5

PL/SQL

Learning Objectives. This chapter focuses on the shortcomings of SQL and how it is overcome by PL/SQL. An introduction to PL/SQL is given in this chapter. After completing this chapter the reader should be familiar with the following concepts in PL/SQL.

– Structure of PL/SQL
– PL/SQL language elements
– Control structure in PL/SQL
– Steps to create PL/SQL program
– Concept of CURSOR
– Basic concepts related to Procedure, Functions
– Basic concept of Trigger

5.1 Introduction

PL/SQL stands for Procedural Language/Structured Query Language, which is provided by Oracle as a procedural extension to SQL. SQL is a declarative language. In SQL, the statements have no control to the program and can be executed in any order. PL/SQL, on the other hand, is a procedural language that makes up for all the missing elements in SQL. PL/SQL arose from the desire of programmers to have a language structure that was more familiar than SQL's purely declarative nature.

5.2 Shortcomings in SQL

We know, SQL is a powerful tool for accessing the database but it suffers from some deficiencies as follows:

(a) SQL statements can be executed only one at a time. Every time to execute a SQL statement, a call is made to Oracle engine, thus it results in an increase in database overheads.

(b) While processing an SQL statement, if an error occurs, Oracle generates its own error message, which is sometimes difficult to understand. If a user wants to display some other meaningful error message, SQL does not have provision for that.
(c) SQL is not able to do the conditional query on RDBMS, this means one cannot use the conditions like if... then, in a SQL statement. Also looping facility (repeating a set of instructions) is not provided by SQL.

5.3 Structure of PL/SQL

PL/SQL is a 4GL (fourth generation) programming language. It offers all features of advanced programming language such as portability, security, data encapsulation, information hiding, etc. A PL/SQL program may consist of more than one SQL statements, while execution of a PL/SQL program makes only one call to Oracle engine, thus it helps in reducing the database overheads. With PL/SQL, one can use the SQL statements together with the control structures (like if... then) for data manipulation. Besides this, user can define his/her own error messages to display. Thus we can say that PL/SQL combines the data manipulation power of SQL with data processing power of procedural language.

PL/SQL is a block structured language. This means a PL/SQL program is made up of *blocks*, where block is a smallest piece of PL/SQL code having logically related statements and declarations. A block consists of three sections namely:

Declare, Begin, and Exception followed by an End statement. We will see the different sections of PL/SQL block.

Declare Section

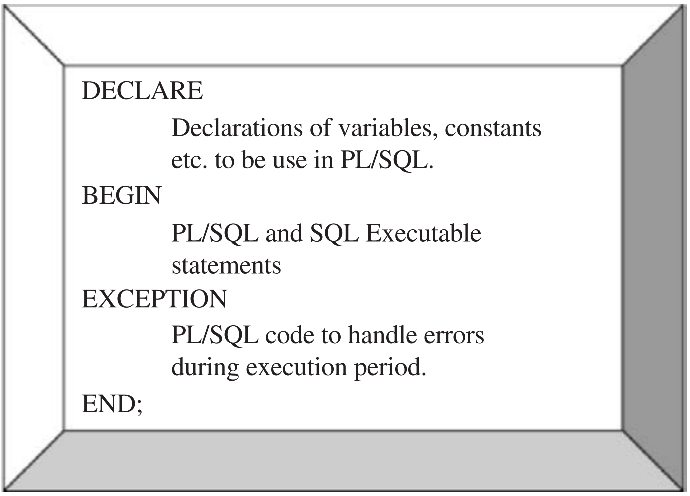
Declare section declares the variables, constants, processes, functions, etc., to be used in the other parts of program. It is an *optional* section.

Begin Section

It is the executable section. It consists of a set of SQL and PL/SQL statements, which is executed when PL/SQL block runs. It is a compulsory section.

Exception Section

This section handles the errors, which occurs during execution of the PL/SQL block. This section allows the user to define his/her own error messages. This section executes only when an error occurs. It is an *optional* section.



```
DECLARE
          Declarations of variables, constants
          etc. to be use in PL/SQL.
BEGIN
          PL/SQL and SQL Executable
          statements
EXCEPTION
          PL/SQL code to handle errors
          during execution period.
END;
```

Fig. 5.1. A PL/SQL block

End Section

This section indicates the end of PL/SQL block.

Every PL/SQL program must consist of at least one block, which may consist of any number of nested sub-blocks. Figure 5.1 shows a typical PL/SQL block.

5.4 PL/SQL Language Elements

Let us start from the basic elements of PL/SQL language. Like other programming languages PL/SQL also have specific character sets, operators, indicators, punctuations, identifiers, comments, etc. In the following sections we will discuss about various language elements of PL/SQL.

Character Set

A PL/SQL program consists of text having specific set of characters. Character set may include the following characters:

- Alphabets, both in upper case [A–Z] and lower case [a–z]
- Numeric digits [0–9]
- Special characters ( ) + − * / < > = ! ~ ˆ ; : . ′ @ % , ″ # $ & _ | { } ? [ ]
- Blank spaces, tabs, and carriage returns.

PL/SQL is not case sensitive, so lowercase letters are equivalent to corresponding uppercase letters except within string and character literals.

Lexical Units

A line of PL/SQL program contains groups of characters known as lexical units, which can be classified as follows:

- Delimiters
- Identifiers
- Literals
- Comments

Delimiters

A *delimiter* is a simple or compound symbol that has a special meaning to PL/SQL. Simple symbol consists of one character, while compound symbol consists of more than one character. For example, to perform the addition and exponentiation operation in PL/SQL, simple symbol delimiter + and compound symbol delimiter ** is used, respectively. PL/SQL supports following simple symbol delimiters:

+ − * / = > < ; % ′ , ( ) @ : ″

Compound symbol delimiters legal in PL/SQL are as follows:

<> ! =∼= ˆ= <= >= := ** .. || << >>

In the following sections we will discuss about these delimiters.

Identifiers

Identifiers are used in the PL/SQL programs to name the PL/SQL program items as constants, variables, cursors, cursor variables, subprograms, etc.

Identifiers can consists of alphabets, numerals, dollar signs, underscores, and number signs only. Any other characters like hyphens, slashes, blank spaces, etc. are illegal. An identifier must begin with an alphabetic letter optionally followed by one or more characters (permissible in identifier). An identifier cannot contain more than 30 characters.

Example

Some of the valid identifiers are as follows:

A – Identifier may consist of a single character
A1 – identifier may consist of numerals after first character
Share$price – dollar sign is permitted
e_mail – under score is permitted
phone# – number sign is permitted

The following identifiers are illegal:

mine&yours – ampersand is illegal
debit-amount – hyphen is illegal
on/off – slash is illegal
user id – space is illegal

However, PL/SQL allows space, slash, hyphen, etc. except double quotes if the identifier is enclosed within double quotes. Thus, the following identifiers are valid:

"A&B"
"TATA INFOTECH"
"True/false"
"Student(s)"
"*** BEGIN ***"

However, the maximum length of a quoted identifier cannot exceed 30 characters, excluding double quotes.

An identifier can consists of lower, upper, or mixed case letters. PL/SQL is not case sensitive except within string and character literals. So, if the only difference between identifiers is the case of corresponding letters, PL/SQL considers the identifiers to be the same. Take for example, a character string "HUMAN" as an identifier; it will be equivalent to each of following identifiers:

Human
human
hUMAN
hUmAn.

An identifier cannot be a reserve word, i.e., the words that have special meaning for PL/SQL. For example, the word DECLARE, which is used for declaring the variables or constants; words BEGIN and END, which enclose the executable part of a block or subprogram are reserve words. An attempt to redefine a reserve word gives an error.

Literals

A *literal* is an explicitly defined character, string, numeric, or Boolean value, which is not represented by an identifier. In the following sections we will discuss about each of these literals in detail:

Numeric Literals

A numeric literal is an integer or a real value. An integer literal may be a positive, negative, or unsigned whole number without a decimal point. Some examples of integer numeric literals are as follows:

100 006 −10 0 +10

A real literal is a positive, negative, or unsigned whole or fractional number with a decimal point. Some examples of real integer literals are as follows:

0.0 −19.0 3.56219 +43.99 .6 7. −4.56

PL/SQL treats a number with decimal point as a real numeric literal, even if the number does not have any numeral after decimal point. Besides integer and real literals, numeric literals can also contain exponential numbers (an optionally signed number suffix with an E (or e) followed by an optionally signed integer). Some examples of exponential numeric literals are as follows:

7E3   2.0E−3   3.14159e1   −2E33   −8.3e−2

where, E stands for "times ten to the power of." For example the exponential literal 7E3 is equivalent to following numeric literal:

7E3 = 7 * 10 ** 3 = 7*10*10*10 = 7000

Another exponential literal −8.3e−2 would be equivalent to following numeric literal:

−8.3e−2 = −8.3 * 10 ** (−2) = −8.3 *0.01 = −0.083

An exponential numeric literal cannot be smaller than 1E−130 and cannot be greater than 10E125. Note that numeric literals cannot contain dollar signs or commas.

Character Literals

A character literal is an individual character enclosed by single quotes (apostrophes). Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols. Some examples of character literals are as follows:

"A" "@" "5" "?" "," "("

PL/SQL is case sensitive within character literals. For example, PL/SQL considers the literals "A" and "a" to be different. Also, the character literals "0"..."9" are not equivalent to integer literals but can be used in arithmetic expressions because PL/SQL implicitly converts them to integers.

String Literals

A character string can be represented by an identifier or explicitly written as a string literal. A string literal is enclosed within single quotes and may consist of one or more characters. Some examples of string literals are as follows:

"Good Morning!"
"TATA INFOTECH LTD"
"04-MAY-00"
"$15,000,000"

All string literals are of character data type.

PL/SQL is case sensitive within string literals. For example, PL/SQL considers the following literals to be different:

"HUMAN"
"Human"

Boolean Literals

Boolean literals are the predefined values TRUE, FALSE, and NULL. Keep in mind Boolean literals are values, not strings. For example a condition: if (x = 10) is TRUE only for the value of x equal to 10, for any other value of x it is FALSE and for no value of x it is NULL.

Comments

Comments are used in the PL/SQL program to improve the readability and understandability of a program. A comment can appear anywhere in the program code. The compiler ignores comments. Generally, comments are used to describe the purpose and use of each code segment. A PL/SQL comment may be a single-line or multiline.

Single-Line Comments

Single-line comments begin with a double hyphen (–) anywhere on a line and extend to the end of the line.

Example

– start calculations

Multiline Comments

Multiline comments begin with a slash-asterisk (/*) and end with an asterisk-slash (*/), and can span multiple lines.

Example

/* Hello World! This is an example of multiline comments in PL/SQL */

Variables and Constants

Variables and constants can be used within PL/SQL block, in procedural statements and in SQL statements. These are used to store the values. As the program executes, the values of variables can change, but the values of constants cannot. However, it is must to declare the variables and constants, before using these in executable portion of PL/SQL. Let us see how to declare variables and constants in PL/SQL.

Declaration

1Variables and constants are declared in the Declaration section of PL/SQL block. These can be any of the SQL data type like CHAR, NUMBER, DATE, etc.

I. Variables Declaration

The syntax for declaring a variable is as follows:

identifier datatype;

Example

To declare the variable name, age, and joining_date as datatype VARCHAR2(10), NUMBER(2), DATE, respectively; declaration statement is as follows:

DECLARE
Name VARCHAR2(10);
Age NUMBER(2);
Joining_date DATE;

Initializing the Variable

By default variables are initialized to NULL at the time of declaration. If we want to initialize the variable by some other value, syntax would be as follows:

Identifier datatype := value;
Or,
Identifier datatype DEFAULT value;

Example

If a number of employees have same joining_date, say 01-JULY-99. It is better to initialize the joining_date rather than entering the same value individually, any of the following declaration can be used:

Joining_date DATE := 01-JULY-99; (or)
Joining_date DATE DEFAULT 01-JULY-99;

Constraining a Variable

Variables can be NOT NULL constrained at the time of declaring these, for example to constrain the joining_date NOT NULL, the declaration statement would be as follows:

Joining_date DATE NOT NULL: = 01-JULY-99;

(NOT NULL constraint must be followed by an initialization clause) thus following declaration will give an error:

Joining_date DATE NOT NULL; – illegal

Declaring Constants

Declaration of constant is similar to declaration of variable, except the keyword CONSTANT precedes the datatype and it must be initialized by some value. The syntax for declaring a constant is as follows:

identifier CONSTANT datatype := value;

Example

To define the age_limit as a constant, having value 30; the declaration statement would be as follows: Age_limit CONSTANT NUMBER := 30;

Restrictions

PL/SQL imposes some restrictions on declaration as follows:

(a) A list of variables that have the same datatype cannot be declared in the same row

Example

A, B, C NUMBER (4,2); – illegal
It should be declared in separate lines as follows:

A NUMBER (4,2);
B NUMBER (4,2);
C NUMBER (4,2);

(b) A variable can reference to other variable if and only if that variable is declared before that variable. The following declaration is illegal:

A NUMBER(2) := B;
B NUMBER(2) := 4;

Correct declaration would be as follows:

B NUMBER(2) := 4;
A NUMBER(2) := B;

(c) In a block same identifier cannot be declared by different datatype. The following declaration is illegal:

DECLARE
X NUMBER(4,2);
X CHAR(4); – illegal

5.5 Data Types

Every constant and variable has a datatype. A datatype specifies the space to be reserved in the memory, type of operations that can be performed, and valid range of values. PL/SQL supports all the built-in SQL datatypes. Apart from those datatypes, PL/SQL provides some other datatypes. Some commonly used PL/SQL datatypes are as follows:

BOOLEAN

One of the mostly used datatype is BOOLEAN. A BOOLEAN datatype is assigned to those variables, which are required for logical operations. A BOOLEAN datatype variable can store only logical values, i.e., TRUE, FALSE, or NULL. A BOOLEAN variable value cannot be inserted in a table; also, a table data cannot be selected or fetched into a BOOLEAN variable.

%Type

The %TYPE attribute provides the datatype of a variable or database column. In the following example, %TYPE provides the datatype of a variable:

```
balance NUMBER(8,2);
minimum_balance balance%TYPE;
```

In the above example PL/SQL will treat the minimum_balance of the same datatype as that of balance, i.e., NUMBER(8,2). The next example shows that a %TYPE declaration can include an initialization clause:

```
balance NUMBER(7,2);
minimum_balance balance%TYPE := 500.00;
```

The %TYPE attribute is particularly useful when declaring variables that refer to database columns. Column in a table can be referenced by %TYPE attribute.

Example

To declare a column my_empno of the same datatype as that of empno column of emp table in scott/tiger user, the declaration statement would be as follows:

```
my_empno scott.emp.empno%TYPE;
```

Using %TYPE to declare my_empno has two advantages. First, the knowledge of exact datatype of empno is not required. Second, if the database definition of empno changes, the datatype of my_empno changes accordingly at run time. But %TYPE variables do not inherit the NOT NULL column constraint, even though the database column empno is defined as NOT NULL, one can assign a null to the variable my_empno.

%Rowtype

The %ROWTYPE attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table.

Example

emp_rec is declared as a record datatype of emp table. emp_rec can store a row selected from the emp table.

emp_rec emp%ROWTYPE;

Expressions

Expressions are constructed using operands and operators. PL/SQL supports all the SQL operators; in addition to those operators it has one more operator, named exponentiation (symbol is **). An operand is a variable, constant, literal, or function call that contributes a value to an expression. An example of simple expression follows:

$$A = B * *3$$

where A, B, and 3 are operand; = and ** are operators. B**3 is equivalent to value of thrice multiplying the B, i.e., B*B*B.

Operators may be unary or binary. Unary operators such as the negation operator (−) operate on one operand; binary operators such as the division operator (/) operate on two operands. PL/SQL evaluates (finds the current value of) an expression by combining the values of operands in ways specified by the operators. This always yields a single value and datatype. PL/SQL determines the datatype by examining the expression and the context in which it appears.

5.6 Operators Precedence

The operations within an expression are done in a particular order depending on their precedence (priority). Table 5.1 lists the operator's level of precedence from top to bottom. Operators listed in the same row have equal precedence.

Operators with higher precedence are applied first, but if parentheses are used, expression within innermost parenthesis is evaluated first. For example the expression $8 + 4/2 * *2$ results in a value 9, because exponentiation has the highest priority followed by division and addition. Now in the same expression if we put parentheses, the expression $8+((4/2)**2)$ results in a value 12 not 9, because now first it will solve the expression within innermost parentheses.

Table 5.1. Order of operations

operator	operation
**, NOT	exponentiation, logical negation
+, −	identity, negation
*, /	multiplication, division
+, −, ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
AND	conjunction
OR	disjunction

5.7 Control Structure

Control structure is an essential part of any programming language. It controls the flow of process. Control structure is broadly divided into three categories:

– Conditional control,
– Iterative control, and
– Sequential control

In the following sections we will discuss about each of these control structures in detail.

Conditional Control

A conditional control structure tests a condition to find out whether it is true or false and accordingly executes the different blocks of SQL statements. Conditional control is generally performed by IF statement. There are three forms of IF statement. IF-THEN, IF-THEN-ELSE, IF-THEN-ELSEIF.

IF-THEN

It is the simplest form of IF condition. The syntax for this statement is as follows:

```
IF condition THEN
Sequence of statements
END IF;
```

Example

To compare the values of two variables A and B and to assign the value of A to HIGH if A is greater than B. The IF construct for this is as follows:

```
IF A > B THEN
HIGH := A;
ENDIF;
```

The sequence of statements is executed only if the condition is true. If the condition is FALSE or NULL, the sequence of statements is skipped and processing continues from statements following END IF statements.

IF-THEN-ELSE

As it is clear with the IF-THEN construct, if condition is FALSE the control exits to next statement out of IF-THEN clause. To execute some other set of statements in case condition evaluates to FALSE, the second form of IF statement is used, it adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
sequence_of_statements1
ELSE
sequence_of_statements2
END IF;
```

Example

To become clear about it, take the previous example, to compare the value of A and B and assign the value of greater number to HIGH. The IF construct for this is as follows:

```
IF A > B THEN
HIGH := A;
ELSE
HIGH := B;
ENDIF;
```

The sequence of statements in the ELSE clause is executed only if the condition is FALSE or NULL.

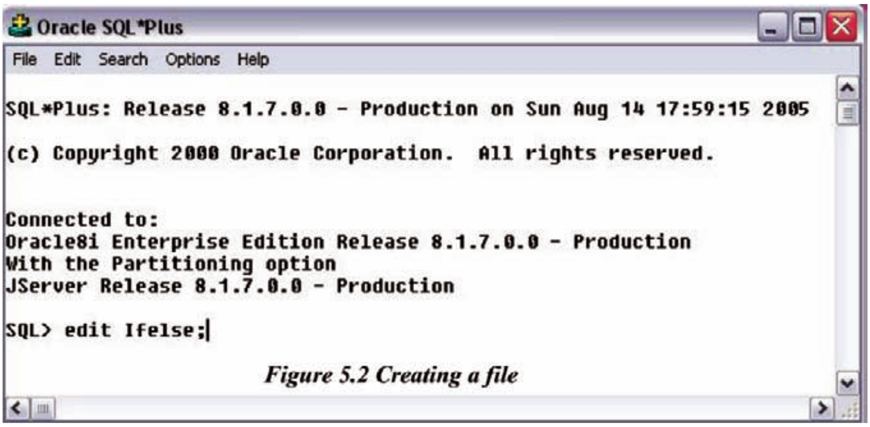
IF-THEN-ELSIF

In the previous constructs of IF, we can check only one condition, whether it is true or false. There is no provision if we want to check some other conditions if first condition evaluates to FALSE; for this purpose third form of IF statement is used. It selects an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
sequence_of_statements1
ELSIF condition2 THEN
sequence_of_statements2
ELSE
sequence_of_statements3
END IF;
```

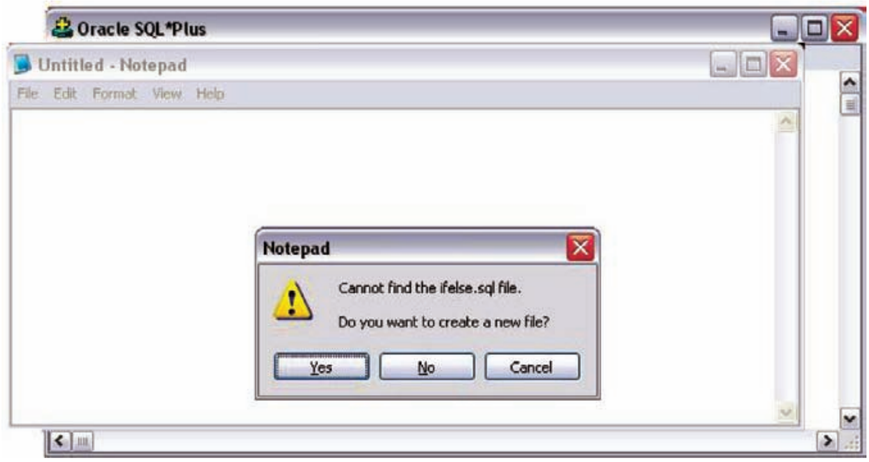
5.8 Steps to Create a PL/SQL Program

1. First a notepad file can be created as typing in the Oracle SQL editor. Figure 5.2 shows the command to create a file,
2. Then a Notepad file will appear and at the same time background Oracle will be disabled. It is shown in Fig. 5.3
3. We can write our PL/SQL program in that file, save that file, and we can execute that program in the Oracle editor as in Fig. 5.4. In this program Cursor (*Cur*rent *S*et *o*f *R*ecords) concept is used which we will see in the following pages. Here content of EMP table is opened by the cursor and they are displayed by the DBMS_OUTPUT package. Command IF is used to check whether the cursor has been opened successfully by using %Found attribute.
4. Then we can execute that file as follows in Fig. 5.5



```
SQL*Plus: Release 8.1.7.0.0 - Production on Sun Aug 14 17:59:15 2005

(c) Copyright 2000 Oracle Corporation.  All rights reserved.


Connected to:
Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production

SQL> edit Ifelse;
```

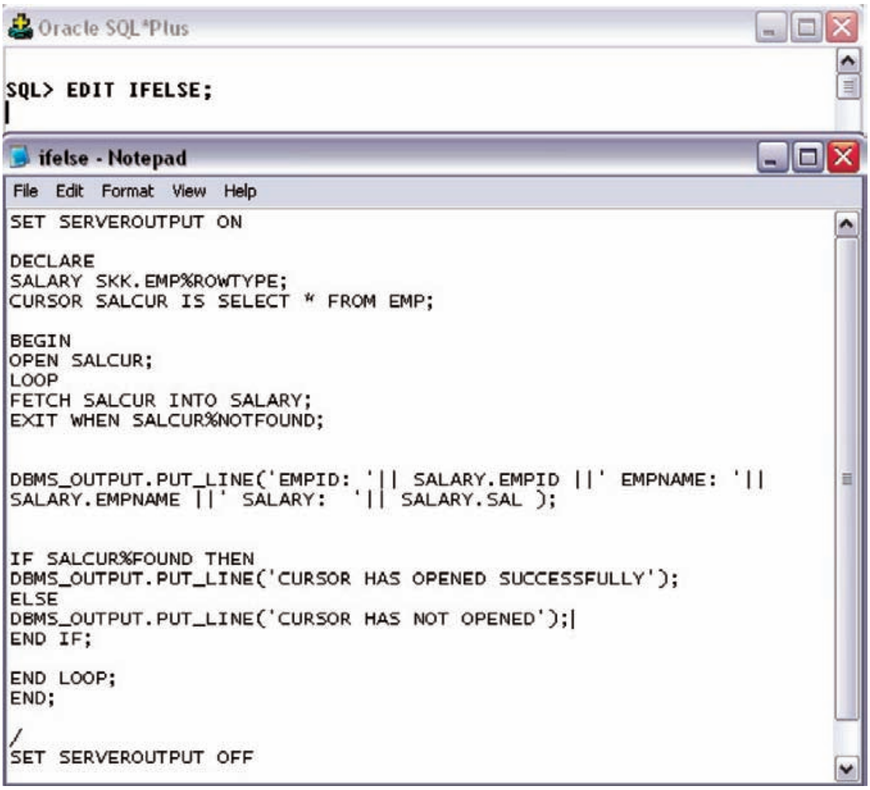
Fig. 5.2. Creating a file



Cannot find the ifelse.sql file.

Do you want to create a new file?

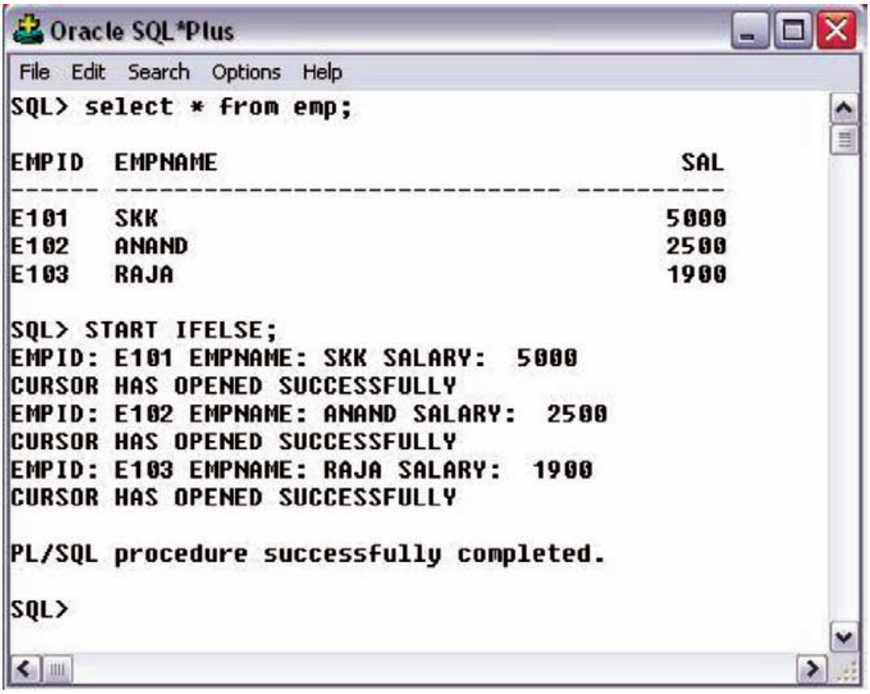
Fig. 5.3. Confirmation for the file created



```
SQL> EDIT IFELSE;
```

```
SET SERVEROUTPUT ON

DECLARE
SALARY SKK.EMP%ROWTYPE;
CURSOR SALCUR IS SELECT * FROM EMP;

BEGIN
OPEN SALCUR;
LOOP
FETCH SALCUR INTO SALARY;
EXIT WHEN SALCUR%NOTFOUND;


DBMS_OUTPUT.PUT_LINE('EMPID: '|| SALARY.EMPID ||' EMPNAME: '||
SALARY.EMPNAME ||' SALARY:  '|| SALARY.SAL );


IF SALCUR%FOUND THEN
DBMS_OUTPUT.PUT_LINE('CURSOR HAS OPENED SUCCESSFULLY');
ELSE
DBMS_OUTPUT.PUT_LINE('CURSOR HAS NOT OPENED');
END IF;

END LOOP;
END;

/
SET SERVEROUTPUT OFF
```

Fig. 5.4. Program writing to the notepad



```
SQL> select * from emp;

EMPID  EMPNAME                                   SAL
------ ------------------------------ ----------
E101   SKK                                      5000
E102   ANAND                                    2500
E103   RAJA                                     1900

SQL> START IFELSE;
EMPID: E101 EMPNAME: SKK SALARY:  5000
CURSOR HAS OPENED SUCCESSFULLY
EMPID: E102 EMPNAME: ANAND SALARY:  2500
CURSOR HAS OPENED SUCCESSFULLY
EMPID: E103 EMPNAME: RAJA SALARY:  1900
CURSOR HAS OPENED SUCCESSFULLY

PL/SQL procedure successfully completed.

SQL>
```

Fig. 5.5. Program execution

5.9 Iterative Control

In iterative control a group of statements are executed repeatedly till certain condition is true, and control exits from loop to next statement when the condition becomes false. There are mainly three types of loop statements:

LOOP, WHILE-LOOP, FOR-LOOP.

LOOP

LOOP is the simplest form of iterative control. It encloses a sequence of statements between the keywords LOOP and END LOOP. The general syntax for LOOP control is as follows:

```
LOOP
sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements gets executed, then control reaches at the top of the loop. But a control structure like this gets entrapped into infinite loop. To avoid this it is must to use the key word EXIT and EXIT-WHEN.

LOOP – EXIT

An EXIT statement within LOOP forces the loop to terminate unconditionally and passes the control to next statements. The general syntax for this is as follows:

```
LOOP
IF condition1 THEN
Sequence of statements1
EXIT;
ELSIF condition2 THEN
Sequence of statements2
EXIT
ELSE
Sequence of statements3
EXIT;
END IF;
END LOOP;
```

LOOP – EXIT WHEN

The EXIT-WHEN statement terminates a loop conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop terminates and control passes to the next statement after the loop. The syntax for this is as follows:

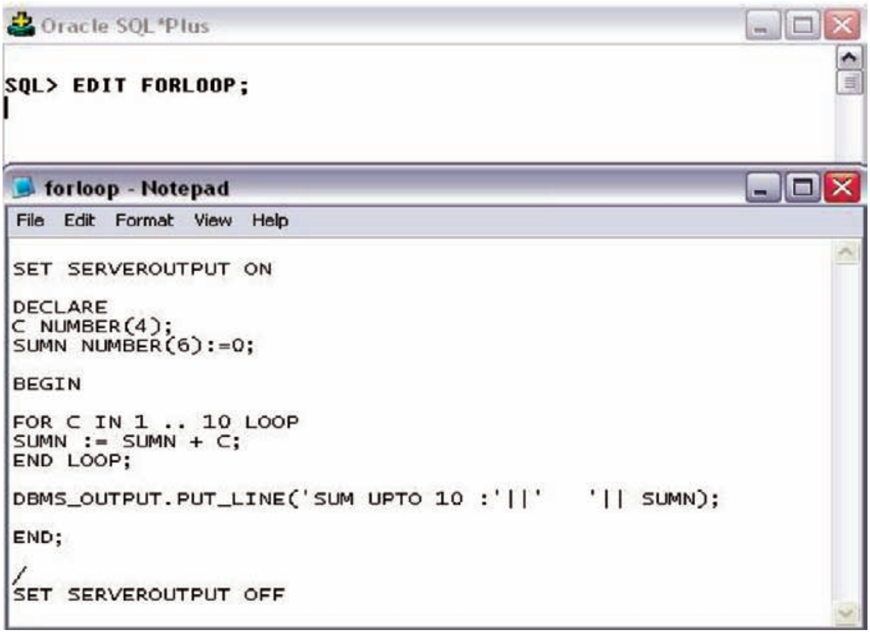
```
LOOP
EXIT WHEN condition
Sequence of statements
END LOOP
```

Example

Figures 5.4 and 5.5 are also the example of LOOP – EXIT WHEN. Condition used here is that the cursor does not return anything by using %NOTFOUND attribute.

WHILE-LOOP

The WHILE statement with LOOP checks the condition. If it is true then only the sequence of statements enclosed within the loop gets executed. Then control resumes at the top of the loop and checks the condition again; if it is true the sequence of statements enclosed within the loop gets executed. The process is repeated till the condition is true. The control passes to the next statement outside the loop for FALSE or NULL condition.



```
SQL> EDIT FORLOOP;
```

```
SET SERVEROUTPUT ON

DECLARE
C NUMBER(4);
SUMN NUMBER(6):=0;

BEGIN

FOR C IN 1 .. 10 LOOP
SUMN := SUMN + C;
END LOOP;

DBMS_OUTPUT.PUT_LINE('SUM UPTO 10 :'||'    '|| SUMN);

END;

/
SET SERVEROUTPUT OFF
```

Fig. 5.6. Example for FOR Loop

```
WHILE condition LOOP
Sequence of statements
END LOOP;
```

FOR-LOOP

FOR loops iterate over a specified range of integers. The range is part of *iteration scheme*, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax is as follows:

```
FOR counter IN lower_limit .. higher_limit LOOP
sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated. The sequence of statements is executed once for each integer in the range. After every iteration, the loop counter is incremented.

Example

To find the sum of natural numbers up to 10, the following program can be used as in Fig. 5.6.

Sequential Control

The sequential control unconditionally passes the control to specified unique label; it can be in the forward direction or in the backward direction. For sequential control GOTO statement is used. Overuse of GOTO statement may increase the complexity, thus as far as possible avoid the use of GOTO statement.

The syntax is as follows:

```
GOTO label;
........
........
<<label>>
Statement
```

5.10 Cursors

Number of rows returned by a query can be zero, one, or many, depending on the query search conditions. In PL/SQL, it is not possible for an SQL statement to return more than one row. In such cases we can use cursors. A cursor is a mechanism that can be used to process the multiple row result sets one row at a time.

In other words, cursors are constructs that enable the user to name a private memory area to hold a specific statement for access at a later time. Cursors are an inherent structure in PL/SQL. Cursors allow users to easily store and process sets of information in PL/SQL program.

Figure 5.7 shows the simple example for the cursor where two rows are selected from the query and they are pointed by the cursor namely All_Lifetime.

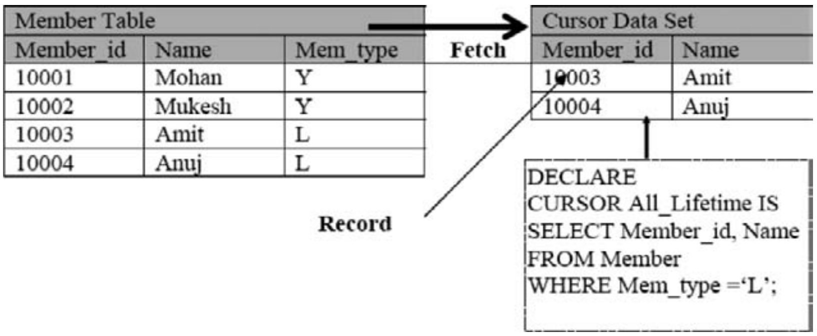


Fig. 5.7. Cursor example

There are two types of cursors in Oracle

1. Implicit cursors
2. Explicit cursors

5.10.1 Implicit Cursors

PL/SQL implicitly declares a cursor for every SQL DML statement, such as INSERT, DELETE, UPDATE, and SELECT statement that is not a part of an explicitly declared cursor, even if the statement processes a single row. PL/SQL allows referencing the most recent cursor or the cursor associated with the most recently executed SQL statement, as the "SQL" cursor. Cursor attributes are used to access information about the most recently executed SQL statement, using SQL cursor.

Implicit Cursor Attributes

In PL/SQL every cursor, implicit or explicit, has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. These cursor attributes can be used in procedural statements (PL/SQL), but not in SQL statements. These attributes let user access information about the most recent execution of INSERT, UPDATE, SELECT INTO, and DELETE commands. These attributes are associated with the implicit "SQL" cursor and can be accessed by appending the attribute name to the implicit cursor name (SQL). Syntax to use cursor attribute is as follows:

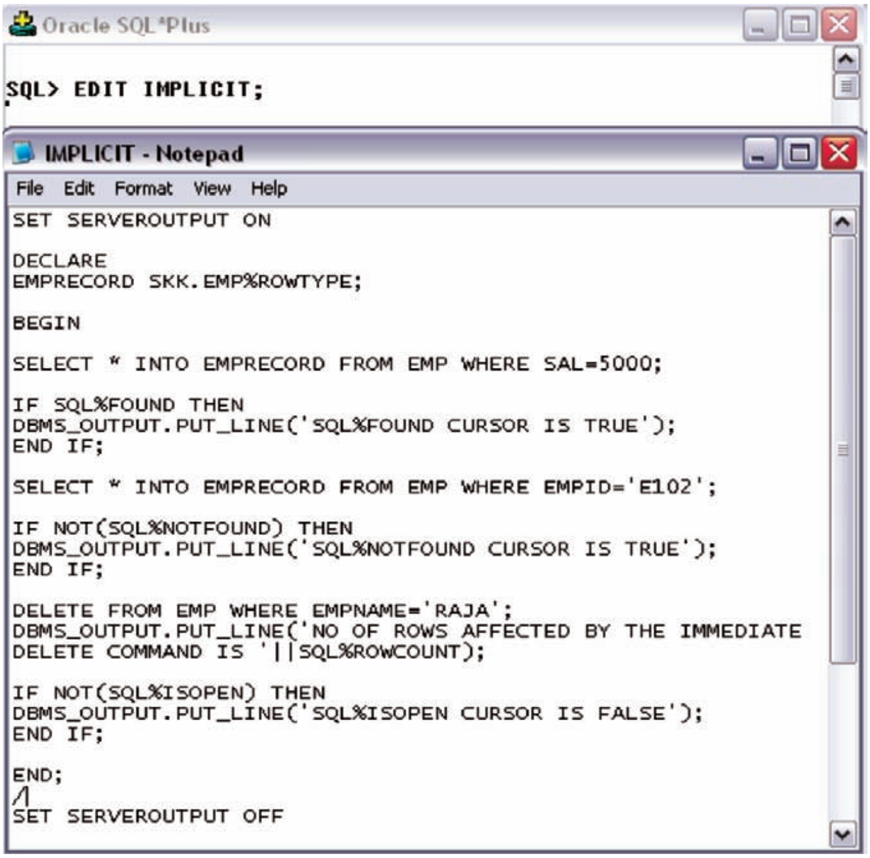
SQL %<attribute name>

%Notfound

This attribute is used to determine if any rows were processed by a SQL DML statement. This attribute evaluates to TRUE if an INSERT, UPDATE, or DELETE affected no rows or a SELECT INTO returned no rows. Otherwise, it returns FALSE. %NOTFOUND attribute can be useful in reporting or processing when no data is affected. If a SELECT statement does not return any data, the predefined exception NO_DATA_FOUND is automatically raised, and program control is sent to an exception handler, if it is present in the program. If a check is made on %NOTFOUND attribute after a SELECT statement, it will be completely skipped when the SELECT statement returns no data.

Example

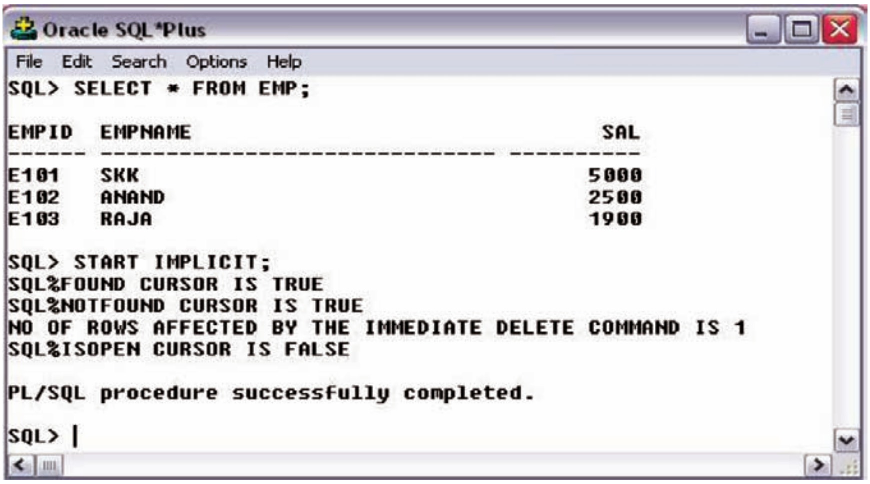
Figures 5.8 and 5.9 show the example of all the implicit cursor attributes. The program will return the status of each cursor attribute depending on the previously executed DML statement.



```
SQL> EDIT IMPLICIT;
```

```
SET SERVEROUTPUT ON

DECLARE
EMPRECORD SKK.EMP%ROWTYPE;

BEGIN

SELECT * INTO EMPRECORD FROM EMP WHERE SAL=5000;

IF SQL%FOUND THEN
DBMS_OUTPUT.PUT_LINE('SQL%FOUND CURSOR IS TRUE');
END IF;

SELECT * INTO EMPRECORD FROM EMP WHERE EMPID='E102';

IF NOT(SQL%NOTFOUND) THEN
DBMS_OUTPUT.PUT_LINE('SQL%NOTFOUND CURSOR IS TRUE');
END IF;

DELETE FROM EMP WHERE EMPNAME='RAJA';
DBMS_OUTPUT.PUT_LINE('NO OF ROWS AFFECTED BY THE IMMEDIATE
DELETE COMMAND IS '||SQL%ROWCOUNT);

IF NOT(SQL%ISOPEN) THEN
DBMS_OUTPUT.PUT_LINE('SQL%ISOPEN CURSOR IS FALSE');
END IF;

END;
/
SET SERVEROUTPUT OFF
```

Fig. 5.8. Implicit cursor example program



```
SQL> SELECT * FROM EMP;

EMPID  EMPNAME                              SAL
------ ------------------------------ ----------
E101   SKK                                  5000
E102   ANAND                                2500
E103   RAJA                                 1900

SQL> START IMPLICIT;
SQL%FOUND CURSOR IS TRUE
SQL%NOTFOUND CURSOR IS TRUE
NO OF ROWS AFFECTED BY THE IMMEDIATE DELETE COMMAND IS 1
SQL%ISOPEN CURSOR IS FALSE

PL/SQL procedure successfully completed.

SQL> |
```

Fig. 5.9. Implicit cursor example execution

%Found

This attribute is used to determine if any rows were processed by a SQL DML statement. In fact %FOUND works just the opposite of %NOTFOUND attribute. Until a SQL DML statement is executed, this attribute evaluates to NULL. It equates to TRUE if an INSERT, UPDATE, or DELETE affects one or more rows or select returns one row. If a select statement returns more than one row, the predefined exception TOO_MANY_ROWS is automatically raised and %FOUND attribute is set to FALSE.

%Rowcount

This attribute is used to determine the number of rows that are processed by an SQL statement. It returns the number of rows affected by an INSERT, UPDATE, or DELETE statement or returned by a SELECT INTO statement. %ROWCOUNT returns zero if the SQL statement affects or returns no rows. If a SELECT statement returns more than one row, the predefined exception TOO_MANY_ROWS is raised automatically. In such a case %ROWCOUNT attribute is set to 1 and not the actual number of rows that satisfy the query.

Example

Figures 5.8 and 5.9 show this example.

%Isopen

%ISOPEN is used to determine if a cursor is already open. It always equates to FALSE in an implicit cursor. Oracle automatically closes implicit cursor after executing its associated SQL statements.

Example

Figures 5.8 and 5.9 show this example.

5.10.2 Explicit Cursor

Explicit cursors are declared by the user and are used to process query results that return multiple rows. Multiple rows returned from a query form a set called an **active set**. PL/SQL defines the size of the active set as the number of rows that have met search criteria. Inherent in every cursor is a pointer that keeps track of the multiple rows being accessed, enabling program to process the rows one at a time. An explicit cursor points to the current row in the active set. This allows the program to process one row at a time.

Multirow query processing is somewhat like file processing. For example, a program opens a file to process records, and then closes the file. Likewise,

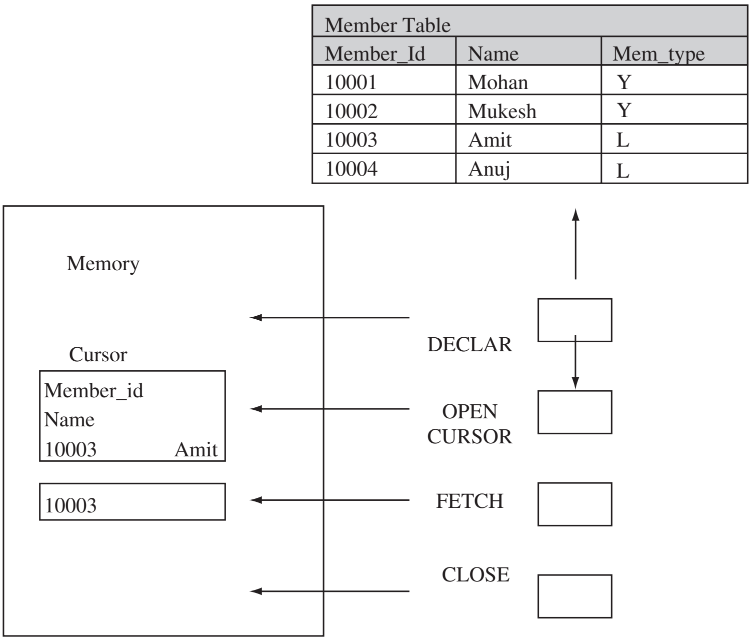


Fig. 5.10. Cursor and memory utilization

a PL/SQL program opens a cursor to process rows returned by a query, and then closes the cursor. Just as a file pointer marks the current position in an open file, a cursor marks the current position in an active set.

After a cursor is declared and opened, the user can FETCH, UPDATE, or DELETE the current row in the active set. The cursor can be CLOSED to disable it and free up any allocated system resources. Three commands are used to control the cursor – OPEN, FETCH, and CLOSE. First the cursor is initialized with an OPEN statement, which identifies the active set. Then, the FETCH statement is used to retrieve the first row. FETCH statement can be executed repeatedly until all rows have been retrieved. When the last row has been processed, the cursor can be released with the CLOSE statement. Figure 5.10 shows the memory utilization by a cursor when each of these statements is given.

5.11 Steps to Create a Cursor

Following are the steps to create a cursor:

5.11.1 Declare the Cursor

In PL/SQL a cursor, like a variable, is declared in the DECLARE section of a PL/SQL block or subprogram. A cursor must be declared before it can be referenced in other statements. A cursor is defined in the declarative part by naming it and specifying a SELECT query to define the active set.

```
CURSOR <cursor_name> IS
SELECT...
```

The SELECT statement associated with a cursor declaration can reference previously declared variables.

Declaring Parameterized Cursors

PL/SQL allows declaration of cursors that can accept input parameters which can be used in the SELECT statement with WHERE clause to select specified rows. Syntax to declare a parameterized cursor:

```
CURSOR <cursor_name> [(parameter......)] IS
SELECT......
WHERE <column_name> = parameter;
```

Parameter is an input parameter defined with the syntax:

```
<variable_name> [IN] <datatype> [{:= | DEFAULT} value]
```

The formal parameters of a cursor must be IN parameters. As in the example above, cursor parameters can be initialized to default values. That way, different numbers of actual parameters can be passed to a cursor, accepting or overriding the default values.

Moreover, new formal parameters can be added without having to change every reference to the cursor. The scope of a cursor parameter is local only to the cursor. A cursor parameter can be referenced only within the SELECT statement associated with the cursor declaration. The values passed to the cursor parameters are used by the SELECT statement when the cursor is opened.

5.11.2 Open the Cursor

After declaration, the cursor is opened with an OPEN statement for processing rows in the cursor. The SELECT statement associated with the cursor is executed when the cursor is opened, and the active set associated with the cursor is created.

The active set is defined when the cursor is declared, and is created when cursor is opened.

The active set consists of all rows that meet the SELECT statement criteria. Syntax of OPEN statement is as follows.

```
OPEN <cursor_name>;
```

5.11.3 Passing Parameters to Cursor

Parameters to a parameterized cursor can be passed when the cursor is opened. For example, given the cursor declaration

CURSOR Mem_detail (MType VARCHAR2) IS SELECT...

Any of the following statements opens the cursor.

OPEN Mem_detail('L');
OPEN Mem_detail(Mem); where Mem is another variable.

Unless default values are to be accepted, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. Formal parameters declared with a default value need not have a corresponding actual parameter. They can simply assume their default values when the OPEN statement is executed. The formal parameters of a cursor must be IN parameters. Therefore, they cannot return values to actual parameters. Each actual parameter must belong to a datatype compatible with the datatype of its corresponding formal parameter.

5.11.4 Fetch Data from the Cursor

After a cursor has been opened, the SELECT statement associated with the cursor is executed and the active set is created. To retrieve the rows in the active set one row at a time, the rows must be fetched individually from the cursor. After each FETCH statement, the cursor advances to the next row in the active set and retrieves it. Syntax of FETCH is:

FETCH <cursor_name> INTO <variable_name>, <variable_name>....

where variable_name is the name of a variable to which a column value is assigned. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. This variable datatype must be compatible with the corresponding database column.

5.11.5 Close the Cursor

After processing the rows in the cursor, it is released with the CLOSE statement. To change the active set in a cursor or the values of the variables referenced in the cursor SELECT statement, the cursor must be released with CLOSE statement. Once a cursor is CLOSEd, it can be reOPENed. The CLOSE statement disables the cursor, and the active set becomes undefined. For example, to CLOSE Mem_detail close statement will be:

```
CLOSE <cursor_name>;
```

Example

Figures 5.4 and 5.5 show the example of declaring, opening, and fetching the cursor called SALCUR.

Explicit Cursor Attributes

It is used to access useful information about the status of an explicit cursor. Explicit cursors have the same set of cursor attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. These attributes can be accessed in PL/SQL statements only, not in SQL statements. Syntax to access an explicit cursor attributes:

```
<cursor_name>%<attribute_name>
```

%Notfound

When a cursor is OPENed, the rows that satisfy the associated query are identified and form the active set. Before the first fetch, %NOTFOUND evaluates to NULL. Rows are FETCHed from the active set one at a time. If the last fetch returned a row, %NOTFOUND evaluates to FALSE. If the last fetch failed to return a row because the active set was empty, %NOTFOUND evaluates to TRUE. FETCH is expected to fail eventually, so when that happens, no exception is raised.

Example

Figures 5.4 and 5.5 show the example for this attribute. In this example, it is used for checking whether all the rows have been fetched or not.

%Found

%FOUND is the logical opposite of %NOTFOUND. After an explicit cursor is open but before the first fetch, %FOUND evaluates to NULL. Thereafter, it evaluates to TRUE if the last fetch returned a row or to FALSE if no row was returned. If a cursor is not open, referencing it with %FOUND raises INVALID_CURSOR exception.

Example

Figures 5.4 and 5.5 show the example for this attribute. In this example, it is used for checking whether the cursor has been opened successfully or not.

%Rowcount

When you open a cursor, %ROWCOUNT is initialized to zero. Before the first fetch, %ROWCOUNT returns a zero. Thereafter, it returns the number of rows fetched so far. The number is incremented if the latest fetch returned a row.

Example

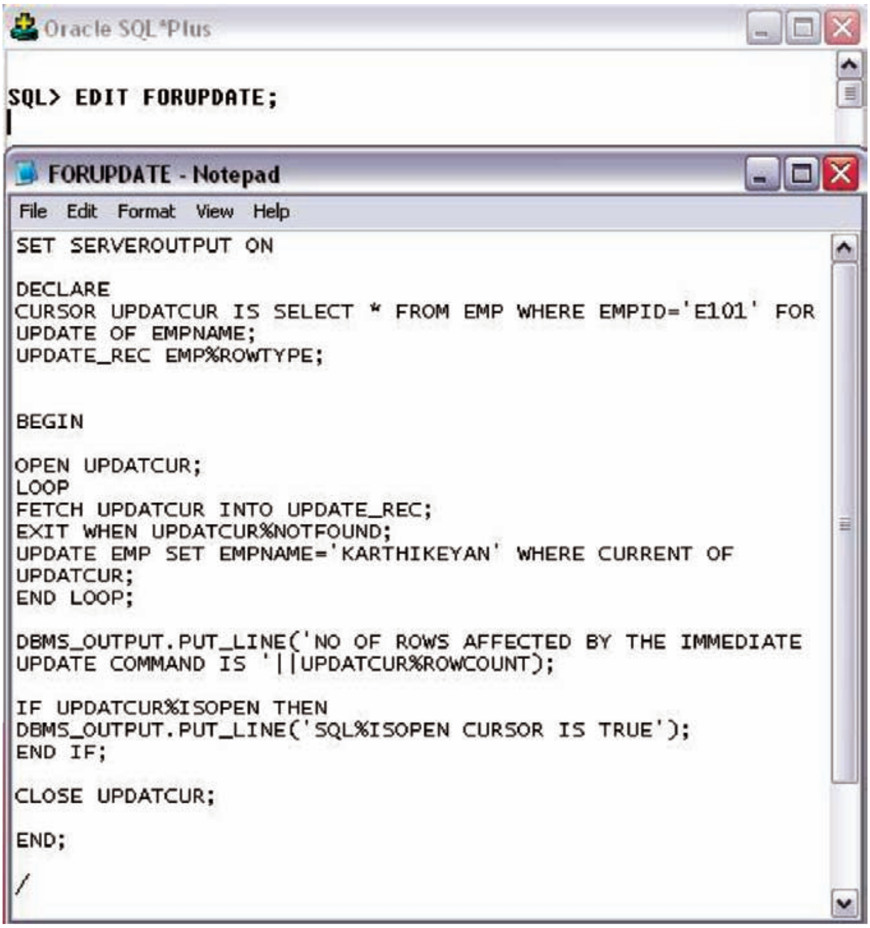
Figures 5.8 and 5.9 show the example of this attribute where cursor updatcur is used.

%Isopen

%ISOPEN evaluates to TRUE if the cursor is open; otherwise, %ISOPEN evaluates to FALSE.

Example

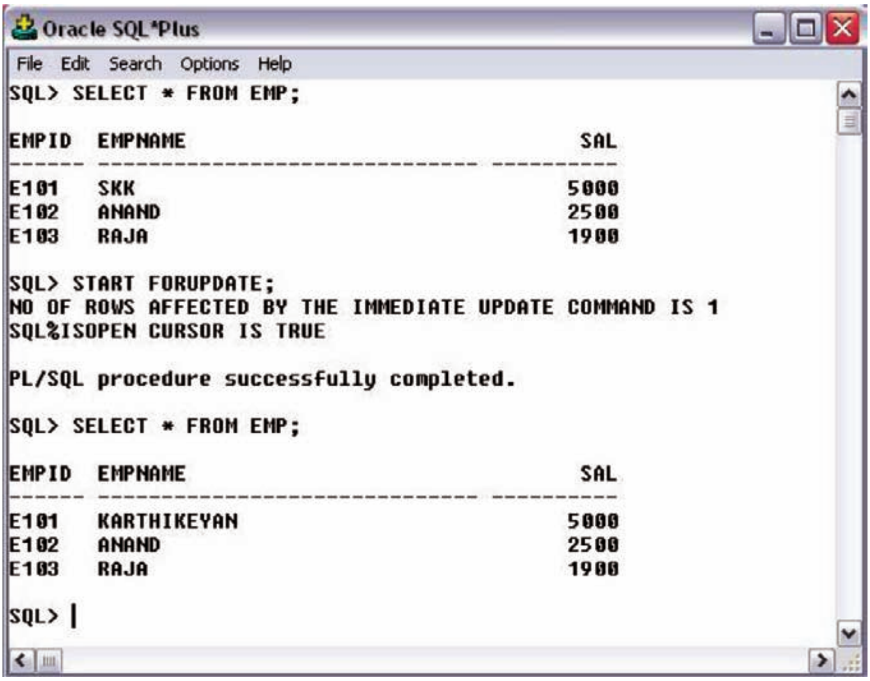
Figures 5.11 and 5.12 show the example of this attribute where cursor updatcur is used.



```
SQL> EDIT FORUPDATE;
```

```
SET SERVEROUTPUT ON

DECLARE
CURSOR UPDATCUR IS SELECT * FROM EMP WHERE EMPID='E101' FOR
UPDATE OF EMPNAME;
UPDATE_REC EMP%ROWTYPE;


BEGIN

OPEN UPDATCUR;
LOOP
FETCH UPDATCUR INTO UPDATE_REC;
EXIT WHEN UPDATCUR%NOTFOUND;
UPDATE EMP SET EMPNAME='KARTHIKEYAN' WHERE CURRENT OF
UPDATCUR;
END LOOP;

DBMS_OUTPUT.PUT_LINE('NO OF ROWS AFFECTED BY THE IMMEDIATE
UPDATE COMMAND IS '||UPDATCUR%ROWCOUNT);

IF UPDATCUR%ISOPEN THEN
DBMS_OUTPUT.PUT_LINE('SQL%ISOPEN CURSOR IS TRUE');
END IF;

CLOSE UPDATCUR;

END;

/
```

Fig. 5.11. Example of FOR UPDATE clause



```
SQL> SELECT * FROM EMP;

EMPID  EMPNAME                               SAL
------ ------------------------------ ----------
E101   SKK                                  5000
E102   ANAND                                2500
E103   RAJA                                 1900

SQL> START FORUPDATE;
NO OF ROWS AFFECTED BY THE IMMEDIATE UPDATE COMMAND IS 1
SQL%ISOPEN CURSOR IS TRUE

PL/SQL procedure successfully completed.

SQL> SELECT * FROM EMP;

EMPID  EMPNAME                               SAL
------ ------------------------------ ----------
E101   KARTHIKEYAN                          5000
E102   ANAND                                2500
E103   RAJA                                 1900

SQL>
```

Fig. 5.12. FOR UPDATE clause execution

Using FOR UPDATE and CURRENT

The FOR UPDATE clause is used to specify that the rows in the active set of a cursor are to be locked for modification. Locking allows the rows in the active set to be modified exclusively by your program. This protects simultaneous modifications until update by one transaction is complete.

CURSOR <cursor_name> IS SELECT <column_name> [.....] FROM.....
FOR UPDATE [OF <column_name> . . . . . .];

FOR UPDATE specifies that the rows of the active set are to be exclusively locked when the cursor is opened and specifies the column names that can be updated. The FOR UPDATE clause must be used in the cursor declaration statement whenever UPDATE or DELETE are to be used after the rows are FETCHed from a cursor.

Syntax of CURRENT clause with UPDATE statement is:

UPDATE <table_name> SET <column_name> = expression [.....]
WHERE CURRENT OF <cursor_name>;

Syntax of CURRENT OF Clause with DELETE Statement is:

DELETE table_name WHERE CURRENT OF cursor_name;

Example

Figures 5.11 and 5.12 show this example where a row of id E101 is locked for updation and its name of the Employee is changed to Karthikeyan.

Cursor FOR Loop

PL/SQL provides FOR loop to manage cursors effectively in situations where the rows in the active set of cursor are to be repeatedly processed in a looping manner. A cursor FOR loop simplifies all aspects of processing a cursor. Cursor FOR loop can be used instead of the OPEN, FETCH, and CLOSE statements.

A cursor FOR loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor, repeatedly fetches rows of values from the active set into fields in the record, and closes the cursor when all rows have been processed. Syntax to declare and process a cursor in a cursor FOR loop is:

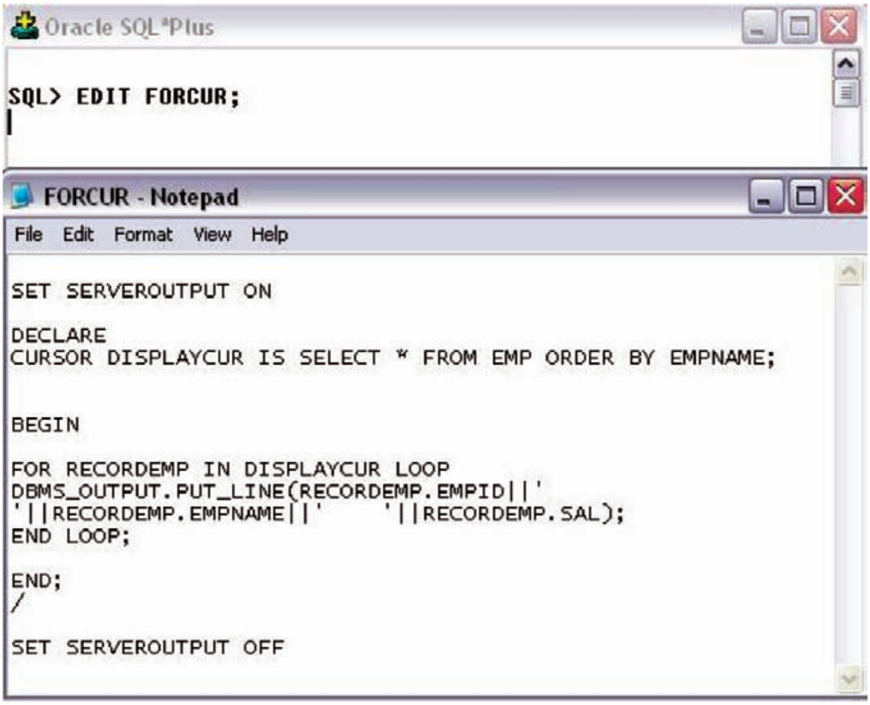
```
FOR <record_name> IN <cursor_name> LOOP
.........
END LOOP;
```

where record_name is the cursor FOR loop index implicitly declared as a record of type %ROWTYPE. Cursor is assumed to be declared in the DECLARE section. In the FOR loop declaration, the FOR loop index is uniquely named and implicitly declared as a record of type %ROWTYPE. This RECORD variable consists of columns referenced in the cursor SELECT statement.

In the FOR loop, the cursor is implicitly opened for processing. No explicit OPEN statement is required. Inside the FOR loop, the column values for each row in the active set can be referenced by the FOR loop index with dot notation in any PL/SQL or SQL statement. Before any iteration of the FOR loop, PL/SQL fetches into the implicitly declared record, which is equivalent to a record declared explicitly. At the end of the active set, the FOR loop implicitly closes the cursor and exits the FOR loop. No explicit CLOSE statement is required. A COMMIT statement is still required to complete the operation. We can pass parameters to a cursor used in a cursor FOR loop. The record is defined only inside the loop. We cannot refer to its fields outside the loop. The sequence of statements inside the loop is executed once for each row that satisfies the query associated with the cursor. On leaving the loop, the cursor is closed automatically. This is true even if an EXIT or GOTO statement is used to leave the loop prematurely or if an exception is raised inside the loop.

Example

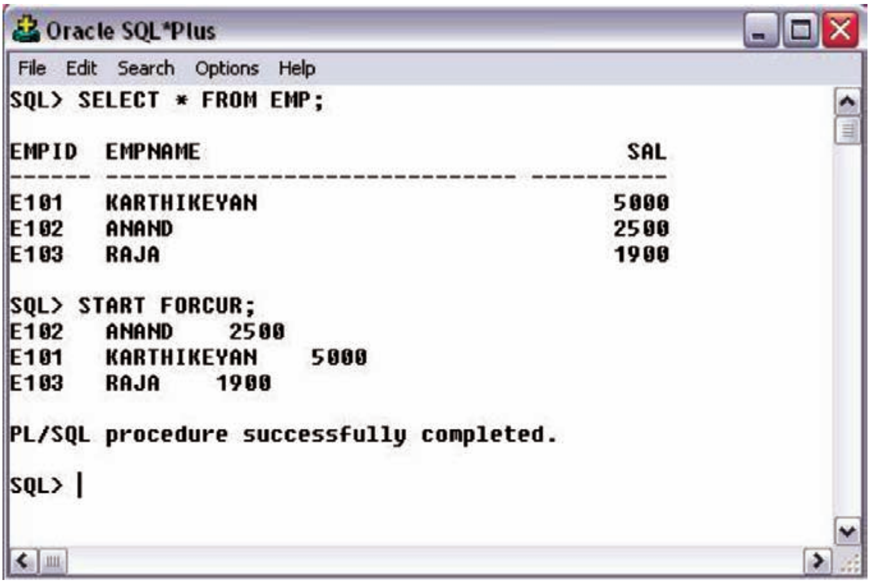
Figures 5.13 and 5.14 show the example of cursor execution using FOR loop.



```
SQL> EDIT FORCUR;
```

```
SET SERVEROUTPUT ON

DECLARE
CURSOR DISPLAYCUR IS SELECT * FROM EMP ORDER BY EMPNAME;


BEGIN

FOR RECORDEMP IN DISPLAYCUR LOOP
DBMS_OUTPUT.PUT_LINE(RECORDEMP.EMPID||'
'||RECORDEMP.EMPNAME||'     '||RECORDEMP.SAL);
END LOOP;

END;
/

SET SERVEROUTPUT OFF
```

Fig. 5.13. Cursor using FOR loop



```
SQL> SELECT * FROM EMP;

EMPID  EMPNAME                              SAL
------ ------------------------------ ----------
E101   KARTHIKEYAN                          5000
E102   ANAND                                2500
E103   RAJA                                 1900

SQL> START FORCUR;
E102   ANAND    2500
E101   KARTHIKEYAN    5000
E103   RAJA    1900

PL/SQL procedure successfully completed.

SQL> 
```

Fig. 5.14. Cursor using FOR loop execution

5.12 Procedure

A procedure is a subprogram that performs some specific task, and stored in the data dictionary. A procedure must have a name, so that it can be invoked or called by any PL/SQL program that appears within an application. Procedures can take parameters from the calling program and perform the specific task. Before the procedure or function is stored, the Oracle engine parses and compiles the procedure or function. When a procedure is created, the Oracle automatically performs the following steps:

1. Compiles the procedure
2. Stores the procedure in the data dictionary

If an error occurs during creation of procedure, Oracle displays a message that procedure is created with compilation errors, but it does not display the errors. To see the errors following statement is used:

SELECT * FROM user_errors;

When the function is invoked, the Oracle loads the compiled procedure in the memory area called system global area (SGA). Once loaded in the SGA other users can also access the same procedure provided they have granted permission for this.

Benefits of Procedures and Functions

Stored procedures and functions have many benefits in addition to *modularizing* application development.

1. It modifies one routine to affect multiple applications.
2. It modifies one routine to eliminate duplicate testing.
3. It ensures that related actions are performed together, or not at all, by doing the activity through a single path.
4. It avoids PL/SQL parsing at runtime by parsing at compile time.
5. It reduces the number of calls to the database and database network traffic by bundling the commands.

Defining and Creating Procedures

A procedure consists of two parts: specification and body. The specification starts with keyword PROCEDURE and ends with parameter list or procedure name. The procedures may accept parameters or may not. Procedures that do not accept parameters are written parentheses.

The procedure body starts with the keyword IS and ends with keyword END. The procedure body is further subdivided into three parts:

1. Declarative part which consists of local declarations placed between keywords IS and BEGIN.

2. Executable part, which consists of actual logic of the procedure, included between keywords BEGIN and EXCEPTION. At least one executable statement is a must in the executable portion of a procedure. Even a single NULL statement will do the job.
3. Error/Exception handling part, an optional part placed between EXCEPTION and END.

The syntax for creating a procedure is follows:

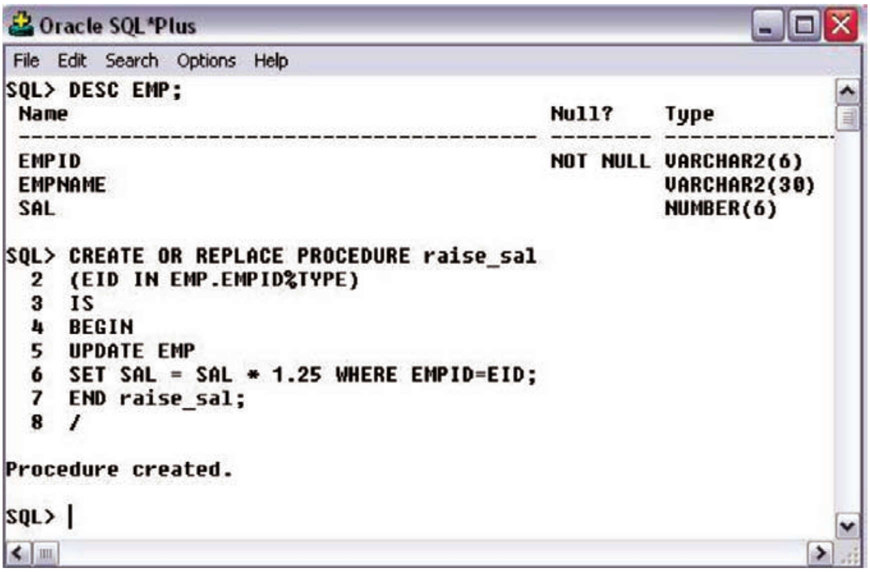
```
CREATE OR REPLACE PROCEDURE [schema.] package_name
[(argument {IN, OUT, IN OUT} data type,.........)] {IS, AS}
[local variable declarations]
BEGIN
executable statements
EXCEPTION
exception handlers
END [procedure name];
```

Create: Creates a new procedure, if a procedure of same name already exists, it gives an error.

Replace: Creates a procedure, if a procedure of same name already exists, it replace the older one by the new procedure definition.

Schema: If the schema is not specified then procedure is created in user's current schema.

Figure 5.15 shows the procedure to raise the salary of the employee. The name of the procedure is raise_sal.



```
SQL> DESC EMP;
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------
 EMPID                                     NOT NULL VARCHAR2(6)
 EMPNAME                                            VARCHAR2(30)
 SAL                                                NUMBER(6)

SQL> CREATE OR REPLACE PROCEDURE raise_sal
  2  (EID IN EMP.EMPID%TYPE)
  3  IS
  4  BEGIN
  5  UPDATE EMP
  6  SET SAL = SAL * 1.25 WHERE EMPID=EID;
  7  END raise_sal;
  8  /

Procedure created.

SQL> |
```

Fig. 5.15. Procedure creation

Argument: It is the name of the argument to the procedure.

IN: Specifies that a value for the argument must be specified when calling the procedure.

OUT: Specifies that the procedure pass a value for this argument back to its calling environment after execution.

IN OUT: Specifies that a value for the argument must be specified when calling the procedure and that the procedure passes a value for this argument back to its calling environment after execution. If no value is specified then it takes the default value IN.

Datatype: It is the unconstrained datatype of an argument. It supports any data type supported by PL/SQL. No constraints like size constraints or NOT NULL constraints can be imposed on the data type. However, you can put on the size constraint indirectly.

Example

To raise the salary of an employee, we can write a procedure as follows.

Declaring Subprograms

Subprograms can be declared inside any valid PL/SQL block. The only thing to be kept in mind is the declaration of programs must be the last part of declarative section of any PL/SQL block; all other declarations should precede the subprogram declarations.

Like any other programming language, PL/SQL also requires that any identifier that is used in PL/SQL program should be declared first before its use. To avoid problems arising due to such malpractices, forward declarations are used.

System and Object Privileges for Procedures

The creator of a procedure must have CREATE PROCEDURE system privilege in his own schema, if the procedure being created refers to his own schema. To create a procedure in other's schema, the creator must have CREATE ANY PROCEDURE system privilege.

To create a procedure without errors (compiling it without errors), the creator of procedure must have required privileges to all the objects he refer to from his procedure. It must be noted that the owner will not get the required privileges through roles, he must be granted those privileges explicitly.

As soon as the privileges granted to the owner of procedure change, the procedure must be reauthenticated in order to bring into picture the new privileges of the owner. If a necessary privilege to an object referenced by a procedure is revoked/withdrawn from the owner of the procedure, the procedure cannot be run.

To EXECUTE any procedure a user must have EXECUTE ANY PROCEDURE privilege. With this privilege he can execute a procedure which belong to some other user.

Executing/Invoking a Procedure

The syntax used to execute a procedure depends on the environment from which the procedure is being called. From within SQLPLUS, a procedure can be executed by using the EXECUTE command, followed by the procedure name. Any arguments to be passed to the procedure must be enclosed in parentheses following the procedure name.

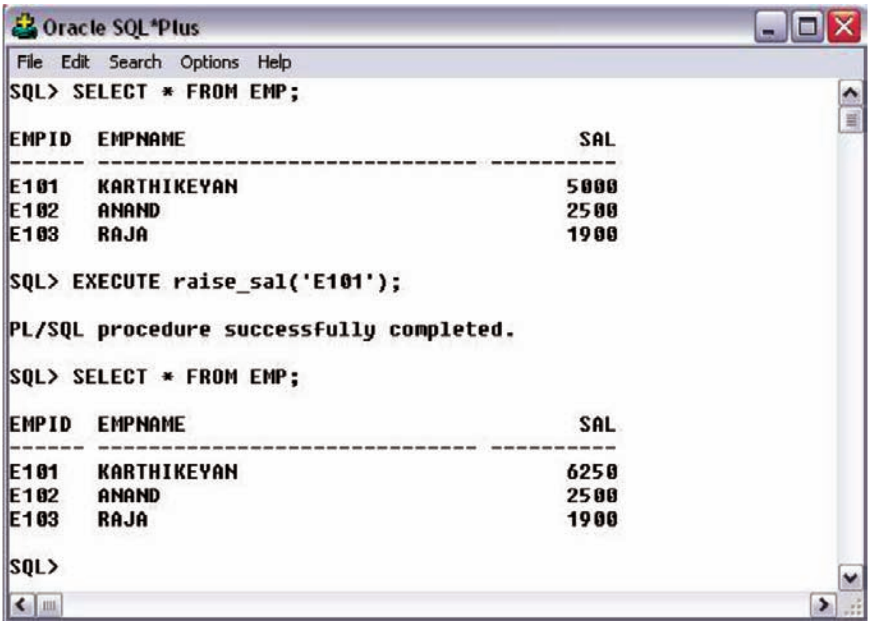
Example

Figure 5.16 shows the execution of procedure raise_sal.

Removing a Procedure

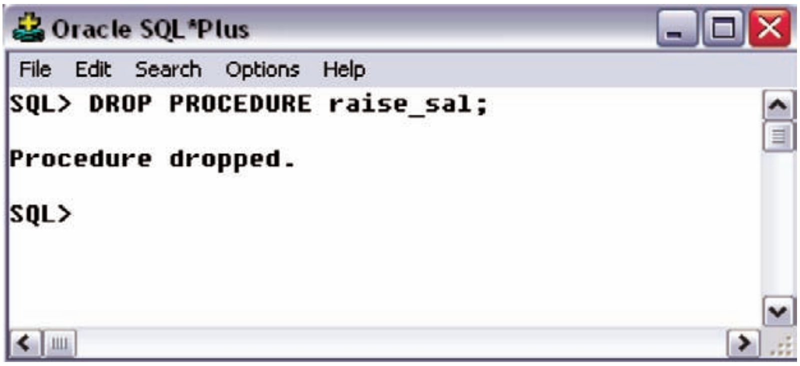
To remove a procedure completely from the database, following command is used:

```
DROP PROCEDURE <PROCEDURE NAME>;
```



```
SQL> SELECT * FROM EMP;

EMPID  EMPNAME                              SAL
------ ------------------------------ ----------
E101   KARTHIKEYAN                         5000
E102   ANAND                               2500
E103   RAJA                                1900

SQL> EXECUTE raise_sal('E101');

PL/SQL procedure successfully completed.

SQL> SELECT * FROM EMP;

EMPID  EMPNAME                              SAL
------ ------------------------------ ----------
E101   KARTHIKEYAN                         6250
E102   ANAND                               2500
E103   RAJA                                1900

SQL>
```

Fig. 5.16. Procedure execution



```
SQL> DROP PROCEDURE raise_sal;

Procedure dropped.

SQL>
```

Fig. 5.17. Dropping of a procedure

To remove a procedure, one must own the procedure he is dropping or he must have DROP ANY PROCEDURE privilege.

Example

To drop a procedure raise_sal. Figure 5.17 indicate the dropping of the procedure raise_sal.

5.13 Function

A Function is similar to procedure except that it must return one and only one value to the calling program. Besides this, a function can be used as part of SQL expression, whereas the procedure cannot.

Difference Between Function and Procedure

Before we look at functions in deep, let us first discuss the major differences between a function and a procedure.

1. A procedure never returns a value to the calling portion of code, whereas a function returns exactly *one value* to the calling program.
2. As functions are capable of returning a value, they can be used as elements of SQL expressions, whereas the procedures cannot. However, user-defined functions cannot be used in CHECK or DEFAULT constraints and cannot manipulate database values, to obey function purity rules.
3. It is mandatory for a function to have at least one RETURN statement, whereas for procedures there is no restriction. A procedure may have a RETURN statement or may not. In case of procedures with RETURN statement, simply the control of execution is transferred back to the portion of code that called the procedure.

The exact syntax for defining a function is given below:

```
CREATE OR REPLACE FUNCTION [schema.] functionname
[(argument IN datatype, ....)] RETURN datatype {IS,AS}
[local variable declarations];
BEGIN
executable statements;
EXCEPTION
exception handlers;
END [functionname];
```

where RETURN datatype is the datatype of the function's return value. It can be any PL/SQL datatype.

Thus a function has two parts: function specification and function body. The function specification begins with keyword FUNCTION and ends with RETURN clause which indicates the datatype of the value returned by the function. Function body is enclosed between the keywords IS and END. Sometimes END is followed by function name, but this is optional. Like procedure, a function body also is composed of three parts: declarative part, executable part, and an optional error/exception handling part.

At least one return statement is a must in a function; otherwise PL/SQL raises PROGRAM_ERROR exception at the run time. A function can have multiple return statements, but can return only one value. In procedures, return statement cannot contain any expression, it simply returns control back to the calling code. However in functions, return statement must contain an expression, which is evaluated and sent to the calling code.

Example

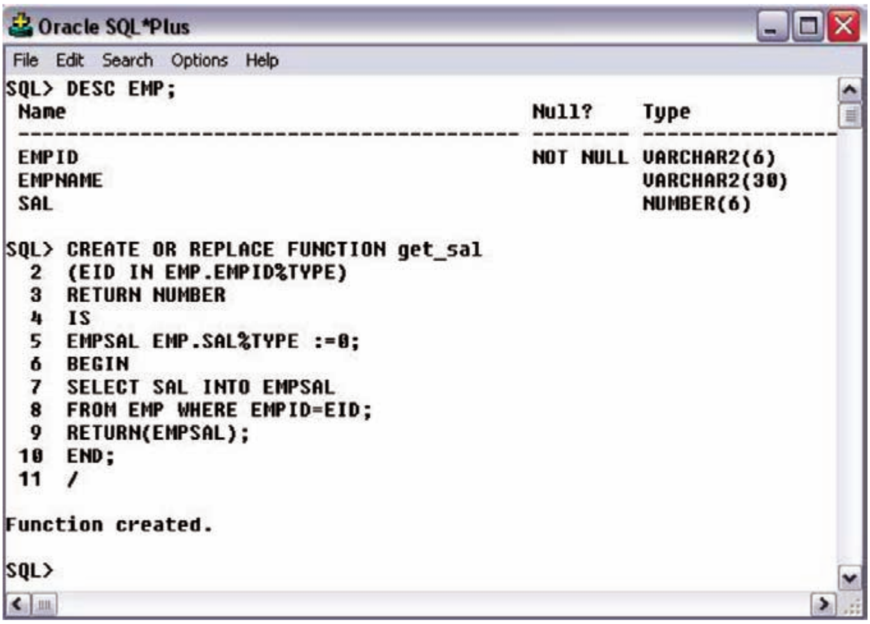
To get a salary of an employee, Fig. 5.18 shows a function.

Figure 5.19 shows that how the calling of a function is different from procedure calling.

Purity of a Function

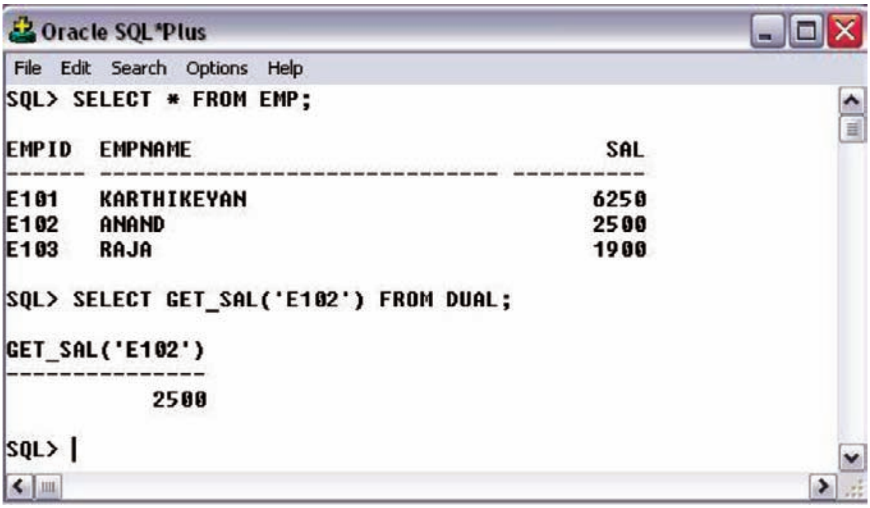
For a function to be eligible for being called in SQL statements, it must satisfy the following requirements, which are known as Purity Rules.

1. When called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
2. When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.



```
SQL> DESC EMP;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 EMPID                                     NOT NULL VARCHAR2(6)
 EMPNAME                                            VARCHAR2(30)
 SAL                                                NUMBER(6)

SQL> CREATE OR REPLACE FUNCTION get_sal
     (EID IN EMP.EMPID%TYPE)
     RETURN NUMBER
     IS
     EMPSAL EMP.SAL%TYPE :=0;
     BEGIN
     SELECT SAL INTO EMPSAL
     FROM EMP WHERE EMPID=EID;
     RETURN(EMPSAL);
     END;
     /

Function created.

SQL>
```

Fig. 5.18. Function creation



```
SQL> SELECT * FROM EMP;

EMPID  EMPNAME                               SAL
------ ------------------------------ ----------
E101   KARTHIKEYAN                          6250
E102   ANAND                                2500
E103   RAJA                                 1900

SQL> SELECT GET_SAL('E102') FROM DUAL;

GET_SAL('E102')
---------------
           2500

SQL>
```

Fig. 5.19. Function execution

3. When called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot

execute DDL statements (such as CREATE) because they are followed by an automatic commit.

If any of the above rules is violated, the function is said to be not following the Purity Rules and the program using such functions receives run time error.

Removing a Function

To remove a function, use following command:

```
DROP FUNCTION <FUNCTION NAME>;
```

Example

Figure 5.20 illustrates the dropping of a function.

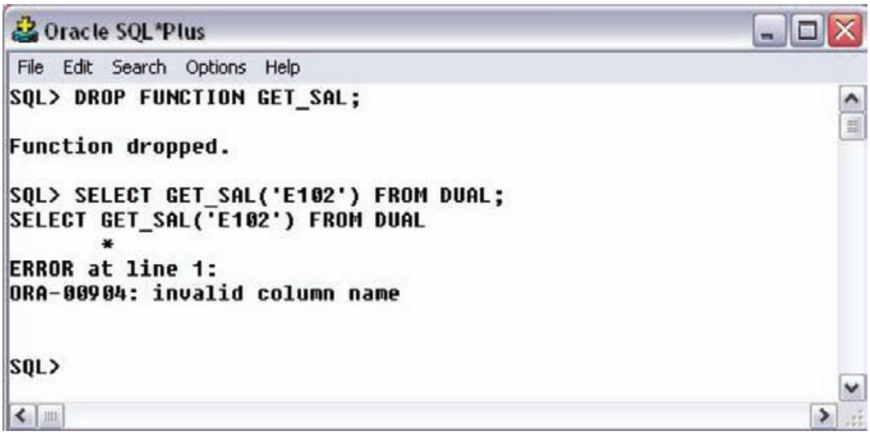
To remove a function, one must own the function to be dropped or he must have DROP ANY FUNCTION privilege.

Parameters

Parameters are the link between a subprogram code and the code calling the subprogram. Lot depends on how the parameters are passed to a subprogram. Hence it is absolutely necessary to know more about parameters, their modes, their default values, and how subprograms can be called without passing all the parameters.

Parameter Modes

Parameter modes define the behavior of formal parameters of subprograms. There are three types of parameter modes: IN, OUT, IN/OUT.



```
SQL> DROP FUNCTION GET_SAL;

Function dropped.

SQL> SELECT GET_SAL('E102') FROM DUAL;
SELECT GET_SAL('E102') FROM DUAL
       *
ERROR at line 1:
ORA-00904: invalid column name


SQL>
```

Fig. 5.20. Dropping the function

IN Mode

IN mode is used to pass values to the called subprogram. In short this is an input to the called subprogram. Inside the called subprogram, an IN parameter acts like a constant and hence it cannot be assigned a new value.

The IN parameter in actual parameter list can be a constant, literal, initialized variable, or an expression. IN parameters can be initialized to default values, which is not the case with IN/OUT or OUT parameters.

It is important to note that IN mode is the default mode of the formal parameters. If we do not specify the mode of a formal parameter it will be treated as an IN mode parameter.

OUT Mode

An OUT parameter returns a value back to the caller subprogram. Inside the subprogram, the parameter specified with OUT mode acts just like any locally declared variable. Its value can be changed or referenced in expressions, just like any other local variables.

The points to be noted for an OUT parameter are:

1. The parameter (in actual argument list) corresponding to OUT parameter must be a variable; it cannot be a constant or literal.
2. Formal OUT parameters are by default initialized to NULL, so we cannot constraint the formal OUT parameters by NOT NULL constraint.
3. The parameter (in actual argument list) corresponding to OUT parameter can have a value before a call to subprogram, but the value is lost as soon as a call is made to the subprogram.

IN/OUT

An IN/OUT parameter performs the duty of both IN parameter as well as OUT parameter. It first passes input value (through actual argument) to the called subprogram and then inside subprogram it receives a new value which will be assigned finally to the actual parameter. In short, inside the called subprogram, the IN/OUT parameter behaves just like an initialized local variable.

Like OUT parameter, the parameter in the actual argument list that corresponds to IN/OUT parameter, must be a variable, it cannot be a constant or an expression. If the subprogram exits successfully, PL/SQL assigns value to actual parameters, however, if the subprogram exits with unhandled exception, PL/SQL does not assign values to actual parameters.

5.14 Packages

A package can be defined as a collection of related program objects such as procedures, functions, and associated cursors and variables together as a unit in the database. In simpler term, a package is a group of related procedures and functions stored together and sharing common variables, as well as local procedures and functions. A package contains two separate parts: the package specification and the package body. The package specification and package body are compiled separately and stored in the data dictionary as two separate objects. The package body is optional and need not to be created if the package specification does not contain any procedures or functions. Applications or users can call packaged procedures and functions explicitly similar to standalone procedures and functions.

Advantages of Packages

Packages offer a lot of advantages. They are as follows.

1. Stored packages allow us to sum up (group logically) related stored procedures, variables, and data types, and so forth in a single-named, stored unit in the database. This provides for better orderliness during the development process. In other words packages and its modules are easily understood because of their logical grouping.
2. Grouping of related procedures, functions, etc. in a package also make privilege management easier. Granting the privilege to use a package makes all components of the package accessible to the grantee.
3. Package helps in achieving data abstraction. Package body hides the details of the package contents and the definition of private program objects so that only the package contents are affected if the package body changes.
4. An entire package is loaded into memory when a procedure within the package is called for the first time. This load is completed in one operation, as opposed to the separate loads required for standalone procedures. Therefore, when calls to related packaged procedures occur, no disk I/O is necessary to execute the compiled code already in memory. This results in faster and efficient operation of programs.
5. Packages provide better performance than stored procedures and functions because public package variables persist in memory for the duration of a session. So that they can be accessed by all procedures and functions that try to access them.
6. Packages allow overloading of its member modules. More than one function in a package can be of same name. The functions are differentiated, depending upon the type and number of parameters it takes.

Units of Packages

As described earlier, a package is used to store together, the logically related PL/SQL units. In general, following units constitute a package.

- Procedures
- Functions
- Triggers
- Cursors
- Variables

Parts of Package

A Package has two parts. They are:

- Package specification
- Package body

Package Specification

The specification declares the types, variables, constants, exceptions, cursors, and subprograms that are public and thus available for use outside the package. In case in the package specification declaration there is only types, constants, exception, or variables, then there is no need for the package body because package specification are sufficient for them. Package body is required when there is subprograms like cursors, functions, etc.

Package Body

The package body fully defines subprograms such as cursors, functions, and procedures. All the private declarations of the package are included in the package body. It implements the package specification. A package specification and the package body are stored separately in the database. This allows calling objects to depend on the specification only, not on both. This separation enables to change the definition of program object in the package body without causing Oracle to interfere with other objects that call or reference the program object. Oracle invalidates the calling object if the package specification is changed.

Creating a Package

A package consists of package specification and package body. Hence creation of a package involves creation of the package specification and then creation of the package body.

The package specification is declared using the CREATE PACKAGE command.

The syntax for package specification declaration is as follows.

```
CREATE[OR REPLACE] PACKAGE <package_name>
[AS/IS]
PL/SQL package specification
```

All the procedures, sub programs, cursors declared in the CREATE PACKAGE command are described and implemented fully in the package body along with private members. The syntax for declaring a package body is as follows:

```
CREATE[OR REPLACE] PACKAGE BODY <package_name>
[AS/IS]
PL/SQL package body
```

Member functions and procedures can be declared in a package and can be made public or private member using the keywords public and private. Use of all the private members of the package is restricted within the package while the public members of the package can be accessed and used outside the package.

Referencing Package Subprograms

Once the package body is created with all members as public, we can access them from outside the program. To access these members outside the packages we have to use the dot operator, by prefixing the package object with the package name. The syntax for referencing any member object is as follows:

```
<PACKAGE_NAME>.<VARIABLE_NAME>
```

To reference procedures we have to use the syntax as follows:

```
EXECUTE <package_name>.<procedure_name(variables)>;
```

But the package member can be referenced by only its name if we reference the member within the package. Moreover the EXECUTE command is not required if procedures are called within PL/SQL. Functions can be referenced similar to that of procedures from outside the package using the dot operator.

Public and Private Members of a Package

A package can consist of public as well as private members. Public members are those members which are accessible outside the package, whereas the private members are accessible only from within the package. Private members are just like local members whose are not visible outside the enclosing code block (in this case, a package).

The place where a package member is declared, also matters in deciding the visibility of that member. Those members whose declaration is found in the package specification are the public members. The package members that are not declared in the package specification but directly defined in the package body become the private members.

Viewing Existing Procedural Objects

The source code for the existing procedures, functions, and packages can be queried from the following data dictionary views.

USER_SOURCE	Procedural objects owned by the user.
ALL_SOURCE	Procedural objects owned by the user or to which the user has been granted access.
DBA_SOURCE	Procedural objects in the database.

Removing a Package

A package can be dropped from the database just like any other table or database object. The exact syntax of the command to be used for dropping a package is:

```
DROP PACKAGE <PACKAGE_NAME>;
```

To drop a package a user either must own the package or he should have DROP ANY PACKAGE privilege.

5.15 Exceptions Handling

During execution of a PL/SQL block of code, Oracle executes every SQL sentence within the PL/SQL block. If an error occurs or an SQL sentence fails, Oracle considers this as an Exception. Oracle engine immediately tries to handle the exception and resolve it, by raising a built-in Exception handler.

Introduction to Exceptions

One can define an EXCEPTION as any error or warning condition that arises during runtime. The main intention of building EXCEPTION technique is to continue the processing of a program even when it encounters runtime error or warning and display suitable messages on console so that user can handle those conditions next time.

In absence of exceptions, unless the error checking is disabled, a program will exit abnormally whenever some runtime error occurs. But with exceptions, if at all some error situation occurs, the exceptional handler unit will flag an appropriate error/warning message and will continue the execution of program and finally come out of the program successfully.

An exception handler is a code block in memory that attempts to resolve the current exception condition. To handle very common and repetitive exception conditions Oracle has about 20 Named Exception Handlers. In addition to these for other exception conditions Oracle has about 20,000 Numbered Exception Handlers, which are identified by four integers preceded by hyphen. Each exception handler, irrespective of how it is defined, (i.e., by Name or Number) has code attached to it that attempts to resolve the exception condition. This is how Oracle's Internal Exception handling strategy works.

Oracle's internal exception handling code can be overridden. When this is done Oracle's internal exception handling code is not executed but the code block that takes care of the exception condition, in the exception section, of the PL/SQL block is executed. As soon as the Oracle invokes an exception handler the exception handler goes back to the PL/SQL block from which the exception condition was raised. The exception handler scans the PL/SQL block for the existence of exception section within the PL/SQL block. If an exception section within the PL/SQL block exists the exception handler scans the first word, after the key word WHEN, within the exception section. If the first word after the key word WHEN is the exception handler's name then the exception handler executes the code contained in the THEN section of the construct, the syntax follows:

```
EXCEPTION
WHEN exception_name THEN
User defined action to be carried out.
```

Exceptions can be internally defined (by the run-time system) or user defined. Internally defined exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise internally defined exceptions. Raised exceptions are handled by separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

Advantages of Using Exceptions

1. Control over abnormal exits of executing programs on encountering error conditions, hence the behavior of application becomes more reliable.
2. Meaningful messages can be flagged so that the developer can become aware of error and warning conditions and act upon them.
3. In traditional error checking system, if same error is to be checked at several places, you are required to code the same error check at all those

places. But with exception handling technique, we will write the exception for that particular error only once in the entire code. Whenever that type error occurs at any place in code, the exceptional handler will automatically raise the defined exception.
4. Being a part of PL/SQL, exceptions can be coded at suitable places and can be coded isolated like procedures and functions. This improves the overall readability of a PL/SQL program.
5. Oracle's internal exception mechanism combined with user-defined exceptions, considerably reduce the development efforts required for cumbersome error handling.

Predefined and User-Defined Exceptions

As discussed earlier there are some predefined or internal exceptions, and a developer can also code user-defined exceptions according to his requirement. In next session we will be looking closely at these two types of exceptions.

Internally (Predefined) Defined Exceptions

An internal exception is raised implicitly whenever a PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines a name for some common errors to raise them as exception. For example, if a SELECT INTO statement returns no rows, PL/SQL raises the predefined exception NO_DATA_FOUND, which has the associated Oracle error number ORA-01403.

Example

Figure 5.21 shows the internally defined exception NO_DATA_FOUND, when we want to get a salary of an employee who is not in the EMP table.

If we execute this query with some emp_name say "XYZ" as input and if emp_name column of employee table does not contain any value "XYZ," Oracle's internal exception handling mechanism will raise NO_DATA_FOUND exception even when we have not coded for it.

PL/SQL declares predefined exceptions globally in package STANDARD, which defines the PL/SQL environment. Some of the commonly used exceptions are as follows:

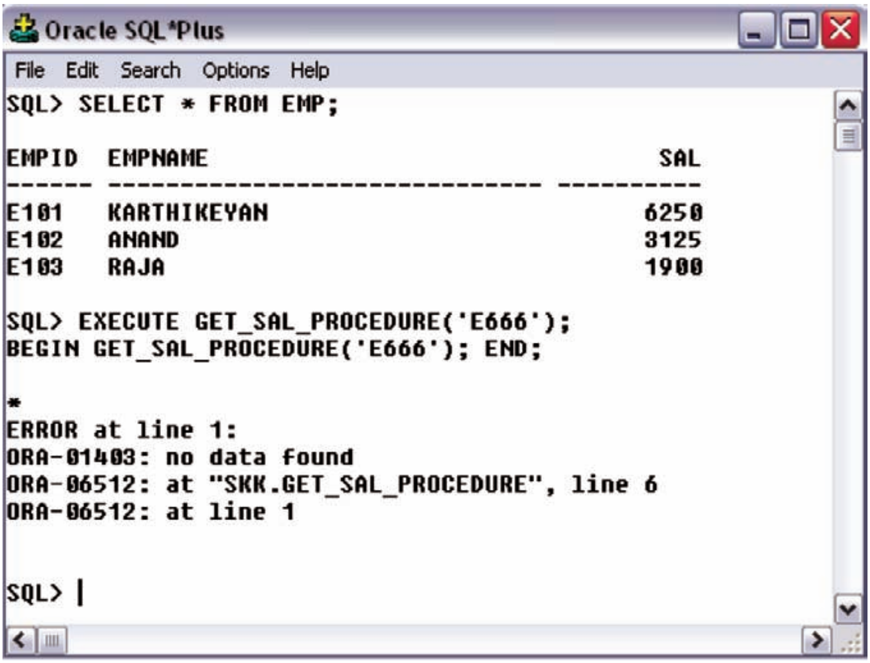
User Defined Exceptions

Unlike internally defined exceptions, user-defined exceptions must be declared and raised explicitly by RAISE statements. Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. An exception is declared by introducing its name, followed by the keyword EXCEPTION.

Name of the exception	Raised when ...
ACCESS_INTO_NULL	Your program attempts to assign values to the attributes of an uninitialized (atomically null) object.
COLLECTION_IS_NULL	Your program attempts to apply collection methods, other than EXISTS to an uninitialized (atomically null) nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	Your program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers. So, your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	Your program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	Your program attempts an illegal cursor operation such as closing an unopened cursor.
INVALID_NUMBER	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.)
LOGIN_DENIED	Your program attempts to log on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. SQL aggregate functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls an aggregate function will never raise NO_DATA_FOUND. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.
NOT_LOGGED_ON	Your program issues a database call without being connected to Oracle.

Continued.

Name of the exception	Raised when ...
ROWTYPE_MISMATCH	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, when an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
PROGRAM_ERROR	PL/SQL has an internal problem.
SELF_IS_NULL	Your program attempts to call a MEMBER method on a null instance. That is, the built-in parameter SELF (which is always the first parameter passed to a MEMBER method) is null.
STORAGE_ERROR	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	Your program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	Your program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
VALUE_ERROR	An arithmetic, conversion, truncation, or size constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	Your program attempts to divide a number by zero.



```
SQL> SELECT * FROM EMP;

EMPID  EMPNAME                                SAL
------ ------------------------------ ----------
E101   KARTHIKEYAN                          6250
E102   ANAND                                3125
E103   RAJA                                 1900

SQL> EXECUTE GET_SAL_PROCEDURE('E666');
BEGIN GET_SAL_PROCEDURE('E666'); END;

*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "SKK.GET_SAL_PROCEDURE", line 6
ORA-06512: at line 1


SQL> |
```

Fig. 5.21. Internally defined exception

The syntax is as follows:

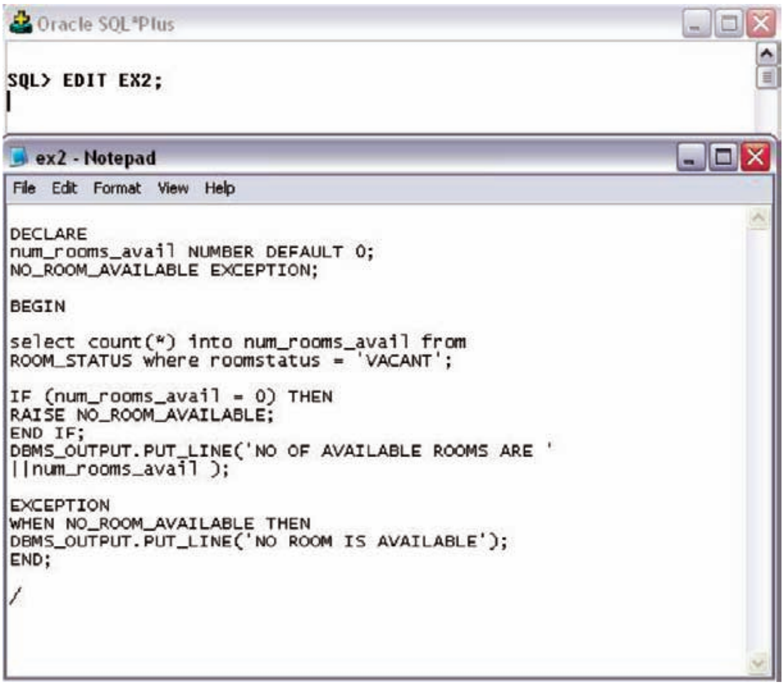
```
DECLARE
<exception_name>EXCEPTION;
```

Exceptions are declared in the same way as the variables. But exceptions cannot be used in assignments or SQL expressions/statements as they are not data items. The visibility of exceptions is governed by the same scope rules which apply to variables also.

Raising User-Defined and Internal Exceptions

As seen in the previous example, one can notice a statement "RAISE Exception1." This statement is used to explicitly raise the exception "Exception1," the reason being, unlike internally defined exceptions which are automatically raised by "OracleS" run time engine, user-defined exceptions have to be raised explicitly by using RAISE statement. However, it is always possible to RAISE predefined (internally defined) exceptions, if needed, in the same way as do the user-defined exceptions, which is illustrated in Fig. 5.22

```
RAISE <exception_name>;
```



```
SQL> EDIT EX2;
```

```
DECLARE
num_rooms_avail NUMBER DEFAULT 0;
NO_ROOM_AVAILABLE EXCEPTION;

BEGIN

select count(*) into num_rooms_avail from
ROOM_STATUS where roomstatus = 'VACANT';

IF (num_rooms_avail = 0) THEN
RAISE NO_ROOM_AVAILABLE;
END IF;
DBMS_OUTPUT.PUT_LINE('NO OF AVAILABLE ROOMS ARE '
||num_rooms_avail );

EXCEPTION
WHEN NO_ROOM_AVAILABLE THEN
DBMS_OUTPUT.PUT_LINE('NO ROOM IS AVAILABLE');
END;

/
```

Fig. 5.22. Exception example

Example

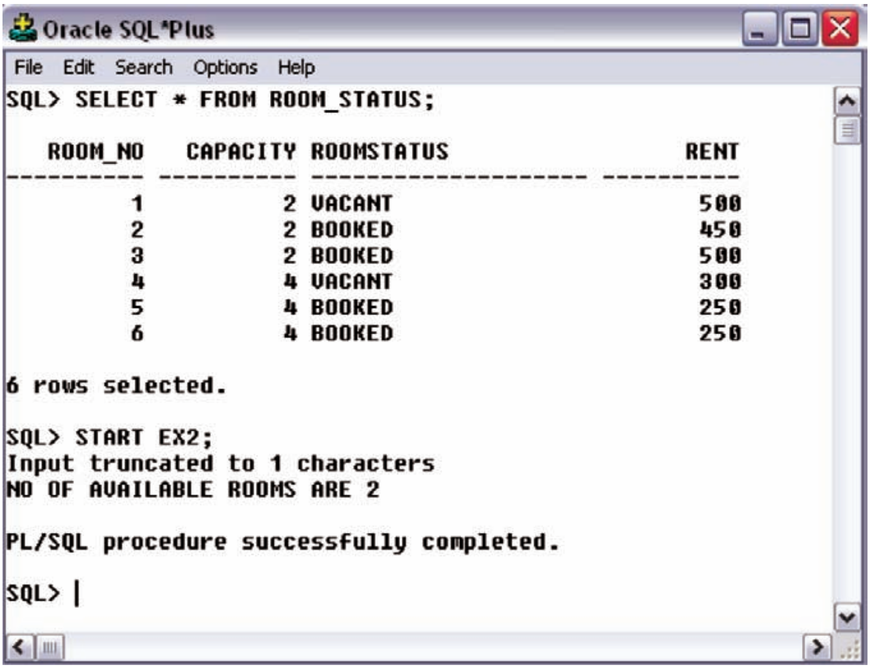
Create a table as follows,

CREATE TABLE ROOM_STATUS (ROOM_NO NUMBER(5)
PRIMARY KEY,
CAPACITY NUMBER(2),
ROOMSTATUS VARCHAR2(20),
RENT NUMBER(4),
CHECK (ROOMSTATUS IN (‘VACANT’,‘BOOKED’)));

User-Defined Error Reporting – Use of Raise_Application_Error

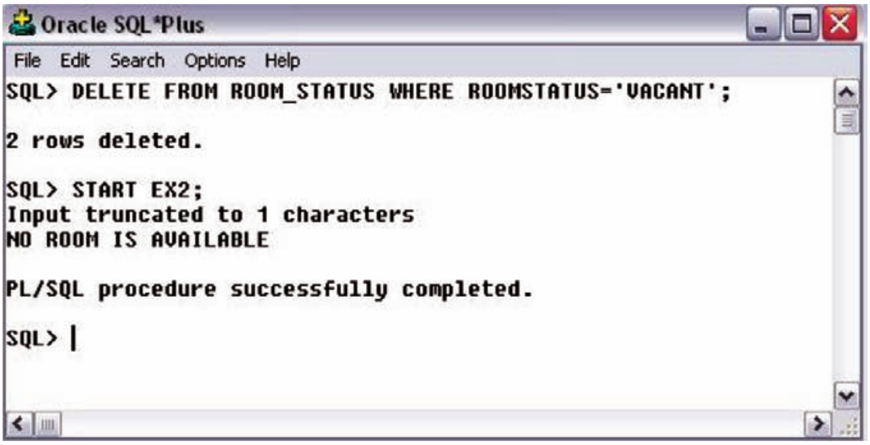
RAISE_APPLICATION_ERROR lets display the messages we want whenever a standard internal error occurs. RAISE_APPILCATION_ERROR associates an Oracle Standard Error Number with the message we define. The syntax for RAISE_APPLICATION_ERROR is as follows:

RAISE_APPLICATION_ERROR (Oracle Error Number, Error Message, TRUE/FALSE);



```
SQL> SELECT * FROM ROOM_STATUS;

   ROOM_NO   CAPACITY ROOMSTATUS                 RENT
---------- ---------- -------------------- ----------
         1          2 VACANT                      500
         2          2 BOOKED                      450
         3          2 BOOKED                      500
         4          4 VACANT                      300
         5          4 BOOKED                      250
         6          4 BOOKED                      250

6 rows selected.

SQL> START EX2;
Input truncated to 1 characters
NO OF AVAILABLE ROOMS ARE 2

PL/SQL procedure successfully completed.

SQL> |
```

Fig. 5.23. Without exception



```
SQL> DELETE FROM ROOM_STATUS WHERE ROOMSTATUS='VACANT';

2 rows deleted.

SQL> START EX2;
Input truncated to 1 characters
NO ROOM IS AVAILABLE

PL/SQL procedure successfully completed.

SQL> |
```

Fig. 5.24. Execution of exception

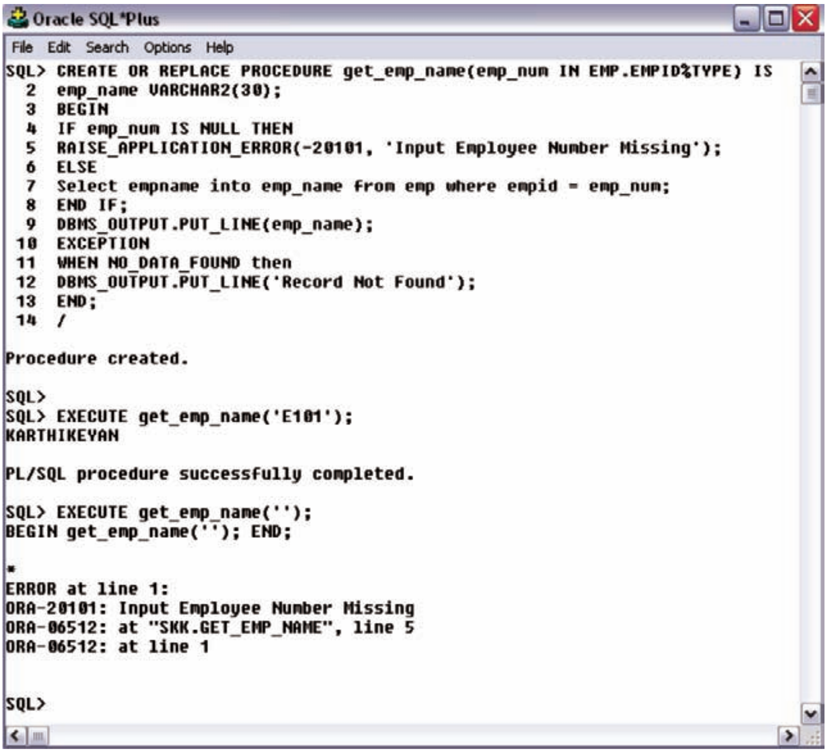
Figures 5.23 and 5.24 shows the output for two conditions ‘Room Available’ and ‘Vacant’.

Oracle error number is the standard Oracle error (−20000 to −20999) that we want to associate with the message (max 2,048 kb) defined, TRUE/FALSE indicates whether to place the error message on previous error stack (TRUE) or to replace all the errors with this message (FALSE).

RAISE_APPLICATION_ERROR can be called only from an executing subprogram. As soon as the subprogram encounters RAISE_APPLICATION_ERROR, the subprogram returns control back to the calling PL/SQL code thereby displaying the error message. We can handle the exception raised in the calling portion of PL/SQL block.

Example

Following Fig. 5.25 illustrates the use of RAISE_APPLICATION_ERROR command with the procedure named get_emp_name.



```
SQL> CREATE OR REPLACE PROCEDURE get_emp_name(emp_num IN EMP.EMPID%TYPE) IS
  2  emp_name VARCHAR2(30);
  3  BEGIN
  4  IF emp_num IS NULL THEN
  5  RAISE_APPLICATION_ERROR(-20101, 'Input Employee Number Missing');
  6  ELSE
  7  Select empname into emp_name from emp where empid = emp_num;
  8  END IF;
  9  DBMS_OUTPUT.PUT_LINE(emp_name);
 10  EXCEPTION
 11  WHEN NO_DATA_FOUND then
 12  DBMS_OUTPUT.PUT_LINE('Record Not Found');
 13  END;
 14  /

Procedure created.

SQL>
SQL> EXECUTE get_emp_name('E101');
KARTHIKEYAN

PL/SQL procedure successfully completed.

SQL> EXECUTE get_emp_name('');
BEGIN get_emp_name(''); END;

*
ERROR at line 1:
ORA-20101: Input Employee Number Missing
ORA-06512: at "SKK.GET_EMP_NAME", line 5
ORA-06512: at line 1


SQL>
```

Fig. 5.25. Raise_application_error example

5.16 Database Triggers

A database trigger is a stored PL/SQL program unit associated with a specific database table. It can perform the role of a constraint, which forces the integrity of data. It is the most practical way to implement routines and granting integrity of data. Unlike the stored procedures or functions, which have to be explicitly invoked, these triggers implicitly get fired whenever the table is affected by the SQL operation. For any event that causes a change in the contents of a table, a user can specify an associated action that the DBMS should carry out. Trigger follows the Event-Condition-Action scheme (ECA scheme).

Privileges Required for Triggers

Creation or alteration of a TRIGGER on a specific table requires TRIGGER privileges as well as table privileges. They are:

1. To create TRIGGER in one's own schema, he must have CREATE TRIGGER privilege. To create a trigger in any other's schema, one must have CREATE ANY TRIGGER system privilege.
2. To create a trigger on table, one must own the table or should have ALTER privilege for that table or should have ALTER ANY TABLE privilege.
3. To ALTER a trigger, one must own that trigger or should have ALTER ANY TRIGGER privilege. Also since the trigger will be operating on some table, one also requires ALTER privilege on that table or ALTER ANY TABLE table privilege.
4. To create a TRIGGER on any database level event, one must have ADMINISTER DATABASE TRIGGER system privilege.

Context to Use Triggers

Following are the situations to use the triggers efficiently:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate the functionality already built into Oracle. For example, do not define triggers to enforce data integrity rules that can be easily enforced using declarative integrity constraints.
- Limit the size of triggers. If the logic for our trigger requires much more than 60 lines of PL/SQL code, then it is better to include most of the code in a stored procedure and call the procedure from the trigger.
- Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
- Do not create recursive triggers which cause the trigger to fire recursively until it has run out of memory.

- Use triggers on DATABASE judiciously. They are executed for every user every time the event occurs on which the trigger is created.

Uniqueness of Trigger

Different types of integrity constraints provide a declarative mechanism to associate "simple" conditions with a table such as a primary key, foreign keys, or domain constraints. Complex integrity constraints that refer to several tables and attributes cannot be specified within table definitions. Triggers, in contrast, provide a procedural technique to specify and maintain integrity constraints.

Triggers even allow users to specify more complex integrity constraints since a trigger essentially is a PL/SQL procedure. Such a procedure is associated with a table and is automatically called by the database system whenever a certain modification (event) occurs on that table.

Simply we can say that trigger is LESS DECLARATIVE AND MORE PROCEDURAL TYPE CONSTRAINT ENFORCEMENT. Triggers are used generally to implement business rules in the database. It is the major difference between Triggers and Integrity Constraints.

Create Trigger Syntax

The Create trigger syntax is as follows:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
[BEFORE/AFTER/INSTEAD OF]
[INSERT/UPDATE/DELETE [of column,..]] ON <table_name>
[REFERENCING [OLD [AS] <old_name> | NEW [AS]
<new_name>]
[FOR EACH STATEMENT/FOR EACH ROW]
[WHEN <condition>]
[BEGIN
–PL/SQL block
END];
```

This syntax can be explained as follows.

Parts of Trigger

A trigger has three basic parts:

- A triggering event or statement
- A trigger restriction
- A trigger action

Trigger Event or Statement

A triggering event or statement is the SQL statement, database event, or user event like update, delete, insert, etc. that causes a trigger to be fired. It also specifies the table to which the trigger is associated. Trigger statement or an event can be any of the following:

1. INSERT, UPDATE, or DELETE on a specific table or view.
2. CREATE, ALTER, or DROP on any schema object.
3. Database activities like startup and shutdown.
4. User activities like logon and logoff.
5. A specific error message on any error message.

Figure 5.26 shows a database application with some SQL statements that implicitly fire several triggers stored in the database. It shows three triggers, which are associated with the INSERT, UPDATE, and DELETE operation in the database table. When these data manipulation commands are given, the corresponding trigger gets automatically fired performing the task described in the corresponding trigger body.

Trigger Restriction

A trigger restriction is any logical expression whose outcome is TRUE/FALSE/UNKNOWN. For a trigger to fire, this logical expression must evaluate to TRUE. Typically, a restriction is a part of trigger declaration that follows the keyword WHEN.

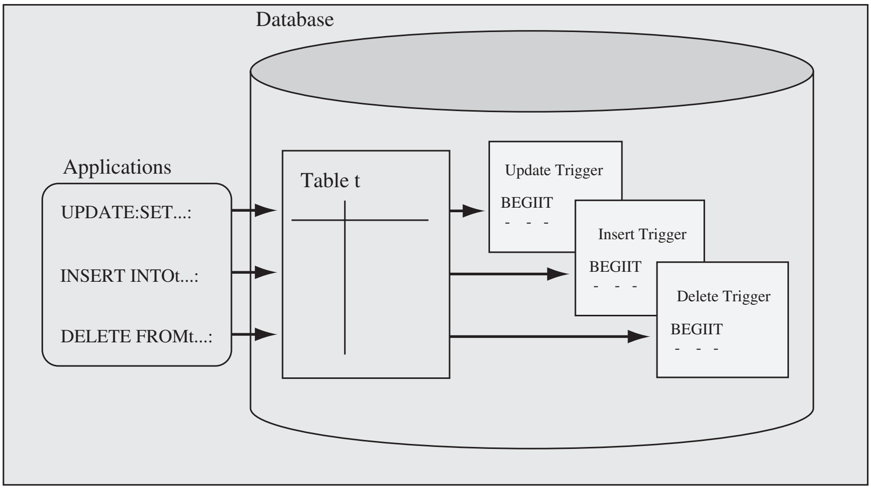


Fig. 5.26. Database application with some SQL statements that implicitly fire several triggers stored in the database

Trigger Action

A trigger action is the PL/SQL block that contains the SQL statements and code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE. It is also called the trigger body. Like stored procedures, a trigger action can contain SQL and PL/SQL.

Following statements will explain the various keywords used in the syntax.

BEFORE and AFTER keyword indicates whether the trigger should be executed before or after the trigger event, where a triggering event can be INSERT, UPDATE, or DELETE. Any combination of triggering events can be included in the same database trigger.

When referring the old and new values of columns, we can use the defaults ("old" and "new") or we can use the REFERENCING clause to specify other names. FOR EACH ROW clause causes the trigger to fire once for each record created, deleted, or modified by the triggering statement. When working with row triggers, the WHEN clause can be used to restrict the records for which the trigger fires.

We can use INSTEAD OF triggers to tell the database what to do instead of performing the actions that invoked the trigger. For example, we can use it on a VIEW to redirect the inserts into a table or to update multiple tables that are parts of the view.

5.17 Types of Triggers

Type of trigger firing, level at which a trigger is executed, and the types of events form the basis classification of triggers into different categories. This section describes the different types of triggers. The broad classification of triggers is as shown below.

On the Basis of Type of Events
- Triggers on System events
- Trigger on User events

On the Basis of the Level at which Triggers are Executed
- Row Level Triggers
- Statement Level Triggers

On the Basis of Type of Trigger/Firing or Triggering Transaction
- BEFORE Triggers
- AFTER Triggers
- INSTEAD OF

Triggers on System Events

System events that can fire triggers are related to instance startup and shutdown and error messages. Triggers created on startup and shutdown events have to be associated with the database; triggers created on error events can be associated with the database or with a schema.

BEFORE Triggers

BEFORE triggers execute the trigger action before the triggering statement is executed. It is used to derive specific column values before completing a triggering DML, DDL statement or to determine whether the triggering statement should be allowed to complete.

Example

We can define a BEFORE trigger on the passengers_detail table that gets fired before deletion of any row. The trigger will check the system date and if the date is Sunday, it will not allow any deletion on the table.

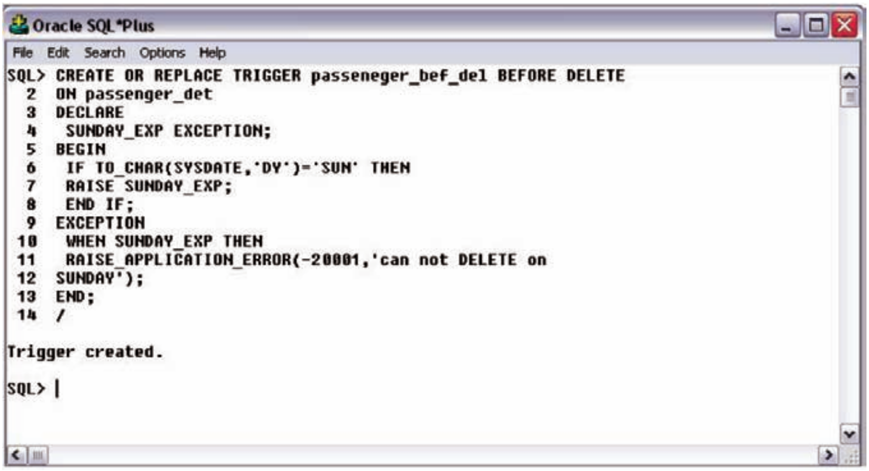
The trigger can be created in Oracle as shown in Fig. 5.27.

The trigger action can be shown as in Fig. 5.28.

As soon as we try to delete a record from passenger_ detail table, the above trigger will be fired and due to SUNDAY_EXP fired, all the changes will be rolled back or undone and the record will not be deleted.

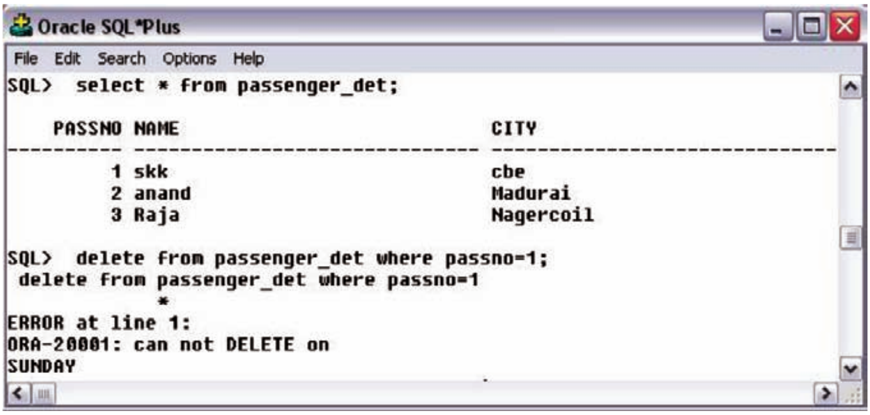
AFTER Triggers

AFTER triggers execute the trigger action after the triggering statement is executed. AFTER triggers are used when we want the triggering statement to complete before executing the trigger action, or to execute some additional logic to the before trigger action.



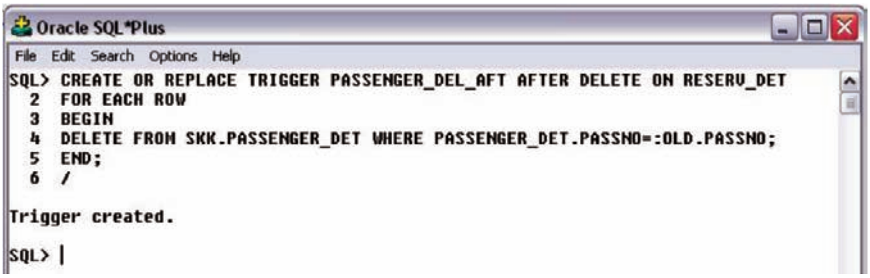
```
SQL> CREATE OR REPLACE TRIGGER passeneger_bef_del BEFORE DELETE
ON passenger_det
DECLARE
 SUNDAY_EXP EXCEPTION;
BEGIN
 IF TO_CHAR(SYSDATE,'DY')='SUN' THEN
 RAISE SUNDAY_EXP;
 END IF;
EXCEPTION
 WHEN SUNDAY_EXP THEN
 RAISE_APPLICATION_ERROR(-20001,'can not DELETE on
SUNDAY');
END;
/

Trigger created.

SQL> |
```

Fig. 5.27. BEFORE trigger creation



```
SQL>  select * from passenger_det;

    PASSNO NAME                           CITY
---------- ------------------------------ ------------------------------
         1 skk                            cbe
         2 anand                          Madurai
         3 Raja                           Nagercoil

SQL>  delete from passenger_det where passno=1;
 delete from passenger_det where passno=1
             *
ERROR at line 1:
ORA-20001: can not DELETE on
SUNDAY
```

Fig. 5.28. BEFORE trigger execution



```
SQL> CREATE OR REPLACE TRIGGER PASSENGER_DEL_AFT AFTER DELETE ON RESERV_DET
  2  FOR EACH ROW
  3  BEGIN
  4  DELETE FROM SKK.PASSENGER_DET WHERE PASSENGER_DET.PASSNO=:OLD.PASSNO;
  5  END;
  6  /

Trigger created.

SQL> |
```

Fig. 5.29. AFTER trigger creation

Example

We can define an AFTER trigger on the reserv_det table that gets fired every time one row is deleted from the table. This trigger will determine the passenger_id of the deleted row and subsequently delete the corresponding row from the passengers_det table with same passenger_id.

Trigger can be created as shown in Fig. 5.29

Trigger action can be shown as in Fig. 5.30. In this figure, the content of the relations passenger_det and reserve_det are shown before and after the triggering event.

Triggers on LOGON and LOGOFF Events

LOGON and LOGOFF triggers can be associated with the database or with a schema. Their attributes include the system event and username, and they can specify simple conditions on USERID and USERNAME.

- LOGON triggers fire after a successful logon of a user.
- LOGOFF triggers fire at the start of a user logoff.

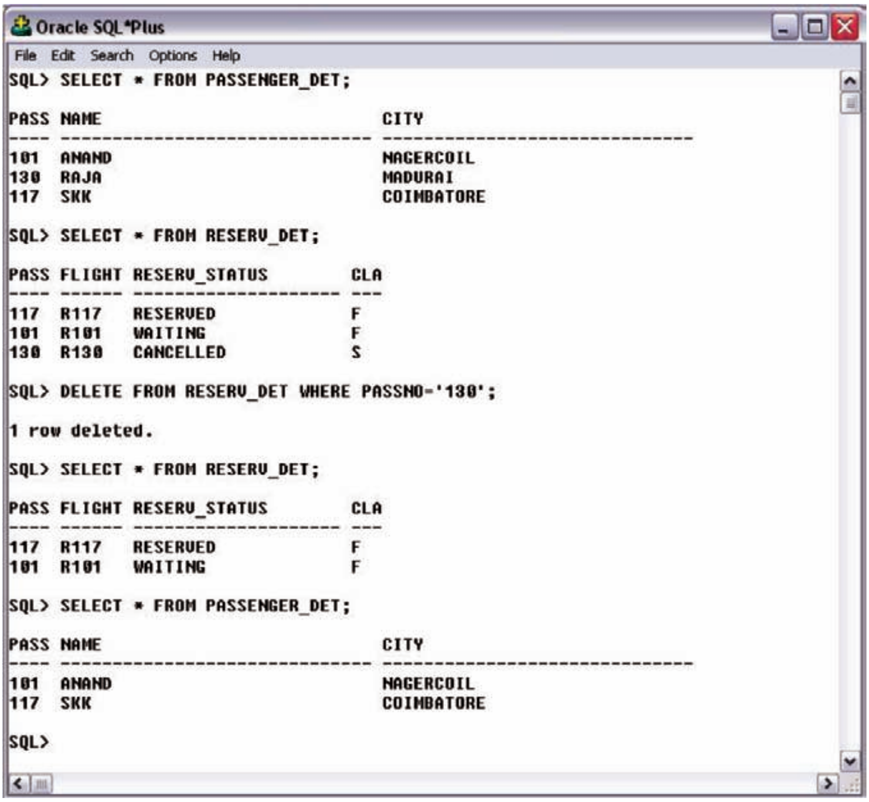

```
SQL> SELECT * FROM PASSENGER_DET;

PASS NAME                           CITY
---- ------------------------------ ------------------------------
101  ANAND                          NAGERCOIL
130  RAJA                           MADURAI
117  SKK                            COIMBATORE

SQL> SELECT * FROM RESERV_DET;

PASS FLIGHT RESERV_STATUS        CLA
---- ------ -------------------- ---
117  R117   RESERVED             F
101  R101   WAITING              F
130  R130   CANCELLED            S

SQL> DELETE FROM RESERV_DET WHERE PASSNO='130';

1 row deleted.

SQL> SELECT * FROM RESERV_DET;

PASS FLIGHT RESERV_STATUS        CLA
---- ------ -------------------- ---
117  R117   RESERVED             F
101  R101   WAITING              F

SQL> SELECT * FROM PASSENGER_DET;

PASS NAME                           CITY
---- ------------------------------ ------------------------------
101  ANAND                          NAGERCOIL
117  SKK                            COIMBATORE

SQL>
```

Fig. 5.30. AFTER trigger execution

Example

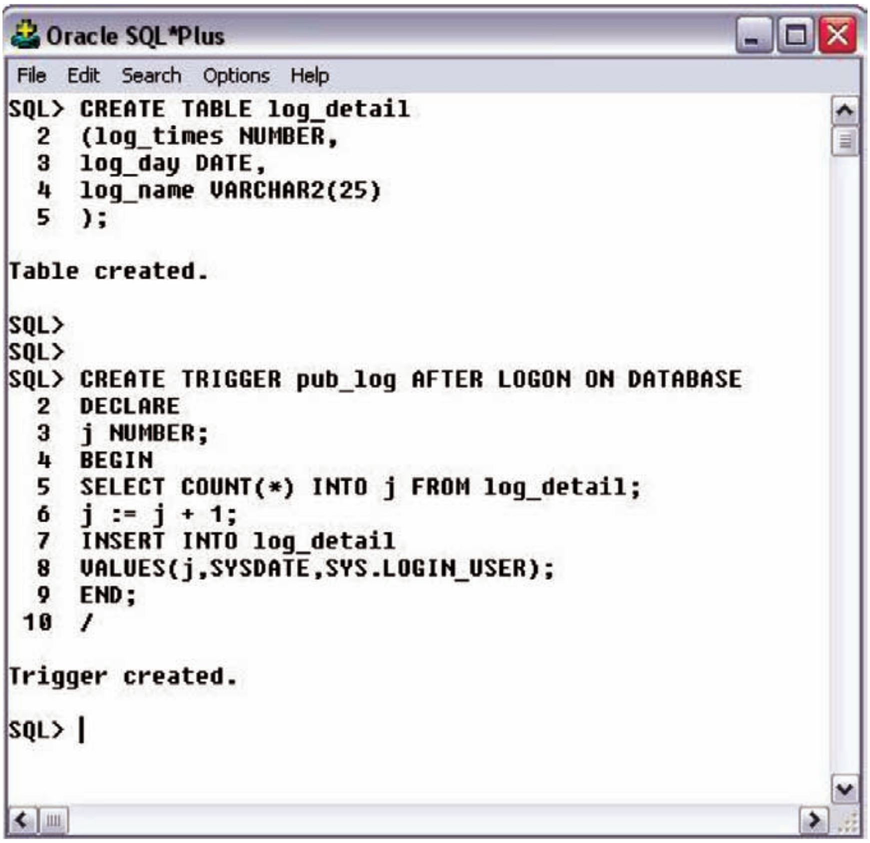
Let us create a trigger on LOGON event called pub_log, which will store the number, date, and user of login done by different user in that particular database. The trigger will store this information in a table called log_detail. The table log_detail must be created before trigger creation by logging into Administrator login. The trigger can be created as shown in Fig. 5.31.

After logging into another login, if we see the content of the relation log_detail it will show who are all logged into database. The value of the attribute log_times would go on increasing with every login into the database which is indicated in Fig. 5.32.

Note The log_detail relation is visible only in Administrator login.

Triggers on DDL Statements

This trigger gets fired when DDL statement such as CREATE, ALTER, or DROP command is issued. DDL triggers can be associated with the database or with a schema. Moreover depending on the time of firing of trigger, this



```
SQL> CREATE TABLE log_detail
  2  (log_times NUMBER,
  3  log_day DATE,
  4  log_name VARCHAR2(25)
  5  );

Table created.

SQL>
SQL>
SQL> CREATE TRIGGER pub_log AFTER LOGON ON DATABASE
  2  DECLARE
  3  j NUMBER;
  4  BEGIN
  5  SELECT COUNT(*) INTO j FROM log_detail;
  6  j := j + 1;
  7  INSERT INTO log_detail
  8  VALUES(j,SYSDATE,SYS.LOGIN_USER);
  9  END;
 10  /

Trigger created.

SQL> |
```

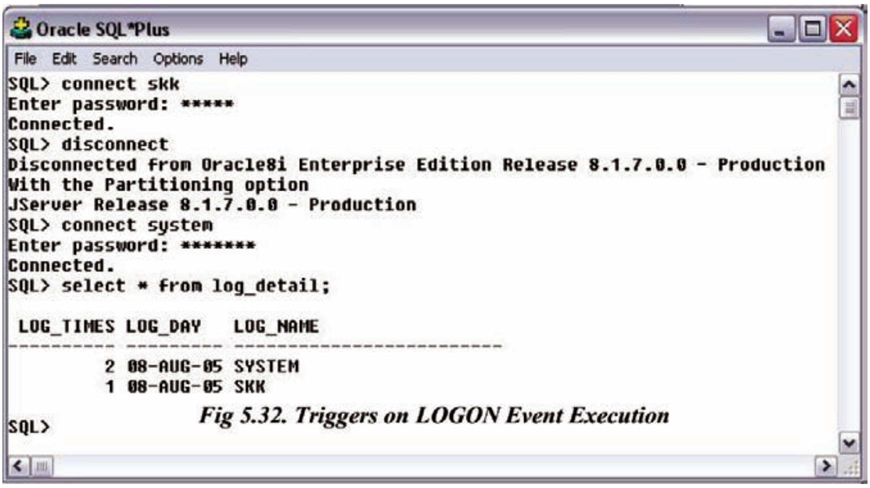
Fig. 5.31. Triggers on LOGON event creation

trigger can be classified into BEFORE and AFTER. Hence the triggers on DDL statements can be as follows:

– BEFORE CREATE and AFTER CREATE triggers fire when a schema object is created in the database or schema.
– BEFORE ALTER and AFTER ALTER triggers fire when a schema object is altered in the database or schema.
– BEFORE DROP and AFTER DROP triggers fire when a schema object is dropped from the database or schema.

Example

Let us create a trigger called “no_drop_pass” that fires before dropping any object on the schema of the user with username “skk.” It checks whether the object type and name. If the object name is “passenger_det” and object type is table, it raises an application error and prevents the dropping of the



```
SQL> connect skk
Enter password: *****
Connected.
SQL> disconnect
Disconnected from Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production
SQL> connect system
Enter password: *******
Connected.
SQL> select * from log_detail;

 LOG_TIMES LOG_DAY   LOG_NAME
---------- --------- ------------------------
         2 08-AUG-05 SYSTEM
         1 08-AUG-05 SKK

SQL>
```

Fig 5.32. Triggers on LOGON Event Execution

Fig. 5.32. Triggers on LOGON event execution

table. The syntax for creating the trigger is as follows. Remember to create the trigger by logging as administrator in the database. The trigger can be created as shown in Fig. 5.33.

The trigger is executed as shown in Fig. 5.34.

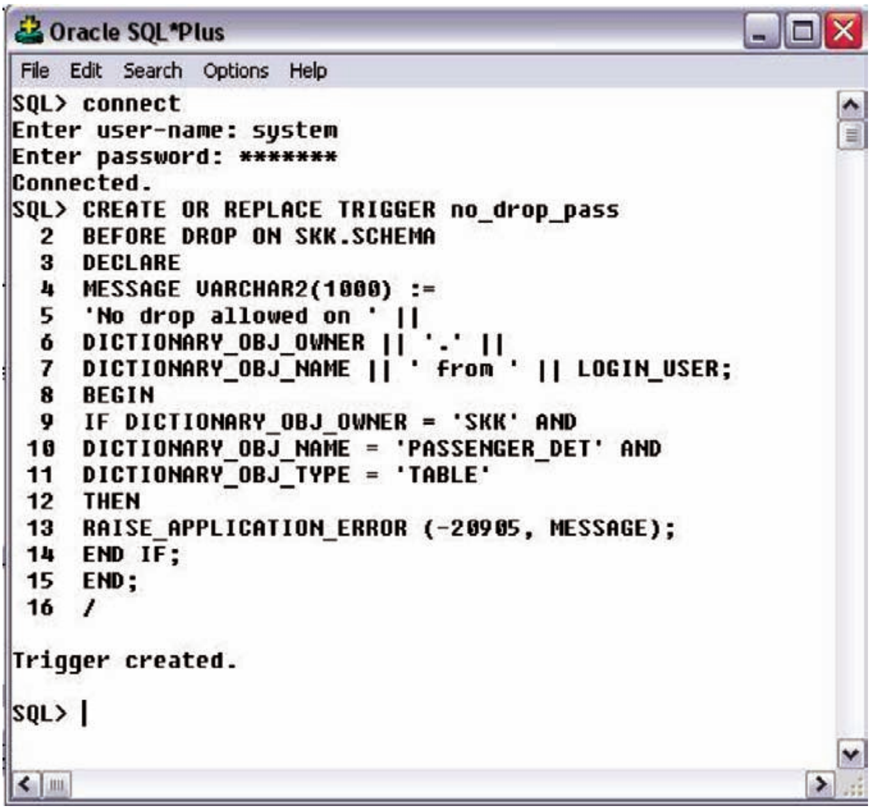
Triggers on DML Statements

This trigger gets fired when DML statement such as INSERT, UPDATE, or DELETE command is issued. DML triggers can be associated with the database or with a schema. Depending on the time of firing of trigger, this trigger can be classified into BEFORE and AFTER. Moreover, when we define a trigger on a DML statement, we can specify the number of times the trigger action is to be executed: once for every row or once for the triggering statement.

Row Level Triggers

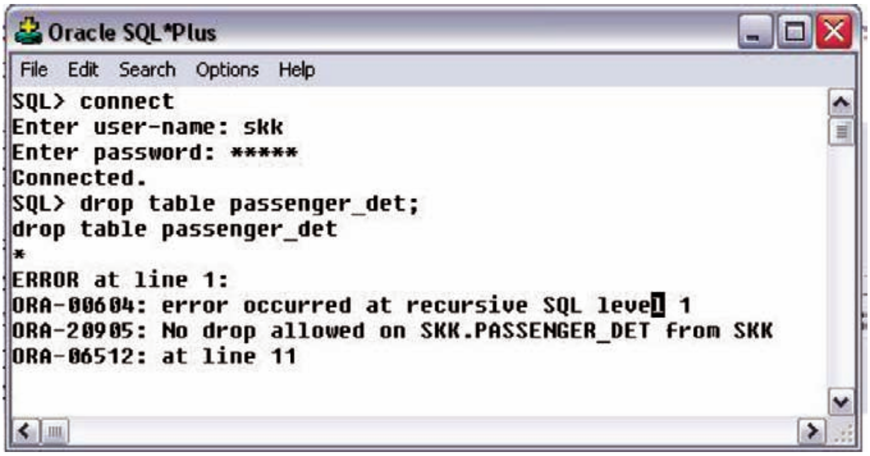
A row level trigger, as its name suggests, is fired for each row that will be affected by the SQL statement, which fires the trigger. Suppose for example if an UPDATE statement updates "N" rows of a table, a row level trigger defined for this UPDATE on that particular table will be fired once for each of those "N" affected rows. If a triggering SQL statement affects no rows, a row trigger is not executed at all. To specify a trigger of row type, FOR EACH ROW clause is used after the name of table.

In row level triggers, the statements in a trigger action have access to column values (new and old) of the current row being processed by the trigger. The names of the new and old values are called correlation names. They allow access to new and old values for each column. By means of new, one refers to the new value with which the row in the tableis updated or inserted. On



```
SQL> connect
Enter user-name: system
Enter password: *******
Connected.
SQL> CREATE OR REPLACE TRIGGER no_drop_pass
  2  BEFORE DROP ON SKK.SCHEMA
  3  DECLARE
  4  MESSAGE VARCHAR2(1000) :=
  5  'No drop allowed on ' ||
  6  DICTIONARY_OBJ_OWNER || '.' ||
  7  DICTIONARY_OBJ_NAME || ' from ' || LOGIN_USER;
  8  BEGIN
  9  IF DICTIONARY_OBJ_OWNER = 'SKK' AND
 10  DICTIONARY_OBJ_NAME = 'PASSENGER_DET' AND
 11  DICTIONARY_OBJ_TYPE = 'TABLE'
 12  THEN
 13  RAISE_APPLICATION_ERROR (-20905, MESSAGE);
 14  END IF;
 15  END;
 16  /

Trigger created.

SQL> |
```

Fig. 5.33. Trigger on DDL statement creation



```
SQL> connect
Enter user-name: skk
Enter password: *****
Connected.
SQL> drop table passenger_det;
drop table passenger_det
*
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20905: No drop allowed on SKK.PASSENGER_DET from SKK
ORA-06512: at line 11
```

Fig. 5.34. Trigger on DDL statement execution

the other hand by means of old, one refers to the old value, which is being updated or deleted. Row level triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected.

Example

The AFTER trigger on reserv_det table that deletes all corresponding rows from passenger_det table with the same passenger_id is a row level trigger as shown in Figs. 5.29 and 5.30, respectively.

Statement Level Triggers

Unlike row level trigger, a statement level trigger is fired only once on behalf of the triggering SQL statement, regardless of the number of rows in the table that the triggering statement affects. Even if the triggering statement affects no rows, the statement level trigger will execute exactly once. For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table. Default type of any trigger is Statement level trigger. Statement level triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.

Example

The BEFORE trigger on passenger_det table that checks that no row should be deleted on Sunday is a statement level trigger as shown in Figs. 5.27 and 5.28, respectively.

INSTEAD-OF Triggers

INSTEAD-OF triggers are used to tell Oracle what to do instead of performing the actions that executed the trigger. It is applicable to both object views and standard relational database. This trigger can be used to redirect table inserts into a different table or to update different tables that are the part of the view. This trigger is used to perform any action instead of the action that executes the trigger. In simpler words if the task associated with this trigger fails, the trigger is fired. It is used mostly for object views rather than tables. This trigger is used to manipulate the tables through the views.

Enabling and Disabling a Trigger

By default, a trigger is enabled when it is created. Only an enabled trigger gets fired whenever the trigger restriction evaluates to TRUE. Disabled triggers do not get fired even when the triggering statement is issued. Thus a trigger can be in either of two distinct modes:

– Enabled (an enabled trigger executes its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE).
– Disabled (a disabled trigger does not execute its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to TRUE).

The need to disable the trigger is there are some situations like heavy data load or partially succeeded load operations. In case of heavy data load condition, disabling trigger may dramatically improve the performance. After load, one has to do all those data operations manually which otherwise a trigger would have done. In case of partial succeeded load, since a part of load is successful, the triggers are already executed for that part. Now when we start the same load fresh, it may be possible that the same trigger would be executed twice which may cause some undesirable effects. So the best way is to disable the trigger and do the operations manually after the entire load is successful.

For enabled triggers, Oracle automatically does the following:

– Prepares a definite plan for execution of triggers of different types.
– Decides time for integrity constraint checking for each type of trigger and ensures that none of the triggers is violating integrity constraints.
– Manages the dependencies among triggers and schema objects referenced in the code of the trigger action.
– No definite order for firing of multiple triggers of same type.

Syntax

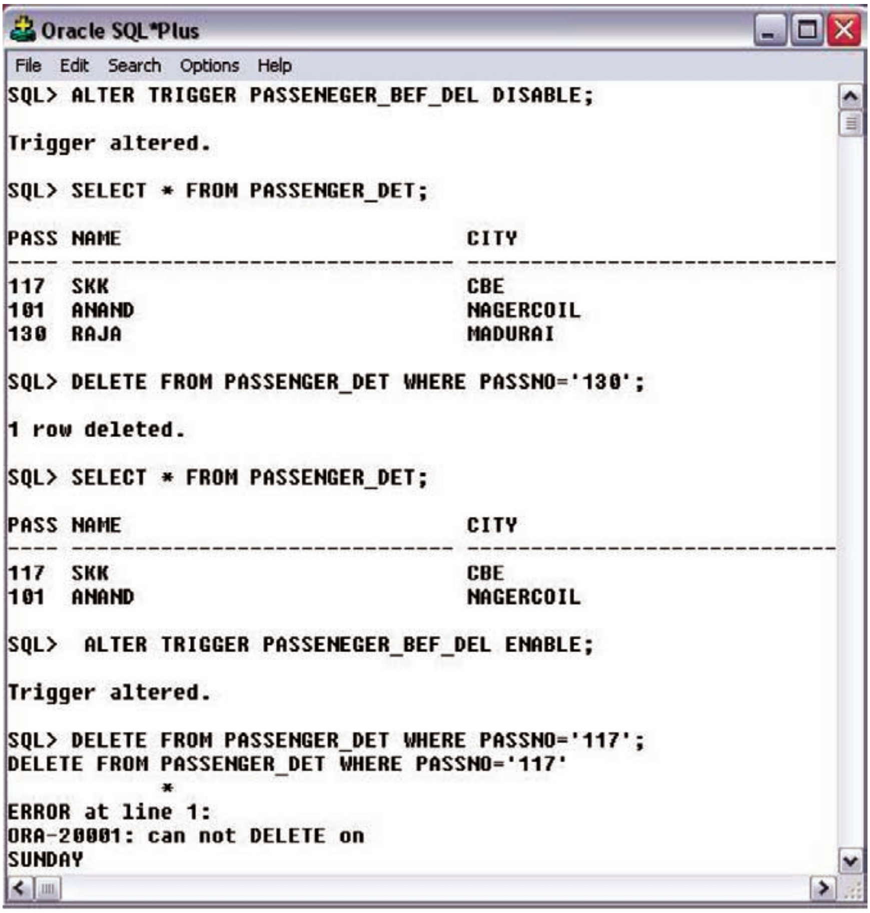
ALTER TRIGGER <Trigger name> ENABLE/DISABLE;

Example

The passenger_bef_del trigger can be disabled and enabled as shown in Fig. 5.35, it shows how Oracle behaves for enabled/disabled triggers.

Replacing Triggers

Triggers cannot be altered explicitly. Triggers have to be replaced with a new definition using OR REPLACE option with CREATE TRIGGER command. In such case the old definition of the trigger is dropped and the new definition is entered in the data dictionary.



```
SQL> ALTER TRIGGER PASSENEGER_BEF_DEL DISABLE;

Trigger altered.

SQL> SELECT * FROM PASSENGER_DET;

PASS NAME                           CITY
---- ------------------------------ ------------------------------
117  SKK                            CBE
101  ANAND                          NAGERCOIL
130  RAJA                           MADURAI

SQL> DELETE FROM PASSENGER_DET WHERE PASSNO='130';

1 row deleted.

SQL> SELECT * FROM PASSENGER_DET;

PASS NAME                           CITY
---- ------------------------------ ------------------------------
117  SKK                            CBE
101  ANAND                          NAGERCOIL

SQL>  ALTER TRIGGER PASSENEGER_BEF_DEL ENABLE;

Trigger altered.

SQL> DELETE FROM PASSENGER_DET WHERE PASSNO='117';
DELETE FROM PASSENGER_DET WHERE PASSNO='117'
            *
ERROR at line 1:
ORA-20001: can not DELETE on
SUNDAY
```

Fig. 5.35. Enabling and disabling the trigger

The exact syntax for replacing the trigger is as follows:

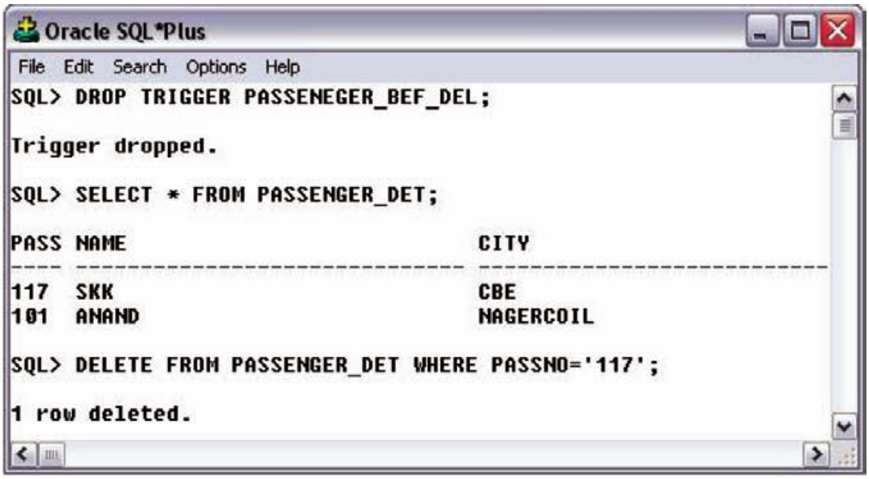
Syntax

CREATE OR REPLACE TRIGGER <trigger_name> AS/IS <trigger_definition>;

The trigger_definition should be as shown in the definition for creating trigger. Alternately the trigger can be dropped and re-created. On dropping a trigger all grants associated with the trigger are dropped as well.

Dropping Triggers

Triggers can be dropped like tables using the drop trigger command. The drop trigger command removes the trigger structure from the database. User needs



```
SQL> DROP TRIGGER PASSENEGER_BEF_DEL;

Trigger dropped.

SQL> SELECT * FROM PASSENGER_DET;

PASS NAME                            CITY
---- ------------------------------- ---------------------------
117  SKK                             CBE
101  ANAND                           NAGERCOIL

SQL> DELETE FROM PASSENGER_DET WHERE PASSNO='117';

1 row deleted.
```

Fig. 5.36. Dropping the trigger

to have DROP ANY TRIGGER system privilege to drop a trigger. The exact syntax for dropping a trigger is as follows.

Syntax

DROP TRIGGER <trigger_name>

Example

We drop the trigger passenger_bef_del as shown in Fig. 5.36.

Summary

This chapter has introduced the concept of PL/SQL. The shortcomings of SQL and the need for PL/SQL are given in detail. PL/SQL combines the data manipulation power of SQL with data processing power of procedural language. The PL/SQL language elements like character sets, operators, indicators, punctuation, identifiers, comments, etc. are introduced with examples in this chapter. The different types of iterative control like FOR loop, WHILE loop, their syntax and concepts are given through examples.

A cursor is a mechanism that can be used to process the multiple row result sets one row at a time. Cursors are an inherent structure in PL/SQL. Cursors allow users to easily store and process sets of information in PL/SQL program. The concept of cursor and different types of cursors like implicit cursor, explicit cursor are given through examples.

A procedure is a subprogram that performs some specific task, and stored in the data dictionary. The concept of procedure, function, the difference between procedure and function are given in this chapter.

A package is a collection of related program objects such as procedures, functions, and associated cursors and variables together as a unit in the database. In simpler term, a package is a group of related procedures and functions stored together and sharing common variables, as well as local procedures and function. In this chapter, the package body and how to create a package are explained with examples.

An EXCEPTION is any error or warning condition that arises during runtime. The main intention of building EXCEPTION technique is to continue the processing of a program even when it encounters runtime error or warning and display suitable messages on console so that user can handle those conditions next time. The advantage of using EXCEPTION, different types of EXCEPTIONS are given through example in this chapter.

A database trigger is a stored PL/SQL program unit associated with a specific database table. It can perform the role of a constraint, which forces the integrity of data. The concept of trigger, the uniqueness of trigger, and the use of trigger are explained with examples in this chapter.

Review Questions

5.1. Mention the key difference between SQL and PL/SQL?

SQL is a declarative language. PL/SQL is a procedural language that makes up for all the missing elements in SQL.

5.2. Mention two drawbacks of SQL?

- SQL statements can be executed only one at a time. Every time to execute a SQL statement, a call is made to Oracle engine, thus it results in an increase in database overheads.
- While processing an SQL statement, if an error occurs, Oracle generates its own error message, which is sometimes difficult to understand. If a user wants to display some other meaningful error message, SQL does not have provision for that.

5.3. Identify which one is not included in PL/SQL Character Set?
(a) * (b)> (c)! (d) \

Answer: (d)

5.4. What are Lexical units related with PL/SQL?

A line of PL/SQL program contains groups of characters known as lexical units, which can be classified as follows:

– Delimiters
– Identifiers
– Literals
– Comments

5.5. What is Delimiter?

A delimiter is a simple or compound symbol that has a special meaning to PL/SQL.

5.6. Identify which identifier is not permitted in PL/SQL?
(a) Bn12 (b) Girt–1 (c) Hay# (d) I am

Answer: (d)

5.7. Give the syntax for single-line comments and multiline comments?

Single line comment: –
Multiline comment: /* ...... Some text...... */

5.8. How you declare a record type variable in PL/SQL?

We can declare record type variable for particular table by using the syntax.
<Variable_Name> <Table_name>%ROWTYPE.
ROWTYPE is a keyword for defining record type variables.

5.9. Find out the error in the following PL/SQL statement?

```
IF condition THEN
sequence_of_statements1
ELSE
sequence_of_statements2
END IF;
```

Answer: No Error in the Statement.

5.10. Mention the facilities available for iterating the statements in PL/SQL?

(a) For-loop
(b) While-loop
(c) Loop-Exit

5.11. What is cursor and mention its types in Oracle?

A cursor is a mechanism that can be used to process the multiple row result sets one row at a time.

In other words, cursors are constructs that enable the user to name a private memory area to hold a specific statement for access at a later time. Cursors are an inherent structure in PL/SQL. Cursors allow users to easily store and process sets of information in PL/SQL program.

There are two types of cursors in Oracle

(a) Implicit and
(b) Explicit cursors.

5.12. Mention the syntax for opening and closing a cursor.

For Opening: Open <cursor name>
For Closing: Close <cursor name>

5.13. Mention some implicit and explicit cursor attributes.

Implicit:
%NOTFOUND, %FOUND, % ROWCOUNT, and %ISOPEN

Explicit:
Similar to Implicit.
%NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN

5.14. What is Procedure in PL/SQL?

A procedure is a subprogram that performs some specific task, and stored in the data dictionary. A procedure must have a name so that it can be invoked or called by any PL/SQL program that appears within an application. Procedures can take parameters from the calling program and perform the specific task. Before the procedure or function is stored, the Oracle engine parses and compiles the procedure or function.

5.15. Mention any four advantages of procedures and function?

1. It modifies one routine to affect multiple applications.
2. It modifies one routine to eliminate duplicate testing.
3. It ensures that related actions are performed together, or not at all, by doing the activity through a single path.
4. It avoids PL/SQL parsing at runtime by parsing at compile time.

5.16. What is the syntax used in PL/SQL for dropping a procedure?

DROP PROCEDURE <PROCEDURE NAME>

5.17. Mention three differences between functions and procedures?

1. A procedure never returns a value to the calling portion of code, whereas a function returns exactly one value to the calling program.
2. As functions are capable of returning a value, they can be used as elements of SQL expressions, whereas the procedures cannot. However, user defined functions cannot be used in CHECK or DEFAULT constraints and can not manipulate database values, to obey function purity rules.

3. It is mandatory for a function to have at least one RETURN statement, whereas for procedures there is no restriction. A procedure may have a RETURN statement or may not. In case of procedures with RETURN statement, simply the control of execution is transferred back to the portion of code that called the procedure.

5.18. What is Purity rule for functions in PL/SQL?

For a function to be eligible for being called in SQL statements, it must satisfy following requirements, which are known as Purity Rules.

1. When called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
2. When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.
3. When called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot execute DDL statements (such as CREATE) because they are followed by an automatic commit.

5.19. What is a syntax for deleting a function in PL/SQL?

DROP FUNCTION <FUNCTION NAME>

5.20. What are parameters?

Parameters are the link between a subprogram code and the code calling the subprogram. Lot depends on how the parameters are passed to a subprogram.

5.21. What are Packages?

A package can be defined as a collection of related program objects such as procedures, functions, and associated cursors and variables together as a unit in the database. In simpler term, a package is a group of related procedures and functions stored together and sharing common variables, as well as local procedures and functions.

5.22. Mention any two advantages of Packages?

1. Stored packages allow you to sum up (group logically) related stored procedures, variables, and datatypes, and so forth in a single-named, stored unit in the database. This provides for better orderliness during the development process. In other words packages and its modules are easily understood because of their logical grouping.

2. Grouping of related procedures, functions, etc. in a package also make privilege management easier. Granting the privilege to use a package makes all components of the package accessible to the grantee.

5.23. Mention how exception handling is done in Oracle?

During execution of a PL/SQL block of code, Oracle executes every SQL sentence within the PL/SQL block. If an error occurs or an SQL sentence fails, Oracle considers this as an Exception. Oracle engine immediately tries to handle the exception and resolve it, by raising a built-in Exception handler.

5.24. Mention two advantages of using exceptions in Oracle?

1. Control over abnormal exits of executing programs on encountering error conditions, hence the behavior of application becomes more reliable.
2. In traditional error checking system, if same error is to be checked at several places, you are required to code the same error check at all those places. But with exception handling technique, you will write the exception for that particular error only once in the entire code. Whenever that type error occurs at any place in code, the exceptional handler will automatically raise the defined exception.