1)

DECLARE

CURSOR customer\_cursor IS

SELECT customer\_id, age, loan\_interest\_rate

FROM customers;

v\_customer\_id customers.customer\_id%TYPE;

v\_age customers.age%TYPE;

v\_loan\_interest\_rate customers.loan\_interest\_rate%TYPE;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

v\_customer\_id := customer\_rec.customer\_id;

v\_age := customer\_rec.age;

v\_loan\_interest\_rate := customer\_rec.loan\_interest\_rate;

IF v\_age > 60 THEN

-- Apply a 1% discount to the current loan interest rate

v\_loan\_interest\_rate := v\_loan\_interest\_rate - 1;

-- Update the loan interest rate in the database

UPDATE customers

SET loan\_interest\_rate = v\_loan\_interest\_rate

WHERE customer\_id = v\_customer\_id;

END IF;

END LOOP;

COMMIT; -- Commit the changes to the database

END;

Scenario 2: Promote Customers to VIP Status Based on Their Balance

plsql

Copy code

DECLARE

CURSOR customer\_cursor IS

SELECT customer\_id, balance

FROM customers;

v\_customer\_id customers.customer\_id%TYPE;

v\_balance customers.balance%TYPE;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

v\_customer\_id := customer\_rec.customer\_id;

v\_balance := customer\_rec.balance;

IF v\_balance > 10000 THEN

-- Set the IsVIP flag to TRUE

UPDATE customers

SET IsVIP = TRUE

WHERE customer\_id = v\_customer\_id;

END IF;

END LOOP;

COMMIT; -- Commit the changes to the database

END;

Scenario 3: Send Reminders for Loans Due Within the Next 30 Days

plsql

Copy code

DECLARE

CURSOR loan\_cursor IS

SELECT loan\_id, customer\_id, due\_date

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30;

v\_loan\_id loans.loan\_id%TYPE;

v\_customer\_id loans.customer\_id%TYPE;

v\_due\_date loans.due\_date%TYPE;

BEGIN

FOR loan\_rec IN loan\_cursor LOOP

v\_loan\_id := loan\_rec.loan\_id;

v\_customer\_id := loan\_rec.customer\_id;

v\_due\_date := loan\_rec.due\_date;

-- Print a reminder message

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || v\_loan\_id ||

' for Customer ID ' || v\_customer\_id ||

' is due on ' || TO\_CHAR(v\_due\_date, 'DD-MON-YYYY'));

END LOOP;

END;

2)

Scenario 1: Handle Exceptions During Fund Transfers Between Accounts

Stored Procedure: SafeTransferFunds

plsql

Copy code

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account\_id IN accounts.account\_id%TYPE,

p\_to\_account\_id IN accounts.account\_id%TYPE,

p\_amount IN NUMBER

)

IS

insufficient\_funds EXCEPTION;

v\_balance accounts.balance%TYPE;

BEGIN

-- Check balance of the source account

SELECT balance INTO v\_balance FROM accounts WHERE account\_id = p\_from\_account\_id FOR UPDATE;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

ELSE

-- Deduct the amount from the source account

UPDATE accounts

SET balance = balance - p\_amount

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

-- Add the amount to the destination account

UPDATE accounts

SET balance = balance + p\_amount

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

COMMIT; -- Commit the transaction

END IF;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Unable to transfer funds. Please try again.');

END;

/

Scenario 2: Manage Errors When Updating Employee Salaries

Stored Procedure: UpdateSalary

plsql

Copy code

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id IN employees.employee\_id%TYPE,

p\_percentage IN NUMBER

)

IS

employee\_not\_found EXCEPTION;

v\_salary employees.salary%TYPE;

BEGIN

-- Check if the employee exists

SELECT salary INTO v\_salary FROM employees WHERE employee\_id = p\_employee\_id FOR UPDATE;

-- Increase the salary by the given percentage

UPDATE employees

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

WHERE employee\_id = p\_employee\_id;

COMMIT; -- Commit the transaction

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Unable to update salary. Please try again.');

END;

/

Scenario 3: Ensure Data Integrity When Adding a New Customer

Stored Procedure: AddNewCustomer

plsql

Copy code

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN customers.customer\_id%TYPE,

p\_name IN customers.name%TYPE,

p\_age IN customers.age%TYPE,

p\_balance IN customers.balance%TYPE

)

IS

customer\_exists EXCEPTION;

BEGIN

-- Insert new customer into the Customers table

INSERT INTO customers (customer\_id, name, age, balance)

VALUES (p\_customer\_id, p\_name, p\_age, p\_balance);

COMMIT; -- Commit the transaction

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Unable to add new customer. Please try again.');

END;

3)

Scenario 1: Process Monthly Interest for All Savings Accounts

Stored Procedure: ProcessMonthlyInterest

plsql

Copy code

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

IS

CURSOR savings\_cursor IS

SELECT account\_id, balance

FROM accounts

WHERE account\_type = 'SAVINGS';

v\_account\_id accounts.account\_id%TYPE;

v\_balance accounts.balance%TYPE;

v\_interest\_rate CONSTANT NUMBER := 0.01; -- 1% interest rate

BEGIN

FOR account\_rec IN savings\_cursor LOOP

v\_account\_id := account\_rec.account\_id;

v\_balance := account\_rec.balance;

-- Calculate new balance with interest

v\_balance := v\_balance + (v\_balance \* v\_interest\_rate);

-- Update the balance in the database

UPDATE accounts

SET balance = v\_balance

WHERE account\_id = v\_account\_id;

END LOOP;

COMMIT; -- Commit the changes to the database

END;

/

Scenario 2: Implement a Bonus Scheme for Employees Based on Performance

Stored Procedure: UpdateEmployeeBonus

plsql

Copy code

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department\_id IN employees.department\_id%TYPE,

p\_bonus\_percentage IN NUMBER

)

IS

CURSOR employee\_cursor IS

SELECT employee\_id, salary

FROM employees

WHERE department\_id = p\_department\_id;

v\_employee\_id employees.employee\_id%TYPE;

v\_salary employees.salary%TYPE;

BEGIN

FOR employee\_rec IN employee\_cursor LOOP

v\_employee\_id := employee\_rec.employee\_id;

v\_salary := employee\_rec.salary;

-- Calculate new salary with bonus

v\_salary := v\_salary + (v\_salary \* p\_bonus\_percentage / 100);

-- Update the salary in the database

UPDATE employees

SET salary = v\_salary

WHERE employee\_id = v\_employee\_id;

END LOOP;

COMMIT; -- Commit the changes to the database

END;

/

Scenario 3: Transfer Funds Between Accounts

Stored Procedure: TransferFunds

plsql

Copy code

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account\_id IN accounts.account\_id%TYPE,

p\_to\_account\_id IN accounts.account\_id%TYPE,

p\_amount IN NUMBER

)

IS

v\_from\_balance accounts.balance%TYPE;

BEGIN

-- Check balance of the source account

SELECT balance INTO v\_from\_balance FROM accounts WHERE account\_id = p\_from\_account\_id FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

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

ELSE

-- Deduct the amount from the source account

UPDATE accounts

SET balance = balance - p\_amount

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

-- Add the amount to the destination account

UPDATE accounts

SET balance = balance + p\_amount

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

COMMIT; -- Commit the transaction

END IF;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Unable to transfer funds.');

END;

/

4).

**Scenario 1: Calculate the Age of Customers for Eligibility Checks**

**Function: CalculateAge**

plsql

Copy code

CREATE OR REPLACE FUNCTION CalculateAge(

p\_date\_of\_birth IN DATE

)

RETURN NUMBER

IS

v\_age NUMBER;

BEGIN

-- Calculate age in years based on the current date and date of birth

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

RETURN v\_age;

END;

/

* **Explanation**: The function CalculateAge calculates the customer's age in years by determining the difference in months between the current date (SYSDATE) and the provided date of birth. It then converts this difference into years using FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_date\_of\_birth) / 12).

**Scenario 2: Compute the Monthly Installment for a Loan**

**Function: CalculateMonthlyInstallment**

plsql

Copy code

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount IN NUMBER,

p\_interest\_rate IN NUMBER,

p\_duration\_years IN NUMBER

)

RETURN NUMBER

IS

v\_monthly\_installment NUMBER;

v\_monthly\_rate NUMBER;

v\_total\_months NUMBER;

BEGIN

-- Convert annual interest rate to a monthly rate

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

-- Total number of months for the loan duration

v\_total\_months := p\_duration\_years \* 12;

-- Calculate the monthly installment using the formula

IF v\_monthly\_rate > 0 THEN

v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_rate / (1 - POWER(1 + v\_monthly\_rate, -v\_total\_months));

ELSE

-- If interest rate is 0, simply divide the loan amount by the number of months

v\_monthly\_installment := p\_loan\_amount / v\_total\_months;

END IF;

RETURN v\_monthly\_installment;

END;

/

* **Explanation**: The function CalculateMonthlyInstallment calculates the monthly installment amount for a loan using the formula for an amortizing loan payment. It considers the loan amount, interest rate (converted to a monthly rate), and the loan duration in years.

**Scenario 3: Check if a Customer Has Sufficient Balance Before Making a Transaction**

**Function: HasSufficientBalance**

plsql

Copy code

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id IN accounts.account\_id%TYPE,

p\_amount IN NUMBER

)

RETURN BOOLEAN

IS

v\_balance accounts.balance%TYPE;

BEGIN

-- Fetch the current balance of the account

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = p\_account\_id;

-- Check if the balance is sufficient

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE; -- If the account is not found, return FALSE

END;

/

5).

Scenario 1: Automatically Update the Last Modified Date When a Customer's Record is Updated

Trigger: UpdateCustomerLastModified

plsql

Copy code

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON customers

FOR EACH ROW

BEGIN

-- Update the LastModified column to the current date

:NEW.LastModified := SYSDATE;

END;

/

Explanation: The UpdateCustomerLastModified trigger is fired before any update on the customers table. It automatically sets the LastModified column to the current date and time (SYSDATE) whenever a customer's record is updated.

Scenario 2: Maintain an Audit Log for All Transactions

Trigger: LogTransaction

plsql

Copy code

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

-- Insert a record into the AuditLog table

INSERT INTO AuditLog (transaction\_id, account\_id, transaction\_date, amount, action\_type)

VALUES (:NEW.transaction\_id, :NEW.account\_id, :NEW.transaction\_date, :NEW.amount, 'INSERT');

END;

/

Explanation: The LogTransaction trigger is fired after a new transaction is inserted into the transactions table. It logs the details of the transaction (such as transaction\_id, account\_id, transaction\_date, and amount) into an AuditLog table with an action type of 'INSERT'.

Scenario 3: Enforce Business Rules on Deposits and Withdrawals

Trigger: CheckTransactionRules

plsql

Copy code

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON transactions

FOR EACH ROW

DECLARE

v\_balance accounts.balance%TYPE;

BEGIN

-- Fetch the current balance of the account

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = :NEW.account\_id

FOR UPDATE;

-- Ensure that withdrawals do not exceed the current balance

IF :NEW.transaction\_type = 'WITHDRAWAL' AND :NEW.amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds for withdrawal.');

END IF;

-- Ensure that deposits are positive amounts

IF :NEW.transaction\_type = 'DEPOSIT' AND :NEW.amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Deposit amount must be positive.');

END IF;

END;

/

6).

Scenario 1: Generate Monthly Statements for All Customers

PL/SQL Block: GenerateMonthlyStatements

plsql

Copy code

DECLARE

CURSOR transaction\_cursor IS

SELECT customer\_id, transaction\_id, transaction\_date, amount, transaction\_type

FROM transactions

WHERE TO\_CHAR(transaction\_date, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY');

v\_customer\_id customers.customer\_id%TYPE;

v\_transaction\_id transactions.transaction\_id%TYPE;

v\_transaction\_date transactions.transaction\_date%TYPE;

v\_amount transactions.amount%TYPE;

v\_transaction\_type transactions.transaction\_type%TYPE;

BEGIN

OPEN transaction\_cursor;

LOOP

FETCH transaction\_cursor INTO v\_customer\_id, v\_transaction\_id, v\_transaction\_date, v\_amount, v\_transaction\_type;

EXIT WHEN transaction\_cursor%NOTFOUND;

-- Print the statement for each customer

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

DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || v\_transaction\_id || ', Date: ' || v\_transaction\_date ||

', Amount: ' || v\_amount || ', Type: ' || v\_transaction\_type);

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

END LOOP;

CLOSE transaction\_cursor;

END;

/

Explanation: The GenerateMonthlyStatements PL/SQL block uses an explicit cursor to retrieve all transactions for the current month. It then prints out a simple statement for each customer, showing transaction details such as the transaction ID, date, amount, and type.

Scenario 2: Apply Annual Fee to All Accounts

PL/SQL Block: ApplyAnnualFee

plsql

Copy code

DECLARE

CURSOR account\_cursor IS

SELECT account\_id, balance

FROM accounts;

v\_account\_id accounts.account\_id%TYPE;

v\_balance accounts.balance%TYPE;

v\_annual\_fee CONSTANT NUMBER := 50; -- Annual maintenance fee

BEGIN

OPEN account\_cursor;

LOOP

FETCH account\_cursor INTO v\_account\_id, v\_balance;

EXIT WHEN account\_cursor%NOTFOUND;

-- Deduct the annual fee from the account balance

UPDATE accounts

SET balance = v\_balance - v\_annual\_fee

WHERE account\_id = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_account\_id || ' charged an annual fee of $' || v\_annual\_fee);

END LOOP;

CLOSE account\_cursor;

COMMIT;

END;

/

Explanation: The ApplyAnnualFee PL/SQL block uses an explicit cursor to fetch all accounts. It then deducts a fixed annual maintenance fee from each account's balance and updates the account in the database.

Scenario 3: Update the Interest Rate for All Loans Based on a New Policy

PL/SQL Block: UpdateLoanInterestRates

plsql

Copy code

DECLARE

CURSOR loan\_cursor IS

SELECT loan\_id, interest\_rate

FROM loans;

v\_loan\_id loans.loan\_id%TYPE;

v\_interest\_rate loans.interest\_rate%TYPE;

v\_new\_interest\_rate NUMBER;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN loan\_cursor%NOTFOUND;

-- Calculate the new interest rate based on the new policy

IF v\_interest\_rate < 5 THEN

v\_new\_interest\_rate := v\_interest\_rate + 1; -- Increase by 1% if the rate is below 5%

ELSE

v\_new\_interest\_rate := v\_interest\_rate \* 1.05; -- Increase by 5% otherwise

END IF;

-- Update the loan's interest rate

UPDATE loans

SET interest\_rate = v\_new\_interest\_rate

WHERE loan\_id = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || v\_loan\_id || ' updated with new interest rate: ' || v\_new\_interest\_rate);

END LOOP;

CLOSE loan\_cursor;

COMMIT;

END;

/

7).

Scenario 1: Group All Customer-Related Procedures and Functions into a Package

Package Specification: CustomerManagement

plsql

Copy code

CREATE OR REPLACE PACKAGE CustomerManagement AS

-- Procedure to add a new customer

PROCEDURE AddCustomer(p\_customer\_id IN customers.customer\_id%TYPE,

p\_name IN customers.name%TYPE,

p\_dob IN customers.date\_of\_birth%TYPE,

p\_address IN customers.address%TYPE);

-- Procedure to update customer details

PROCEDURE UpdateCustomer(p\_customer\_id IN customers.customer\_id%TYPE,

p\_name IN customers.name%TYPE,

p\_address IN customers.address%TYPE);

-- Function to get customer balance

FUNCTION GetCustomerBalance(p\_customer\_id IN customers.customer\_id%TYPE)

RETURN NUMBER;

END CustomerManagement;

/

Package Body: CustomerManagement

plsql

Copy code

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_customer\_id IN customers.customer\_id%TYPE,

p\_name IN customers.name%TYPE,

p\_dob IN customers.date\_of\_birth%TYPE,

p\_address IN customers.address%TYPE) IS

BEGIN

INSERT INTO customers (customer\_id, name, date\_of\_birth, address)

VALUES (p\_customer\_id, p\_name, p\_dob, p\_address);

END AddCustomer;

PROCEDURE UpdateCustomer(p\_customer\_id IN customers.customer\_id%TYPE,

p\_name IN customers.name%TYPE,

p\_address IN customers.address%TYPE) IS

BEGIN

UPDATE customers

SET name = p\_name, address = p\_address

WHERE customer\_id = p\_customer\_id;

END UpdateCustomer;

FUNCTION GetCustomerBalance(p\_customer\_id IN customers.customer\_id%TYPE)

RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(balance) INTO v\_total\_balance

FROM accounts

WHERE customer\_id = p\_customer\_id;

RETURN NVL(v\_total\_balance, 0);

END GetCustomerBalance;

END CustomerManagement;

/

Explanation: The CustomerManagement package groups customer-related operations:

AddCustomer: Inserts a new customer into the customers table.

UpdateCustomer: Updates an existing customer's details.

GetCustomerBalance: Returns the total balance of all accounts associated with a specific customer.

Scenario 2: Create a Package to Manage Employee Data

Package Specification: EmployeeManagement

plsql

Copy code

CREATE OR REPLACE PACKAGE EmployeeManagement AS

-- Procedure to hire a new employee

PROCEDURE HireEmployee(p\_employee\_id IN employees.employee\_id%TYPE,

p\_name IN employees.name%TYPE,

p\_position IN employees.position%TYPE,

p\_salary IN employees.salary%TYPE);

-- Procedure to update employee details

PROCEDURE UpdateEmployee(p\_employee\_id IN employees.employee\_id%TYPE,

p\_name IN employees.name%TYPE,

p\_position IN employees.position%TYPE,

p\_salary IN employees.salary%TYPE);

-- Function to calculate annual salary

FUNCTION CalculateAnnualSalary(p\_employee\_id IN employees.employee\_id%TYPE)

RETURN NUMBER;

END EmployeeManagement;

/

Package Body: EmployeeManagement

plsql

Copy code

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_employee\_id IN employees.employee\_id%TYPE,

p\_name IN employees.name%TYPE,

p\_position IN employees.position%TYPE,

p\_salary IN employees.salary%TYPE) IS

BEGIN

INSERT INTO employees (employee\_id, name, position, salary)

VALUES (p\_employee\_id, p\_name, p\_position, p\_salary);

END HireEmployee;

PROCEDURE UpdateEmployee(p\_employee\_id IN employees.employee\_id%TYPE,

p\_name IN employees.name%TYPE,

p\_position IN employees.position%TYPE,

p\_salary IN employees.salary%TYPE) IS

BEGIN

UPDATE employees

SET name = p\_name, position = p\_position, salary = p\_salary

WHERE employee\_id = p\_employee\_id;

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_employee\_id IN employees.employee\_id%TYPE)

RETURN NUMBER IS

v\_monthly\_salary employees.salary%TYPE;

v\_annual\_salary NUMBER;

BEGIN

SELECT salary INTO v\_monthly\_salary

FROM employees

WHERE employee\_id = p\_employee\_id;

v\_annual\_salary := v\_monthly\_salary \* 12;

RETURN v\_annual\_salary;

END CalculateAnnualSalary;

END EmployeeManagement;

/

Explanation: The EmployeeManagement package handles employee data:

HireEmployee: Adds a new employee to the employees table.

UpdateEmployee: Updates existing employee details.

CalculateAnnualSalary: Computes the annual salary of an employee based on their monthly salary.

Scenario 3: Group All Account-Related Operations into a Package

Package Specification: AccountOperations

plsql

Copy code

CREATE OR REPLACE PACKAGE AccountOperations AS

-- Procedure to open a new account

PROCEDURE OpenAccount(p\_account\_id IN accounts.account\_id%TYPE,

p\_customer\_id IN accounts.customer\_id%TYPE,

p\_balance IN accounts.balance%TYPE);

-- Procedure to close an account

PROCEDURE CloseAccount(p\_account\_id IN accounts.account\_id%TYPE);

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

FUNCTION GetTotalCustomerBalance(p\_customer\_id IN accounts.customer\_id%TYPE)

RETURN NUMBER;

END AccountOperations;

/

Package Body: AccountOperations

plsql

Copy code

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_account\_id IN accounts.account\_id%TYPE,

p\_customer\_id IN accounts.customer\_id%TYPE,

p\_balance IN accounts.balance%TYPE) IS

BEGIN

INSERT INTO accounts (account\_id, customer\_id, balance)

VALUES (p\_account\_id, p\_customer\_id, p\_balance);

END OpenAccount;

PROCEDURE CloseAccount(p\_account\_id IN accounts.account\_id%TYPE) IS

BEGIN

DELETE FROM accounts

WHERE account\_id = p\_account\_id;

END CloseAccount;

FUNCTION GetTotalCustomerBalance(p\_customer\_id IN accounts.customer\_id%TYPE)

RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(balance) INTO v\_total\_balance

FROM accounts

WHERE customer\_id = p\_customer\_id;

RETURN NVL(v\_total\_balance, 0);

END GetTotalCustomerBalance;

END AccountOperations;

/