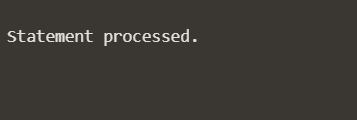
**Exercise 1: Control Structures**

**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 customer\_cursor IS          SELECT c.CustomerID, l.InterestRate          FROM Customers c          JOIN Loans l ON c.CustomerID = l.CustomerID          WHERE FLOOR(MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12) > 60;        v\_customer\_id Customers.CustomerID%TYPE;      v\_loan\_interest\_rate Loans.InterestRate%TYPE;    BEGIN      OPEN customer\_cursor;      LOOP          FETCH customer\_cursor INTO v\_customer\_id, v\_loan\_interest\_rate;          EXIT WHEN customer\_cursor%NOTFOUND;          v\_loan\_interest\_rate := v\_loan\_interest\_rate - 1;          UPDATE Loans          SET InterestRate = v\_loan\_interest\_rate          WHERE CustomerID = v\_customer\_id;            DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount to customer ' || v\_customer\_id || ' new interest rate is ' || v\_loan\_interest\_rate);      END LOOP;      CLOSE customer\_cursor;        COMMIT;  END;  / |

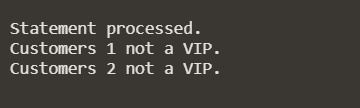


Output 🡪

**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.

|  |
| --- |
| DECLARE  CURSOR customer\_cursor IS  SELECT CustomerID, Balance  FROM Customers;    v\_customer\_id Customers.CustomerID%TYPE;  v\_balance Customers.Balance%TYPE;    BEGIN  OPEN customer\_cursor;  LOOP  FETCH customer\_cursor INTO v\_customer\_id, v\_balance;  EXIT WHEN customer\_cursor%NOTFOUND;    IF v\_balance > 10000 THEN  UPDATE Customers  SET IsVIP = 'T'  WHERE CustomerID = v\_customer\_id;    DBMS\_OUTPUT.PUT\_LINE('Customers ' || v\_customer\_id || ' promoted to VIP.');  ELSE  UPDATE Customers  SET IsVIP = 'F'  WHERE CustomerID = v\_customer\_id;    DBMS\_OUTPUT.PUT\_LINE('Customers ' || v\_customer\_id || ' not a VIP.');  END IF;  END LOOP;  CLOSE customer\_cursor;    COMMIT;  END;  / |

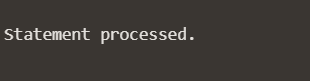


Output 🡪

**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  CURSOR loan\_cursor IS  SELECT l.LoanID, l.CustomerID, c.Name, l.EndDate  FROM Loans l  JOIN Customers c ON l.CustomerID = c.CustomerID  WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;    v\_loan\_id Loans.LoanID%TYPE;  v\_customer\_id Loans.CustomerID%TYPE;  v\_customer\_name Customers.Name%TYPE;  v\_due\_date Loans.EndDate%TYPE;    BEGIN  OPEN loan\_cursor;  LOOP  FETCH loan\_cursor INTO v\_loan\_id, v\_customer\_id, v\_customer\_name, v\_due\_date;  EXIT WHEN loan\_cursor%NOTFOUND;    DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || v\_loan\_id || ' for customer ' || v\_customer\_name || ' (ID: ' || v\_customer\_id || ') is due on ' || TO\_CHAR(v\_due\_date, 'YYYY-MM-DD'));  END LOOP;  CLOSE loan\_cursor;  END;  / |



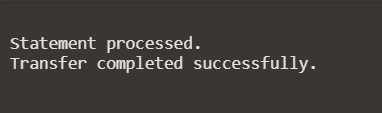
Output 🡪

**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  ) AS  v\_from\_balance NUMBER;  BEGIN  IF p\_amount <= 0 THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Transfer amount must be greater than zero.');  END IF;  SELECT Balance INTO v\_from\_balance  FROM Accounts  WHERE AccountID = p\_from\_account\_id  FOR UPDATE;  IF v\_from\_balance < p\_amount THEN  RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds for transfer.');  END IF;  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;  DBMS\_OUTPUT.PUT\_LINE('Transfer completed successfully.');  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('Error: One of the account IDs does not exist.');  WHEN OTHERS THEN  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('Error during fund transfer: ' || SQLERRM);  END SafeTransferFunds;  /  --Executing the procedure  BEGIN  SafeTransferFunds(2, 6, 1000);  END;  / |

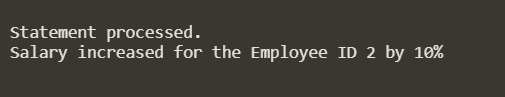
****

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 (  p\_employee\_id IN NUMBER,  p\_percentage IN NUMBER  ) AS  BEGIN  IF p\_percentage <= 0 THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Percentage increase must be greater than zero.');  END IF;  UPDATE Employees  SET Salary = Salary \* (1 + p\_percentage / 100)  WHERE EmployeeID = p\_employee\_id;  IF SQL%ROWCOUNT = 0 THEN  RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID is not there.');  END IF;  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('Salary increased for the Employee ID ' || p\_employee\_id || ' by ' || p\_percentage || '%');  EXCEPTION  WHEN OTHERS THEN  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('Error during salary update: ' || SQLERRM);  END UpdateSalary;  /  BEGIN  UpdateSalary(2, 10);  END;  / |

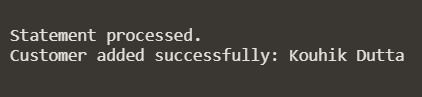
****

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 (  p\_customer\_id IN NUMBER,  p\_name IN VARCHAR2,  p\_dob IN DATE,  p\_balance IN NUMBER  ) AS  BEGIN  INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('Customer added successfully: ' || p\_name);  EXCEPTION  WHEN DUP\_VAL\_ON\_INDEX THEN  DBMS\_OUTPUT.PUT\_LINE('Error: Customer with ID ' || p\_customer\_id || ' already exists.');  ROLLBACK;  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error during customer insertion: ' || SQLERRM);  ROLLBACK;  END AddNewCustomer;  /  --Adding a new Customer Koushik Dutta at Employee Id 3  BEGIN  AddNewCustomer(3, 'Kouhik Dutta', TO\_DATE('2004-05-14', 'YYYY-MM-DD'), 1000);  END;  / |



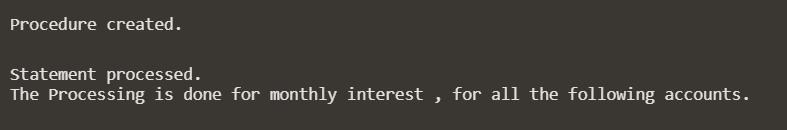
Output🡪

**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 AS  BEGIN  UPDATE Accounts  SET Balance = Balance \* 1.01,  LastModified = SYSDATE  WHERE AccountType = 'Savings';  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('The Processing is done for monthly interest , for all the following accounts.');  EXCEPTION  WHEN OTHERS THEN  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('Error found: ' || SQLERRM);  END ProcessMonthlyInterest;  /  BEGIN  ProcessMonthlyInterest;  END;  / |

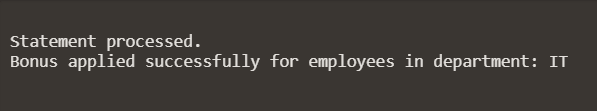


Output 🡪

**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 (  p\_department IN VARCHAR2,  p\_bonus\_percentage IN NUMBER  ) AS  BEGIN  IF p\_bonus\_percentage <= 0 THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Bonus percentage must be greater than zero.');  END IF;  UPDATE Employees  SET Salary = Salary \* (1 + p\_bonus\_percentage / 100)  WHERE Department = p\_department;  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('Bonus applied successfully for employees in department: ' || p\_department);  EXCEPTION  WHEN OTHERS THEN  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('Error during bonus update: ' || SQLERRM);  END UpdateEmployeeBonus;  /  --Applying 100% to employees working in IT  BEGIN  UpdateEmployeeBonus('IT', 100);  END;  / |

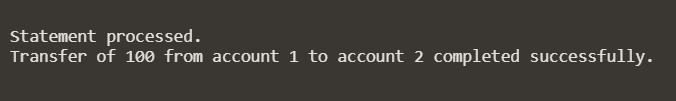


Output 🡪

**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 (  p\_from\_account\_id IN NUMBER,  p\_to\_account\_id IN NUMBER,  p\_amount IN NUMBER  ) AS  v\_from\_balance NUMBER;  BEGIN  IF p\_amount <= 0 THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Transfer amount must be greater than zero.');  END IF;  SELECT Balance INTO v\_from\_balance  FROM Accounts  WHERE AccountID = p\_from\_account\_id  FOR UPDATE;  IF v\_from\_balance < p\_amount THEN  RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds in the source account.');  END IF;  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;  DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' from account ' || p\_from\_account\_id || ' to account ' || p\_to\_account\_id || ' completed successfully.');  EXCEPTION  WHEN OTHERS THEN  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('Error during fund transfer: ' || SQLERRM);  END TransferFunds;  /  --Sending 100 From Account 1 to 2  BEGIN  TransferFunds(1, 2, 100);  END;  / |



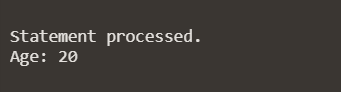
Output 🡪

**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 (  p\_dob IN DATE  ) RETURN NUMBER  IS  v\_age NUMBER;  v\_current\_date DATE := SYSDATE;  BEGIN  v\_age := FLOOR(MONTHS\_BETWEEN(v\_current\_date, p\_dob) / 12);  RETURN v\_age;  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error calculating age: ' || SQLERRM);  RETURN NULL;  END CalculateAge;  /  DECLARE  v\_dob DATE := TO\_DATE('2004-05-14', 'YYYY-MM-DD');  v\_age NUMBER;  BEGIN  v\_age := CalculateAge(v\_dob);  DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);  END;  / |

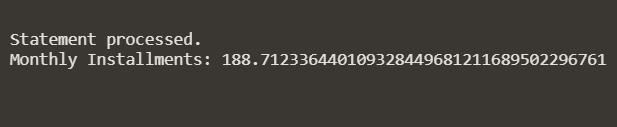


Output🡪

**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 (  p\_loan\_amount IN NUMBER,  p\_annual\_interest\_rate IN NUMBER,  p\_duration\_years IN NUMBER  ) RETURN NUMBER  IS  v\_monthly\_interest\_rate NUMBER;  v\_total\_installments NUMBER;  v\_installment NUMBER;  BEGIN  v\_monthly\_interest\_rate := p\_annual\_interest\_rate / 100 / 12;  v\_total\_installments := p\_duration\_years \* 12;  IF v\_monthly\_interest\_rate = 0 THEN  v\_installment := p\_loan\_amount / v\_total\_installments;  ELSE  v\_installment := p\_loan\_amount \* (v\_monthly\_interest\_rate + (v\_monthly\_interest\_rate / (POWER(1 + v\_monthly\_interest\_rate, v\_total\_installments) - 1)));  END IF;  RETURN v\_installment;  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error calculating monthly installment: ' || SQLERRM);  RETURN NULL;  END CalculateMonthlyInstallment;  /  --Calculating Monthly Instalments  DECLARE  v\_loan\_amount NUMBER := 10000;  v\_annual\_interest\_rate NUMBER := 5;  v\_duration\_years NUMBER := 5;  v\_installment NUMBER;  BEGIN  v\_installment := CalculateMonthlyInstallment(v\_loan\_amount, v\_annual\_interest\_rate, v\_duration\_years);  DBMS\_OUTPUT.PUT\_LINE('Monthly Installments: ' || v\_installment);  END;  / |

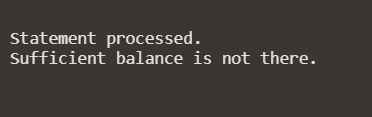


Output 🡪

**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 (  p\_account\_id IN NUMBER,  p\_amount IN NUMBER  ) RETURN BOOLEAN  IS  v\_balance NUMBER;  BEGIN  SELECT Balance INTO v\_balance  FROM Accounts  WHERE AccountID = p\_account\_id;    IF v\_balance >= p\_amount THEN  RETURN TRUE;  ELSE  RETURN FALSE;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  -- Account does not exist  RETURN FALSE;  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error checking balance: ' || SQLERRM);  RETURN FALSE;  END HasSufficientBalance;  /  v\_account\_id NUMBER := 3;  v\_amount NUMBER := 1000;  v\_result BOOLEAN;  BEGIN  v\_result := HasSufficientBalance(v\_account\_id, v\_amount);  IF v\_result THEN  DBMS\_OUTPUT.PUT\_LINE('Sufficient balance is there.');  ELSE  DBMS\_OUTPUT.PUT\_LINE('Sufficient balance is not there.');  END IF;  END;  / |



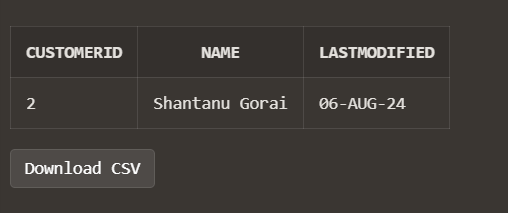
Output 🡪

**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;  /  --Updating Record to a new customer Shantanu Gorai  UPDATE Customers  SET Name = 'Shantanu Gorai'  WHERE CustomerID = 2;  --To Check for Last Modified  SELECT CustomerID, Name, LastModified  FROM Customers  WHERE CustomerID = 2; |

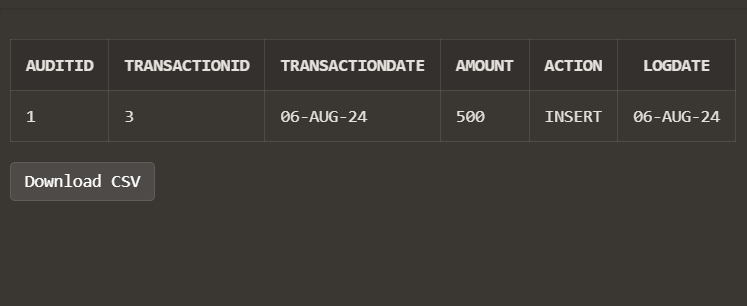


Output 🡪

**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.

|  |
| --- |
| --Creating the audit log First  CREATE TABLE AuditLog (  AuditID NUMBER PRIMARY KEY,  TransactionID NUMBER,  TransactionDate DATE,  Amount NUMBER,  Action VARCHAR2(10),  LogDate DATE  );  --Creating the Sequence  CREATE SEQUENCE AuditLog\_SEQ  START WITH 1  INCREMENT BY 1  NOCACHE  NOCYCLE;  --Creating the trigger  CREATE OR REPLACE TRIGGER LogTransaction  AFTER INSERT ON Transactions  FOR EACH ROW  DECLARE  v\_audit\_id NUMBER;  BEGIN  SELECT AuditLog\_SEQ.NEXTVAL INTO v\_audit\_id FROM DUAL;  INSERT INTO AuditLog (  AuditID,  TransactionID,  TransactionDate,  Amount,  Action,  LogDate  ) VALUES (  v\_audit\_id, -- Use the value generated by the sequence  :NEW.TransactionID, -- TransactionID from the newly inserted row  :NEW.TransactionDate, -- TransactionDate from the newly inserted row  :NEW.Amount, -- Amount from the newly inserted row  'INSERT', -- Action type  SYSDATE -- LogDate as the current date  );  EXCEPTION  WHEN DUP\_VAL\_ON\_INDEX THEN  -- Handle the unique constraint violation  DBMS\_OUTPUT.PUT\_LINE('Duplicate value error: AuditID must be unique.');  WHEN OTHERS THEN  -- Handle other exceptions  DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);  END;  /  --Testing the trigger  BEGIN  -- Insert a new transaction into the Transactions table  INSERT INTO Transactions (  TransactionID,  AccountID,  TransactionDate,  Amount,  TransactionType  ) VALUES (  TransactionID\_SEQ.NEXTVAL, -- Use a unique value for TransactionID  1, -- Example AccountID  SYSDATE, -- Current date as TransactionDate  500, -- Example Amount  'Deposit' -- Example TransactionType  );  COMMIT;  -- Output a confirmation message  DBMS\_OUTPUT.PUT\_LINE('New transaction inserted and audit log updated.');  END;  /  --Finally Printing it  SELECT \* FROM AuditLog; |

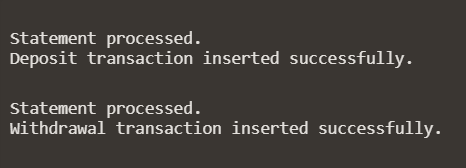


Output 🡪

**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  v\_current\_balance NUMBER;  BEGIN  -- First Finding the Current Balance  SELECT Balance INTO v\_current\_balance  FROM Accounts  WHERE AccountID = :NEW.AccountID;  -- Checking for deposit rule  IF :NEW.TransactionType = 'Deposit' THEN  IF :NEW.Amount <= 0 THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be positive.');  END IF;  -- Checking for withdrawal rule  ELSIF :NEW.TransactionType = 'Withdrawal' THEN  IF :NEW.Amount > v\_current\_balance THEN  RAISE\_APPLICATION\_ERROR(-20002, 'Withdrawal amount exceeds current balance.');  END IF;  ELSE  RAISE\_APPLICATION\_ERROR(-20003, 'Invalid transaction type.');  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RAISE\_APPLICATION\_ERROR(-20004, 'Account not found.');  WHEN OTHERS THEN  RAISE\_APPLICATION\_ERROR(-20005, 'An unexpected error occurred: ' || SQLERRM);  END;  /  --Testing the Trigger  BEGIN  INSERT INTO Transactions (  TransactionID,  AccountID,  TransactionDate,  Amount,  TransactionType  ) VALUES (  TransactionID\_SEQ.NEXTVAL, -- Ensure a unique TransactionID  1, -- Example AccountID  SYSDATE, -- Current date as TransactionDate  100, -- Example Deposit Amount  'Deposit' -- TransactionType  );  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('Deposit transaction inserted successfully.');  END;  /  -- Checking for valid Withdrawal  BEGIN  INSERT INTO Transactions (  TransactionID,  AccountID,  TransactionDate,  Amount,  TransactionType  ) VALUES (  TransactionID\_SEQ.NEXTVAL, -- Ensure a unique TransactionID  1, -- Example AccountID  SYSDATE, -- Current date as TransactionDate  50, -- Example Withdrawal Amount  'Withdrawal' -- TransactionType  );  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('Withdrawal transaction inserted successfully.');  END;  /  -- Checking for invalid withdrawal  BEGIN  INSERT INTO Transactions (  TransactionID,  AccountID,  TransactionDate,  Amount,  TransactionType  ) VALUES (  TransactionID\_SEQ.NEXTVAL, -- Ensure a unique TransactionID  1, -- Example AccountID  SYSDATE, -- Current date as TransactionDate  5000, -- Withdrawal Amount exceeding balance  'Withdrawal' -- TransactionType  );  COMMIT;  DBMS\_OUTPUT.PUT\_LINE('Withdrawal transaction inserted successfully.');  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE(SQLERRM);  END;  / |



Output 🡪

**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers.

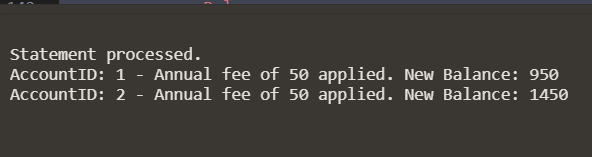
**Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

|  |
| --- |
| DECLARE  CURSOR c\_monthly\_transactions IS  SELECT t.TransactionID,  t.AccountID,  t.TransactionDate,  t.Amount,  t.TransactionType,  a.CustomerID,  c.Name AS CustomerName  FROM Transactions t  JOIN Accounts a ON t.AccountID = a.AccountID  JOIN Customers c ON a.CustomerID = c.CustomerID  WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)  AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);  r\_transaction c\_monthly\_transactions%ROWTYPE;  BEGIN  -- Open the cursor  OPEN c\_monthly\_transactions;  LOOP  FETCH c\_monthly\_transactions INTO r\_transaction;  EXIT WHEN c\_monthly\_transactions%NOTFOUND;  -- Print the statement for each customer  DBMS\_OUTPUT.PUT\_LINE('Statement for Customer: ' || r\_transaction.CustomerName);  DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || r\_transaction.TransactionID);  DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || r\_transaction.AccountID);  DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || TO\_CHAR(r\_transaction.TransactionDate, 'YYYY-MM-DD'));  DBMS\_OUTPUT.PUT\_LINE('Amount: ' || r\_transaction.Amount);  DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || r\_transaction.TransactionType);  DBMS\_OUTPUT.PUT\_LINE('-----------------------------------');  END LOOP;  -- Close the cursor  CLOSE c\_monthly\_transactions;  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);  END;  Output 🡪  Statement processed. Statement for Customer: John Doe Transaction ID: 4 Account ID: 1 Transaction Date: 2024-08-06 Amount: 100 Transaction Type: Deposit ----------------------------------- Statement for Customer: John Doe Transaction ID: 5 Account ID: 1 Transaction Date: 2024-08-06 Amount: 50 Transaction Type: Withdrawal ----------------------------------- Statement for Customer: John Doe Transaction ID: 1 Account ID: 1 Transaction Date: 2024-08-06 Amount: 200 Transaction Type: Deposit ----------------------------------- Statement for Customer: Jane Smith Transaction ID: 2 Account ID: 2 Transaction Date: 2024-08-06 Amount: 300 Transaction Type: Withdrawal ----------------------------------- Statement for Customer: John Doe Transaction ID: 3 Account ID: 1 Transaction Date: 2024-08-06 Amount: 500 Transaction Type: Deposit -----------------------------------  / |

**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 to retrieve all accounts  CURSOR c\_accounts IS  SELECT AccountID,  Balance  FROM Accounts;  r\_account c\_accounts%ROWTYPE;  annual\_fee NUMBER := 50; -- Example fee amount, adjust as needed  BEGIN    OPEN c\_accounts;  LOOP  FETCH c\_accounts INTO r\_account;  EXIT WHEN c\_accounts%NOTFOUND;  UPDATE Accounts  SET Balance = Balance - annual\_fee  WHERE AccountID = r\_account.AccountID;  DBMS\_OUTPUT.PUT\_LINE('AccountID: ' || r\_account.AccountID ||  ' - Annual fee of ' || annual\_fee ||  ' applied. New Balance: ' ||  (r\_account.Balance - annual\_fee));  END LOOP;  CLOSE c\_accounts;  COMMIT;  EXCEPTION  WHEN OTHERS THEN  -- Rollback changes in case of an error  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);  END;  / |

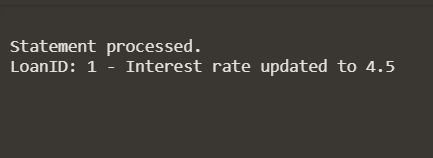


Output 🡪

**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 c\_loans IS  SELECT LoanID,  InterestRate  FROM Loans;  r\_loan c\_loans%ROWTYPE;  -- Defining the new interest rates to the policy  new\_interest\_rate NUMBER := 4.5; -- Example new interest rate, adjust as needed  BEGIN  -- Open the cursor  OPEN c\_loans;  -- Looping through the record using the cursor  LOOP  FETCH c\_loans INTO r\_loan;  EXIT WHEN c\_loans%NOTFOUND;  UPDATE Loans  SET InterestRate = new\_interest\_rate  WHERE LoanID = r\_loan.LoanID;  DBMS\_OUTPUT.PUT\_LINE('LoanID: ' || r\_loan.LoanID ||  ' - Interest rate updated to ' || new\_interest\_rate);  END LOOP;  CLOSE c\_loans;  COMMIT;  EXCEPTION  WHEN OTHERS THEN  -- Rollback changes in case of an error  ROLLBACK;  DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);  END;  / |



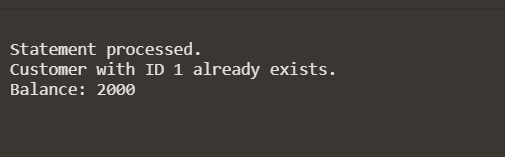
Output 🡪

**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 AS  PROCEDURE AddNewCustomer(  p\_CustomerID IN NUMBER,  p\_Name IN VARCHAR2,  p\_DOB IN DATE,  p\_Balance IN NUMBER  );  PROCEDURE UpdateCustomerDetails(  p\_CustomerID IN NUMBER,  p\_Name IN VARCHAR2,  p\_Balance IN NUMBER  );  FUNCTION GetCustomerBalance(  p\_CustomerID IN NUMBER  ) RETURN NUMBER;  END CustomerManagement;  /  --Creating the body Customer management  CREATE OR REPLACE PACKAGE BODY CustomerManagement AS  PROCEDURE AddNewCustomer(  p\_CustomerID IN NUMBER,  p\_Name IN VARCHAR2,  p\_DOB IN DATE,  p\_Balance IN NUMBER  ) IS  BEGIN  BEGIN  INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);  COMMIT;  EXCEPTION  WHEN DUP\_VAL\_ON\_INDEX THEN  DBMS\_OUTPUT.PUT\_LINE('Customer with ID ' || p\_CustomerID || ' already exists.');  END;  END AddNewCustomer;  PROCEDURE UpdateCustomerDetails(  p\_CustomerID IN NUMBER,  p\_Name IN VARCHAR2,  p\_Balance IN NUMBER  ) IS  BEGIN  BEGIN  UPDATE Customers  SET Name = p\_Name,  Balance = p\_Balance,  LastModified = SYSDATE  WHERE CustomerID = p\_CustomerID;  IF SQL%ROWCOUNT = 0 THEN  DBMS\_OUTPUT.PUT\_LINE('Customer with ID ' || p\_CustomerID || ' does not exist.');  ELSE  COMMIT;  END IF;  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error updating customer details: ' || SQLERRM);  END;  END UpdateCustomerDetails;  FUNCTION GetCustomerBalance(  p\_CustomerID IN NUMBER  ) RETURN NUMBER IS  v\_Balance NUMBER;  BEGIN  BEGIN  SELECT Balance INTO v\_Balance  FROM Customers  WHERE CustomerID = p\_CustomerID;  RETURN v\_Balance;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  DBMS\_OUTPUT.PUT\_LINE('Customer with ID ' || p\_CustomerID || ' does not exist.');  RETURN NULL;  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error retrieving customer balance: ' || SQLERRM);  RETURN NULL;  END;  END GetCustomerBalance;  END CustomerManagement;  /  --Testing the package  BEGIN  CustomerManagement.AddNewCustomer(1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1500);  CustomerManagement.UpdateCustomerDetails(1, 'John Doe', 2000);  DBMS\_OUTPUT.PUT\_LINE('Balance: ' || CustomerManagement.GetCustomerBalance(1));  END;  / |

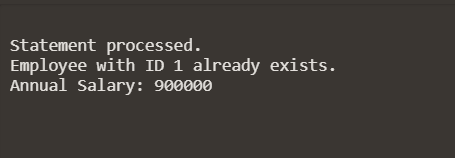


Output 🡪

**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 AS  PROCEDURE HireEmployee(  p\_EmployeeID IN NUMBER,  p\_Name IN VARCHAR2,  p\_Position IN VARCHAR2,  p\_Salary IN NUMBER,  p\_Department IN VARCHAR2  );  PROCEDURE UpdateEmployeeDetails(  p\_EmployeeID IN NUMBER,  p\_Name IN VARCHAR2,  p\_Position IN VARCHAR2,  p\_Salary IN NUMBER,  p\_Department IN VARCHAR2  );  FUNCTION CalculateAnnualSalary(  p\_EmployeeID IN NUMBER  ) RETURN NUMBER;  END EmployeeManagement;  /  CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS  PROCEDURE HireEmployee(  p\_EmployeeID IN NUMBER,  p\_Name IN VARCHAR2,  p\_Position IN VARCHAR2,  p\_Salary IN NUMBER,  p\_Department IN VARCHAR2  ) IS  BEGIN  BEGIN  INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)  VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, SYSDATE);  COMMIT;  EXCEPTION  WHEN DUP\_VAL\_ON\_INDEX THEN  DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || p\_EmployeeID || ' already exists.');  END;  END HireEmployee;  PROCEDURE UpdateEmployeeDetails(  p\_EmployeeID IN NUMBER,  p\_Name IN VARCHAR2,  p\_Position IN VARCHAR2,  p\_Salary IN NUMBER,  p\_Department IN VARCHAR2  ) IS  BEGIN  BEGIN  UPDATE Employees  SET Name = p\_Name,  Position = p\_Position,  Salary = p\_Salary,  Department = p\_Department,  HireDate = SYSDATE  WHERE EmployeeID = p\_EmployeeID;  IF SQL%ROWCOUNT = 0 THEN  DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || p\_EmployeeID || ' does not exist.');  ELSE  COMMIT;  END IF;  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error updating employee details: ' || SQLERRM);  END;  END UpdateEmployeeDetails;  FUNCTION CalculateAnnualSalary(  p\_EmployeeID IN NUMBER  ) RETURN NUMBER IS  v\_Salary NUMBER;  v\_AnnualSalary NUMBER;  BEGIN  BEGIN  SELECT Salary INTO v\_Salary  FROM Employees  WHERE EmployeeID = p\_EmployeeID;  v\_AnnualSalary := v\_Salary \* 12;  RETURN v\_AnnualSalary;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || p\_EmployeeID || ' does not exist.');  RETURN NULL;  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error calculating annual salary: ' || SQLERRM);  RETURN NULL;  END;  END CalculateAnnualSalary;  END EmployeeManagement;  /  BEGIN  EmployeeManagement.HireEmployee(1, 'Alice Johnson', 'Manager', 70000, 'HR');  EmployeeManagement.UpdateEmployeeDetails(1, 'Alice Johnson', 'Senior Manager', 75000, 'HR');  DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || EmployeeManagement.CalculateAnnualSalary(1));  END;  / |

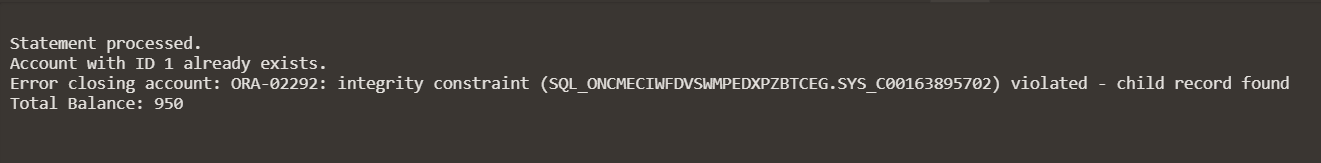


Output 🡪

**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 AS  PROCEDURE OpenAccount(  p\_AccountID IN NUMBER,  p\_CustomerID IN NUMBER,  p\_AccountType IN VARCHAR2,  p\_Balance IN NUMBER  );  PROCEDURE CloseAccount(  p\_AccountID IN NUMBER  );  FUNCTION GetTotalBalance(  p\_CustomerID IN NUMBER  ) RETURN NUMBER;  END AccountOperations;  /  CREATE OR REPLACE PACKAGE BODY AccountOperations AS  PROCEDURE OpenAccount(  p\_AccountID IN NUMBER,  p\_CustomerID IN NUMBER,  p\_AccountType IN VARCHAR2,  p\_Balance IN NUMBER  ) IS  BEGIN  BEGIN  INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)  VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);  COMMIT;  EXCEPTION  WHEN DUP\_VAL\_ON\_INDEX THEN  DBMS\_OUTPUT.PUT\_LINE('Account with ID ' || p\_AccountID || ' already exists.');  END;  END OpenAccount;  PROCEDURE CloseAccount(  p\_AccountID IN NUMBER  ) IS  BEGIN  BEGIN  DELETE FROM Accounts  WHERE AccountID = p\_AccountID;  IF SQL%ROWCOUNT = 0 THEN  DBMS\_OUTPUT.PUT\_LINE('Account with ID ' || p\_AccountID || ' does not exist.');  ELSE  COMMIT;  END IF;  EXCEPTION  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error closing account: ' || SQLERRM);  END;  END CloseAccount;  FUNCTION GetTotalBalance(  p\_CustomerID IN NUMBER  ) RETURN NUMBER IS  v\_TotalBalance NUMBER;  BEGIN  BEGIN  SELECT SUM(Balance) INTO v\_TotalBalance  FROM Accounts  WHERE CustomerID = p\_CustomerID;  RETURN NVL(v\_TotalBalance, 0); -- Return 0 if no accounts found  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  DBMS\_OUTPUT.PUT\_LINE('Customer with ID ' || p\_CustomerID || ' does not exist.');  RETURN 0;  WHEN OTHERS THEN  DBMS\_OUTPUT.PUT\_LINE('Error retrieving total balance: ' || SQLERRM);  RETURN 0;  END;  END GetTotalBalance;  END AccountOperations;  /  BEGIN  AccountOperations.OpenAccount(1, 1, 'Savings', 1000);  AccountOperations.CloseAccount(1);  DBMS\_OUTPUT.PUT\_LINE('Total Balance: ' || AccountOperations.GetTotalBalance(1));  END;  / |



Output 🡪