| **Exercise 1: Control Structures — All 3 Scenarios Combined** This script includes:  1. CustomerDiscount: apply 1% discount to loan interest for customers above 60 years old  2. VIPPromotion: promote customers to VIP status if balance exceeds ₹10,000  3. LoanReminder: print reminders for loans due within the next 30 days  **Code:**  -- Table Creation  CREATE TABLE customers (  customerid SERIAL PRIMARY KEY,  name VARCHAR(100),  dob DATE,  balance NUMERIC,  lastmodified DATE  );  CREATE TABLE loans (  loanid SERIAL PRIMARY KEY,  customerid INTEGER REFERENCES customers(customerid),  loanamount NUMERIC,  interestrate NUMERIC,  startdate DATE,  enddate DATE  );  -- Sample Data  INSERT INTO customers (name, dob, balance, lastmodified)  VALUES  ('John Senior', '1950-01-01', 5000, CURRENT\_DATE),  ('Alice Young', '1995-06-10', 7000, CURRENT\_DATE);  INSERT INTO loans (customerid, loanamount, interestrate, startdate, enddate)  VALUES  (1, 10000, 8, CURRENT\_DATE, CURRENT\_DATE + INTERVAL '1 year'),  (2, 8000, 7, CURRENT\_DATE, CURRENT\_DATE + INTERVAL '1 year');  -- Apply 1% Discount for Customers Above 60  DO  $$  DECLARE  cust RECORD;  BEGIN  FOR cust IN  SELECT c.customerid, c.name,  DATE\_PART('year', AGE(c.dob)) AS age,  l.loanid, l.interestrate  FROM customers c  JOIN loans l ON c.customerid = l.customerid  LOOP  IF cust.age > 60 THEN  UPDATE loans  SET interestrate = interestrate - 1  WHERE loanid = cust.loanid;  RAISE NOTICE 'Discount applied for Customer: %', cust.name;  END IF;  END LOOP;  END  $$;  -- Add IsVIP column  ALTER TABLE customers ADD COLUMN isvip VARCHAR(5);  -- Set IsVIP = TRUE for balance > 10000  DO  $$  DECLARE  cust RECORD;  BEGIN  FOR cust IN (SELECT customerid, balance FROM customers) LOOP  IF cust.balance > 10000 THEN  UPDATE customers  SET isvip = 'TRUE'  WHERE customerid = cust.customerid;  RAISE NOTICE 'Customer % promoted to VIP.', cust.customerid;  END IF;  END LOOP;  END  $$;  **Output:**   **Exercise 2: Error Handling — All 3 Scenarios Combined** This script includes:   1. A safe fund transfer between accounts with error handling. 2. A salary update procedure with missing employee ID detection. 3. A customer insertion procedure that prevents duplicate IDs.   **Code:**  -- Step 1: Create required tables  CREATE TABLE customers (  customerid INT PRIMARY KEY,  name VARCHAR(100),  dob DATE,  balance NUMERIC,  lastmodified DATE  );  CREATE TABLE accounts (  accountid SERIAL PRIMARY KEY,  customerid INT REFERENCES customers(customerid),  balance NUMERIC  );  CREATE TABLE employees (  employeeid INT PRIMARY KEY,  name VARCHAR(100),  salary NUMERIC,  department VARCHAR(50)  );  -- Step 2: Sample data  INSERT INTO customers VALUES  (1, 'John Doe', '1980-01-01', 5000, CURRENT\_DATE),  (2, 'Jane Smith', '1990-01-01', 6000, CURRENT\_DATE);  INSERT INTO accounts (customerid, balance) VALUES  (1, 3000),  (2, 2000);  INSERT INTO employees VALUES  (101, 'Alice', 70000, 'IT'),  (102, 'Bob', 50000, 'HR');  -- Step 3: SafeTransferFunds procedure  CREATE OR REPLACE FUNCTION SafeTransferFunds(p\_from INT, p\_to INT, p\_amount NUMERIC) RETURNS VOID AS  $$  DECLARE  v\_from\_balance NUMERIC;  BEGIN  SELECT balance INTO v\_from\_balance FROM accounts WHERE accountid = p\_from FOR UPDATE;  IF v\_from\_balance < p\_amount THEN  RAISE EXCEPTION 'Insufficient funds in account %', p\_from;  END IF;  UPDATE accounts SET balance = balance - p\_amount WHERE accountid = p\_from;  UPDATE accounts SET balance = balance + p\_amount WHERE accountid = p\_to;  RAISE NOTICE 'Funds transferred successfully.';  EXCEPTION  WHEN OTHERS THEN  RAISE NOTICE 'Transfer failed: %', SQLERRM;  END;  $$ LANGUAGE plpgsql;  -- Step 4: UpdateSalary procedure  CREATE OR REPLACE FUNCTION UpdateSalary(p\_emp\_id INT, p\_percent NUMERIC) RETURNS VOID AS  $$  BEGIN  UPDATE employees  SET salary = salary + (salary \* p\_percent / 100)  WHERE employeeid = p\_emp\_id;  IF NOT FOUND THEN  RAISE EXCEPTION 'Employee ID % not found.', p\_emp\_id;  ELSE  RAISE NOTICE 'Salary updated for employee %.', p\_emp\_id;  END IF;  END;  $$ LANGUAGE plpgsql;  -- Step 5: AddNewCustomer procedure  CREATE OR REPLACE FUNCTION AddNewCustomer(p\_id INT, p\_name TEXT, p\_dob DATE, p\_balance NUMERIC) RETURNS VOID AS  $$  BEGIN  INSERT INTO customers (customerid, name, dob, balance, lastmodified)  VALUES (p\_id, p\_name, p\_dob, p\_balance, CURRENT\_DATE);  RAISE NOTICE 'Customer % added successfully.', p\_name;  EXCEPTION  WHEN unique\_violation THEN  RAISE NOTICE 'Error: Customer ID % already exists.', p\_id;  WHEN OTHERS THEN  RAISE NOTICE 'Other error: %', SQLERRM;  END;  $$ LANGUAGE plpgsql;  -- Step 6: Call procedures (Test Output)  -- SafeTransferFunds: Try successful and failed transfer  SELECT SafeTransferFunds(1, 2, 1000); -- Successful  SELECT SafeTransferFunds(2, 1, 5000); -- Insufficient funds  -- UpdateSalary: Try valid and invalid employee  SELECT UpdateSalary(101, 10); -- Success  SELECT UpdateSalary(999, 5); -- Invalid ID  -- AddNewCustomer: Try new and duplicate insert  SELECT AddNewCustomer(3, 'Charlie', '2000-01-01', 4000); -- New customer  SELECT AddNewCustomer(1, 'Duplicate', '1990-01-01', 1000);-- Duplicate ID  **Output:**     **Exercise 3: Stored Procedures – All 3 Scenarios in One Script** This script covers:   1. Monthly interest processing for savings accounts 2. Bonus update for employees in a department 3. Fund transfer between customer accounts with balance check   **Code:**  -- Step 1: Create required tables  CREATE TABLE accounts (  accountid SERIAL PRIMARY KEY,  customerid INT,  accounttype VARCHAR(20),  balance NUMERIC  );  CREATE TABLE employees (  employeeid SERIAL PRIMARY KEY,  name VARCHAR(100),  salary NUMERIC,  department VARCHAR(50)  );  -- Step 2: Insert sample data  INSERT INTO accounts (customerid, accounttype, balance) VALUES  (1, 'Savings', 10000),  (2, 'Savings', 15000),  (3, 'Checking', 8000);  INSERT INTO employees (name, salary, department) VALUES  ('Alice', 70000, 'IT'),  ('Bob', 50000, 'HR'),  ('Charlie', 60000, 'IT');  -- Step 3: ProcessMonthlyInterest procedure (1% interest for Savings accounts)  CREATE OR REPLACE FUNCTION ProcessMonthlyInterest() RETURNS VOID AS  $$  BEGIN  UPDATE accounts  SET balance = balance + (balance \* 0.01)  WHERE accounttype = 'Savings';    RAISE NOTICE 'Monthly interest applied to Savings accounts.';  END;  $$ LANGUAGE plpgsql;  -- Step 4: UpdateEmployeeBonus procedure (apply bonus % to employees in dept)  CREATE OR REPLACE FUNCTION UpdateEmployeeBonus(p\_dept TEXT, p\_bonus NUMERIC) RETURNS VOID AS  $$  BEGIN  UPDATE employees  SET salary = salary + (salary \* p\_bonus / 100)  WHERE department = p\_dept;    RAISE NOTICE 'Bonus of % applied to % department.', p\_bonus, p\_dept;  END;  $$ LANGUAGE plpgsql;  -- Step 5: TransferFunds procedure (balance check before transfer)  CREATE OR REPLACE FUNCTION TransferFunds(p\_from INT, p\_to INT, p\_amount NUMERIC) RETURNS VOID AS  $$  DECLARE  v\_balance NUMERIC;  BEGIN  SELECT balance INTO v\_balance FROM accounts WHERE accountid = p\_from FOR UPDATE;    IF v\_balance < p\_amount THEN  RAISE EXCEPTION 'Insufficient balance in source account %', p\_from;  END IF;  UPDATE accounts SET balance = balance - p\_amount WHERE accountid = p\_from;  UPDATE accounts SET balance = balance + p\_amount WHERE accountid = p\_to;  RAISE NOTICE 'Transferred % from account % to account %.', p\_amount, p\_from, p\_to;  END;  $$ LANGUAGE plpgsql;  -- Step 6: Call all 3 procedures  -- Scenario 1: Apply interest  SELECT ProcessMonthlyInterest();  -- Scenario 2: Give 10% bonus to IT department  SELECT UpdateEmployeeBonus('IT', 10);  -- Scenario 3: Transfer funds with and without error  SELECT TransferFunds(1, 3, 2000); -- Success  SELECT TransferFunds(3, 1, 9000); -- Fail due to insufficient funds  **Output:**     **Exercise 4: Functions — All 3 Scenarios Combined** This script includes:   1. CalculateAge – returns customer's age from DOB 2. CalculateMonthlyInstallment – computes EMI 3. HasSufficientBalance – returns boolean if account has enough funds   **Code:**  -- Step 1: Create necessary tables  CREATE TABLE customers (  customerid SERIAL PRIMARY KEY,  name VARCHAR(100),  dob DATE  );  CREATE TABLE accounts (  accountid SERIAL PRIMARY KEY,  customerid INT,  balance NUMERIC  );  -- Step 2: Insert sample data  INSERT INTO customers (name, dob) VALUES  ('John Doe', '1985-06-15'),  ('Alice Smith', '2000-01-01');  INSERT INTO accounts (customerid, balance) VALUES  (1, 5000),  (2, 15000);  -- Step 3: CalculateAge function  CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURNS INT AS  $$  BEGIN  RETURN DATE\_PART('year', AGE(p\_dob));  END;  $$ LANGUAGE plpgsql;  -- Step 4: CalculateMonthlyInstallment function  -- Formula: EMI = (P \* R \* (1+R)^N) / ((1+R)^N - 1)  -- R is monthly interest rate (annual rate / 12 / 100)  CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_amount NUMERIC, p\_rate NUMERIC, p\_years INT) RETURNS NUMERIC AS  $$  DECLARE  r NUMERIC := p\_rate / 12 / 100;  n INT := p\_years \* 12;  BEGIN  RETURN ROUND((p\_amount \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1), 2);  END;  $$ LANGUAGE plpgsql;  -- Step 5: HasSufficientBalance function  CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_account\_id INT, p\_amount NUMERIC) RETURNS BOOLEAN AS  $$  DECLARE  v\_balance NUMERIC;  BEGIN  SELECT balance INTO v\_balance FROM accounts WHERE accountid = p\_account\_id;  IF v\_balance IS NULL THEN  RETURN FALSE;  END IF;  RETURN v\_balance >= p\_amount;  END;  $$ LANGUAGE plpgsql;  -- Step 6: Function calls with test output  SELECT 'Age of John Doe' AS label, CalculateAge('1985-06-15') AS result;  SELECT 'Monthly Installment (Loan ₹100000, 10% for 2 years)' AS label,  CalculateMonthlyInstallment(100000, 10, 2) AS EMI;  SELECT 'Sufficient balance in Account 2 for ₹10000?' AS label,  HasSufficientBalance(2, 10000) AS is\_sufficient;  SELECT 'Sufficient balance in Account 1 for ₹6000?' AS label,  HasSufficientBalance(1, 6000) AS is\_sufficient;  **Output:**   **Exercise 5: Triggers — All 3 Scenarios Combined** This includes:   1. Trigger to auto-update lastmodified when a customer's data changes 2. Trigger to log all transactions into an audit table 3. Trigger to enforce deposit/withdrawal business rules   **Code:**  -- Step 1: Create required tables  -- Customers Table  CREATE TABLE customers (  customerid SERIAL PRIMARY KEY,  name VARCHAR(100),  dob DATE,  balance NUMERIC,  lastmodified TIMESTAMP  );  -- Transactions Table  CREATE TABLE transactions (  transactionid SERIAL PRIMARY KEY,  accountid INT,  transactiondate TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  amount NUMERIC,  transactiontype VARCHAR(10) -- 'Deposit' or 'Withdrawal'  );  -- AuditLog Table  CREATE TABLE auditlog (  logid SERIAL PRIMARY KEY,  transactionid INT,  logtime TIMESTAMP,  description TEXT  );  -- Accounts Table  CREATE TABLE accounts (  accountid SERIAL PRIMARY KEY,  customerid INT,  balance NUMERIC  );  -- Step 2: Insert sample data  INSERT INTO customers (name, dob, balance, lastmodified) VALUES  ('John Doe', '1980-01-01', 5000, CURRENT\_TIMESTAMP);  INSERT INTO accounts (customerid, balance) VALUES  (1, 5000);  -- Step 3: Trigger 1 – Auto-update `lastmodified` on customer update  CREATE OR REPLACE FUNCTION update\_last\_modified()  RETURNS TRIGGER AS  $$  BEGIN  NEW.lastmodified := CURRENT\_TIMESTAMP;  RETURN NEW;  END;  $$ LANGUAGE plpgsql;  CREATE TRIGGER trg\_update\_last\_modified  BEFORE UPDATE ON customers  FOR EACH ROW  EXECUTE FUNCTION update\_last\_modified();  -- Step 4: Trigger 2 – Log transactions into AuditLog  CREATE OR REPLACE FUNCTION log\_transaction()  RETURNS TRIGGER AS  $$  BEGIN  INSERT INTO auditlog (transactionid, logtime, description)  VALUES (NEW.transactionid, CURRENT\_TIMESTAMP,  'Transaction of ' || NEW.amount || ' as ' || NEW.transactiontype || ' on account ' || NEW.accountid);  RETURN NEW;  END;  $$ LANGUAGE plpgsql;  CREATE TRIGGER trg\_log\_transaction  AFTER INSERT ON transactions  FOR EACH ROW  EXECUTE FUNCTION log\_transaction();  -- Step 5: Trigger 3 – Check transaction rules (withdraw ≤ balance, deposit > 0)  CREATE OR REPLACE FUNCTION check\_transaction\_rules()  RETURNS TRIGGER AS  $$  DECLARE  current\_balance NUMERIC;  BEGIN  SELECT balance INTO current\_balance FROM accounts WHERE accountid = NEW.accountid;  IF NEW.transactiontype = 'Withdrawal' THEN  IF NEW.amount > current\_balance THEN  RAISE EXCEPTION 'Withdrawal amount exceeds account balance.';  END IF;  ELSIF NEW.transactiontype = 'Deposit' THEN  IF NEW.amount <= 0 THEN  RAISE EXCEPTION 'Deposit amount must be positive.';  END IF;  ELSE  RAISE EXCEPTION 'Invalid transaction type: %', NEW.transactiontype;  END IF;  RETURN NEW;  END;  $$ LANGUAGE plpgsql;  CREATE TRIGGER trg\_check\_transaction\_rules  BEFORE INSERT ON transactions  FOR EACH ROW  EXECUTE FUNCTION check\_transaction\_rules();  -- Step 6: Test Triggers  -- Update customer info (should auto-update lastmodified)  UPDATE customers SET name = 'John D.' WHERE customerid = 1;  -- Valid deposit  INSERT INTO transactions (accountid, amount, transactiontype)  VALUES (1, 1000, 'Deposit');  -- Valid withdrawal  INSERT INTO transactions (accountid, amount, transactiontype)  VALUES (1, 3000, 'Withdrawal');  -- Invalid withdrawal (more than balance)  -- Should raise error  -- INSERT INTO transactions (accountid, amount, transactiontype)  -- VALUES (1, 10000, 'Withdrawal');  -- Invalid deposit (zero amount)  -- Should raise error  -- INSERT INTO transactions (accountid, amount, transactiontype)  -- VALUES (1, 0, 'Deposit');  -- Invalid transaction type  -- Should raise error  -- INSERT INTO transactions (accountid, amount, transactiontype)  -- VALUES (1, 1000, 'Transfer');  -- Check Audit Log  SELECT \* FROM auditlog;  **Output:**   **Exercise 6: Cursors — All 3 Scenarios Combined** This includes:   1. GenerateMonthlyStatements – for each customer, print their transactions for the month 2. ApplyAnnualFee – deduct an annual maintenance fee from each account 3. UpdateLoanInterestRates – update loan interest based on new policy   **Code:**  -- Step 1: Create necessary tables  -- Customers  CREATE TABLE customers (  customerid SERIAL PRIMARY KEY,  name VARCHAR(100)  );  -- Accounts  CREATE TABLE accounts (  accountid SERIAL PRIMARY KEY,  customerid INT,  balance NUMERIC  );  -- Transactions  CREATE TABLE transactions (  transactionid SERIAL PRIMARY KEY,  accountid INT,  transactiondate DATE,  amount NUMERIC,  transactiontype VARCHAR(10)  );  -- Loans  CREATE TABLE loans (  loanid SERIAL PRIMARY KEY,  customerid INT,  loanamount NUMERIC,  interestrate NUMERIC  );  -- Step 2: Insert sample data  INSERT INTO customers (name) VALUES  ('Alice'), ('Bob');  INSERT INTO accounts (customerid, balance) VALUES  (1, 10000),  (2, 8000);  INSERT INTO transactions (accountid, transactiondate, amount, transactiontype) VALUES  (1, CURRENT\_DATE - INTERVAL '5 days', 500, 'Deposit'),  (1, CURRENT\_DATE - INTERVAL '2 days', 200, 'Withdrawal'),  (2, CURRENT\_DATE - INTERVAL '15 days', 1000, 'Deposit');  INSERT INTO loans (customerid, loanamount, interestrate) VALUES  (1, 5000, 6),  (2, 10000, 5);  -- Step 3: Cursor 1 – GenerateMonthlyStatements  DO  $$  DECLARE  trans RECORD;  BEGIN  RAISE NOTICE '--- Monthly Statements ---';  FOR trans IN  SELECT c.name, t.transactiondate, t.amount, t.transactiontype  FROM customers c  JOIN accounts a ON c.customerid = a.customerid  JOIN transactions t ON a.accountid = t.accountid  WHERE t.transactiondate >= CURRENT\_DATE - INTERVAL '30 days'  ORDER BY c.name, t.transactiondate  LOOP  RAISE NOTICE 'Customer: %, Date: %, Type: %, Amount: %',  trans.name, trans.transactiondate, trans.transactiontype, trans.amount;  END LOOP;  END  $$;  -- Step 4: Cursor 2 – ApplyAnnualFee  DO  $$  DECLARE  acc RECORD;  annual\_fee CONSTANT NUMERIC := 100;  BEGIN  RAISE NOTICE '--- Applying Annual Fee ---';  FOR acc IN SELECT accountid, balance FROM accounts LOOP  UPDATE accounts SET balance = balance - annual\_fee WHERE accountid = acc.accountid;  RAISE NOTICE 'Annual fee of % deducted from account %', annual\_fee, acc.accountid;  END LOOP;  END  $$;  -- Step 5: Cursor 3 – UpdateLoanInterestRates  -- Let's assume: loans under ₹8000 get +0.5%, others get +1%  DO  $$  DECLARE  loan\_rec RECORD;  BEGIN  RAISE NOTICE '--- Updating Loan Interest Rates ---';  FOR loan\_rec IN SELECT loanid, loanamount, interestrate FROM loans LOOP  IF loan\_rec.loanamount < 8000 THEN  UPDATE loans SET interestrate = interestrate + 0.5 WHERE loanid = loan\_rec.loanid;  RAISE NOTICE 'Loan %: Low amount, interest increased by 0.5%%', loan\_rec.loanid;  ELSE  UPDATE loans SET interestrate = interestrate + 1 WHERE loanid = loan\_rec.loanid;  RAISE NOTICE 'Loan %: High amount, interest increased by 1%%', loan\_rec.loanid;  END IF;  END LOOP;  END  $$;  -- Step 6: Check updates  SELECT \* FROM accounts;  SELECT \* FROM loans;  **Output:**   **Exercise 7: Packages — All 3 Scenarios Combined** This script includes:   1. CustomerManagement: procedures/functions to manage customers 2. EmployeeManagement: procedures/functions to manage employees 3. AccountOperations: procedures/functions to manage accounts   **Code:**  -- Step 1: Create schemas to simulate packages  CREATE SCHEMA customermanagement;  CREATE SCHEMA employeemanagement;  CREATE SCHEMA accountoperations;  -- Step 2: Create common tables  CREATE TABLE customers (  customerid SERIAL PRIMARY KEY,  name VARCHAR(100),  balance NUMERIC  );  CREATE TABLE employees (  employeeid SERIAL PRIMARY KEY,  name VARCHAR(100),  salary NUMERIC,  department VARCHAR(50)  );  CREATE TABLE accounts (  accountid SERIAL PRIMARY KEY,  customerid INT,  balance NUMERIC  );  -- Step 3: Sample data  INSERT INTO customers (name, balance) VALUES  ('Alice', 10000),  ('Bob', 5000);  INSERT INTO employees (name, salary, department) VALUES  ('John', 60000, 'IT'),  ('Jane', 70000, 'HR');  INSERT INTO accounts (customerid, balance) VALUES  (1, 10000),  (2, 5000);  -- Step 4: CustomerManagement package (schema)  -- Procedure: Add new customer  CREATE OR REPLACE FUNCTION customermanagement.add\_customer(p\_name TEXT, p\_balance NUMERIC)  RETURNS VOID AS  $$  BEGIN  INSERT INTO customers (name, balance) VALUES (p\_name, p\_balance);  RAISE NOTICE 'Customer % added.', p\_name;  END;  $$ LANGUAGE plpgsql;  -- Procedure: Update customer details  CREATE OR REPLACE FUNCTION customermanagement.update\_customer(p\_id INT, p\_name TEXT, p\_balance NUMERIC)  RETURNS VOID AS  $$  BEGIN  UPDATE customers SET name = p\_name, balance = p\_balance WHERE customerid = p\_id;  RAISE NOTICE 'Customer % updated.', p\_id;  END;  $$ LANGUAGE plpgsql;  -- Function: Get customer balance  CREATE OR REPLACE FUNCTION customermanagement.get\_balance(p\_id INT)  RETURNS NUMERIC AS  $$  DECLARE  v\_balance NUMERIC;  BEGIN  SELECT balance INTO v\_balance FROM customers WHERE customerid = p\_id;  RETURN v\_balance;  END;  $$ LANGUAGE plpgsql;  -- Step 5: EmployeeManagement package  -- Procedure: Hire employee  CREATE OR REPLACE FUNCTION employeemanagement.hire\_employee(p\_name TEXT, p\_salary NUMERIC, p\_dept TEXT)  RETURNS VOID AS  $$  BEGIN  INSERT INTO employees (name, salary, department)  VALUES (p\_name, p\_salary, p\_dept);  RAISE NOTICE 'Employee % hired in %.', p\_name, p\_dept;  END;  $$ LANGUAGE plpgsql;  -- Procedure: Update employee  CREATE OR REPLACE FUNCTION employeemanagement.update\_employee(p\_id INT, p\_salary NUMERIC)  RETURNS VOID AS  $$  BEGIN  UPDATE employees SET salary = p\_salary WHERE employeeid = p\_id;  RAISE NOTICE 'Employee % salary updated.', p\_id;  END;  $$ LANGUAGE plpgsql;  -- Function: Get annual salary  CREATE OR REPLACE FUNCTION employeemanagement.annual\_salary(p\_id INT)  RETURNS NUMERIC AS  $$  DECLARE  monthly NUMERIC;  BEGIN  SELECT salary INTO monthly FROM employees WHERE employeeid = p\_id;  RETURN monthly \* 12;  END;  $$ LANGUAGE plpgsql;  -- Step 6: AccountOperations package  -- Procedure: Open account  CREATE OR REPLACE FUNCTION accountoperations.open\_account(p\_custid INT, p\_balance NUMERIC)  RETURNS VOID AS  $$  BEGIN  INSERT INTO accounts (customerid, balance) VALUES (p\_custid, p\_balance);  RAISE NOTICE 'Account opened for customer %.', p\_custid;  END;  $$ LANGUAGE plpgsql;  -- Procedure: Close account  CREATE OR REPLACE FUNCTION accountoperations.close\_account(p\_accid INT)  RETURNS VOID AS  $$  BEGIN  DELETE FROM accounts WHERE accountid = p\_accid;  RAISE NOTICE 'Account % closed.', p\_accid;  END;  $$ LANGUAGE plpgsql;  -- Function: Get total balance for a customer  CREATE OR REPLACE FUNCTION accountoperations.total\_balance(p\_custid INT)  RETURNS NUMERIC AS  $$  DECLARE  total NUMERIC;  BEGIN  SELECT SUM(balance) INTO total FROM accounts WHERE customerid = p\_custid;  RETURN total;  END;  $$ LANGUAGE plpgsql;  -- Step 7: Test all package functions/procedures  -- CustomerManagement  SELECT customermanagement.add\_customer('Charlie', 8000);  SELECT customermanagement.update\_customer(1, 'Alice A.', 11000);  SELECT 'Balance of Customer 1:' AS label, customermanagement.get\_balance(1);  -- EmployeeManagement  SELECT employeemanagement.hire\_employee('Emily', 65000, 'Finance');  SELECT employeemanagement.update\_employee(1, 75000);  SELECT 'Annual salary of Employee 1:' AS label, employeemanagement.annual\_salary(1);  -- AccountOperations  SELECT accountoperations.open\_account(2, 3000);  SELECT accountoperations.close\_account(1);  SELECT 'Total balance for Customer 2:' AS label, accountoperations.total\_balance(2);  **Output:** |
| --- |