolling back changes to maintain data consistency.

-- PL/SQL code for Write a PL/SQL block using an explicit cursor UpdateLoanInterestRates that fetches all loans and updates their interest rates based on the new policy.  
DECLARE  
 CURSOR c\_loans IS SELECT LoanID, InterestRate FROM Loans FOR UPDATE;  
 v\_new\_rate NUMBER;  
BEGIN  
 FOR rec IN c\_loans LOOP  
 -- Example new policy: If current rate is 5, set to 4.5; otherwise, increase by 0.1  
 IF rec.InterestRate = 5 THEN  
 v\_new\_rate := 4.5;  
 ELSE  
 v\_new\_rate := rec.InterestRate + 0.1;  
 END IF;  
  
 UPDATE Loans  
 SET InterestRate = v\_new\_rate  
 WHERE LoanID = rec.LoanID;  
 DBMS\_OUTPUT.PUT\_LINE('Updated interest rate for Loan ID: ' || rec.LoanID || ' from ' || rec.InterestRate || ' to ' || v\_new\_rate);  
 END LOOP;  
 COMMIT;  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error updating loan interest rates: ' || SQLERRM);  
END;  
/

### Output

-- Simulated output for Write a PL/SQL block using an explicit cursor UpdateLoanInterestRates that fetches all loans and updates their interest rates based on the new policy.  
Updated interest rate for Loan ID: 1 from 5 to 4.5  
PL/SQL procedure successfully completed.

### Output Narrative

The output shows the result of applying the new interest rate policy. For Loan ID 1, the interest rate was successfully updated from 5% to 4.5%. This confirms that the PL/SQL block correctly identified the loan and applied the specified rate adjustment, with the final message indicating successful completion of the entire process.

# Week 7 Packages Hands-On

## Scenario: Group all customer-related procedures and functions into a package.

### Question: Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance.

### Explanation

The CustomerManagement package is a comprehensive solution for organizing and managing all customer-related operations within a bank’s database. It encapsulates three key functionalities: AddCustomer (to securely add new customers, handling duplicates), UpdateCustomerDetails (to modify existing customer information like name or balance), and GetCustomerBalance (a function to retrieve a customer’s current balance). By grouping these related procedures and functions into a single package, it promotes modularity, reusability, and easier maintenance of the codebase, making customer data management more efficient and less prone to errors.

-- PL/SQL code for Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance.  
CREATE OR REPLACE PACKAGE CustomerManagement AS  
 PROCEDURE AddCustomer(  
 p\_customer\_id IN NUMBER,  
 p\_name IN VARCHAR2,  
 p\_dob IN DATE,  
 p\_balance IN NUMBER  
 );  
 PROCEDURE UpdateCustomerDetails(  
 p\_customer\_id IN NUMBER,  
 p\_new\_name IN VARCHAR2 DEFAULT NULL,  
 p\_new\_balance IN NUMBER DEFAULT NULL  
 );  
 FUNCTION GetCustomerBalance(  
 p\_customer\_id IN NUMBER  
 ) RETURN NUMBER;  
END CustomerManagement;  
/  
  
CREATE OR REPLACE PACKAGE BODY CustomerManagement AS  
 PROCEDURE AddCustomer(  
 p\_customer\_id IN NUMBER,  
 p\_name IN VARCHAR2,  
 p\_dob IN DATE,  
 p\_balance IN NUMBER  
 ) IS  
 BEGIN  
 INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
 VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Customer ' || p\_name || ' added.');  
 EXCEPTION  
 WHEN DUP\_VAL\_ON\_INDEX THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_customer\_id || ' already exists.');  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error adding customer: ' || SQLERRM);  
 END AddCustomer;  
  
 PROCEDURE UpdateCustomerDetails(  
 p\_customer\_id IN NUMBER,  
 p\_new\_name IN VARCHAR2 DEFAULT NULL,  
 p\_new\_balance IN NUMBER DEFAULT NULL  
 ) IS  
 BEGIN  
 UPDATE Customers  
 SET  
 Name = NVL(p\_new\_name, Name),  
 Balance = NVL(p\_new\_balance, Balance),  
 LastModified = SYSDATE  
 WHERE CustomerID = p\_customer\_id;  
 COMMIT;  
 IF SQL%ROWCOUNT > 0 THEN  
 DBMS\_OUTPUT.PUT\_LINE('Customer ' || p\_customer\_id || ' details updated.');  
 ELSE  
 DBMS\_OUTPUT.PUT\_LINE('Customer ' || p\_customer\_id || ' not found.');  
 END IF;  
 EXCEPTION  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error updating customer details: ' || SQLERRM);  
 END UpdateCustomerDetails;  
  
 FUNCTION GetCustomerBalance(  
 p\_customer\_id IN NUMBER  
 ) RETURN NUMBER IS  
 v\_balance NUMBER;  
 BEGIN  
 SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_customer\_id;  
 RETURN v\_balance;  
 EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 DBMS\_OUTPUT.PUT\_LINE('Customer ' || p\_customer\_id || ' not found.');  
 RETURN NULL;  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error getting customer balance: ' || SQLERRM);  
 RETURN NULL;  
 END GetCustomerBalance;  
END CustomerManagement;  
/

### Output

-- Simulated output for Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance.  
Package created.  
Package body created.  
  
-- Example Usage:  
BEGIN  
 CustomerManagement.AddCustomer(3, 'Charlie Brown', TO\_DATE('1995-11-22', 'YYYY-MM-DD'), 2500);  
 CustomerManagement.UpdateCustomerDetails(1, p\_new\_name => 'Johnathan Doe');  
 DBMS\_OUTPUT.PUT\_LINE('Balance for Customer 2: ' || CustomerManagement.GetCustomerBalance(2));  
END;  
/  
  
Customer Charlie Brown added.  
Customer 1 details updated.  
Balance for Customer 2: 1500  
PL/SQL procedure successfully completed.

### Output Narrative

The output demonstrates the practical application of the CustomerManagement package. It shows a new customer, ‘Charlie Brown’, being successfully added. Then, Customer ID 1’s name is updated to ‘Johnathan Doe’, with a confirmation message. Finally, the balance for Customer ID 2 is retrieved and displayed as ‘1500’. This sequence illustrates how the package streamlines common customer management tasks, providing clear feedback on each operation.

# Week 7 Packages Hands-On

## Scenario: Create a package to manage employee data.

### Question: Write a package EmployeeManagement with procedures to hire new employees, update employee details, and a function to calculate annual salary.

### Explanation

The EmployeeManagement package provides a centralized and efficient way to handle various employee-related data operations. It includes procedures like HireEmployee for adding new staff members (with duplicate ID checks) and UpdateEmployeeDetails for modifying existing employee information such as position or salary. Additionally, it offers a CalculateAnnualSalary function to quickly determine an employee’s yearly earnings. This package streamlines HR processes, ensuring data consistency and simplifying common tasks related to employee records.

-- PL/SQL code for Write a package EmployeeManagement with procedures to hire new employees, update employee details, and a function to calculate annual salary.  
CREATE OR REPLACE PACKAGE EmployeeManagement AS  
 PROCEDURE HireEmployee(  
 p\_employee\_id IN NUMBER,  
 p\_name IN VARCHAR2,  
 p\_position IN VARCHAR2,  
 p\_salary IN NUMBER,  
 p\_department IN VARCHAR2,  
 p\_hire\_date IN DATE  
 );  
 PROCEDURE UpdateEmployeeDetails(  
 p\_employee\_id IN NUMBER,  
 p\_new\_position IN VARCHAR2 DEFAULT NULL,  
 p\_new\_salary IN NUMBER DEFAULT NULL  
 );  
 FUNCTION CalculateAnnualSalary(  
 p\_employee\_id IN NUMBER  
 ) RETURN NUMBER;  
END EmployeeManagement;  
/  
  
CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS  
 PROCEDURE HireEmployee(  
 p\_employee\_id IN NUMBER,  
 p\_name IN VARCHAR2,  
 p\_position IN VARCHAR2,  
 p\_salary IN NUMBER,  
 p\_department IN VARCHAR2,  
 p\_hire\_date IN DATE  
 ) IS  
 BEGIN  
 INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)  
 VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_name || ' hired.');  
 EXCEPTION  
 WHEN DUP\_VAL\_ON\_INDEX THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_employee\_id || ' already exists.');  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error hiring employee: ' || SQLERRM);  
 END HireEmployee;  
  
 PROCEDURE UpdateEmployeeDetails(  
 p\_employee\_id IN NUMBER,  
 p\_new\_position IN VARCHAR2 DEFAULT NULL,  
 p\_new\_salary IN NUMBER DEFAULT NULL  
 ) IS  
 BEGIN  
 UPDATE Employees  
 SET  
 Position = NVL(p\_new\_position, Position),  
 Salary = NVL(p\_new\_salary, Salary)  
 WHERE EmployeeID = p\_employee\_id;  
 COMMIT;  
 IF SQL%ROWCOUNT > 0 THEN  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_employee\_id || ' details updated.');  
 ELSE  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_employee\_id || ' not found.');  
 END IF;  
 EXCEPTION  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error updating employee details: ' || SQLERRM);  
 END UpdateEmployeeDetails;  
  
 FUNCTION CalculateAnnualSalary(  
 p\_employee\_id IN NUMBER  
 ) RETURN NUMBER IS  
 v\_salary NUMBER;  
 BEGIN  
 SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_employee\_id;  
 RETURN v\_salary \* 12; -- Assuming monthly salary is stored  
 EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_employee\_id || ' not found.');  
 RETURN NULL;  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error calculating annual salary: ' || SQLERRM);  
 RETURN NULL;  
 END CalculateAnnualSalary;  
END EmployeeManagement;  
/

### Output

-- Simulated output for Write a package EmployeeManagement with procedures to hire new employees, update employee details, and a function to calculate annual salary.  
Package created.  
Package body created.  
  
-- Example Usage:  
BEGIN  
 EmployeeManagement.HireEmployee(3, 'David Lee', 'Analyst', 5000, 'Finance', TO\_DATE('2020-01-01', 'YYYY-MM-DD'));  
 EmployeeManagement.UpdateEmployeeDetails(1, p\_new\_salary => 75000);  
 DBMS\_OUTPUT.PUT\_LINE('Annual Salary for Employee 2: ' || EmployeeManagement.CalculateAnnualSalary(2));  
END;  
/  
  
Employee David Lee hired.  
Employee 1 details updated.  
Annual Salary for Employee 2: 720000  
PL/SQL procedure successfully completed.

### Output Narrative

The output demonstrates the functionality of the EmployeeManagement package. It shows the successful hiring of a new employee, ‘David Lee’. Then, Employee ID 1’s salary is updated to 75,000, with a confirmation message. Finally, the annual salary for Employee ID 2 is calculated and displayed as 720,000 (assuming a monthly salary of 60,000). This illustrates how the package simplifies HR tasks and provides clear feedback on employee data changes.

# Week 7 Packages Hands-On

## Scenario: Group all account-related operations into a package.

### Question: Create a package AccountOperations with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

### Explanation

The AccountOperations package centralizes all essential banking account functionalities, making it easier to manage customer accounts. It provides procedures for OpenNewAccount (to create new accounts for customers, handling potential duplicates), CloseAccount (to deactivate accounts), and a function GetTotalCustomerBalance (to sum up a customer’s balance across all their accounts). This modular approach enhances code organization, promotes reusability, and simplifies the development and maintenance of banking applications, ensuring consistent and reliable account management.

-- PL/SQL code for Create a package AccountOperations with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.  
CREATE OR REPLACE PACKAGE AccountOperations AS  
 PROCEDURE OpenNewAccount(  
 p\_account\_id IN NUMBER,  
 p\_customer\_id IN NUMBER,  
 p\_account\_type IN VARCHAR2,  
 p\_initial\_balance IN NUMBER  
 );  
 PROCEDURE CloseAccount(  
 p\_account\_id IN NUMBER  
 );  
 FUNCTION GetTotalCustomerBalance(  
 p\_customer\_id IN NUMBER  
 ) RETURN NUMBER;  
END AccountOperations;  
/  
  
CREATE OR REPLACE PACKAGE BODY AccountOperations AS  
 PROCEDURE OpenNewAccount(  
 p\_account\_id IN NUMBER,  
 p\_customer\_id IN NUMBER,  
 p\_account\_type IN VARCHAR2,  
 p\_initial\_balance IN NUMBER  
 ) IS  
 BEGIN  
 INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)  
 VALUES (p\_account\_id, p\_customer\_id, p\_account\_type, p\_initial\_balance, SYSDATE);  
 COMMIT;  
 DBMS\_OUTPUT.PUT\_LINE('New ' || p\_account\_type || ' account opened for Customer ' || p\_customer\_id || '.');  
 EXCEPTION  
 WHEN DUP\_VAL\_ON\_INDEX THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error: Account with ID ' || p\_account\_id || ' already exists.');  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error opening new account: ' || SQLERRM);  
 END OpenNewAccount;  
  
 PROCEDURE CloseAccount(  
 p\_account\_id IN NUMBER  
 ) IS  
 BEGIN  
 DELETE FROM Accounts WHERE AccountID = p\_account\_id;  
 COMMIT;  
 IF SQL%ROWCOUNT > 0 THEN  
 DBMS\_OUTPUT.PUT\_LINE('Account ' || p\_account\_id || ' closed.');  
 ELSE  
 DBMS\_OUTPUT.PUT\_LINE('Account ' || p\_account\_id || ' not found.');  
 END IF;  
 EXCEPTION  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error closing account: ' || SQLERRM);  
 END CloseAccount;  
  
 FUNCTION GetTotalCustomerBalance(  
 p\_customer\_id IN NUMBER  
 ) RETURN NUMBER IS  
 v\_total\_balance NUMBER;  
 BEGIN  
 SELECT SUM(Balance) INTO v\_total\_balance FROM Accounts WHERE CustomerID = p\_customer\_id;  
 RETURN NVL(v\_total\_balance, 0);  
 EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 RETURN 0;  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error getting total customer balance: ' || SQLERRM);  
 RETURN NULL;  
 END GetTotalCustomerBalance;  
END AccountOperations;  
/

### Output

-- Simulated output for Create a package AccountOperations with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.  
Package created.  
Package body created.  
  
-- Example Usage:  
BEGIN  
 AccountOperations.OpenNewAccount(3, 1, 'Checking', 500);  
 AccountOperations.CloseAccount(2);  
 DBMS\_OUTPUT.PUT\_LINE('Total balance for Customer 1: ' || AccountOperations.GetTotalCustomerBalance(1));  
END;  
/  
  
New Checking account opened for Customer 1.  
Account 2 closed.  
Total balance for Customer 1: 1500  
PL/SQL procedure successfully completed.

### Output Narrative

The output demonstrates the practical use of the AccountOperations package. It shows a new ‘Checking’ account being opened for Customer ID 1, followed by the successful closure of Account ID 2. Finally, the total balance for Customer ID 1 across all their accounts is retrieved and displayed as ‘1500’. This sequence illustrates how the package streamlines common account management tasks, providing clear feedback on each operation.