**TABLES**

**Customers Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CustomerID** | **Name** | **DOB** | **Balance** | **LastModified** |
| 1 | John Doe | 15-05-1985 | 1000 | 2024-08-07 |
| 2 | Jane Smith | 20-07-1990 | 1500 | 2024-08-07 |

**Accounts Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AccountID** | **CustomerID** | **AccountType** | **Balance** | **LastModified** |
| 1 | 1 | Savings | 1000 | 2024-08-07 |
| 2 | 2 | Checking | 1500 | 2024-08-07 |

**Transactions Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TransactionID** | **AccountID** | **TransactionDate** | **Amount** | **TransactionType** |
| 1 | 1 | 2024-08-07 | 200 | Deposit |
| 2 | 2 | 2024-08-07 | 300 | Withdrawal |

**Loans Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **LoanID** | **CustomerID** | **LoanAmount** | **InterestRate** | **StartDate** | **EndDate** |
| 1 | 1 | 5000 | 5 | 2024-08-07 | 2029-08-07 |

**Employees Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EmployeeID** | **Name** | **Position** | **Salary** | **Department** | **HireDate** |
| 1 | Alice Johnson | Manager | 70000 | HR | 15-06-2015 |
| 2 | Bob Brown | Developer | 60000 | IT | 20-03-2017 |

**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 to select all customers

CURSOR c\_customers IS

SELECT CustomerID, DOB FROM Customers;

v\_customer\_id Customers.CustomerID%TYPE;

v\_dob Customers.DOB%TYPE;

v\_age NUMBER;

BEGIN

-- Loop through each customer

OPEN c\_customers;

LOOP

FETCH c\_customers INTO v\_customer\_id, v\_dob;

EXIT WHEN c\_customers%NOTFOUND;

-- Calculate the age

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, v\_dob) / 12);

-- Check if age is greater than 60

IF v\_age > 60 THEN

-- Apply a 1% discount to their current loan interest rates

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = v\_customer\_id;

END IF;

END LOOP;

CLOSE c\_customers;

DBMS\_OUTPUT.PUT\_LINE('Loan interest rates updated for customers above 60 years old.');

END;

/

**OUTPUT**

Loan interest rates updated for customers above 60 years old.

**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.**

-- Add the IsVIP column to the Customers table

ALTER TABLE Customers ADD (IsVIP CHAR(1));

-- PL/SQL block to promote customers to VIP status

BEGIN

-- Update the IsVIP flag for customers with balance over $10,000

UPDATE Customers

SET IsVIP = 'Y'

WHERE Balance > 10000;

DBMS\_OUTPUT.PUT\_LINE('VIP status updated for customers with balance over $10,000.');

END;

/

**OUTPUT:**

VIP status updated for customers with balance over $10,000.

**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 to select loans due in the next 30 days

CURSOR c\_loans\_due IS

SELECT L.CustomerID, L.LoanID, C.Name, L.EndDate

FROM Loans L

JOIN Customers C ON L.CustomerID = C.CustomerID

WHERE L.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

v\_customer\_id Loans.CustomerID%TYPE;

v\_loan\_id Loans.LoanID%TYPE;

v\_customer\_name Customers.Name%TYPE;

v\_end\_date Loans.EndDate%TYPE;

BEGIN

-- Loop through each loan due in the next 30 days

OPEN c\_loans\_due;

LOOP

FETCH c\_loans\_due INTO v\_customer\_id, v\_loan\_id, v\_customer\_name, v\_end\_date;

EXIT WHEN c\_loans\_due%NOTFOUND;

-- Print reminder message

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || v\_customer\_name || ' (ID: ' || v\_customer\_id || ') has a loan (ID: ' || v\_loan\_id || ') due on ' || TO\_CHAR(v\_end\_date, 'YYYY-MM-DD') || '.');

END LOOP;

CLOSE c\_loans\_due;

DBMS\_OUTPUT.PUT\_LINE('Reminders sent for loans due within the next 30 days.');

END;

/

**OUTPUT:**

Reminder: Customer John Doe (ID: 1) has a loan (ID: 1) due on 2024-08-25.

Reminders sent for loans due within the next 30 days.

**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

) IS

v\_from\_balance Accounts.Balance%TYPE;

v\_to\_balance Accounts.Balance%TYPE;

BEGIN

-- Start a transaction

BEGIN

-- Check the balance of the source account

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id FOR UPDATE;

-- Ensure the source account has sufficient funds

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the source account.');

END IF;

-- Check the balance of the destination account

SELECT Balance INTO v\_to\_balance

FROM Accounts

WHERE AccountID = p\_to\_account\_id FOR UPDATE;

-- Update balances

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

-- Commit transaction

COMMIT;

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Account(s) not found.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

END;

/

**OUTPUT:**

**If the transfer is successful:**

Funds transferred successfully.

**If there is an error (e.g., insufficient funds or account not found):**

Error: Insufficient funds in the source account.

**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

) IS

BEGIN

-- Start a transaction

BEGIN

-- Update the salary

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID does not exist.');

END IF;

-- Commit transaction

COMMIT;

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID does not exist.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

END;

/

**OUTPUT:**

**If the salary is updated successfully:**

Salary updated successfully.

**If the employee ID does not exist:**

Error: Employee ID does not exist.

**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\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

-- Start a transaction

BEGIN

-- Attempt to insert the new customer

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

VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);

-- Commit transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('New customer added successfully.');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Customer with this ID already exists.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

END;

/

**OUTPUT:**

**If the customer is added successfully:**

New customer added successfully.

**If a customer with the same ID already exists:**

Error: Customer with this ID already exists.

**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 Accounts

SET Balance = Balance \* 1.01

WHERE AccountType = 'Savings';

COMMIT; -- Commit the transaction to save changes

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings accounts.');

END;

/

**OUTPUT:**

Monthly interest applied to all savings accounts.

**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 VARCHAR2, p\_bonus\_percentage NUMBER) AS

BEGIN

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

UPDATE Employees

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

WHERE Department = p\_department;

COMMIT; -- Commit the transaction to save changes

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to employees in department ' || p\_department || '.');

END;

/

**OUTPUT:**

Bonus applied to employees in department HR.

**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\_from\_account\_id NUMBER, p\_to\_account\_id NUMBER, p\_amount NUMBER) AS

v\_balance NUMBER;

BEGIN

-- Check if the source account has sufficient balance

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_account\_id;

IF v\_balance < p\_amount THEN

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance for the transfer.');

RETURN;

END IF;

-- Transfer the funds

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

COMMIT; -- Commit the transaction to save changes

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' completed from account ' || p\_from\_account\_id || ' to account ' || p\_to\_account\_id || '.');

END;

/

**OUTPUT:**

Transfer of 200 completed from account 1 to account 2.

**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.**

**FUNCTION:**

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

v\_age NUMBER;

BEGIN

-- Calculate the age using the date of birth

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

RETURN v\_age;

END;

/

**SQL**

-- Test the function

DECLARE

v\_age NUMBER;

BEGIN

-- Test with customer 1

v\_age := CalculateAge(TO\_DATE('1985-05-15', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Age of customer with DOB 1985-05-15: ' || v\_age);

-- Test with customer 2

v\_age := CalculateAge(TO\_DATE('1990-07-20', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Age of customer with DOB 1990-07-20: ' || v\_age);

END;

/

**OUTPUT:**

Age of customer with DOB 1985-05-15: 39

Age of customer with DOB 1990-07-20: 34

**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.**

**FUNCTION:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount NUMBER,

p\_annual\_interest\_rate NUMBER,

p\_duration\_years NUMBER

) RETURN NUMBER IS

v\_monthly\_interest\_rate NUMBER;

v\_number\_of\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

-- Calculate monthly interest rate and number of payments

v\_monthly\_interest\_rate := p\_annual\_interest\_rate / 12 / 100;

v\_number\_of\_payments := p\_duration\_years \* 12;

-- Calculate the monthly installment using the formula

v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_interest\_rate /

(1 - POWER(1 + v\_monthly\_interest\_rate, -v\_number\_of\_payments));

RETURN v\_monthly\_installment;

END;

/

**SQL**

-- Test the function

DECLARE

v\_installment NUMBER;

BEGIN

v\_installment := CalculateMonthlyInstallment(5000, 5, 5);

DBMS\_OUTPUT.PUT\_LINE('Monthly installment for a loan of 5000 at 5% interest for 5 years: ' || v\_installment);

END;

/

**OUTPUT:**

Monthly installment for a loan of 5000 at 5% interest for 5 years: 94.56

**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.**

**FUNCTION:**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

-- Retrieve the balance of the account

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

-- Check if the balance is sufficient

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE; -- Account not found

END;

/

**SQL:**

-- Test the function

DECLARE

v\_sufficient BOOLEAN;

BEGIN

-- Test with AccountID 1 for amount 500

v\_sufficient := HasSufficientBalance(1, 500);

DBMS\_OUTPUT.PUT\_LINE('AccountID 1 has sufficient balance for 500: ' || CASE WHEN v\_sufficient THEN 'YES' ELSE 'NO' END);

-- Test with AccountID 2 for amount 2000

v\_sufficient := HasSufficientBalance(2, 2000);

DBMS\_OUTPUT.PUT\_LINE('AccountID 2 has sufficient balance for 2000: ' || CASE WHEN v\_sufficient THEN 'YES' ELSE 'NO' END);

END;

/

**OUTPUT:**

AccountID 1 has sufficient balance for 500: YES

AccountID 2 has sufficient balance for 2000: NO

**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.**

**TRIGGER:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

**SQL:**

UPDATE Customers

SET Name = 'Johnathan Doe'

WHERE CustomerID = 1;

**OUTPUT:**

Customer record updated. LastModified column set to 2024-08-07.

**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 TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

TransactionID NUMBER,

ActionDate DATE,

ActionType VARCHAR2(50),

FOREIGN KEY (TransactionID) REFERENCES Transactions(TransactionID)

);

**TRIGGER:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (LogID, TransactionID, ActionDate, ActionType)

VALUES (AuditLog\_SEQ.NEXTVAL, :NEW.TransactionID, SYSDATE, 'INSERT');

END;

/

**SQL:**

CREATE SEQUENCE AuditLog\_SEQ

START WITH 1

INCREMENT BY 1;

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (3, 1, SYSDATE, 500, 'Deposit');

**OUTPUT:**

Audit log entry added for TransactionID 3 on 2024-08-07 with action 'INSERT'.

**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

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

-- Check if withdrawal does not exceed balance

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance for withdrawal.');

END IF;

END;

ELSIF :NEW.TransactionType = 'Deposit' THEN

-- Ensure deposits are positive

IF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

END;

/

**OUTPUT:**

-- Valid deposit

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (4, 1, SYSDATE, 200, 'Deposit');

Transaction of 200 as Deposit completed successfully.

-- Invalid withdrawal (more than balance)

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (5, 1, SYSDATE, 1500, 'Withdrawal');

ORA-20001: Insufficient balance for withdrawal.

-- Invalid deposit (non-positive amount)

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (6, 1, SYSDATE, -100, 'Deposit');

ORA-20002: Deposit amount must be positive.

**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 c\_transactions IS

SELECT C.CustomerID, C.Name, T.TransactionDate, T.Amount, T.TransactionType

FROM Customers C

JOIN Accounts A ON C.CustomerID = A.CustomerID

JOIN Transactions T ON A.AccountID = T.AccountID

WHERE EXTRACT(MONTH FROM T.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM T.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

v\_customer\_id Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_transaction\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_transaction\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN c\_transactions;

LOOP

FETCH c\_transactions INTO v\_customer\_id, v\_name, v\_transaction\_date, v\_amount, v\_transaction\_type;

EXIT WHEN c\_transactions%NOTFOUND;

-- Print the statement for each customer

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_customer\_id || ', Name: ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || TO\_CHAR(v\_transaction\_date, 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Amount: ' || v\_amount || ', Type: ' || v\_transaction\_type);

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

END LOOP;

CLOSE c\_transactions;

DBMS\_OUTPUT.PUT\_LINE('Monthly statements generated for all customers.');

END;

/

**OUTPUT**

Customer ID: 1, Name: John Doe

Transaction Date: 2024-08-07

Amount: 200, Type: Deposit

----------------------------------------

Customer ID: 2, Name: Jane Smith

Transaction Date: 2024-08-07

Amount: 300, Type: Withdrawal

----------------------------------------

Monthly statements generated for all customers.

**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 c\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

annual\_fee NUMBER := 50; -- Example annual fee

BEGIN

OPEN c\_accounts;

LOOP

FETCH c\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN c\_accounts%NOTFOUND;

-- Deduct annual maintenance fee from the balance

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = v\_account\_id;

END LOOP;

CLOSE c\_accounts;

COMMIT; -- Commit the transaction to save changes

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to all accounts.');

END;

/

**OUTPUT:**

Annual fee applied to all accounts.

**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 c\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

new\_interest\_rate NUMBER := 6; -- Example new interest rate policy

BEGIN

OPEN c\_loans;

LOOP

FETCH c\_loans INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN c\_loans%NOTFOUND;

-- Update the interest rate based on the new policy

UPDATE Loans

SET InterestRate = new\_interest\_rate

WHERE LoanID = v\_loan\_id;

END LOOP;

CLOSE c\_loans;

COMMIT; -- Commit the transaction to save changes

DBMS\_OUTPUT.PUT\_LINE('Interest rates updated based on the new policy.');

END;

/

**OUTPUT:**

Interest rates updated based on the new policy.

**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 AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER);

PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_balance NUMBER);

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

**PACKAGE BODY**

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

BEGIN

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

VALUES (p\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer added: ' || p\_name);

END AddCustomer;

PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_name, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer updated: ' || p\_id);

END UpdateCustomer;

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

/

**OUTPUT:**

BEGIN

-- Add a new customer

CustomerManagement.AddCustomer(3, 'Michael Clark', TO\_DATE('1980-02-22', 'YYYY-MM-DD'), 2000);

-- Update existing customer

CustomerManagement.UpdateCustomer(1, 'John Doe Updated', 1200);

-- Get customer balance

DBMS\_OUTPUT.PUT\_LINE('Balance of Customer 1: ' || CustomerManagement.GetCustomerBalance(1));

END;

/

Customer added: Michael Clark

Customer updated: 1

Balance of Customer 1: 1200

**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\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE);

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_salary NUMBER);

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE) IS

BEGIN

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

VALUES (p\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Employee hired: ' || p\_name);

END HireEmployee;

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_salary NUMBER) IS

BEGIN

UPDATE Employees

SET Name = p\_name, Salary = p\_salary

WHERE EmployeeID = p\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Employee updated: ' || p\_id);

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_id;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

/

BEGIN

-- Hire a new employee

EmployeeManagement.HireEmployee(3, 'Emily Davis', 'Analyst', 55000, 'Finance', TO\_DATE('2024-07-01', 'YYYY-MM-DD'));

-- Update existing employee

EmployeeManagement.UpdateEmployee(1, 'Alice Johnson Updated', 75000);

-- Calculate annual salary

DBMS\_OUTPUT.PUT\_LINE('Annual Salary of Employee 1: ' || EmployeeManagement.CalculateAnnualSalary(1));

END;

/

**OUTPUT:**

Employee hired: Emily Davis

Employee updated: 1

Annual Salary of Employee 1: 900000

**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\_id NUMBER, p\_customer\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_id NUMBER);

FUNCTION GetTotalBalance(p\_customer\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_id NUMBER, p\_customer\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

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

VALUES (p\_id, p\_customer\_id, p\_type, p\_balance, SYSDATE);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Account opened: ' || p\_id);

END OpenAccount;

PROCEDURE CloseAccount(p\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Account closed: ' || p\_id);

END CloseAccount;

FUNCTION GetTotalBalance(p\_customer\_id NUMBER) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN NVL(v\_total\_balance, 0); -- Return 0 if no accounts found

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END GetTotalBalance;

END AccountOperations;

/

BEGIN

-- Open a new account

AccountOperations.OpenAccount(3, 1, 'Savings', 500);

-- Close an existing account

AccountOperations.CloseAccount(2);

-- Get total balance for a customer

DBMS\_OUTPUT.PUT\_LINE('Total balance for Customer 1: ' || AccountOperations.GetTotalBalance(1));

END;

/

**OUTPUT:**

Account opened: 3

Account closed: 2

Total balance for Customer 1: 1310