View Definition

- A relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

- A view is defined using the **create view** statement which has the form

  create view v **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by v.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

- **A view consisting of branches and their customers**

```
  create view all_customer as
                (select branch_name, customer_name
                 from depositor, account
                where depositor.account_number =
                               account.account_number )
                union
                (select branch_name, customer_name
                from borrower, loan
                where borrower.loan_number = loan.loan_number )
```

- **Find all customers of the Perryridge branch**

```
        select customer_name
                from all_customer
                where branch_name = 'Perryridge'
```

Uses of Views

- Hiding some information from some users
  - Consider a user who needs to know a customer’s name, loan number and branch name, but has no need to see the loan amount.
  - Define a view

```
(create view cust_loan_data as
 select customer_name, borrower.loan_number, branch_name
  from borrower, loan
  where borrower.loan_number = loan.loan_number )
```

  - Grant the user permission to **read cust_loan_data, but not borrower or loan**
  - Predefined queries to make writing of other queries easier
  - Common example: Aggregate queries used for statistical analysis of data

Processing of Views

- **When a view is created**
  - **Query expression is stored in the database along with the view name**
  - **Expression is substituted into any query using the view**
- **Views definitions containing views**
  - **One view may be used in the expression defining another view**
  - **A view relation v_1 is said to depend directly on a view relation v_2 if v_2 is used in the expression defining v_1**
  - **A view relation v_1 is said to depend on view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2**
  - **A view relation v is said to be recursive if it depends on itself.**

View Expansion

- A way to define the meaning of views defined in terms of other views.

- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.

- View expansion of an expression repeats the following replacement step:

```
repeat
   Find any view relation $v_i$ in $e_1$
   Replace the view relation $v_i$ by the expression defining $v_i$
until no more view relations are present in $e_1$
```

As long as the view definitions are not recursive, this loop will terminate

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

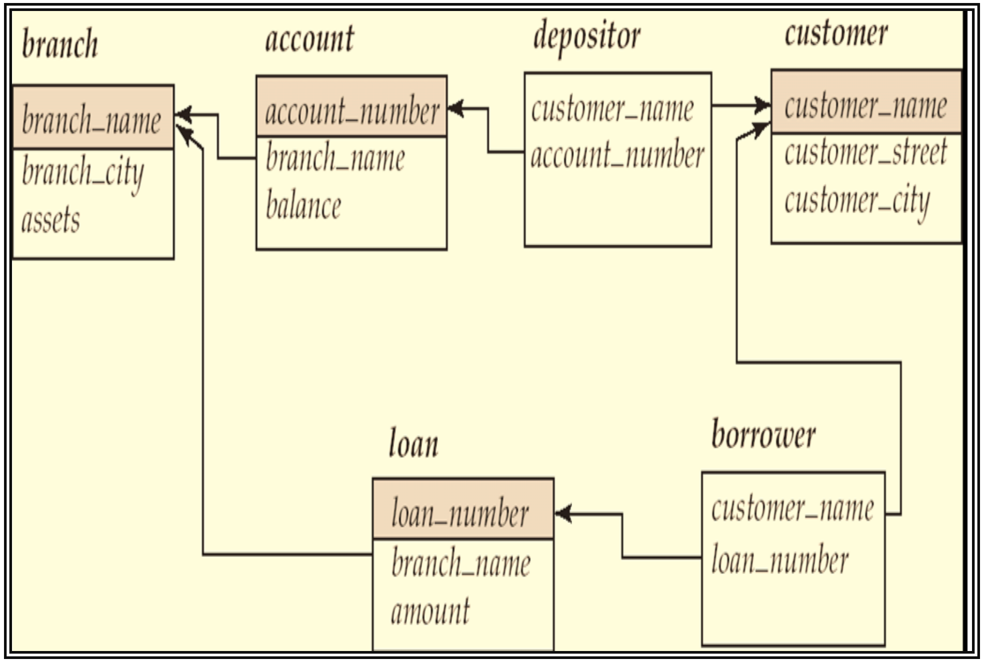
- **Find all accounts with the maximum balance**

```
with max_balance (value) as
    select max (balance)
    from account
select account_number
from account, max_balance
where account.balance = max_balance.value
```

Complex Queries using With Clause

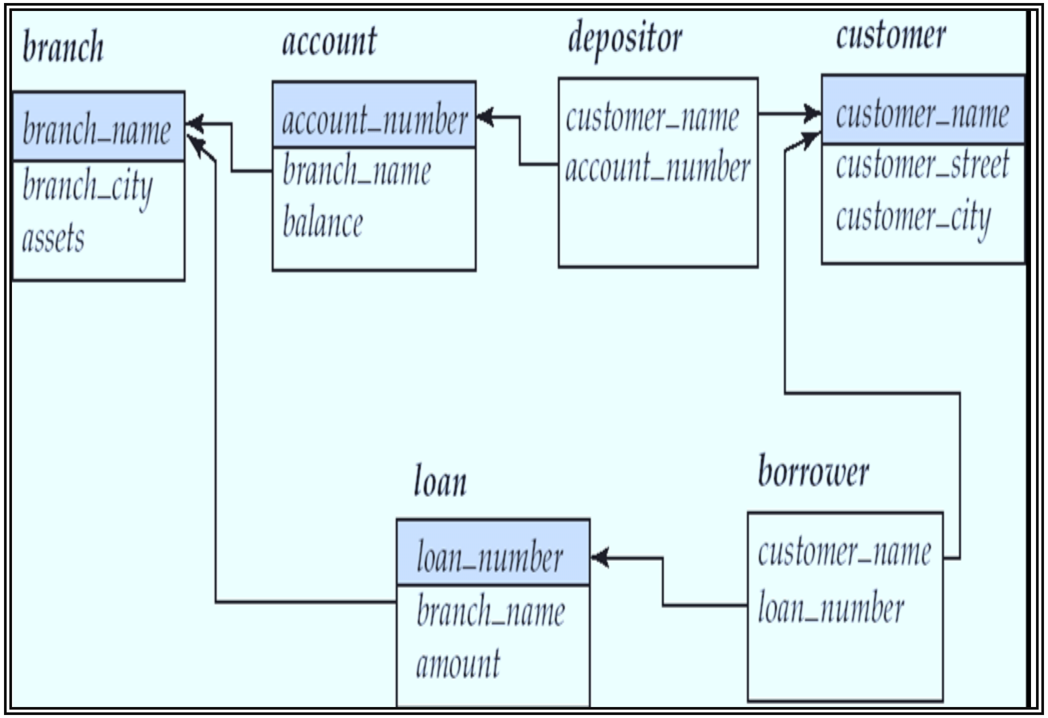
- **Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.**

```
with branch_total (branch_name, value) as
        select branch_name, sum (balance)
        from account
        group by branch_name


with branch_total_avg (value) as
        select avg (value)
        from branch_total


select branch_name
 from branch_total, branch_total_avg
 where branch_total.value >= branch_total_avg.value
```



Update of a View

- **Create a view of all loan data in the loan relation, hiding the amount attribute**



```
create view loan_branch as
select loan_number, branch_name
 from loan
```

Add a new tuple to loan_branch

```
insert into loan_branch
 values ('L-37‘, 'Perryridge‘)
```

This insertion must be represented by the insertion of the tuple

('L-37', 'Perryridge', null)

into the loan relation

Updates Through Views ..contd.

- **Some updates through views are impossible to translate into updates on the database relations**

  -
    ```
    create view v as
                    select loan_number, branch_name, amount
                    from loan
                    where branch_name = ‘Perryridge’
    ```

  ```
  insert into v values  ( 'L-99','Downtown', '23')
  ```

- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Null Values

- It is possible for tuples to have a null value, denoted by null, for some of their attributes
- Null signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.

Find all loan number which appear in the loan relation with null values for amount.

```
select loan_number
from loan
where amount is null
```

The result of any arithmetic expression involving null is null

Example: 5 + null returns null

However, aggregate functions simply ignore nulls

Null Values and Three Valued Logic

- **Any comparison with null returns unknown**
  - Example: 5 < null or null <> null or null = null

- **Three-valued logic using the truth value unknown**:
  - OR: (unknown **or** true) = true,
    (unknown **or** false) = unknown
    (unknown **or** unknown) = unknown
  - AND: (true **and** unknown) = unknown,
    (false **and** unknown) = false,
    (unknown **and** unknown) = unknown
  - NOT: (**not** unknown) = unknown
  - "P **is unknown**" evaluates to true if predicate P evaluates to unknown

- **Result of where clause predicate is treated as false if it evaluates to unknown**

Null Values and Aggregates

- Total all loan amounts

select sum (amount)
from loan

  - Above statement ignores null amounts
  - Result is null if there is no non-null amount

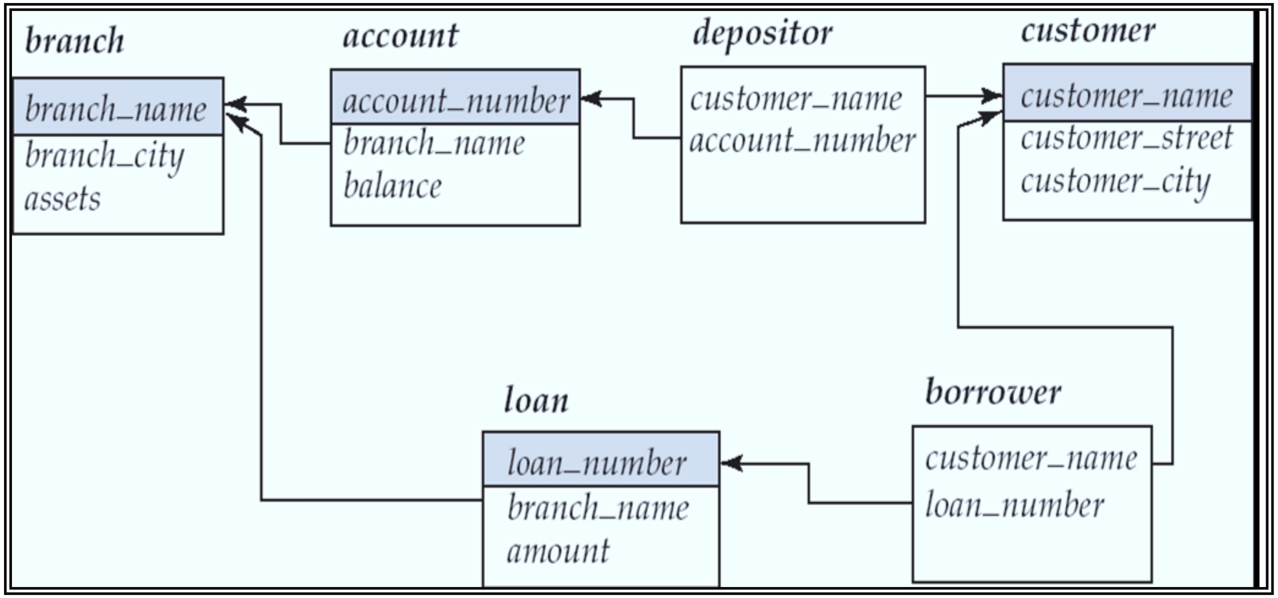
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

The where Clause . . . Contd.

- SQL includes a **between** comparison operator

- Find the loan number of those loans with loan amounts between Rs 90,000 and Rs 100,000 (that is, >= Rs 90,000 and <= Rs100,000)

```
select loan_number
        from loan
        where amount between 90000 and 100000
```



Cursors

- When ever a select statement is written in PL/SQL block the data returned by that select statement is stored in buffer in an area called context area.

- The rows that are stored in buffer in an area and are the result of the select statement are combined called as activeset.

- Cursor is a pointer point to the activeset returned by an SQL statement.

- Cursors are classified into two types
  - **Implicit Cursors:**
    - An implicit cursors are those that are automatically created and provided by oracle.
    - An implicit cursors is created automatically whenever a DML operation is performed or a select statement was returned.

  - **Explicit Cursors:**
    - Explicit cursors must be declared and write appropriate code by the user.

Cursors

- **Steps to create and write code for an explicit cursors:**

  1.CURSOR DECLARATION

  2.OPENING CURSOR

  3.FETCHING DATA

  4.CLOSING CURSOR

1. Cursor Declaration: Before using an explicit cursor it must be declared within the declaration section. By declaration of cursor, cursor associated with an SQL statement to which the cursor has to point.

Syntax: CURSOR C_Name IS select statement;

2. OPENING CURSOR: Before accessing the data using a cursor, cursor must be opened during opening a cursor, the select statement associated with that cursor is executed and cursor points to the first row of the active set returned by that select statement.

Syntax: OPEN C_name;

3. FETCHING DATA: To access data by using cursor first we have to copy the values of a row to which cursor points into local variables. This process is called fetching data from the cursor.

Syntax: FETCHING DATA C_Name INTO local variable list;

4. CLOSING CURSOR: After completion of work with the cursor we close the cursor by which memory occupied by cursor points will be released.

Syntax: CLOSE C_Name

Cursors…contd.

- **Cursors Attributes:**

Cursor attributes are used to find whether data is retrieved by the fetch statement or whether cursor is already opened or the number of rows fetched from the cursor until now.

1. **%FOUND: This attribute returns true if the previous fetch statement retrieves a row. Otherwise false.**

2. **%NOT FOUND: This attribute returns true if the previous fetch statement does not return any rows otherwise returns false.**

3. **%IS OPEN: This attribute returns true if the specified cursor is already open otherwise returns false.**

4. **%ROW COUNT: This attribute returns the number of rows fetched from the specified cursor until now.**

Cursors…contd.

- **Write a PL/SQL cursors to update the salaries of employees by using following criteria. Manager-1000, Analyst-750, any other -500. [USING WHILE]**

```
DECLARE
            CURSOR My_cur IS SELECT EMPNO, JOB, SAL FROM EMP;
             E_NO EMP.EMPNO%TYPE;
             J  EMP.JOB%TYPE;
             S  EMP.SAL%TYPE
 BEGIN
          OPEN My_cur;
           FETCH My_cur INTO E_NO, J, S;
           WHILE My_cur %FOUND
           LOOP
               IF J:='MANAGER' THEN
                  S:=S+1000;
                  ELSEIF J:= 'ANALYST' THEN
                  S := S+750;
                  ELSE
                 S:=  S+500
              END IF;
              UPDATE EMP SET SAL=S WHERE EMPNO:=E_NO;
              FETCH My_cur INTO E_NO, J, S;
           END LOOP;
 CLSOE My_cur;
END
```

Cursors…contd.

- **Write a PL/SQL cursors to update the salaries of employees by using following criteria. Manager-1000, Analyst-750, any other -500.** [USING LOOP]

```
DECLARE
            CURSOR My_cur IS SELECT EMPNO, JOB, SAL FROM EMP;
             E_NO EMP.EMPNO%TYPE;
             J  EMP.JOB%TYPE;
             S  EMP.SAL%TYPE
 BEGIN
         OPEN My_cur;
          LOOP
          FETCH My_cur INTO E_NO, J, S;
          EXIT WHEN My_cur % NOT FOUND
           IF J:=‘MANAGER’ THEN
                   S:=S+1000;
                   ELSEIF J:= ‘ANALYST’ THEN
                   S := S+750;
                  ELSE
                 S:=  S+500
           END IF;
           UPDATE EMP SET SAL=S WHERE EMPNO:=E_NO;
          END LOOP;
 CLSOE My_cur;
END
```

Cursors…contd.

- **Write a PL/SQL cursors to update the salaries of employees by using following criteria. Manager-1000, Analyst-750, any other -500.** [USING FOR LOOP]

```
DECLARE
          CURSOR My_cur IS SELECT *  FROM EMP;
           E_ROW EMP % ROW TYPE;
BEGIN
        OPEN My_cur;
        FOR E_ROW IN My_cur
        LOOP
        IF E_ROW.JOB:=‘MANAGER’ THEN
                E_ROW.SAL:=E_ROW.SAL+1000;
                ELSEIF E_ROW. JOB:= ‘ANALYST’ THEN
                E_ROW.SAL := E_ROW.SAL+750;
                ELSE
                E_ROW.SAL := E_ROW.SAL+500
         END IF;
         UPDATE EMP SET SAL=E_ROW.SAL WHERE EMPNO:=E_ROW.EMPNO;
        END LOOP;
 CLSOE My_cur;
END
```

Cursors…contd.

Parameterized Cursors: We can pass arguments to a cursor like passing arguments to a function by defining parameters at the time of cursor declaration.

Cursor Declaration:

```
CURSOR  My_cur (p1 datatype, p2 datatype,…,pn datatype) IS select statement;
```

Opening Cursor:

```
OPEN My_cur (arg1, arg2,…,argn);
```

```
DECLARE                    /* Print the details of student who doesn’t paid fee  */
CURSOR My_cur(p_course  course%type ) IS select * from Student where course = p_course;
SROW   Student % ROW_TYPE;
C    Student. Course%type := ‘&course’;
BEGIN
     OPEN My_cur;
     FETCH My_cur INTO SROW.Sno, SROW.Sname, SROW.course, SROW.Fee;
     LOOP
         IF SROW.Fee>0 THEN
            DBMS_OUPUT.PUT_LINE(SROW.Sno||”||SROW.Sname||’’||SROW.COURSE||’’||SROW.Fee);
          ENDIF
      FETCH My_cur INTO SROW.Sno, SROW.Sname, SROW.course,  SROW.Fee;
      END LOOP;
      CLOSE My_cur;
  END;
```

Procedures…contd.

Parameter modes:

Parameters can be passed to a procedure or function in three modes IN, OUT, INOUT. The behavior of parameters when they are passed to a function or procedure in any of the three modes is as follows:

IN: When a parameter is passed in IN mode then that parameter is read only within the procedure or function. This means we can read the value present in that parameter but it is not possible to change the value of that parameter.

OUT: When a parameter is passed as OUT parameter then that parameter is write only within the procedure or function. This means that we can change the value of that parameter but it is not possible to read the value of that parameter. The change made to the out parameter will reflect that change in its corresponding argument.

INOUT :When a parameter is passed in INOUT mode then that parameter is read/write within the procedure or function. This means that we can read the value of that parameter as well as we can change the value of that parameter. Any change made to the INOUT parameter within the procedure or function will reflect that changes in its corresponding arguments.

Subprograms

- PL/SQL blocks that are created with a name and are stored within the database are called as subprograms.
- Unlike anonymous blocks subprograms can be used any number of times as they are stored within the database and have a name.
- Subprograms include functions, procedures and packages.

Procedures: Procedures are the PL/SQL subprograms that performs a task and doesn't return any value to the place from which it is called.

- Creating a procedure:

```
CREATE OR REPLACE PROCEDURE P_name ( PARM1 IN/OUT/INOUT DATATYPE,
                                     PARM2 IN/OUT/INOUT DATATYPE,
                                     PARM3 IN/OUT/INOUT DATATYPE,
                                     ….
                                    PARMn IN/OUT/INOUT DATATYPE) IS
LOCAL VARIABLE DECLARATIONS
BEGIN
EXECUTABLE STAEMENTS
EXCEPTION
ERROR HANDLING ROUTINES
END P_name
```

Where: - P_name is the procedure name,
 - prameter1 to parametern are the names of parameters that hold the arguments passed to the procedure.
 - IN, OUT, INOUT are the modes of parameters. Procedure doesn't require the keyword "DECLARE" for declaring the local variables.

Cursor…contd.

Calling a procedure: There are two approaches

- We call procedure from the SQL prompt then execute command is used.

  General Syntax: EXECUTE P_name (Arg1,Arg2,…,ARGn);

  E.g.: EXECUTE ADDSTUDENT(1, ‘THEORY’, ORACLE,2000);

- If we are calling a procedure from another PL/SQL block then we call directly i.e., use procedure name without using any commands.

  General Syntax: P_Name(Arg1, Arg2,…Argn);

Procedure…contd.

Write a procedure that updates the commission of all employees on following criteria:

Salesman 40% of Salary

Clerk 20% of salary

Others 10% of salary

```
CREATE OR REPLACE PROCEDURE COMMISION IS
CURSOR My_cur IS SELECT E_No, Job, Sal, Comm FROM EMPLOYEE;
EROW  EMP%ROW TYPE;
BEGIN
OPEN My_cur;
FETCH My_cur INTO EROW.E_No, EORW.Job, EORW.Sal, EROW.Comm;
WHILE My_cur%FOUND
LOOP
      IF EROW.Job=‘SALESMAN’ THEN
          EROW.Comm=Sal*40/100;
      ELSE IF EROW.Job=‘CLERK’  THEN
         EROW.Comm=EORW.Sal*20/100;
     ELSE EROW.Comm=EROW.Sal*10/100;
    ENDIF
 UPDATE EMP SET Comm=EROW.Comm WHERE E_No=EROW.E_No;
FETCH My_cur INTO EROW.E_No, EORW.Job, EORW.Sal, EROW.Comm;
END LOOP;
CLOSE My_cur;
END;
```

Subprograms…contd.

- Function is a PL/SQL block that performs a given task and will return a value to the place which it is called.

- Creating a function:

```
CREATE OR REPLACE FUNCTION F_name ( PARM1 IN/OUT/INOUT DATATYPE,
                                     PARM2 IN/OUT/INOUT DATATYPE,
                                     PARM3 IN/OUT/INOUT DATATYPE,
                                   ….
                                    PARMn IN/OUT/INOUT DATATYPE)
                                    RETURN DATATYPE IS/AS
LOCAL VARIABLE DECLARATIONS
BEGIN
EXECUTABLE STAEMENTS
EXCEPTION
ERROR HANDLING ROUTINES
END F_name
```

Functions

- Write a function accepts three numeric values and returns the total of these values.

```
 CREATE  OR REPLACE FUNCTION TOTAL (P_M1 MARKS.M1%TYPE,
                                    P_M2 MARKS.M2%TYPE
                                    P_M3 MARKS.M3%TYPE) RETURN NUMBER IS
     BEGIN
     RETURN (P_M1+P_M2+P_M3);
     END TOTAL;
```

- Write a function that accepts total marks of a student in three subjects and returns average.

```
    CREATE  OR REPLACE FUNCTION AVEG (P_TOT MARKS.TOT%TYPE) RETURN NUMBER IS
     BEGIN
     RETURN (P_TOT/3);
     END  AVEG;
```

- Write a function that accepts average marks of a student and returns his grade.

```
      CREATE  OR REPLACE FUNCTION Grade (P_AVG  MARKS.AVEG%TYPE) RETURN VARCHAR2 IS
     G    MARKS.GRADE%TYPE;
     BEGIN
     IF P_AVG>= 90 THEN
        G:= ‘DISTINCTION’
     ELSEIF  P_AVEG>=65 THEN
        G:= ‘FIRST CLASS’
     ELSE
       G :=‘ PASS’
     ENDIF
     RETURN (G);
     END Grade
```

Packages

- Packages are another type of PL/SQL blocks that are used to group related functions and procedures.

- Packages are also used to declare global variables, curser and exceptions.

- Like subprograms packages are also stored in the database but unlike subprograms they cannot be executed and they doesn’t receive any arguments.

- A package consists of two parts, package specification and package body.

  -Package Specification contains variables, cursors, exception declarations, functions and procedure definitions.

  -Package Body consists of the code to be executed for the functions and procedures declared within the package specification.

-Package specification and package body must be created separately but both must have the same name.

Packages…contd.

Package specification:

Syntax:

```
          CREATE OR REPLACE PACKAGE P_Name IS/AS
          VAR DECLARATIONS
          CURSOR DECLARATIONS
          EXCEPTION DECLARATIONS
          FUNCTION DECLARATIONS
          PROCEDURE DECLARATIONS
          END P-N ame
```

E.g.,

```
CREATE OR REPLACE PACKAGE Stu_Pack AS
        PROCEDURE ADD_STUDENT(P_SNO STUDENT.SNo%TYPE, P_SNAME STUDENT.SName%TYPE,
                                  P_COURSE STUDENT.Course);
         FUNCTION TOTAL (P_M1 MARKS.M1%TYPE, P_M2 MARKS.M2%TYPE, P_M3 MARKS.M3%TYPE)
                              RETURN NUMBER;
          FUNCTION AVEG(P_TOT, MARKS.TOT%TYPE) RETURN NUMBER;
          FUNCTION GRADE(P_AVG MARKS.AVEG%TYPE) RETURN VARCHAR2;
          PROCEDURE ADD_MARKS(P_SNO MARKS.Sno%TYPE, P_M1   MARKS.M1%TYPE,
                                  P_M2   MARKS.M2%TYPE, P_M3   MARKS.M3%TYPE);
END Stu_Pack;
```

Packages…contd.

Package body:

```
Syntax:     CREATE OR REPLACE PACKAGE BODY PB_Name IS/AS
             FUNCTION DEFINITIONS
              PROCEDURE DEFINITIONS
              END PB-N ame
```

```
CREATE OR REPLACE PACKAGE BODY  Stu_Pack AS
        PROCEDURE ADD_STUDENT(P_SNo STUDENT.SNo%TYPE, P_SName STUDENT.SName%TYPE,
                                      P_Course STUDENT.Course) IS
         BEGIN
           INSERT INTO STUDENT VALUES(P_Sno, P_SName, P_Course);
         END ADD_STUDENT



 FUNCTION TOTAL (P_M1 MARKS.M1%TYPE, P_M2 MARKS.M2%TYPE, P_M3 MARKS.M3%TYPE)
  BEGIN
    RETURN (P_M1+P_M2+P_M3);
  END TOTAL

      FUNCTION AVEG(P_TOT, MARKS.TOT%TYPE)
       BEGIN
         RETURN (P_TOT/3);
      END AVEG
```

Packages…contd.

```
 FUNCTION GRADE(P_AVG MARKS.AVEG%TYPE) RETURN VARCHAR2;
        G MARKS.GRADE%TYPE
                IF P_AVG >=90 THEN
                G:= ‘Distinction’;
                ELSEIF P_AVG>= 65 THEN
                G:= ‘First Class’;
                ELSEIF P_AVG>= 55 THEN
                G:= ‘Second Class’;
                ELSEIF P_AVG>= 35 THEN
                G:= ‘Pass Class’;
                ELSE
                G:= ‘Fail’;
                ENDIF
        RETURN(G);
        END GRADE;
PROCEDURE ADD_MARKS(P_SNO MARKS.Sno%TYPE, P_M1   MARKS.M1%TYPE,
                          P_M2   MARKS.M2%TYPE, P_M3   MARKS.M3%TYPE)IS
T   MARKS.TOT%TYPE:= Stu_Pack.TOTAL(P_M1,P_M2,P_M3);
A    MARKS.AGEV%TYPE:=Stu_Pack.AVEG(T);
G    MARKS.GRADE%TYPE:= Stu_Pack.GRADE(A);
BEGIN
INSERT INTO MARKS VALUES(P_Sno, P_M1, P_M2, P_M3,T, A, G);
END ADD_MARKS
```

Packages…contd.

Package Specification:

```
CREATE OR REPLACE PACKAGE c_package AS
           -- Adds a customer
  PROCEDURE addCustomer(c_id   customers.id%type,
                          c_name  customers.name%type,
                         c_age  customers.age%type,
                         c_addr customers.address%type,
                         c_sal  customers.salary%type);

          -- Removes a customer
    PROCEDURE delCustomer(c_id  customers.id%TYPE);

        --Lists all customers
    PROCEDURE listCustomer;

END c_package;
```

Packages…contd.

CREATING THE PACKAGE BODY:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
     c_name customers.name%type,
     c_age  customers.age%type,
     c_addr  customers.address%type,
     c_sal   customers.salary%type)
  IS
  BEGIN
     INSERT INTO customers (id,name,age,address,salary)
        VALUES(c_id, c_name, c_age, c_addr, c_sal);
  END addCustomer;

  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
      DELETE FROM customers
        WHERE id = c_id;
  END delCustomer;
```

```
 PROCEDURE listCustomer IS
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list is TABLE OF
customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
   BEGIN
      FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('
||counter|| ')'||name_list(counter));
      END LOOP;
   END listCustomer;

END c_package;
```