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.

```
SQL> SET SERVEROUT ON;
SQL> DECLARE
  2  age NUMBER;
  3  BEGIN
  4  FOR rec IN (SELECT CustomerID, DOB, Balance FROM Customers) LOOP
  5  age := FLOOR(MONTHS_BETWEEN(SYSDATE, rec.DOB) / 12);
  6  IF age > 60 THEN
  7  UPDATE Customers
  8  SET Balance = Balance * 0.99,
  9  LastModified = SYSDATE
 10  WHERE CustomerID = rec.CustomerID;
 11  DBMS_OUTPUT.PUT_LINE('Applied discount to Customer ID: ' || rec.CustomerID);
 12  END IF;
 13  END LOOP;
 14  COMMIT;
 15  END;
 16  /
```

```
Applied 1% discount to CustomerID: 3, New Balance: 1980

PL/SQL procedure successfully completed.

SQL> SELECT * FROM Customers;

CUSTOMERID
----------
NAME
--------------------------------------------------------------------------------
DOB          BALANCE LASTMODIF
--------- ---------- ---------
         1
John Doe
15-MAY-85       1000 27-JUN-25

         2
Jane Smith
20-JUL-90       1500 27-JUN-25

CUSTOMERID
----------
NAME
--------------------------------------------------------------------------------
DOB          BALANCE LASTMODIF
--------- ---------- ---------

         3
Robert Brown
01-FEB-50       1980 27-JUN-25
```

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.

```
SQL> ALTER TABLE Customers ADD isVIP VARCHAR(5) DEFAULT 'FALSE';

Table altered.
```

```
SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
  2  FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP
  3  IF rec.Balance > 10000 THEN
  4  UPDATE Customers
  5  SET IsVIP = 'TRUE'
  6  WHERE CustomerID = rec.CustomerID;
  7  DBMS_OUTPUT.PUT_LINE('CustomerID ' || rec.CustomerID || 'got promoted to VIP.');
  8  END IF;
  9  END LOOP;
 10  COMMIT;
 11  END;
 12  /

PL/SQL procedure successfully completed.
```

```
SQL> SELECT CustomerID, Name, Balance, IsVIP FROM Customers WHERE IsVIP = 'TRUE';

no rows selected
```

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.

```
SQL> ALTER TABLE Customers ADD Loan_Due_Date DATE;

Table altered.
```

```
SQL> UPDATE Customers SET Loan_Due_Date = SYSDATE + 10 WHERE CustomerID = 1;

1 row updated.

SQL> UPDATE Customers SET Loan_Due_Date = SYSDATE + 35 WHERE CustomerID = 2;

1 row updated.

SQL> UPDATE Customers SET Loan_Due_Date = SYSDATE + 5 WHERE CustomerID = 3;

1 row updated.
```

```
SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
  2  FOR rec IN (
  3  SELECT CustomerID, Name, Loan_Due_Date
  4  FROM Customers
  5  WHERE Loan_Due_Date BETWEEN SYSDATE AND SYSDATE + 30
  6  ) LOOP
  7  DBMS_OUTPUT.PUT_LINE('Remainder: Loan for Customer "' || rec.Name || '" (ID: ' || rec.CustomerID || ') is due on ' || TO_CHAR(rec.Loan_Due_Date, 'DD-MO
N-YYYY')
  8  );
  9  END LOOP;
 10  END;
 11  /
Remainder: Loan for Customer "John Doe" (ID: 1) is due on 07-JUL-2025
Remainder: Loan for Customer "Robert Brown" (ID: 3) is due on 02-JUL-2025

PL/SQL procedure successfully completed.
```

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.

```
SQL> CREATE OR REPLACE PROCEDURE SafeTransferFunds(
  2  fromaccountid IN NUMBER,
  3  toaccountid IN NUMBER,
  4  amount IN NUMBER
  5  ) IS
  6  frombalance NUMBER;
  7  BEGIN
  8  SELECT Balance INTO frombalance
  9  FROM Accounts
 10  WHERE AccountID = fromaccountid
 11  FOR UPDATE;
 12  IF frombalance < amount THEN
 13  RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds in source account.');
 14  END IF;
 15  UPDATE Accounts
 16  SET Balance = Balance - amount,
 17  LastModified = SYSDATE
 18  WHERE AccountID = fromaccountid;
 19  UPDATE Accounts
 20  SET Balance = Balance + amount,
 21  LastModified = SYSDATE
 22  WHERE AccountID = toaccountid;
 23  COMMIT;
 24  DBMS_OUTPUT.PUT_LINE('Transfer successful from Account ' || fromaccountid || ' to Account ' || toaccountid || ' of amount ' || amount);
 25  EXCEPTION
 26  WHEN OTHERS THEN
 27  ROLLBACK;
 28  INSERT INTO TransferErrors(ErrorMessage)
 29  VALUES ('Error during fund transfer: ' || SQLERRM);
 30  DBMS_OUTPUT.PUT_LINE('Transfer failed: ' ||SQLERRM);
 31  END;
 32  /
```

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.

```
SQL> CREATE OR REPLACE PROCEDURE UpdateSalary (
  2      p_EmployeeID IN NUMBER,
  3      p_Percent    IN NUMBER
  4  ) IS
  5      v_Salary NUMBER;
  6  BEGIN
  7      SELECT Salary INTO v_Salary
  8      FROM Employees
  9      WHERE EmployeeID = p_EmployeeID;
 10
 11      UPDATE Employees
 12      SET Salary = Salary + (Salary * p_Percent / 100)
 13      WHERE EmployeeID = p_EmployeeID;
 14
 15      DBMS_OUTPUT.PUT_LINE('Salary updated successfully for EmployeeID ' || p_EmployeeID);
 16
 17  EXCEPTION
 18      WHEN NO_DATA_FOUND THEN
 19          INSERT INTO SalaryUpdateErrors(EmployeeID, ErrorMsg)
 20          VALUES (p_EmployeeID, 'Employee does not exist.');
 21          DBMS_OUTPUT.PUT_LINE('Error: Employee not found. Logged in SalaryUpdateErrors.');
 22
 23      WHEN OTHERS THEN
 24          INSERT INTO SalaryUpdateErrors(EmployeeID, ErrorMsg)
 25          VALUES (p_EmployeeID, 'Unexpected error: ' || SQLERRM);
 26          DBMS_OUTPUT.PUT_LINE('Unexpected error occurred. Logged in SalaryUpdateErrors.');
 27  END;
 28  /
```

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.

```
SQL> CREATE OR REPLACE PROCEDURE AddNewCustomer (
  2      p_CustomerID IN NUMBER,
  3      p_Name       IN VARCHAR2,
  4      p_DOB        IN DATE,
  5      p_Balance    IN NUMBER
  6  ) IS
  7  BEGIN
  8      INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
  9      VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
 10
 11      DBMS_OUTPUT.PUT_LINE('Customer added successfully with ID: ' || p_CustomerID);
 12
 13  EXCEPTION
 14      WHEN DUP_VAL_ON_INDEX THEN
 15          INSERT INTO CustomerErrors (CustomerID, ErrorMessage)
 16          VALUES (p_CustomerID, 'Duplicate CustomerID. Insertion prevented.');
 17          DBMS_OUTPUT.PUT_LINE('Error: Duplicate CustomerID. Logged in CustomerErrors.');
 18
 19      WHEN OTHERS THEN
 20          INSERT INTO CustomerErrors (CustomerID, ErrorMessage)
 21          VALUES (p_CustomerID, 'Unexpected error: ' || SQLERRM);
 22          DBMS_OUTPUT.PUT_LINE('Unexpected error occurred. Logged in CustomerErrors.');
 23  END;
 24  /
```

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.

```
SQL> CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
  2  BEGIN
  3  UPDATE Accounts
  4  SET Balance = Balance + (Balance * 0.01),
  5  LastModified = SYSDATE
  6  WHERE AccountType = 'Savings';
  7  COMMIT;
  8  DBMS_OUTPUT.PUT_LINE('Monthly interest processed for all savings accounts.');
  9  EXCEPTION
 10  WHEN OTHERS THEN
 11  ROLLBACK;
 12  DBMS_OUTPUT.PUT_LINE('Error during interest processing: ' || SQLERRM);
 13  END;
 14  /

Procedure created.

SQL> EXEC ProcessMonthlyInterest;

PL/SQL procedure successfully completed.
```

```
SQL> SELECT AccountID, AccountType, Balance FROM Accounts WHERE AccountType = 'Savings';

 ACCOUNTID ACCOUNTTYPE            BALANCE
---------- -------------------- ----------
         1 Savings                    1010
```

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.

```
SQL> CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (
  2  department IN VARCHAR2,
  3  bonuspercent IN NUMBER
  4  ) IS
  5  affectedcount NUMBER;
  6  BEGIN
  7  UPDATE Employees
  8  SET Salary = Salary + (Salary * bonuspercent / 100)
  9  WHERE Department = department;
 10  affectedcount := SQL%ROWCOUNT;
 11  COMMIT;
 12  DBMS_OUTPUT.PUT_LINE('Bonus of ' || bonuspercent || '% applied to ' || affectedcount || ' employees in department "' || department || '".');
 13  EXCEPTION
 14  WHEN OTHERS THEN
 15  ROLLBACK;
 16  DBMS_OUTPUT.PUT_LINE('Error applying bonus: ' || SQLERRM);
 17  END;
 18  /

Procedure created.
```

```
SQL> SELECT EmployeeID, Name, Department, Salary FROM Employees WHERE Department = 'IT';

EMPLOYEEID
----------
NAME
--------------------------------------------------------------------------------
DEPARTMENT                                           SALARY
-------------------------------------------------- ----------
         2
Bob Brown
IT                                                    60000
```

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.

```
SQL> CREATE OR REPLACE PROCEDURE TransferFunds(
  2  fromaccountid IN NUMER,
  3  toaccountid IN NUMBER,
  4  amount IN NUMBER
  5  ) IS
  6  frombalance NUMBER;
  7  BEGIN
  8  SELECT Balance INTO frombalance
  9  FROM Accounts
 10  WHERE AccountID = fromaccountid
 11  FOR UPDATE;
 12  IF frombalance < amount THEN
 13  RAISE_APPLICATION_ERROR(-20001, 'Insufficient balance in source account.');
 14  END IF;
 15  UPDATE Accounts
 16  SET Balance = Balance - amount,
 17  LastModified = SYSDATE
 18  WHERE AccountID = fromaccountid;
 19  UPDATE Accounts
 20  SET Balance = Balance + amount,
 21  LastModified = SYSDATE
 22  WHERE AccountID = toaccountid;
 23  COMMIT;
 24  DBMS_OUTPUT.PUT_LINE('Transfer successful from Account' || fromaccountid || ' to Account ' || toaccountid || ' of amount' || amount);
 25  EXCEPTION
 26  WHEN NO_DATA_FOUND THEN
 27  ROLLBACK;
 28  DBMS_OUTPUT.PUT_LINE('Error: One or both accounts do not exists.');
 29  WHEN OTHERS THEN
 30  ROLLBACK;
 31  DBMS_OUTPUT.PUT_LINE('Transfer failed: ' || SQLERRM);
 32  END;
 33  /
```

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.

```
SQL> CREATE OR REPLACE FUNCTION CalculateAge(
  2  DOB IN DATE
  3  ) RETURN NUMBER IS
  4  Age NUMBER;
  5  BEGIN
  6  Age := TRUNC(MONTHS_BETWEEN(SYSDATE, DOB) / 12);
  7  RETURN Age;
  8  END;
  9  /

Function created.
```

```
SQL> SELECT Name, DOB, CalculateAge(DOB) AS Age FROM Customers;

NAME
--------------------------------------------------------------------------------
DOB              AGE
--------- ----------
John Doe
15-MAY-85         40

Jane Smith
20-JUL-90         34

Robert Brown
01-FEB-50         75
```

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.

```
SQL> CREATE OR REPLACE FUNCTION CaculateMonthlyInstallment(
  2  loanamount IN NUMBER,
  3  annualrate IN NUMBER,
  4  durationyears IN NUMBER
  5  ) RETURN NUMBER IS
  6
  7  monthlyrate NUMBER;
  8  totalmonths NUMBER;
  9  emi NUMBER;
 10  BEGIN
 11  monthlyrate := annualrate /12 / 100;
 12  totalmonths := durationyears * 12;
 13  emi := (loanamount * monthlyrate * POWER(1 + monthlyrate, totalmonths)) / (POWER(1 + monthlyrate, totalmonths) - 1);
 14  RETURN ROUND(emi, 2);
 15  EXCEPTION
 16  WHEN ZERO_DIVIDE THEN
 17  RETURN 0;
 18  WHEN OTHERS THEN
 19  RETURN -1;
 20  END;
 21  /

Function created.
```

```
SQL> SELECT CaculateMonthlyInstallment(500000, 7.5, 5) AS EMI FROM dual;

       EMI
----------
  10018.97
```

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.

```
SQL> CREATE OR REPLACE FUNCTION HasSufficientBalance(
  2  accountid IN NUMBER,
  3  amount IN NUMBER
  4  ) RETURN BOOLEAN IS
  5  balance NUMBER;
  6  BEGIN
  7  SELECT Balance INTO balance
  8  FROM Accounts
  9  WHERE AccountID = accountid;
 10  RETURN balance >= amount;
 11  EXCEPTION
 12  WHEN NO_DATA_FOUND THEN
 13  RETURN FALSE;
 14  WHEN OTHERS THEN
 15  RETURN FALSE;
 16  END;
 17  /

Function created.
```

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.

```
SQL> CREATE OR REPLACE TRIGGER UpdateCustomerLastModified
  2  BEFORE UPDATE ON Customers
  3  FOR EACH ROW
  4  BEGIN
  5  :NEW.LastModified := SYSDATE;
  6  END;
  7  /
```

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.

```
SQL> CREATE TABLE AuditLog (
  2      AuditID         NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  3      TransactionID   NUMBER,
  4      AccountID       NUMBER,
  5      Action          VARCHAR2(50),
  6      LogDate         DATE DEFAULT SYSDATE
  7  );

Table created.
```

```
SQL> CREATE OR REPLACE TRIGGER LogTransaction
  2  AFTER INSERT ON Transactions
  3  FOR EACH ROW
  4  BEGIN
  5      INSERT INTO AuditLog (TransactionID, AccountID, Action, LogDate)
  6      VALUES (:NEW.TransactionID, :NEW.AccountID, 'INSERTED TRANSACTION', SYSDATE);
  7  END;
  8  /
```

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.

```
SQL> CREATE OR REPLACE TRIGGER CheckTransactionRules
  2  BEFORE INSERT ON Transactions
  3  FOR EACH ROW
  4  DECLARE
  5      v_Balance NUMBER;
  6  BEGIN
  7      SELECT Balance INTO v_Balance
  8      FROM Accounts
  9      WHERE AccountID = :NEW.AccountID;
 10
 11      IF UPPER(:NEW.TransactionType) = 'DEPOSIT' AND :NEW.Amount <= 0 THEN
 12          RAISE_APPLICATION_ERROR(-20001, 'Deposit amount must be greater than 0.');
 13      END IF;
 14
 15      IF UPPER(:NEW.TransactionType) = 'WITHDRAWAL' AND :NEW.Amount > v_Balance THEN
 16          RAISE_APPLICATION_ERROR(-20002, 'Insufficient balance for withdrawal.');
 17      END IF;
 18  END;
 19  /
```

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.

```
SQL> SET SERVEROUTPUT ON;
SQL>
SQL> DECLARE
  2      -- Cursor to fetch current month's transactions with customer info
  3      CURSOR cur_monthly IS
  4          SELECT c.CustomerID, c.Name, t.AccountID, t.Amount, t.TransactionType, t.TransactionDate
  5          FROM Customers c
  6          JOIN Accounts a ON c.CustomerID = a.CustomerID
  7          JOIN Transactions t ON a.AccountID = t.AccountID
  8          WHERE TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM')
  9          ORDER BY c.CustomerID, t.TransactionDate;
 10
 11      -- Record variable for cursor
 12      rec cur_monthly%ROWTYPE;
 13
 14      -- To track current customer
 15      v_last_customer_id Customers.CustomerID%TYPE := NULL;
 16  BEGIN
 17      OPEN cur_monthly;
 18
 19      LOOP
 20          FETCH cur_monthly INTO rec;
 21          EXIT WHEN cur_monthly%NOTFOUND;
 22
 23          -- Print customer header only when customer changes
 24          IF v_last_customer_id IS NULL OR v_last_customer_id != rec.CustomerID THEN
 25              DBMS_OUTPUT.PUT_LINE('-----------------------------------------------');
 26              DBMS_OUTPUT.PUT_LINE('Monthly Statement for Customer: ' || rec.Name || ' (ID: ' || rec.CustomerID || ')');
 27              DBMS_OUTPUT.PUT_LINE('Date        | Type       | Amount | Account');
 28              DBMS_OUTPUT.PUT_LINE('-----------------------------------------------');
 29              v_last_customer_id := rec.CustomerID;
 30          END IF;
 31
 32          -- Print transaction line
 33          DBMS_OUTPUT.PUT_LINE(TO_CHAR(rec.TransactionDate, 'DD-MON-YYYY') || ' | ' ||
 34                               RPAD(rec.TransactionType, 10) || ' | ' ||
 35                               LPAD(TO_CHAR(rec.Amount, '99999.99'), 7) || ' | ' ||
 36                               rec.AccountID);
 37      END LOOP;
 38
 39      CLOSE cur_monthly;
 40  END;
```

```
-----------------------------------------------
Monthly Statement for Customer: John Doe (ID: 1)
Date        | Type       | Amount | Account
-----------------------------------------------
28-JUN-2025 | Deposit    |    200. | 1
-----------------------------------------------
Monthly Statement for Customer: Jane Smith (ID: 2)
Date        | Type       | Amount | Account
-----------------------------------------------
28-JUN-2025 | Withdrawal |    300. | 2
```

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.

```
SQL> SET SERVEROUTPUT ON;
SQL>
SQL> DECLARE
  2      -- Annual maintenance fee to be deducted
  3      annual_fee CONSTANT NUMBER := 200;
  4
  5      -- Cursor to fetch all accounts
  6      CURSOR account_cursor IS
  7          SELECT AccountID, Balance
  8          FROM Accounts;
  9
 10      -- Variable to hold each account record
 11      acct_rec account_cursor%ROWTYPE;
 12  BEGIN
 13      OPEN account_cursor;
 14
 15      LOOP
 16          FETCH account_cursor INTO acct_rec;
 17          EXIT WHEN account_cursor%NOTFOUND;
 18
 19          -- Skip deduction if balance is less than the fee
 20          IF acct_rec.Balance >= annual_fee THEN
 21              UPDATE Accounts
 22              SET Balance = Balance - annual_fee,
 23                  LastModified = SYSDATE
 24              WHERE AccountID = acct_rec.AccountID;
 25
 26              DBMS_OUTPUT.PUT_LINE('Annual fee applied to AccountID: ' || acct_rec.AccountID);
 27          ELSE
 28              DBMS_OUTPUT.PUT_LINE('Skipped AccountID: ' || acct_rec.AccountID || ' due to insufficient balance.');
 29          END IF;
 30      END LOOP;
 31
 32      CLOSE account_cursor;
 33  END;
 34  /
```

```
Annual fee applied to AccountID: 1
Annual fee applied to AccountID: 2
```

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.

```
SQL> SET SERVEROUTPUT ON;
SQL>
SQL> DECLARE
  2      CURSOR loan_cursor IS
  3          SELECT LoanID, LoanType, InterestRate
  4          FROM Loans;
  5
  6      loan_rec loan_cursor%ROWTYPE;
  7      new_rate NUMBER;
  8  BEGIN
  9      OPEN loan_cursor;
 10
 11      LOOP
 12          FETCH loan_cursor INTO loan_rec;
 13          EXIT WHEN loan_cursor%NOTFOUND;
 14
 15          CASE UPPER(loan_rec.LoanType)
 16              WHEN 'HOME' THEN
 17                  new_rate := 6.5;
 18              WHEN 'AUTO' THEN
 19                  new_rate := 7.0;
 20              WHEN 'PERSONAL' THEN
 21                  new_rate := 9.5;
 22              ELSE
 23                  new_rate := loan_rec.InterestRate;
 24          END CASE;
 25
 26          IF loan_rec.InterestRate != new_rate THEN
 27              UPDATE Loans
 28              SET InterestRate = new_rate
 29              WHERE LoanID = loan_rec.LoanID;
 30
 31              DBMS_OUTPUT.PUT_LINE('Updated LoanID ' || loan_rec.LoanID ||
 32                                   ' from ' || loan_rec.InterestRate || '% to ' || new_rate || '%.');
 33          ELSE
 34              DBMS_OUTPUT.PUT_LINE('LoanID ' || loan_rec.LoanID || ' already has correct rate.');
 35          END IF;
 36      END LOOP;
 37
 38      CLOSE loan_cursor;
 39  END;
 40  /
```

```
Updated LoanID 1 from 5% to 6.5%.
```

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.

```
SQL> CREATE OR REPLACE PACKAGE CustomerManagement IS
  2      PROCEDURE AddCustomer(
  3          p_CustomerID   IN NUMBER,
  4          p_Name         IN VARCHAR2,
  5          p_DOB          IN DATE,
  6          p_Balance      IN NUMBER
  7      );
  8
  9      PROCEDURE UpdateCustomer(
 10          p_CustomerID   IN NUMBER,
 11          p_Name         IN VARCHAR2,
 12          p_DOB          IN DATE
 13      );
 14
 15      FUNCTION GetCustomerBalance(
 16          p_CustomerID   IN NUMBER
 17      ) RETURN NUMBER;
 18  END CustomerManagement;
 19  /

Package created.
```

```
SQL> CREATE OR REPLACE PACKAGE BODY CustomerManagement IS
  2
  3      PROCEDURE AddCustomer(
  4          p_CustomerID   IN NUMBER,
  5          p_Name         IN VARCHAR2,
  6          p_DOB          IN DATE,
  7          p_Balance      IN NUMBER
  8      ) IS
  9      BEGIN
 10          INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
 11          VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
 12      END AddCustomer;
 13
 14      PROCEDURE UpdateCustomer(
 15          p_CustomerID   IN NUMBER,
 16          p_Name         IN VARCHAR2,
 17          p_DOB          IN DATE
 18      ) IS
 19      BEGIN
 20          UPDATE Customers
 21          SET Name = p_Name,
 22              DOB = p_DOB,
 23              LastModified = SYSDATE
 24          WHERE CustomerID = p_CustomerID;
 25      END UpdateCustomer;
 26
 27      FUNCTION GetCustomerBalance(
 28          p_CustomerID   IN NUMBER
 29      ) RETURN NUMBER IS
 30          v_Balance NUMBER;
 31      BEGIN
 32          SELECT Balance INTO v_Balance
 33          FROM Customers
 34          WHERE CustomerID = p_CustomerID;
 35
 36          RETURN v_Balance;
 37      EXCEPTION
 38          WHEN NO_DATA_FOUND THEN
 39              RETURN NULL;
 40      END GetCustomerBalance;
 41
 42  END CustomerManagement;
 43  /
```

```
SQL> BEGIN
  2      CustomerManagement.UpdateCustomer(101, 'Alice Johnson', TO_DATE('1990-01-01', 'YYYY-MM-DD'));
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> DECLARE
  2      v_bal NUMBER;
  3  BEGIN
  4      v_bal := CustomerManagement.GetCustomerBalance(101);
  5      DBMS_OUTPUT.PUT_LINE('Balance: ' || v_bal);
  6  END;
  7  /
Balance:

PL/SQL procedure successfully completed.
```

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.

```
SQL> CREATE OR REPLACE PACKAGE EmployeeManagement IS
  2      PROCEDURE HireEmployee(
  3          p_EmployeeID IN NUMBER,
  4          p_Name       IN VARCHAR2,
  5          p_Position   IN VARCHAR2,
  6          p_Salary     IN NUMBER,
  7          p_Department IN VARCHAR2,
  8          p_HireDate   IN DATE
  9      );
 10
 11      PROCEDURE UpdateEmployeeDetails(
 12          p_EmployeeID IN NUMBER,
 13          p_Name       IN VARCHAR2,
 14          p_Position   IN VARCHAR2,
 15          p_Department IN VARCHAR2
 16      );
 17
 18      FUNCTION GetAnnualSalary(
 19          p_EmployeeID IN NUMBER
 20      ) RETURN NUMBER;
 21  END EmployeeManagement;
 22  /

Package created.
```

```
SQL> CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS
  2
  3      PROCEDURE HireEmployee(
  4          p_EmployeeID IN NUMBER,
  5          p_Name       IN VARCHAR2,
  6          p_Position   IN VARCHAR2,
  7          p_Salary     IN NUMBER,
  8          p_Department IN VARCHAR2,
  9          p_HireDate   IN DATE
 10      ) IS
 11      BEGIN
 12          INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
 13          VALUES (p_EmployeeID, p_Name, p_Position, p_Salary, p_Department, p_HireDate);
 14      END HireEmployee;
 15
 16      PROCEDURE UpdateEmployeeDetails(
 17          p_EmployeeID IN NUMBER,
 18          p_Name       IN VARCHAR2,
 19          p_Position   IN VARCHAR2,
 20          p_Department IN VARCHAR2
 21      ) IS
 22      BEGIN
 23          UPDATE Employees
 24          SET Name = p_Name,
 25              Position = p_Position,
 26              Department = p_Department
 27          WHERE EmployeeID = p_EmployeeID;
 28      END UpdateEmployeeDetails;
 29
 30      FUNCTION GetAnnualSalary(
 31          p_EmployeeID IN NUMBER
 32      ) RETURN NUMBER IS
 33          v_Salary NUMBER;
 34      BEGIN
 35          SELECT Salary INTO v_Salary
 36          FROM Employees
 37          WHERE EmployeeID = p_EmployeeID;
 38
 39          RETURN v_Salary * 12; -- monthly to annual
 40      EXCEPTION
 41          WHEN NO_DATA_FOUND THEN
 42              RETURN NULL;
 43      END GetAnnualSalary;
```

```
SQL> BEGIN
  2      EmployeeManagement.UpdateEmployeeDetails(
  3          201, 'David Clark', 'Senior Analyst', 'Corporate Finance'
  4      );
  5  END;
  6  /

PL/SQL procedure successfully completed.
```

```
SQL> DECLARE
  2      v_annual NUMBER;
  3  BEGIN
  4      v_annual := EmployeeManagement.GetAnnualSalary(201);
  5      DBMS_OUTPUT.PUT_LINE('Annual Salary: ' || v_annual);
  6  END;
  7  /
Annual Salary:

PL/SQL procedure successfully completed.
```

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.

```
SQL> CREATE OR REPLACE PACKAGE AccountOperations IS
  2      PROCEDURE OpenAccount(
  3          p_AccountID   IN NUMBER,
  4          p_CustomerID  IN NUMBER,
  5          p_AccountType IN VARCHAR2,
  6          p_Balance     IN NUMBER
  7      );
  8
  9      PROCEDURE CloseAccount(
 10          p_AccountID IN NUMBER
 11      );
 12
 13      FUNCTION GetTotalBalance(
 14          p_CustomerID IN NUMBER
 15      ) RETURN NUMBER;
 16  END AccountOperations;
 17  /

Package created.
```

```
SQL> CREATE OR REPLACE PACKAGE BODY AccountOperations IS
  2
  3      PROCEDURE OpenAccount(
  4          p_AccountID   IN NUMBER,
  5          p_CustomerID  IN NUMBER,
  6          p_AccountType IN VARCHAR2,
  7          p_Balance     IN NUMBER
  8      ) IS
  9      BEGIN
 10          INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
 11          VALUES (p_AccountID, p_CustomerID, p_AccountType, p_Balance, SYSDATE);
 12      END OpenAccount;
 13
 14      PROCEDURE CloseAccount(
 15          p_AccountID IN NUMBER
 16      ) IS
 17      BEGIN
 18          DELETE FROM Accounts
 19          WHERE AccountID = p_AccountID;
 20      END CloseAccount;
 21
 22      FUNCTION GetTotalBalance(
 23          p_CustomerID IN NUMBER
 24      ) RETURN NUMBER IS
 25          v_total NUMBER := 0;
 26      BEGIN
 27          SELECT NVL(SUM(Balance), 0)
 28          INTO v_total
 29          FROM Accounts
 30          WHERE CustomerID = p_CustomerID;
 31
 32          RETURN v_total;
 33      END GetTotalBalance;
 34
 35  END AccountOperations;
 36  /

Package body created.
```

```
SQL> DECLARE
  2      v_balance NUMBER;
  3  BEGIN
  4      v_balance := AccountOperations.GetTotalBalance(101);
  5      DBMS_OUTPUT.PUT_LINE('Total Balance for Customer 101: ' || v_balance);
  6  END;
  7  /
Total Balance for Customer 101: 0

PL/SQL procedure successfully completed.
```

Schema to be Created

CREATE TABLE Customers (
CustomerID NUMBER PRIMARY KEY,
Name VARCHAR2(100),
DOB DATE,

```
   Balance NUMBER,
   LastModified DATE
);
```

```
SQL> CREATE TABLE Customers(
  2  CustomerID NUMBER PRIMARY KEY,
  3  Name VARCHAR2(100),
  4  DOB DATE,
  5  Balance NUMBER,
  6  LastModified DATE
  7  );

Table created.
```

```
CREATE TABLE Accounts (
   AccountID NUMBER PRIMARY KEY,
   CustomerID NUMBER,
   AccountType VARCHAR2(20),
   Balance NUMBER,
   LastModified DATE,
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

```
SQL> CREATE TABLE Accounts(
  2  AccountID NUMBER PRIMARY KEY,
  3  CustomerID NUMBER,
  4  AccountType VARCHAR2(20),
  5  Balance NUMBER,
  6  LastModified DATE,
  7  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
  8  );

Table created.
```

```
CREATE TABLE Transactions (
   TransactionID NUMBER PRIMARY KEY,
   AccountID NUMBER,
   TransactionDate DATE,
```

```
  Amount NUMBER,
  TransactionType VARCHAR2(10),
  FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
);
```

```
SQL> CREATE TABLE Transactions (
  2      TransactionID NUMBER PRIMARY KEY,
  3      AccountID NUMBER,
  4      TransactionDate DATE,
  5      Amount NUMBER,
  6      TransactionType VARCHAR2(10),
  7      FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
  8  );

Table created.
```

```
CREATE TABLE Loans (
  LoanID NUMBER PRIMARY KEY,
  CustomerID NUMBER,
  LoanAmount NUMBER,
  InterestRate NUMBER,
  StartDate DATE,
  EndDate DATE,
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

```
SQL> CREATE TABLE Loans (
  2      LoanID NUMBER PRIMARY KEY,
  3      CustomerID NUMBER,
  4      LoanAmount NUMBER,
  5      InterestRate NUMBER,
  6      StartDate DATE,
  7      EndDate DATE,
  8      FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
  9  );

Table created.
```

```
CREATE TABLE Employees (
  EmployeeID NUMBER PRIMARY KEY,
  Name VARCHAR2(100),
  Position VARCHAR2(50),
  Salary NUMBER,
  Department VARCHAR2(50),
```

HireDate DATE
);

```
SQL> CREATE TABLE Employees (
  2      EmployeeID NUMBER PRIMARY KEY,
  3      Name VARCHAR2(100),
  4      Position VARCHAR2(50),
  5      Salary NUMBER,
  6      Department VARCHAR2(50),
  7      HireDate DATE
  8  );

Table created.
```

Example Scripts for Sample Data Insertion

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (1, 'John Doe', TO_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (2, 'Jane Smith', TO_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);

```
SQL> INSERT INTO Customers(CustomerID, Name, DOB, Balance, LastModified) VALUES (1, 'John Doe', TO_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);
1 row created.
SQL> INSERT INTO Customers(CustomerID, Name, DOB, Balance, LastModified) VALUES (2, 'Jane Smith', TO_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);
1 row created.
```

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
VALUES (1, 1, 'Savings', 1000, SYSDATE);
INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)
VALUES (2, 2, 'Checking', 1500, SYSDATE);

```
SQL> INSERT INTO Accounts(AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (1, 1, 'Savings', 1000, SYSDATE);
1 row created.
SQL> INSERT INTO Accounts(AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (2, 2, 'Checking', 1500, SYSDATE);
1 row created.
```

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (1, 1, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

```
SQL> INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
  2  VALUES (1, 1, SYSDATE, 200, 'Deposit');

1 row created.

SQL> INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)
  2  VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

1 row created.
```

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)
VALUES (1, 1, 5000, 5, SYSDATE, ADD_MONTHS(SYSDATE, 60));

```
SQL> INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)
  2  VALUES (1, 1, 5000, 5, SYSDATE, ADD_MONTHS(SYSDATE, 60));

1 row created.
```

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO_DATE('2015-06-15', 'YYYY-MM-DD'));
INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO_DATE('2017-03-20', 'YYYY-MM-DD'));

```
SQL> INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
  2  VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO_DATE('2015-06-15', 'YYYY-MM-DD'));

1 row created.

SQL> INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)
  2  VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO_DATE('2017-03-20', 'YYYY-MM-DD'));

1 row created.
```