**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 cur\_customers IS
  + SELECT CustomerID, DOB
  + FROM Customers;
  + v\_customerID Customers.CustomerID%TYPE;
  + v\_dob Customers.DOB%TYPE;
  + v\_age NUMBER;
  + BEGIN
  + FOR rec IN cur\_customers LOOP
  + v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, rec.DOB) / 12);
  + IF v\_age > 60 THEN
  + UPDATE Loans
  + SET InterestRate = InterestRate - 1
  + WHERE CustomerID = rec.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.
  + DECLARE
  + CURSOR cur\_customers IS
  + SELECT CustomerID, DOB
  + FROM Customers;
  + v\_age NUMBER;
  + BEGIN
  + FOR rec IN cur\_customers LOOP
  + v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, rec.DOB) / 12);
  + IF v\_age > 60 THEN
  + UPDATE Loans
  + SET InterestRate = InterestRate - 1
  + WHERE CustomerID = rec.CustomerID;
  + END IF;
  + 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.
  + DECLARE
  + CURSOR cur\_loans IS
  + SELECT CustomerID, LoanID, EndDate
  + FROM Loans
  + WHERE EndDate <= SYSDATE + 30;
  + BEGIN
  + FOR rec IN cur\_loans LOOP
  + DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID || ' for Customer ' || rec.CustomerID || ' is due on ' || TO\_CHAR(rec.EndDate, 'YYYY-MM-DD'));
  + 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.
  + CREATE OR REPLACE PROCEDURE SafeTransferFunds(
  + p\_fromAccountID IN NUMBER,
  + p\_toAccountID IN NUMBER,
  + p\_amount IN NUMBER
  + ) IS
  + v\_fromBalance NUMBER;
  + v\_toBalance NUMBER;
  + BEGIN
  + BEGIN
  + SELECT Balance INTO v\_fromBalance FROM Accounts WHERE AccountID = p\_fromAccountID FOR UPDATE;
  + IF v\_fromBalance < p\_amount THEN
  + RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the source account.');
  + END IF;
  + SELECT Balance INTO v\_toBalance FROM Accounts WHERE AccountID = p\_toAccountID FOR UPDATE;
  + UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_fromAccountID;
  + UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_toAccountID;
  + COMMIT;
  + EXCEPTION
  + WHEN OTHERS THEN
  + ROLLBACK;
  + DBMS\_OUTPUT.PUT\_LINE('Error during fund transfer: ' || SQLERRM);
  + RAISE;
  + END;
  + END SafeTransferFunds;
  + /

**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\_employeeID IN NUMBER,
  + p\_percentage IN NUMBER
  + ) IS
  + BEGIN
  + UPDATE Employees
  + SET Salary = Salary \* (1 + p\_percentage / 100)
  + WHERE EmployeeID = p\_employeeID;
  + IF SQL%ROWCOUNT = 0 THEN
  + RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID does not exist.');
  + END IF;
  + COMMIT;
  + EXCEPTION
  + WHEN OTHERS THEN
  + ROLLBACK;
  + DBMS\_OUTPUT.PUT\_LINE('Error updating salary: ' || SQLERRM);
  + RAISE;
  + END UpdateSalary;
  + /

**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\_customerID 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\_customerID, p\_name, p\_dob, p\_balance, SYSDATE);
  + COMMIT;
  + EXCEPTION
  + WHEN DUP\_VAL\_ON\_INDEX THEN
  + DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_customerID || ' already exists.');
  + ROLLBACK;
  + WHEN OTHERS THEN
  + DBMS\_OUTPUT.PUT\_LINE('Error adding new customer: ' || SQLERRM);
  + ROLLBACK;
  + END AddNewCustomer;
  + /

**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 IS
  + BEGIN
  + UPDATE Accounts
  + SET Balance = Balance \* 1.01
  + WHERE AccountType = 'Savings';
  + COMMIT;
  + END ProcessMonthlyInterest;
  + /

**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 IN VARCHAR2,
  + p\_bonusPercentage IN NUMBER
  + ) IS
  + BEGIN
  + UPDATE Employees
  + SET Salary = Salary \* (1 + p\_bonusPercentage / 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.
  + CREATE OR REPLACE PROCEDURE TransferFunds(
  + p\_fromAccountID IN NUMBER,
  + p\_toAccountID IN NUMBER,
  + p\_amount IN NUMBER
  + ) IS
  + v\_fromBalance NUMBER;
  + BEGIN
  + BEGIN
  + SELECT Balance INTO v\_fromBalance FROM Accounts WHERE AccountID = p\_fromAccountID FOR UPDATE;
  + IF v\_fromBalance < p\_amount THEN
  + RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the source account.');
  + END IF;
  + UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_fromAccountID;
  + UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_toAccountID;
  + COMMIT;
  + EXCEPTION
  + WHEN OTHERS THEN
  + ROLLBACK;
  + DBMS\_OUTPUT.PUT\_LINE('Error during fund transfer: ' || SQLERRM);
  + RAISE;
  + END;
  + END TransferFunds;
  + /

**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\_dob DATE
  + ) RETURN NUMBER IS
  + v\_age NUMBER;
  + BEGIN
  + v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);
  + RETURN v\_age;
  + 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.
  + CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(
  + p\_loanAmount NUMBER,
  + p\_interestRate NUMBER,
  + p\_loanDurationYears NUMBER
  + ) RETURN NUMBER IS
  + v\_monthlyInstallment NUMBER;
  + v\_monthlyInterestRate NUMBER;
  + v\_totalMonths NUMBER;
  + BEGIN
  + v\_totalMonths := p\_loanDurationYears \* 12;
  + v\_monthlyInterestRate := p\_interestRate / 1200;
  + v\_monthlyInstallment := (p\_loanAmount \* v\_monthlyInterestRate) / (1 - POWER(1 + v\_monthlyInterestRate, -v\_totalMonths));
  + RETURN v\_monthlyInstallment;
  + 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.
  + CREATE OR REPLACE FUNCTION HasSufficientBalance(
  + p\_accountID NUMBER,
  + p\_amount NUMBER
  + ) RETURN BOOLEAN IS
  + v\_balance NUMBER;
  + BEGIN
  + SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_accountID;
  + RETURN v\_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.
  + CREATE OR REPLACE TRIGGER UpdateCustomerLastModified
  + BEFORE UPDATE ON Customers
  + FOR EACH ROW
  + BEGIN
  + :NEW.LastModified := SYSDATE;
  + END UpdateCustomerLastModified;
  + /

**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 OR REPLACE TRIGGER LogTransaction
  + AFTER INSERT ON Transactions
  + FOR EACH ROW
  + BEGIN
  + DBMS\_OUTPUT.PUT\_LINE('TransactionID: ' || :NEW.TransactionID || ', Action: INSERT, Date: ' || TO\_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS'));
  + 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.
  + CREATE OR REPLACE TRIGGER CheckTransactionRules
  + BEFORE INSERT ON Transactions
  + FOR EACH ROW
  + DECLARE
  + v\_balance NUMBER;
  + BEGIN
  + IF :NEW.TransactionType = 'Withdrawal' THEN
  + SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;
  + IF v\_balance < :NEW.Amount THEN
  + RAISE\_APPLICATION\_ERROR(-20003, 'Withdrawal amount exceeds balance.');
  + END IF;
  + ELSIF :NEW.TransactionType = 'Deposit' THEN
  + IF :NEW.Amount <= 0 THEN
  + RAISE\_APPLICATION\_ERROR(-20004, '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.
  + DECLARE
  + CURSOR cur\_transactions IS
  + SELECT a.CustomerID, t.TransactionID, t.TransactionDate, t.Amount, t.TransactionType
  + FROM Transactions t
  + JOIN Accounts a ON t.AccountID = a.AccountID
  + WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)
  + AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);
  + v\_customerID Customers.CustomerID%TYPE;
  + v\_statement VARCHAR2(4000);
  + BEGIN
  + FOR rec IN cur\_transactions LOOP
  + v\_customerID := rec.CustomerID;
  + v\_statement := 'Customer ID: ' || v\_customerID ||
  + ', Transaction ID: ' || rec.TransactionID ||
  + ', Date: ' || TO\_CHAR(rec.TransactionDate, 'YYYY-MM-DD') ||
  + ', Amount: ' || rec.Amount ||
  + ', Type: ' || rec.TransactionType;
  + DBMS\_OUTPUT.PUT\_LINE(v\_statement);
  + 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.
  + DECLARE
  + CURSOR cur\_accounts IS
  + SELECT AccountID, Balance
  + FROM Accounts;
  + v\_fee NUMBER := 50; -- Example annual fee amount
  + BEGIN
  + FOR rec IN cur\_accounts LOOP
  + UPDATE Accounts
  + SET Balance = Balance - v\_fee
  + WHERE AccountID = rec.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.
  + DECLARE
  + CURSOR cur\_loans IS
  + SELECT LoanID, InterestRate
  + FROM Loans;
  + v\_newInterestRate NUMBER;
  + BEGIN
  + FOR rec IN cur\_loans LOOP
  + v\_newInterestRate := rec.InterestRate + 0.5; -- Example policy: increase by 0.5%
  + UPDATE Loans
  + SET InterestRate = v\_newInterestRate
  + WHERE LoanID = rec.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.
  + CREATE OR REPLACE PACKAGE CustomerManagement AS
  + PROCEDURE AddNewCustomer(
  + p\_customerID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_dob IN DATE,
  + p\_balance IN NUMBER
  + );
  + PROCEDURE UpdateCustomer(
  + p\_customerID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_dob IN DATE,
  + p\_balance IN NUMBER
  + );
  + FUNCTION GetCustomerBalance(
  + p\_customerID IN NUMBER
  + ) RETURN NUMBER;
  + END CustomerManagement;
  + /
  + CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
  + PROCEDURE AddNewCustomer(
  + p\_customerID 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\_customerID, p\_name, p\_dob, p\_balance, SYSDATE);
  + COMMIT;
  + END AddNewCustomer;
  + PROCEDURE UpdateCustomer(
  + p\_customerID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_dob IN DATE,
  + p\_balance IN NUMBER
  + ) IS
  + BEGIN
  + UPDATE Customers
  + SET Name = p\_name,
  + DOB = p\_dob,
  + Balance = p\_balance,
  + LastModified = SYSDATE
  + WHERE CustomerID = p\_customerID;
  + COMMIT;
  + END UpdateCustomer;
  + FUNCTION GetCustomerBalance(
  + p\_customerID IN NUMBER
  + ) RETURN NUMBER IS
  + v\_balance NUMBER;
  + BEGIN
  + SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_customerID;
  + RETURN v\_balance;
  + EXCEPTION
  + WHEN NO\_DATA\_FOUND THEN
  + RETURN NULL;
  + 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 HireEmployee(
  + p\_employeeID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_position IN VARCHAR2,
  + p\_salary IN NUMBER,
  + p\_department IN VARCHAR2,
  + p\_hireDate IN DATE
  + );
  + PROCEDURE UpdateEmployee(
  + p\_employeeID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_position IN VARCHAR2,
  + p\_salary IN NUMBER,
  + p\_department IN VARCHAR2
  + );
  + FUNCTION CalculateAnnualSalary(
  + p\_employeeID IN NUMBER
  + ) RETURN NUMBER;
  + END EmployeeManagement;
  + /
  + CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
  + PROCEDURE HireEmployee(
  + p\_employeeID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_position IN VARCHAR2,
  + p\_salary IN NUMBER,
  + p\_department IN VARCHAR2,
  + p\_hireDate IN DATE
  + ) IS
  + BEGIN
  + INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
  + VALUES (p\_employeeID, p\_name, p\_position, p\_salary, p\_department, p\_hireDate);
  + COMMIT;
  + END HireEmployee;
  + PROCEDURE UpdateEmployee(
  + p\_employeeID IN NUMBER,
  + p\_name IN VARCHAR2,
  + p\_position IN VARCHAR2,
  + p\_salary IN NUMBER,
  + p\_department IN VARCHAR2
  + ) IS
  + BEGIN
  + UPDATE Employees
  + SET Name = p\_name,
  + Position = p\_position,
  + Salary = p\_salary,
  + Department = p\_department
  + WHERE EmployeeID = p\_employeeID;
  + COMMIT;
  + END UpdateEmployee;
  + FUNCTION CalculateAnnualSalary(
  + p\_employeeID IN NUMBER
  + ) RETURN NUMBER IS
  + v\_salary NUMBER;
  + BEGIN
  + SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_employeeID;
  + RETURN v\_salary \* 12;
  + EXCEPTION
  + WHEN NO\_DATA\_FOUND THEN
  + RETURN NULL;
  + 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 OpenAccount(
  + p\_accountID IN NUMBER,
  + p\_customerID IN NUMBER,
  + p\_accountType IN VARCHAR2,
  + p\_balance IN NUMBER
  + );
  + PROCEDURE CloseAccount(
  + p\_accountID IN NUMBER
  + );
  + FUNCTION GetTotalBalance(
  + p\_customerID IN NUMBER
  + ) RETURN NUMBER;
  + END AccountOperations;
  + /
  + CREATE OR REPLACE PACKAGE BODY AccountOperations AS
  + PROCEDURE OpenAccount(
  + p\_accountID IN NUMBER,
  + p\_customerID IN NUMBER,
  + p\_accountType IN VARCHAR2,
  + p\_balance IN NUMBER
  + ) IS
  + BEGIN
  + INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
  + VALUES (p\_accountID, p\_customerID, p\_accountType, p\_balance, SYSDATE);
  + COMMIT;
  + END OpenAccount;
  + PROCEDURE CloseAccount(
  + p\_accountID IN NUMBER
  + ) IS
  + BEGIN
  + DELETE FROM Accounts WHERE AccountID = p\_accountID;
  + COMMIT;
  + END CloseAccount;
  + FUNCTION GetTotalBalance(
  + p\_customerID IN NUMBER
  + ) RETURN NUMBER IS
  + v\_totalBalance NUMBER;
  + BEGIN
  + SELECT SUM(Balance) INTO v\_totalBalance FROM Accounts WHERE CustomerID = p\_customerID;
  + RETURN v\_totalBalance;
  + EXCEPTION
  + WHEN NO\_DATA\_FOUND THEN
  + RETURN 0;
  + END GetTotalBalance;
  + END AccountOperations;
  + /