**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.**

**Code:**

**BEGIN**

**FOR person IN (SELECT CustomerID, DOB FROM Customers) LOOP**

**IF MONTHS\_BETWEEN(SYSDATE, person.DOB) / 12 > 60 THEN**

**UPDATE Loans**

**SET InterestRate = InterestRate - 1**

**WHERE CustomerID = person.CustomerID;**

**END IF;**

**END LOOP;**

**END;**

**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.**

**Code:**

**BEGIN**

**FOR person IN (SELECT CustomerID, Balance FROM Customers WHERE Balance > 10000) LOOP**

**UPDATE Customers**

**SET IsVIP = TRUE**

**WHERE CustomerID = person.CustomerID;**

**END LOOP;**

**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.**

**Code:**

**BEGIN**

**FOR loan IN (**

**SELECT CustomerID, LoanID**

**FROM Loans**

**WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30**

**) LOOP**

**DBMS\_OUTPUT.PUT\_LINE(**

**'Alert: Loan ' || loan.LoanID ||**

**' for customer ' || loan.CustomerID ||**

**' is approaching its end date.'**

**);**

**END LOOP;**

**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.**

**Code:**

**CREATE OR REPLACE PROCEDURE ExecuteFundTransfer (**

**src\_account NUMBER,**

**dest\_account NUMBER,**

**transfer\_amount NUMBER**

**) AS**

**low\_balance EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(low\_balance, -20001);**

**current\_balance NUMBER;**

**BEGIN**

**SELECT Balance INTO current\_balance**

**FROM Accounts**

**WHERE AccountID = src\_account;**

**IF current\_balance < transfer\_amount THEN**

**RAISE low\_balance;**

**END IF;**

**UPDATE Accounts**

**SET Balance = Balance - transfer\_amount**

**WHERE AccountID = src\_account;**

**UPDATE Accounts**

**SET Balance = Balance + transfer\_amount**

**WHERE AccountID = dest\_account;**

**COMMIT;**

**EXCEPTION**

**WHEN low\_balance THEN**

**DBMS\_OUTPUT.PUT\_LINE('Insufficient funds in the source account.');**

**ROLLBACK;**

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('Unexpected error because of some issues: ' || SQLERRM);**

**ROLLBACK;**

**END ExecuteFundTransfer;**

**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.**

**Code:**

**CREATE OR REPLACE PROCEDURE AdjustEmployeeSalary (**

**emp\_id NUMBER,**

**pct\_increase NUMBER**

**) AS**

**not\_found EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(not\_found, -20002);**

**current\_salary NUMBER;**

**BEGIN**

**SELECT Salary INTO current\_salary**

**FROM Employees**

**WHERE EmployeeID = emp\_id;**

**UPDATE Employees**

**SET Salary = Salary + (Salary \* pct\_increase / 100)**

**WHERE EmployeeID = emp\_id;**

**COMMIT;**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**RAISE not\_found;**

**WHEN not\_found THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: Employee not found.');**

**ROLLBACK;**

**WHEN OTHERS THEN**

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

**ROLLBACK;**

**END AdjustEmployeeSalary;**

**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.**

**Code:**

**CREATE OR REPLACE PROCEDURE RegisterCustomer (**

**new\_customer\_id NUMBER,**

**new\_name VARCHAR2,**

**new\_dob DATE,**

**initial\_balance NUMBER**

**) AS**

**already\_exists EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(already\_exists, -20003);**

**customer\_count NUMBER;**

**BEGIN**

**SELECT COUNT(\*) INTO customer\_count**

**FROM Customers**

**WHERE CustomerID = new\_customer\_id;**

**IF customer\_count > 0 THEN RAISE already\_exists;**

**END IF;**

**INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)**

**VALUES (new\_customer\_id, new\_name, new\_dob, initial\_balance, SYSDATE);**

**COMMIT;**

**EXCEPTION**

**WHEN already\_exists THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: Customer already exists.');**

**ROLLBACK;**

**WHEN OTHERS THEN**

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

**ROLLBACK;**

**END RegisterCustomer;**

**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.**

**Code:**

**CREATE OR REPLACE PROCEDURE ApplyMonthlyInterest AS**

**BEGIN**

**FOR account IN (**

**SELECT AccountID, Balance**

**FROM Accounts**

**WHERE AccountType = 'Savings'**

**) LOOP**

**UPDATE Accounts**

**SET Balance = Balance + (Balance \* 0.01)**

**WHERE AccountID = account.AccountID;**

**END LOOP;**

**COMMIT;**

**END ApplyMonthlyInterest;**

**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.**

**Code:**

**CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (p\_department VARCHAR2, p\_bonus\_percentage NUMBER) AS**

**BEGIN**

**UPDATE Employees**

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

**WHERE Department = p\_department;**

**COMMIT;**

**END UpdateEmployeeBonus;**

**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.**

**Code:**

**CREATE OR REPLACE PROCEDURE ExecuteTransfer (**

**source\_account NUMBER,**

**target\_account NUMBER,**

**transfer\_amount NUMBER**

**) AS**

**funds\_shortage EXCEPTION;**

**PRAGMA EXCEPTION\_INIT(funds\_shortage, -20001);**

**current\_balance NUMBER;**

**BEGIN**

**SELECT Balance INTO current\_balance**

**FROM Accounts**

**WHERE AccountID = source\_account;**

**IF current\_balance < transfer\_amount THEN**

**RAISE funds\_shortage;**

**END IF;**

**UPDATE Accounts**

**SET Balance = Balance - transfer\_amount**

**WHERE AccountID = source\_account;**

**UPDATE Accounts**

**SET Balance = Balance + transfer\_amount**

**WHERE AccountID = target\_account;**

**COMMIT;**

**EXCEPTION**

**WHEN funds\_shortage THEN**

**DBMS\_OUTPUT.PUT\_LINE('Error: Not enough funds in the source account.');**

**ROLLBACK;**

**WHEN OTHERS THEN**

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

**ROLLBACK;**

**END ExecuteTransfer;**

**Exercise 4: Functions**

**Scenario 1: Calculate the age of customers for eligibility checks.**

**Question: Write a function CalculateAge that takes a customer&#39;s date of birth as input and returns their age in years.**

**Code:**

**CREATE OR REPLACE FUNCTION CalculateAge (p\_dob DATE) RETURN NUMBER IS**

**calage NUMBER;**

**BEGIN**

**calage := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);**

**RETURN calage;**

**END CalculateAge;**

**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.**

**Code:**

**CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (**

**p\_loan\_amount NUMBER,**

**p\_interest\_rate NUMBER,**

**p\_duration\_years NUMBER**

**) RETURN NUMBER IS**

**l\_monthly\_installment NUMBER;**

**l\_monthly\_rate NUMBER;**

**l\_num\_payments NUMBER;**

**BEGIN**

**IF p\_interest\_rate = 0 THEN**

**l\_monthly\_installment := p\_loan\_amount / (p\_duration\_years \* 12);**

**ELSE**

**l\_monthly\_rate := p\_interest\_rate / 1200;**

**l\_num\_payments := p\_duration\_years \* 12;**

**l\_monthly\_installment := p\_loan\_amount \* l\_monthly\_rate /**

**(1 - POWER(1 + l\_monthly\_rate, -l\_num\_payments));**

**END IF;**

**RETURN l\_monthly\_installment;**

**END CalculateMonthlyInstallment;**

**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.**

**Code:**

**CREATE OR REPLACE FUNCTION HasSufficientBalance (p\_account\_id NUMBER, p\_amount NUMBER) RETURN BOOLEAN IS**

**l\_balance NUMBER;**

**BEGIN**

**SELECT Balance INTO l\_balance FROM Accounts WHERE AccountID = p\_account\_id;**

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

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**RETURN FALSE;**

**END HasSufficientBalance;**

**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.**

**Code:**

**CREATE OR REPLACE TRIGGER UpdateCustomerLastModified**

**BEFORE UPDATE ON Customers**

**FOR EACH ROW**

**BEGIN**

**:NEW.LastModified := SYSDATE;**

**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.**

**Code:**

**CREATE OR REPLACE TRIGGER LogTransaction AFTER INSERT ON Transactions**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO AuditLog (TransactionID, AccountID, TransactionDate, Amount, TransactionType)**

**VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType);**

**END LogTransaction;**

**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.**

**Code:**

**CREATE OR REPLACE TRIGGER CheckTransactionRules**

**BEFORE INSERT ON Transactions**

**FOR EACH ROW**

**BEGIN**

**IF :NEW.TransactionType = 'Withdrawal' THEN**

**DECLARE l\_balance NUMBER;**

**BEGIN**

**SELECT Balance INTO l\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;**

**IF l\_balance < :NEW.Amount THEN**

**RAISE\_APPLICATION\_ERROR(-20000, 'Insufficient balance for withdrawal.');**

**END IF;**

**END;**

**ELSIF :NEW.TransactionType = 'Deposit' THEN**

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

**RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be positive.');**

**END IF;**

**END IF;**

**END CheckTransactionRules;**

**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.**

**Code:**

**BEGIN**

**FOR mon\_cust IN (SELECT CustomerID FROM Customers) LOOP**

**DBMS\_OUTPUT.PUT\_LINE('Statement for Customer ID: ' || mon\_cust.CustomerID);**

**FOR txn IN (**

**SELECT \***

**FROM Transactions**

**WHERE AccountID IN (**

**SELECT AccountID**

**FROM Accounts**

**WHERE CustomerID = mon\_cust.CustomerID**

**)**

**AND TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE)**

**) LOOP**

**DBMS\_OUTPUT.PUT\_LINE(**

**'Transaction ID: ' || txn.TransactionID ||**

**', Amount: ' || txn.Amount ||**

**', Type: ' || txn.TransactionType**

**);**

**END LOOP;**

**END LOOP;**

**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.**

**Code:**

**BEGIN**

**FOR acc IN (SELECT AccountID FROM Accounts) LOOP**

**UPDATE Accounts**

**SET Balance = Balance - 100**

**WHERE AccountID = acc.AccountID;**

**END LOOP;**

**COMMIT;**

**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.**

**Code:**

**BEGIN**

**FOR loan IN (SELECT LoanID FROM Loans) LOOP**

**UPDATE Loans**

**SET InterestRate = InterestRate + 0.5**

**WHERE LoanID = loan.LoanID;**

**END LOOP;**

**COMMIT;**

**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.**

**Code:**

**CREATE OR REPLACE PACKAGE CustomerManagement AS**

**PROCEDURE AddClient (**

**p\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_birthdate DATE,**

**p\_account\_balance NUMBER**

**);**

**PROCEDURE ModifyClient (**

**p\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_birthdate DATE,**

**p\_account\_balance NUMBER**

**);**

**FUNCTION RetrieveClientBalance (**

**p\_id NUMBER**

**) RETURN NUMBER;**

**END CustomerManagement;**

**CREATE OR REPLACE PACKAGE BODY CustomerManagement AS**

**PROCEDURE AddClient (**

**p\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_birthdate DATE,**

**p\_account\_balance NUMBER**

**) IS**

**BEGIN**

**INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)**

**VALUES (p\_id, p\_full\_name, p\_birthdate, p\_account\_balance, SYSDATE);**

**COMMIT;**

**END AddClient;**

**PROCEDURE ModifyClient (**

**p\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_birthdate DATE,**

**p\_account\_balance NUMBER**

**) IS**

**BEGIN**

**UPDATE Customers**

**SET Name = p\_full\_name,**

**DOB = p\_birthdate,**

**Balance = p\_account\_balance,**

**LastModified = SYSDATE**

**WHERE CustomerID = p\_id;**

**COMMIT;**

**END ModifyClient;**

**FUNCTION RetrieveClientBalance (**

**p\_id NUMBER**

**) RETURN NUMBER IS**

**l\_current\_balance NUMBER;**

**BEGIN**

**BEGIN**

**SELECT Balance INTO l\_current\_balance**

**FROM Customers**

**WHERE CustomerID = p\_id;**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**l\_current\_balance := NULL;**

**END;**

**RETURN l\_current\_balance;**

**END RetrieveClientBalance;**

**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.**

**Code:**

**CREATE OR REPLACE PACKAGE EmployeeManagement AS**

**PROCEDURE AddNewEmployee (**

**p\_emp\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_job\_title VARCHAR2,**

**p\_monthly\_salary NUMBER,**

**p\_dept VARCHAR2,**

**p\_start\_date DATE**

**);**

**PROCEDURE ModifyEmployeeDetails (**

**p\_emp\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_job\_title VARCHAR2,**

**p\_monthly\_salary NUMBER,**

**p\_dept VARCHAR2**

**);**

**FUNCTION ComputeAnnualSalary (**

**p\_emp\_id NUMBER**

**) RETURN NUMBER;**

**END EmployeeManagement;**

**CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS**

**PROCEDURE AddNewEmployee (**

**p\_emp\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_job\_title VARCHAR2,**

**p\_monthly\_salary NUMBER,**

**p\_dept VARCHAR2,**

**p\_start\_date DATE**

**) IS**

**BEGIN**

**INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)**

**VALUES (p\_emp\_id, p\_full\_name, p\_job\_title, p\_monthly\_salary, p\_dept, p\_start\_date);**

**COMMIT;**

**END AddNewEmployee;**

**PROCEDURE ModifyEmployeeDetails (**

**p\_emp\_id NUMBER,**

**p\_full\_name VARCHAR2,**

**p\_job\_title VARCHAR2,**

**p\_monthly\_salary NUMBER,**

**p\_dept VARCHAR2**

**) IS**

**BEGIN**

**UPDATE Employees**

**SET Name = p\_full\_name,**

**Position = p\_job\_title,**

**Salary = p\_monthly\_salary,**

**Department = p\_dept**

**WHERE EmployeeID = p\_emp\_id;**

**COMMIT;**

**END ModifyEmployeeDetails;**

**FUNCTION ComputeAnnualSalary (**

**p\_emp\_id NUMBER**

**) RETURN NUMBER IS**

**l\_monthly\_salary NUMBER;**

**BEGIN**

**SELECT Salary INTO l\_monthly\_salary**

**FROM Employees**

**WHERE EmployeeID = p\_emp\_id;**

**RETURN l\_monthly\_salary \* 12;**

**END ComputeAnnualSalary;**

**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.**

**Code:**

**CREATE OR REPLACE PACKAGE AccountManagement AS**

**PROCEDURE CreateAccount (**

**p\_acc\_id NUMBER,**

**p\_client\_id NUMBER,**

**p\_acc\_type VARCHAR2,**

**p\_initial\_balance NUMBER**

**);**

**PROCEDURE RemoveAccount (**

**p\_acc\_id NUMBER**

**);**

**FUNCTION CalculateTotalBalance (**

**p\_client\_id NUMBER**

**) RETURN NUMBER;**

**END AccountManagement;**

**/**

**CREATE OR REPLACE PACKAGE BODY AccountManagement AS**

**PROCEDURE CreateAccount (**

**p\_acc\_id NUMBER,**

**p\_client\_id NUMBER,**

**p\_acc\_type VARCHAR2,**

**p\_initial\_balance NUMBER**

**) IS**

**BEGIN**

**INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)**

**VALUES (p\_acc\_id, p\_client\_id, p\_acc\_type, p\_initial\_balance, SYSDATE);**

**COMMIT;**

**END CreateAccount;**

**PROCEDURE RemoveAccount (**

**p\_acc\_id NUMBER**

**) IS**

**BEGIN**

**DELETE FROM Accounts WHERE AccountID = p\_acc\_id;**

**COMMIT;**

**END RemoveAccount;**

**FUNCTION CalculateTotalBalance (**

**p\_client\_id NUMBER**

**) RETURN NUMBER IS**

**l\_balance\_sum NUMBER;**

**BEGIN**

**SELECT SUM(Balance) INTO l\_balance\_sum**

**FROM Accounts**

**WHERE CustomerID = p\_client\_id;**

**RETURN l\_balance\_sum;**

**END CalculateTotalBalance;**

**END AccountManagement;**

**/**