**Exercise 1: Control Structures**

**Scenario 1:** 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.

**DECLARE**

**CURSOR customers IS**

**SELECT customer\_id, date\_of\_birth, loan\_interest\_rate**

**FROM customers**

**WHERE date\_of\_birth IS NOT NULL AND loan\_interest\_rate IS NOT NULL;**

**customer customers%ROWTYPE;**

**BEGIN**

**OPEN customers;**

**LOOP**

**FETCH customers INTO customer;**

**EXIT WHEN customers%NOTFOUND;**

**IF TRUNC(MONTHS\_BETWEEN(SYSDATE, customer.date\_of\_birth) / 12) > 60 THEN**

**UPDATE customers**

**SET loan\_interest\_rate = loan\_interest\_rate - 0.01**

**WHERE customer\_id = customer.customer\_id;**

**END IF;**

**END LOOP;**

**CLOSE customers;**

**END;**

Top of Form **Scenario 2:** 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.

**DECLARE**

**CURSOR customers IS**

**SELECT customer\_id, balance**

**FROM customers**

**WHERE balance IS NOT NULL;**

**customer customers%ROWTYPE;**

**BEGIN**

**OPEN customers;**

**LOOP**

**FETCH customers INTO customer;**

**EXIT WHEN customers%NOTFOUND;**

**IF customer.balance > 10000 THEN**

**UPDATE customers**

**SET IsVIP = 'TRUE'**

**WHERE customer\_id = customer.customer\_id;**

**END IF;**

**END LOOP;**

**CLOSE customers;**

**END;**

**Scenario 3:** 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.

**DECLARE**

**CURSOR loans\_due IS**

**SELECT customer\_name, loan\_due\_date**

**FROM customers**

**WHERE loan\_due\_date IS NOT NULL AND loan\_due\_date <= SYSDATE + 30;**

**loan loans\_due%ROWTYPE;**

**BEGIN**

**OPEN loans\_due;**

**LOOP**

**FETCH loans\_due INTO loan;**

**EXIT WHEN loans\_due%NOTFOUND;**

**DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan due for ' || loan.customer\_name || ' on ' || TO\_CHAR(loan.loan\_due\_date, 'DD-MON-YYYY'));**

**END LOOP;**

**CLOSE loans\_due;**

**END;**

**Exercise 2: Error Handling**

**Scenario 1:** 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.

**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;**

**v\_to\_balance NUMBER;**

**v\_insufficient\_funds EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(v\_insufficient\_funds, -20001);**

**BEGIN**

**-- Check if the from account has sufficient funds**

**SELECT balance INTO v\_from\_balance**

**FROM accounts**

**WHERE account\_id = p\_from\_account\_id;**

**IF v\_from\_balance < p\_amount THEN**

**RAISE v\_insufficient\_funds;**

**END IF;**

**-- Perform the transfer**

**UPDATE accounts**

**SET balance = balance - p\_amount**

**WHERE account\_id = p\_from\_account\_id;**

**UPDATE accounts**

**SET balance = balance + p\_amount**

**WHERE account\_id = p\_to\_account\_id;**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Transfer successful');**

**EXCEPTION**

**WHEN v\_insufficient\_funds THEN**

**ROLLBACK;**

**DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in account ' || p\_from\_account\_id);**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**Scenario 2:** 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.

**CREATE OR REPLACE PROCEDURE UpdateSalary(p\_employee\_id IN NUMBER,**

**p\_percentage IN NUMBER**

**)**

**AS**

**v\_current\_salary NUMBER;**

**v\_new\_salary NUMBER;**

**v\_employee\_not\_found EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(v\_employee\_not\_found, -2291);**

**BEGIN**

**-- Retrieve the current salary of the employee**

**SELECT salary INTO v\_current\_salary**

**FROM employees**

**WHERE employee\_id = p\_employee\_id;**

**-- Calculate the new salary**

**v\_new\_salary := v\_current\_salary \* (1 + p\_percentage / 100);**

**-- Update the salary**

**UPDATE employees**

**SET salary = v\_new\_salary**

**WHERE employee\_id = p\_employee\_id;**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully');**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**ROLLBACK;**

**DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID ' || p\_employee\_id || ' does not exist');**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**Scenario 3**: 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.

**CREATE OR REPLACE PROCEDURE AddNewCustomer(**

**p\_customer\_id IN NUMBER,**

**p\_name IN VARCHAR2,**

**p\_email IN VARCHAR2,**

**p\_phone IN VARCHAR2**

**)**

**AS**

**v\_duplicate\_customer EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(v\_duplicate\_customer, -1);**

**BEGIN**

**-- Check if a customer with the same ID already exists**

**IF EXISTS (SELECT 1 FROM customers WHERE customer\_id = p\_customer\_id) THEN**

**RAISE v\_duplicate\_customer;**

**END IF;**

**-- Insert the new customer**

**INSERT INTO customers (customer\_id, name, email, phone)**

**VALUES (p\_customer\_id, p\_name, p\_email, p\_phone);**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Customer added successfully');**

**EXCEPTION**

**WHEN v\_duplicate\_customer THEN**

**ROLLBACK;**

**DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_customer\_id || ' already exists');**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**Exercise 3: Stored Procedures**

**Scenario 1:** 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.

**CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest**

**AS**

**BEGIN**

**-- Update the balance of all savings accounts by applying an interest rate of 1%**

**UPDATE savings\_accounts**

**SET balance = balance \* 1.01;**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed successfully');**

**EXCEPTION**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**Scenario 2:** 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.

**CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(**

**p\_department\_id IN NUMBER,**

**p\_bonus\_percentage IN NUMBER**

**)**

**AS**

**BEGIN**

**-- Update the salary of employees in the given department by adding the bonus percentage**

**UPDATE employees**

**SET salary = salary \* (1 + p\_bonus\_percentage / 100)**

**WHERE department\_id = p\_department\_id;**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Employee bonuses updated successfully for department ' || p\_department\_id);**

**EXCEPTION**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**Scenario 3:** 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.

**CREATE OR REPLACE PROCEDURE TransferFunds(**

**p\_source\_account\_id IN NUMBER,**

**p\_target\_account\_id IN NUMBER,**

**p\_amount IN NUMBER**

**)**

**AS**

**v\_source\_balance NUMBER;**

**v\_target\_balance NUMBER;**

**BEGIN**

**-- Check if the source and target accounts exist**

**IF NOT EXISTS (SELECT 1 FROM accounts WHERE account\_id = p\_source\_account\_id) THEN**

**RAISE\_APPLICATION\_ERROR(-20001, 'Source account does not exist');**

**ELSIF NOT EXISTS (SELECT 1 FROM accounts WHERE account\_id = p\_target\_account\_id) THEN**

**RAISE\_APPLICATION\_ERROR(-20002, 'Target account does not exist');**

**END IF;**

**-- Check if the source account has sufficient balance**

**SELECT balance INTO v\_source\_balance FROM accounts WHERE account\_id = p\_source\_account\_id;**

**IF v\_source\_balance < p\_amount THEN**

**RAISE\_APPLICATION\_ERROR(-20003, 'Insufficient balance in source account');**

**END IF;**

**-- Debit the source account**

**UPDATE accounts**

**SET balance = balance - p\_amount**

**WHERE account\_id = p\_source\_account\_id;**

**-- Credit the target account**

**UPDATE accounts**

**SET balance = balance + p\_amount**

**WHERE account\_id = p\_target\_account\_id;**

**COMMIT;**

**DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully');**

**EXCEPTION**

**WHEN OTHERS THEN**

**ROLLBACK;**

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

**END;**

**Exercise 4: Functions**

**Scenario 1:** 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.

**CREATE OR REPLACE FUNCTION CalculateAge(**

**p\_date\_of\_birth IN DATE**

**)**

**RETURN NUMBER**

**AS**

**v\_age NUMBER;**

**BEGIN**

**v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_date\_of\_birth) / 12);**

**RETURN v\_age;**

**END;**

**Scenario 2:** 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.

**CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(**

**p\_loan\_amount IN NUMBER,**

**p\_interest\_rate IN NUMBER,**

**p\_loan\_duration IN NUMBER**

**)**

**RETURN NUMBER**

**AS**

**v\_monthly\_installment NUMBER;**

**BEGIN**

**v\_monthly\_installment := p\_loan\_amount \* (p\_interest\_rate / 1200) \* POWER(1 + (p\_interest\_rate / 1200), p\_loan\_duration \* 12) / (POWER(1 + (p\_interest\_rate / 1200), p\_loan\_duration \* 12) - 1);**

**RETURN v\_monthly\_installment;**

**END;**

**Scenario 3:** 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.

**CREATE OR REPLACE FUNCTION HasSufficientBalance(**

**p\_account\_id IN NUMBER,**

**p\_amount IN NUMBER**

**)**

**RETURN BOOLEAN**

**AS**

**v\_balance NUMBER;**

**BEGIN**

**SELECT balance**

**INTO v\_balance**

**FROM accounts**

**WHERE account\_id = p\_account\_id;**

**RETURN v\_balance >= p\_amount;**

**END;**

**Exercise 5: Triggers**

**Scenario 1:** 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.

**CREATE TRIGGER UpdateCustomerLastModified**

**BEFORE UPDATE ON Customers**

**FOR EACH ROW**

**BEGIN**

**:NEW.LastModified := SYSTIMESTAMP;**

**END;**

**Scenario 2:** 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.

**CREATE TRIGGER LogTransaction**

**AFTER INSERT ON Transactions**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO AuditLog (TransactionId, TransactionType, OldValue, NewValue, Timestamp)**

**VALUES (:NEW.TransactionId, 'INSERT', NULL, :NEW.Amount, SYSTIMESTAMP);**

**END;**

**Scenario 3:** 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.

**CREATE TRIGGER CheckTransactionRules**

**BEFORE INSERT ON Transactions**

**FOR EACH ROW**

**BEGIN**

**IF :NEW.Type = 'WITHDRAWAL' THEN**

**IF :NEW.Amount > (SELECT Balance FROM Accounts WHERE AccountId = :NEW.AccountId) THEN**

**RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal amount exceeds balance');**

**END IF;**

**ELSIF :NEW.Type = 'DEPOSIT' THEN**

**IF :NEW.Amount <= 0 THEN**

**RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be positive');**

**END IF;**

**END IF;**

**END;**

**Exercise 6: Cursors**

**Scenario 1:** 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.

**DECLARE**

**CURSOR cur\_transactions IS**

**SELECT t.AccountId, a.CustomerName, t.TransactionDate, t.Type, t.Amount**

**FROM Transactions t**

**JOIN Accounts a ON t.AccountId = a.AccountId**

**WHERE TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSTIMESTAMP, 'MM')**

**ORDER BY t.AccountId, t.TransactionDate;**

**cur\_rec cur\_transactions%ROWTYPE;**

**v\_customer\_name VARCHAR2(50);**

**v\_account\_id NUMBER;**

**v\_balance NUMBER := 0;**

**BEGIN**

**OPEN cur\_transactions;**

**LOOP**

**FETCH cur\_transactions INTO cur\_rec;**

**EXIT WHEN cur\_transactions%NOTFOUND;**

**IF cur\_rec.AccountId != v\_account\_id THEN**

**IF v\_account\_id IS NOT NULL THEN**

**DBMS\_OUTPUT.PUT\_LINE('Statement for ' || v\_customer\_name || ':');**

**DBMS\_OUTPUT.PUT\_LINE('---------------------------');**

**DBMS\_OUTPUT.PUT\_LINE('Date | Type | Amount');**

**DBMS\_OUTPUT.PUT\_LINE('---------------------------');**

**END IF;**

**v\_customer\_name := cur\_rec.CustomerName;**

**v\_account\_id := cur\_rec.AccountId;**

**v\_balance := 0;**

**END IF;**

**DBMS\_OUTPUT.PUT\_LINE(RPAD(TO\_CHAR(cur\_rec.TransactionDate, 'DD/MM/YYYY'),10)||**

**' | ' || cur\_rec.Type || ' | ' || TO\_CHAR(cur\_rec.Amount, '999999.99'));**

**v\_balance := v\_balance + CASE cur\_rec.Type WHEN 'DEPOSIT' THEN cur\_rec.Amount ELSE -cur\_rec.Amount**

**END;**

**IF cur\_transactions%ROWCOUNT = cur\_transactions%ROWCOUNT THEN**

**DBMS\_OUTPUT.PUT\_LINE('---------------------------');**

**DBMS\_OUTPUT.PUT\_LINE('Balance: ' || TO\_CHAR(v\_balance, '999999.99'));**

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

**END IF;**

**END LOOP;**

**CLOSE cur\_transactions;**

**END;**

**Scenario 2:** 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.

**DECLARE**

**CURSOR cur\_accounts IS**

**SELECT AccountId, Balance**

**FROM Accounts;**

**cur\_rec cur\_accounts%ROWTYPE;**

**v\_annual\_fee NUMBER := 25; -- Annual maintenance fee**

**BEGIN**

**OPEN cur\_accounts;**

**LOOP**

**FETCH cur\_accounts INTO cur\_rec;**

**EXIT WHEN cur\_accounts%NOTFOUND;**

**-- Check if the balance is sufficient to deduct the annual fee**

**IF cur\_rec.Balance >= v\_annual\_fee THEN**

**UPDATE Accounts**

**SET Balance = Balance - v\_annual\_fee**

**WHERE AccountId = cur\_rec.AccountId;**

**-- Insert a transaction record for the annual fee**

**INSERT INTO Transactions (AccountId, TransactionDate, Type, Amount)**

**VALUES (cur\_rec.AccountId, SYSTIMESTAMP, 'ANNUAL FEE', v\_annual\_fee);**

**DBMS\_OUTPUT.PUT\_LINE('Annual fee of ' || v\_annual\_fee || ' deducted from account ' || cur\_rec.AccountId);**

**ELSE**

**DBMS\_OUTPUT.PUT\_LINE('Insufficient balance in account ' || cur\_rec.AccountId || ' to deduct annual fee');**

**END IF;**

**END LOOP;**

**CLOSE cur\_accounts;**

**END;**

**Scenario 3:** 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.

**DECLARE**

**CURSOR cur\_loans IS**

**SELECT LoanId, InterestRate, LoanType**

**FROM Loans;**

**cur\_rec cur\_loans%ROWTYPE;**

**v\_new\_interest\_rate NUMBER;**

**BEGIN**

**OPEN cur\_loans;**

**LOOP**

**FETCH cur\_loans INTO cur\_rec;**

**EXIT WHEN cur\_loans%NOTFOUND;**

**-- Update interest rate based on the new policy**

**CASE cur\_rec.LoanType**

**WHEN 'PERSONAL' THEN**

**v\_new\_interest\_rate := 5.5;**

**WHEN 'HOME' THEN**

**v\_new\_interest\_rate := 4.25;**

**WHEN 'AUTO' THEN**

**v\_new\_interest\_rate := 3.75;**

**ELSE**

**v\_new\_interest\_rate := cur\_rec.InterestRate;**

**END CASE;**

**UPDATE Loans**

**SET InterestRate = v\_new\_interest\_rate**

**WHERE LoanId = cur\_rec.LoanId;**

**DBMS\_OUTPUT.PUT\_LINE('Interest rate for loan ' || cur\_rec.LoanId || ' updated to ' || v\_new\_interest\_rate);**

**END LOOP;**

**CLOSE cur\_loans;**

**END;**

**Exercise 7: Packages**

**Scenario 1:** 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.

**CREATE OR REPLACE PACKAGE CustomerManagement AS**

**-- Procedure to add a new customer**

**PROCEDURE AddCustomer(**

**p\_CustomerId IN Customers.CustomerId%TYPE,**

**p\_Name IN Customers.Name%TYPE,**

**p\_Address IN Customers.Address%TYPE,**

**p\_Phone IN Customers.Phone%TYPE,**

**p\_Email IN Customers.Email%TYPE**

**);**

**-- Procedure to update customer details**

**PROCEDURE UpdateCustomer(**

**p\_CustomerId IN Customers.CustomerId%TYPE,**

**p\_Name IN Customers.Name%TYPE DEFAULT NULL,**

**p\_Address IN Customers.Address%TYPE DEFAULT NULL,**

**p\_Phone IN Customers.Phone%TYPE DEFAULT NULL,**

**p\_Email IN Customers.Email%TYPE DEFAULT NULL**

**);**

**-- Function to get customer balance**

**FUNCTION GetCustomerBalance(p\_CustomerId IN Customers.CustomerId%TYPE)**

**RETURN NUMBER;**

**END CustomerManagement;**

**CREATE OR REPLACE PACKAGE BODY CustomerManagement AS**

**-- Procedure to add a new customer**

**PROCEDURE AddCustomer(**

**p\_CustomerId IN Customers.CustomerId%TYPE,**

**p\_Name IN Customers.Name%TYPE,**

**p\_Address IN Customers.Address%TYPE,**

**p\_Phone IN Customers.Phone%TYPE,**

**p\_Email IN Customers.Email%TYPE**

**)**

**AS**

**BEGIN**

**INSERT INTO Customers (CustomerId, Name, Address, Phone, Email)**

**VALUES (p\_CustomerId, p\_Name, p\_Address, p\_Phone, p\_Email);**

**END AddCustomer;**

**-- Procedure to update customer details**

**PROCEDURE UpdateCustomer(**

**p\_CustomerId IN Customers.CustomerId%TYPE,**

**p\_Name IN Customers.Name%TYPE DEFAULT NULL,**

**p\_Address IN Customers.Address%TYPE DEFAULT NULL,**

**p\_Phone IN Customers.Phone%TYPE DEFAULT NULL,**

**p\_Email IN Customers.Email%TYPE DEFAULT NULL**

**)**

**AS**

**BEGIN**

**UPDATE Customers**

**SET Name = NVL(p\_Name, Name),**

**Address = NVL(p\_Address, Address),**

**Phone = NVL(p\_Phone, Phone),**

**Email = NVL(p\_Email, Email)**

**WHERE CustomerId = p\_CustomerId;**

**END UpdateCustomer;**

**-- Function to get customer balance**

**FUNCTION GetCustomerBalance(p\_CustomerId IN Customers.CustomerId%TYPE)**

**RETURN NUMBER AS**

**v\_Balance NUMBER;**

**BEGIN**

**SELECT SUM(NVL(Balance, 0))**

**INTO v\_Balance**

**FROM Accounts**

**WHERE CustomerId = p\_CustomerId;**

**RETURN v\_Balance;**

**END GetCustomerBalance;**

**END CustomerManagement;**

**Scenario 2:** 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.

**CREATE OR REPLACE PACKAGE EmployeeManagement AS**

**-- Procedure to hire a new employee**

**PROCEDURE HireEmployee(**

**p\_EmployeeId IN Employees.EmployeeId%TYPE,**

**p\_Name IN Employees.Name%TYPE,**

**p\_Department IN Employees.Department%TYPE,**

**p\_JobTitle IN Employees.JobTitle%TYPE,**

**p\_HireDate IN Employees.HireDate%TYPE,**

**p\_Salary IN Employees.Salary%TYPE**

**);**

**-- Procedure to update employee details**

**PROCEDURE UpdateEmployee(**

**p\_EmployeeId IN Employees.EmployeeId%TYPE,**

**p\_Name IN Employees.Name%TYPE DEFAULT NULL,**

**p\_Department IN Employees.Department%TYPE DEFAULT NULL,**

**p\_JobTitle IN Employees.JobTitle%TYPE DEFAULT NULL,**

**p\_Salary IN Employees.Salary%TYPE DEFAULT NULL**

**);**

**-- Function to calculate annual salary**

**FUNCTION CalculateAnnualSalary(p\_Salary IN Employees.Salary%TYPE)**

**RETURN NUMBER;**

**END EmployeeManagement;**

**CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS**

**-- Procedure to hire a new employee**

**PROCEDURE HireEmployee(**

**p\_EmployeeId IN Employees.EmployeeId%TYPE,**

**p\_Name IN Employees.Name%TYPE,**

**p\_Department IN Employees.Department%TYPE,**

**p\_JobTitle IN Employees.JobTitle%TYPE,**

**p\_HireDate IN Employees.HireDate%TYPE,**

**p\_Salary IN Employees.Salary%TYPE**

**)AS**

**BEGIN**

**INSERT INTO Employees (EmployeeId, Name, Department, JobTitle, HireDate, Salary)**

**VALUES (p\_EmployeeId, p\_Name, p\_Department, p\_JobTitle, p\_HireDate, p\_Salary);**

**END HireEmployee;**

**-- Procedure to update employee details**

**PROCEDURE UpdateEmployee(**

**p\_EmployeeId IN Employees.EmployeeId%TYPE,**

**p\_Name IN Employees.Name%TYPE DEFAULT NULL,**

**p\_Department IN Employees.Department%TYPE DEFAULT NULL,**

**p\_JobTitle IN Employees.JobTitle%TYPE DEFAULT NULL,**

**p\_Salary IN Employees.Salary%TYPE DEFAULT NULL**

**) AS**

**BEGIN**

**UPDATE Employees**

**SET Name = NVL(p\_Name, Name),**

**Department = NVL(p\_Department, Department),**

**JobTitle = NVL(p\_JobTitle, JobTitle),**

**Salary = NVL(p\_Salary, Salary)**

**WHERE EmployeeId = p\_EmployeeId;**

**END UpdateEmployee;**

**-- Function to calculate annual salary**

**FUNCTION CalculateAnnualSalary(p\_Salary IN Employees.Salary%TYPE)**

**RETURN NUMBER AS**

**BEGIN**

**RETURN p\_Salary \* 12;**

**END CalculateAnnualSalary;**

**END EmployeeManagement;**

**Scenario 3:** 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.

**CREATE OR REPLACE PACKAGE AccountOperations AS**

**-- Procedure to open a new account**

**PROCEDURE OpenAccount(**

**p\_AccountId IN Accounts.AccountId%TYPE,**

**p\_CustomerId IN Accounts.CustomerId%TYPE,**

**p\_AccountType IN Accounts.AccountType%TYPE,**

**p\_OpeningBalance IN Accounts.Balance%TYPE**

**);**

**-- Procedure to close an account**

**PROCEDURE CloseAccount(p\_AccountId IN Accounts.AccountId%TYPE);**

**-- Function to get the total balance of a customer across all accounts**

**FUNCTION GetTotalBalance(p\_CustomerId IN Accounts.CustomerId%TYPE)**

**RETURN NUMBER;**

**END AccountOperations;**

**CREATE OR REPLACE PACKAGE BODY AccountOperations AS**

**-- Procedure to open a new account**

**PROCEDURE OpenAccount(**

**p\_AccountId IN Accounts.AccountId%TYPE,**

**p\_CustomerId IN Accounts.CustomerId%TYPE,**

**p\_AccountType IN Accounts.AccountType%TYPE,**

**p\_OpeningBalance IN Accounts.Balance%TYPE**

**) AS**

**BEGIN**

**INSERT INTO Accounts (AccountId, CustomerId, AccountType, Balance)**

**VALUES (p\_AccountId, p\_CustomerId, p\_AccountType, p\_OpeningBalance);**

**END OpenAccount;**

**-- Procedure to close an account**

**PROCEDURE CloseAccount(p\_AccountId IN Accounts.AccountId%TYPE) AS**

**BEGIN**

**UPDATE Accounts**

**SET Status = 'CLOSED'**

**WHERE AccountId = p\_AccountId;**

**END CloseAccount;**

**-- Function to get the total balance of a customer across all accounts**

**FUNCTION GetTotalBalance(p\_CustomerId IN Accounts.CustomerId%TYPE)**

**RETURN NUMBER AS**

**v\_TotalBalance NUMBER;**

**BEGIN**

**SELECT SUM(Balance)**

**INTO v\_TotalBalance**

**FROM Accounts**

**WHERE CustomerId = p\_CustomerId AND Status = 'ACTIVE';**

**RETURN v\_TotalBalance;**

**END GetTotalBalance;**

**END AccountOperations;**