**PL/SQL Assignment Solutions**

**First, Create the Database Tables**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE

);

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Employees1 (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

INSERT INTO Customers VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

INSERT INTO Customers VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);

INSERT INTO Customers VALUES (3, 'Bob Wilson', TO\_DATE('1960-03-10', 'YYYY-MM-DD'), 15000, SYSDATE);

INSERT INTO Accounts VALUES (1, 1, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts VALUES (2, 2, 'Checking', 1500, SYSDATE);

INSERT INTO Accounts VALUES (3, 3, 'Savings', 15000, SYSDATE);

INSERT INTO Loans VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

INSERT INTO Loans VALUES (2, 3, 10000, 4, SYSDATE, ADD\_MONTHS(SYSDATE, 36));

INSERT INTO Employees1 VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees1 VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));

COMMIT;

**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

v\_age NUMBER;

BEGIN

FOR customer IN (SELECT CustomerID, DOB FROM Customers) LOOP

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, customer.DOB) / 12);

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = customer.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Discount given to Customer ' || customer.CustomerID);

END IF;

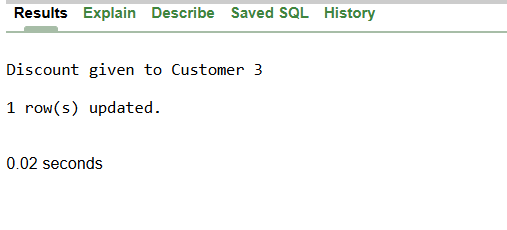
END LOOP;

COMMIT;

END;

/

**Output:**



**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.**

ALTER TABLE Customers ADD IsVIP VARCHAR2(1) DEFAULT 'N';

DECLARE

BEGIN

FOR customer IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF customer.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = customer.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Customer ' || customer.CustomerID || ' is now VIP');

END IF;

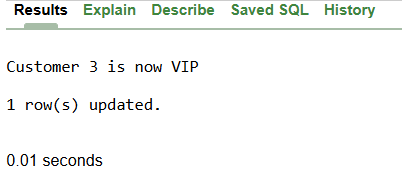
END LOOP;

COMMIT;

END;

/

**Output:**



**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

BEGIN

FOR loan IN (SELECT l.LoanID, c.Name, l.EndDate

FROM Loans l, Customers c

WHERE l.CustomerID = c.CustomerID

AND l.EndDate BETWEEN SYSDATE AND SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('REMINDER: ' || loan.Name ||

', your loan ' || loan.LoanID ||

' is due on ' || loan.EndDate);

END LOOP;

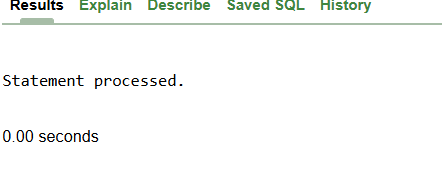
END;

/

**Output:**

REMINDER: John Doe, your loan 1 is due on 22-JUN-30

REMINDER: Bob Wilson, your loan 2 is due on 22-JUN-28



**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(

source\_account NUMBER,

target\_account NUMBER,

amount NUMBER

) IS

source\_balance NUMBER;

BEGIN

SELECT Balance INTO source\_balance

FROM Accounts

WHERE AccountID = source\_account;

IF source\_balance < amount THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Not enough money in account');

RETURN;

END IF;

UPDATE Accounts SET Balance = Balance - amount

WHERE AccountID = source\_account;

UPDATE Accounts SET Balance = Balance + amount

WHERE AccountID = target\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful: $' || amount);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Account does not exist');

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('ERROR: Transfer failed');

END;

/

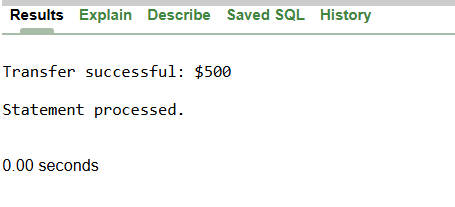
BEGIN

SafeTransferFunds(1, 2, 500);

END;

/

**Output:**



**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(

emp\_id NUMBER,

percentage NUMBER

) IS

current\_salary NUMBER;

BEGIN

SELECT Salary INTO current\_salary

FROM Employees1

WHERE EmployeeID = emp\_id;

UPDATE Employees1

SET Salary = current\_salary \* (1 + percentage/100)

WHERE EmployeeID = emp\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated for employee ' || emp\_id);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Employee ' || emp\_id || ' not found');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Cannot update salary');

END;

/

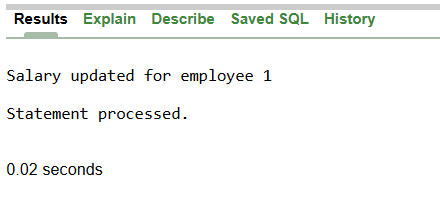
BEGIN

UpdateSalary(1, 10);

END;

/

**Output:**



**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(

cust\_id NUMBER,

cust\_name VARCHAR2,

birth\_date DATE,

initial\_balance NUMBER

) IS

BEGIN

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

VALUES (cust\_id, cust\_name, birth\_date, initial\_balance, SYSDATE);

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Customer ID ' || cust\_id || ' already exists');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Cannot add customer');

END;

/

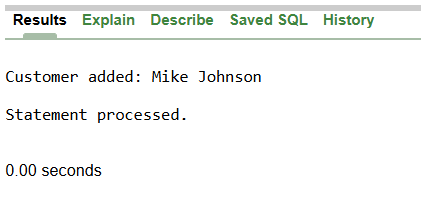
BEGIN

AddNewCustomer(4, 'Mike Johnson', TO\_DATE('1985-12-01', 'YYYY-MM-DD'), 2000);

END;

/

**Output:**



**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

FOR account IN (SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings') LOOP

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountID = account.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Interest added to Account ' || account.AccountID);

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processing done');

END;

/

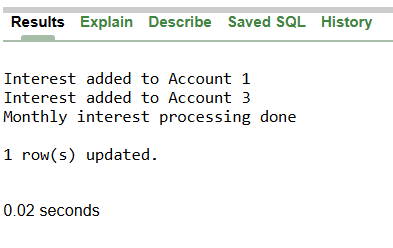
BEGIN

ProcessMonthlyInterest;

END;

/

**Output:**



**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(

dept\_name VARCHAR2,

bonus\_percent NUMBER

) IS

emp\_count NUMBER := 0;

BEGIN

FOR emp IN (SELECT EmployeeID, Name FROM Employees1 WHERE Department = dept\_name) LOOP

UPDATE Employees1

SET Salary = Salary + (Salary \* bonus\_percent/100)

WHERE EmployeeID = emp.EmployeeID;

emp\_count := emp\_count + 1;

DBMS\_OUTPUT.PUT\_LINE('Bonus given to ' || emp.Name);

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus given to ' || emp\_count || ' Employees1');

END;

/

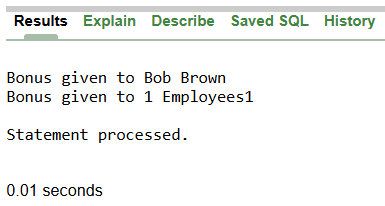
BEGIN

UpdateEmployeeBonus('IT', 5);

END;

/

**Output:**



**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(

from\_account NUMBER,

to\_account NUMBER,

transfer\_amount NUMBER

) IS

account\_balance NUMBER;

BEGIN

SELECT Balance INTO account\_balance FROM Accounts WHERE AccountID = from\_account;

IF account\_balance < transfer\_amount THEN

DBMS\_OUTPUT.PUT\_LINE('ERROR: Not enough money');

RETURN;

END IF;

UPDATE Accounts SET Balance = Balance - transfer\_amount WHERE AccountID = from\_account;

UPDATE Accounts SET Balance = Balance + transfer\_amount WHERE AccountID = to\_account;

COMMIT;

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

END;

/

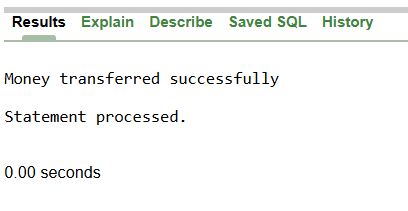
BEGIN

TransferFunds(3, 1, 1000);

END;

/

**Output:**



**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(birth\_date DATE) RETURN NUMBER IS

customer\_age NUMBER;

BEGIN

customer\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, birth\_date) / 12);

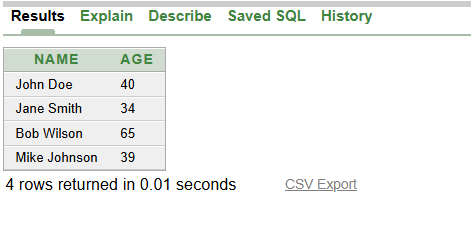
RETURN customer\_age;

END;

/

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

**Output:**



**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(

loan\_amount NUMBER,

yearly\_rate NUMBER,

years NUMBER

) RETURN NUMBER IS

monthly\_rate NUMBER;

total\_months NUMBER;

monthly\_payment NUMBER;

BEGIN

monthly\_rate := (yearly\_rate / 100) / 12;

total\_months := years \* 12;

IF monthly\_rate = 0 THEN

monthly\_payment := loan\_amount / total\_months;

ELSE

monthly\_payment := loan\_amount \*

(monthly\_rate \* POWER(1 + monthly\_rate, total\_months)) /

(POWER(1 + monthly\_rate, total\_months) - 1);

END IF;

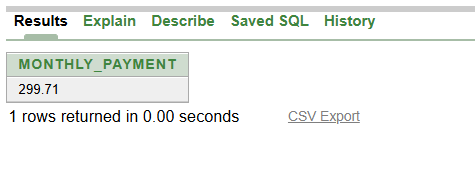
RETURN ROUND(monthly\_payment, 2);

END;

/

SELECT CalculateMonthlyInstallment(10000, 5, 3) AS Monthly\_Payment FROM DUAL;

**Output:**



**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(

acc\_id NUMBER,

needed\_amount NUMBER

) RETURN VARCHAR2 IS

acc\_balance NUMBER;

BEGIN

SELECT Balance INTO acc\_balance FROM Accounts WHERE AccountID = acc\_id;

IF acc\_balance >= needed\_amount THEN

RETURN 'YES';

ELSE

RETURN 'NO';

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

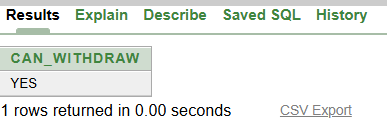
RETURN 'ACCOUNT NOT FOUND';

END;

/

SELECT HasSufficientBalance(1, 500) AS Can\_Withdraw FROM DUAL;

**Output:**



**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;

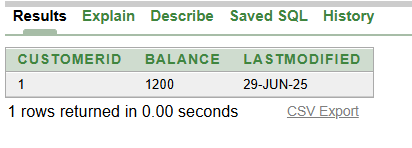
END;

/

UPDATE Customers SET Balance = 1200 WHERE CustomerID = 1;

SELECT CustomerID, Balance, LastModified FROM Customers WHERE CustomerID = 1;

**Output:**



**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.**

DROP TRIGGER LogTransaction;

DROP SEQUENCE audit\_seq;

DROP TABLE AuditLog;

CREATE TABLE AuditLog (

AuditID NUMBER PRIMARY KEY,

TransactionID NUMBER,

Action VARCHAR2(50),

LogTime DATE

);

CREATE SEQUENCE audit\_seq START WITH 1;

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog VALUES (audit\_seq.NEXTVAL, :NEW.TransactionID, 'NEW TRANSACTION', SYSDATE);

DBMS\_OUTPUT.PUT\_LINE('Transaction logged: ' || :NEW.TransactionID);

END;

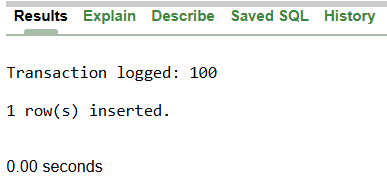
/

INSERT INTO Transactions VALUES (100, 1, SYSDATE, 250, 'Deposit');

SELECT \* FROM AuditLog;

SELECT \* FROM Transactions;

**Output:**



**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

acc\_balance NUMBER;

BEGIN

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

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > acc\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Not enough money for withdrawal');

END IF;

IF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN

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

END IF;

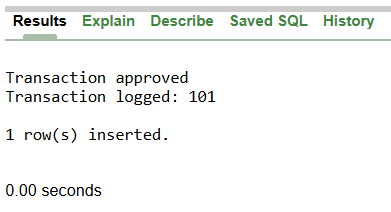
DBMS\_OUTPUT.PUT\_LINE('Transaction approved');

END;

/

INSERT INTO Transactions VALUES (101, 1, SYSDATE, 100, 'Deposit');

**Output:**



**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 monthly\_cursor IS

SELECT c.Name, t.TransactionDate, t.Amount, t.TransactionType

FROM Customers c, Accounts a, Transactions t

WHERE c.CustomerID = a.CustomerID

AND a.AccountID = t.AccountID

AND EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE);

BEGIN

DBMS\_OUTPUT.PUT\_LINE('=== MONTHLY STATEMENTS ===');

FOR stmt IN monthly\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || stmt.Name ||

', Date: ' || stmt.TransactionDate ||

', Type: ' || stmt.TransactionType ||

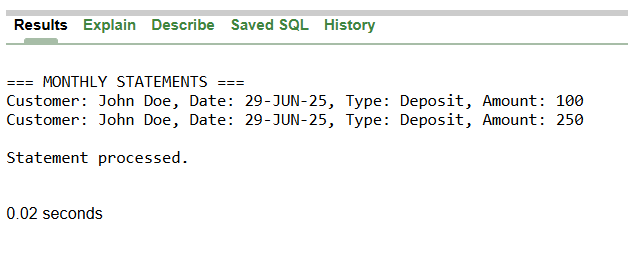
', Amount: ' || stmt.Amount);

END LOOP;

END;

/

**Output:**



**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 account\_cursor IS SELECT AccountID FROM Accounts;

annual\_fee NUMBER := 50;

BEGIN

FOR acc IN account\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = acc.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to Account ' || acc.AccountID);

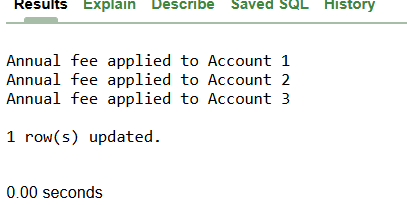
END LOOP;

COMMIT;

END;

/

**Output:**



**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 loan\_cursor IS SELECT LoanID, LoanAmount FROM Loans;

new\_rate NUMBER;

BEGIN

FOR loan IN loan\_cursor LOOP

IF loan.LoanAmount < 7000 THEN

new\_rate := 6;

ELSE

new\_rate := 4;

END IF;

UPDATE Loans SET InterestRate = new\_rate WHERE LoanID = loan.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Loan ' || loan.LoanID || ' rate updated to ' || new\_rate || '%');

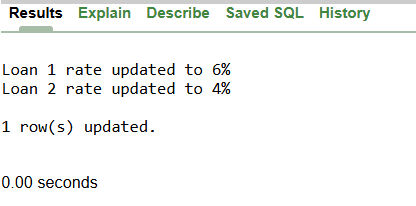
END LOOP;

COMMIT;

END;

/

**Output:**



**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 IS

PROCEDURE AddNewCustomer(cust\_id NUMBER, cust\_name VARCHAR2, birth\_date DATE, balance NUMBER);

PROCEDURE UpdateCustomerDetails(cust\_id NUMBER, cust\_name VARCHAR2, balance NUMBER);

FUNCTION GetCustomerBalance(cust\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

PROCEDURE AddNewCustomer(cust\_id NUMBER, cust\_name VARCHAR2, birth\_date DATE, balance NUMBER) IS

BEGIN

INSERT INTO Customers VALUES (cust\_id, cust\_name, birth\_date, balance, SYSDATE, 'N');

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(cust\_id NUMBER, cust\_name VARCHAR2, balance NUMBER) IS

BEGIN

UPDATE Customers SET Name = cust\_name, Balance = balance, LastModified = SYSDATE

WHERE CustomerID = cust\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer updated');

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(cust\_id NUMBER) RETURN NUMBER IS

cust\_balance NUMBER;

BEGIN

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

RETURN cust\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END GetCustomerBalance;

END CustomerManagement;

/

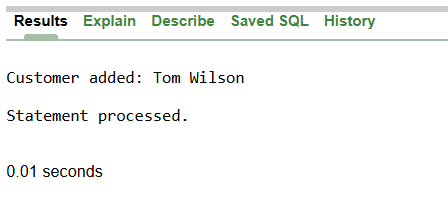
BEGIN

CustomerManagement.AddNewCustomer(5, 'Tom Wilson', TO\_DATE('1980-01-01', 'YYYY-MM-DD'), 3000);

END;

/

**Output:**



**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 IS

PROCEDURE HireNewEmployee(emp\_id NUMBER, emp\_name VARCHAR2, position VARCHAR2, salary NUMBER, dept VARCHAR2);

PROCEDURE UpdateEmployeeDetails(emp\_id NUMBER, emp\_name VARCHAR2, position VARCHAR2, salary NUMBER, dept VARCHAR2);

FUNCTION CalculateAnnualSalary(emp\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

PROCEDURE HireNewEmployee(emp\_id NUMBER, emp\_name VARCHAR2, position VARCHAR2, salary NUMBER, dept VARCHAR2) IS

BEGIN

INSERT INTO Employees1 VALUES (emp\_id, emp\_name, position, salary, dept, SYSDATE);

COMMIT;

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

END HireNewEmployee;

PROCEDURE UpdateEmployeeDetails(emp\_id NUMBER, emp\_name VARCHAR2, position VARCHAR2, salary NUMBER, dept VARCHAR2) IS

BEGIN

UPDATE Employees1 SET Name = emp\_name, Position = position, Salary = salary, Department = dept

WHERE EmployeeID = emp\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Employee details updated');

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(emp\_id NUMBER) RETURN NUMBER IS

monthly\_sal NUMBER;

BEGIN

SELECT Salary INTO monthly\_sal FROM Employees1 WHERE EmployeeID = emp\_id;

RETURN monthly\_sal \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END CalculateAnnualSalary;

END EmployeeManagement;

/

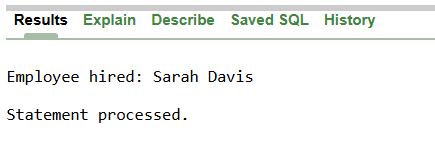
BEGIN

EmployeeManagement.HireNewEmployee(3, 'Sarah Davis', 'Analyst', 55000, 'Finance');

END;

/

**Output:**



**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 IS

PROCEDURE OpenNewAccount(acc\_id NUMBER, cust\_id NUMBER, acc\_type VARCHAR2, balance NUMBER);

PROCEDURE CloseAccount(acc\_id NUMBER);

FUNCTION GetTotalCustomerBalance(cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

PROCEDURE OpenNewAccount(acc\_id NUMBER, cust\_id NUMBER, acc\_type VARCHAR2, balance NUMBER) IS

BEGIN

INSERT INTO Accounts VALUES (acc\_id, cust\_id, acc\_type, balance, SYSDATE);

COMMIT;

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

END OpenNewAccount;

PROCEDURE CloseAccount(acc\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = acc\_id;

COMMIT;

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

END CloseAccount;

FUNCTION GetTotalCustomerBalance(cust\_id NUMBER) RETURN NUMBER IS

total\_balance NUMBER := 0;

BEGIN

SELECT SUM(Balance) INTO total\_balance FROM Accounts WHERE CustomerID = cust\_id;

RETURN NVL(total\_balance, 0);

END GetTotalCustomerBalance;

END AccountOperations;

/

BEGIN

AccountOperations.OpenNewAccount(4, 1, 'Checking', 500);

END;

/

**Output:**

