## 10.5 Programming Oracle Applications

Programming in Oracle is done in several ways:

- Writing interactive SQL queries in the SQL query mode.
- Writing programs in a host language like COBOL, C, or PASCAL, and embedding SQL within the program. A precompiler such as PRO*COBOL or PRO*C is used to link the application to Oracle.
- Writing in PL/SQL, which is Oracle's own procedural language.
- Using Oracle Call Interface (OCI) and the Oracle runtime library SQLLIB.

### 10.5.1 Programming in PL/SQL

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL offers software engineering features such as data encapsulation, information hiding, overloading, and exception handling to the developers. It is the most heavily used technique for application development in Oracle.

PL/SQL is a block-structured language. That is, the basic units—procedures, functions and anonymous blocks—that make up a PL/SQL program are logical blocks, which can contain any number of nested subblocks. A block or subblock groups logically related declarations and statements. The declarations are local to the block and cease to exist when the block completes. As illustrated below, a PL/SQL block has three parts: (1) a **declaration part** where variables and objects are declared, (2) an **executable part** where these variables are manipulated, and (3) an **exception part** where exceptions or errors raised during execution can be handled.

```
[ DECLARE
        ---declarations ]
BEGIN
        ---statements
[ EXCEPTION
        ---handlers ]
END ;
```

In the declaration part—which is optional—variables are declared. Variables can have any SQL data type as well as additional PL/SQL data types. Variables can also be assigned values in this section. Objects are manipulated in the executable part, which is the only required part. Here data can be processed using conditional, iterative, and sequential flow-of-control statements such as IF-THEN-ELSE, FOR-LOOP, WHILE-LOOP, EXIT-WHEN, and GO-TO. The exception part handles any error conditions raised in the executable part. The exception could be user-defined errors or database errors or exceptions. When an error or exception occurs, an exception is raised and the normal execution stops and control transfers to the exception-handling part of the PL/SQL block or subprogram.

Suppose we want to write PL/SQL programs to process the database of Figure 7.5. As a first example, E1, we write a program segment that prints out some information about an employee who has the highest salary as follows:

```
E1:
DECLARE
     v_fname      employee.fname%TYPE;
     v_minit      employee.minit%TYPE;
     v_lname      employee.lname%TYPE;
     v_address    employee.address%TYPE;
     v_salary     employee.salary%TYPE;

BEGIN
     SELECT  fname, minit, lname, address, salary
     INTO    v_fname, v_minit, v_lname, v_address , v_salary
     FROM    EMPLOYEE
     WHERE   salary = (select max (salary) from employee) ;

     DBMS_OUTPUT.PUT_LINE (v_fname, v_minit, v_lname, v_address,
                           v_salary);

EXCEPTION
     WHEN OTHERS
     DBMS_OUTPUT.PUT_LINE ('Error Detected');
END;
```

In E1, we need to declare program variables to match the types of the database attributes that the program will process. These program variables may or may not have names that are identical to their corresponding attributes. The %TYPE in each variable declaration means that that variable is of the same type as the corresponding column in the table. DBMS_OUTPUT.PUT_LINE is PL/SQL's print function. The error handling part prints out an error message if Oracle detects an error—in this case, if more than one employee is selected—while executing the SQL. The program needs an INTO clause, which specifies the program variables into which attribute values from the database are retrieved.

In the next example, E2, we write a simple program to increase the salary of employees whose salaries are less than the average salary by 10 percent. The program recomputes and prints out the average salary if it exceeds 50000 after the above update.

```
E2:
DECLARE
     avg_salary NUMBER;

BEGIN
     SELECT avg(salary) INTO avg_salary
     FROM employee;

     UPDATE employee
       SET salary = salary*1.1
```

```
        WHERE salary < avg_salary;

    SELECT avg(salary) INTO avg_salary
    FROM employee;

    IF avg_salary > 50000 THEN
    dbms_output.put_line ('Average Salary is ' || avg_salary) ;
    END IF;

    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
    dbms_output.put_line ('Error in Salary update ')
    ROLLBACK;

END;
```

In E2, avg_salary is defined as a variable and it gets the value of the average of the employees' salary from the first SELECT statement and this value is used to choose which of the employees will have their salaries updated. The EXCEPTION part rolls back the whole transaction (that is, removes any effect of the transaction on the database) if an error of any type occurs during execution.

## 10.5.2 Cursors in PL/SQL

The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet the search criteria. When a query returns multiple rows, it is necessary to explicitly declare a **cursor** to process the rows. A cursor is similar to a *file variable* or *file pointer*, which points to a single row (tuple) from the result of a query. Cursors should be declared in the declarative part and are controlled by three commands: OPEN, FETCH, and CLOSE. The cursor is initialized with the OPEN statement, which executes the query, retrieves the resulting set of rows, and sets the cursor to a position before the first row in the result of the query. This becomes the current row for the cursor. The FETCH statement, when executed for the first time, retrieves the first row into the program variables and sets the cursor to point to that row. Subsequent executions of FETCH advance the cursor to the next row in the result set, and retrieve that row into the program variables. This is similar to the traditional record-at-a-time file processing. When the last row has been processed, the cursor is released with the CLOSE statement. Example E3 displays the SSN of employees whose salary is greater than their supervisor's salary.

```
E3:
DECLARE
   emp_salary NUMBER;
   emp_super_salary NUMBER;
   emp_ssn CHAR (9);
   emp_superssn CHAR (9);
   CURSOR salary_cursor IS
```

```
    SELECT ssn, salary, superssn FROM employee;
BEGIN
  OPEN salary_cursor;

  LOOP
    FETCH salary_cursor INTO emp_ssn, emp_salary, emp_superssn;
    EXIT WHEN salary_cursor%NOTFOUND;

    IF emp_superssn is NOT NULL THEN
        SELECT salary INTO emp_super_salary
        FROM employee
        WHERE ssn = emp_superssn;

        IF emp_salary > emp_super_salary THEN
        dbms_output.put_line(emp_ssn);
        END IF;
    END IF;
  END LOOP;
  IF salary_cursor%ISOPEN THEN CLOSE salary_cursor;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
  dbms_output.put_line ('Errors with ssn ' || emp_ssn);
  IF salary_cursor%ISOPEN THEN CLOSE salary_cursor;

END;
```

In the above example, the SALARY_CURSOR loops through the entire employee table until the cursor fetches no further rows. The exception part handles the situation where an incorrect supervisor ssn may be assigned to an employee. The %NOTFOUND is one of the four cursor attributes, which are the following:

- %ISOPEN returns TRUE if the cursor is already open.
- %FOUND returns TRUE if the last FETCH returned a row, and returns FALSE if the last FETCH failed to return a row.
- %NOTFOUND is the logical opposite of %FOUND.
- %ROWCOUNT yields the number of rows fetched.

As a final example, E4 shows a program segment that gets a list of all the employees, increments each employee's salary by 10 percent, and displays the old and the new salary.

```
E4:
DECLARE
  v_fname    employee.fname%TYPE;
  v_minit    employee.minit%TYPE;
  v_lname    employee.lname%TYPE;
  v_address  employee.address%TYPE;
  v_salary   employee.salary%TYPE;
```

```
  CURSOR EMP IS
      SELECT ssn, fname, minit, lname, salary
      FROM employee;

BEGIN
  OPEN EMP ;

  LOOP
      FETCH EMP INTO v_ssn, v_fname, v_minit, v_lname,
       v_salary ;
      EXIT WHEN EMP%NOTFOUND;

      dbms_output.putline('SSN:' || v_ssn || 'Old salary :'
                          || v_salary);

      UPDATE employee
      SET    salary = salary*1.1
      WHERE  ssn = v_ssn;
      COMMIT;
      dbms_output.putline('SSN:' || v_ssn || ' New salary :'
                          || v_salary*1.1);

  END LOOP;
  CLOSE EMP;

EXCEPTION
  WHEN OTHERS
  dbms_output.put_line ('Error Detected');
END:
```