Tech Mahindra

# Oracle

## PL/SQL Study Material

For ***Oracle*** and ***BI*** streams

# Introduction to PL/SQL

## What Is PL/SQL?

A computer language is a particular way of giving instructions to (that is, programming) a computer. Computer languages tend to have a small vocabulary compared to regular human language. In addition, the way you can use the language vocabulary—that is, the grammar—is much less flexible than human language. These limitations occur because computers take everything literally; they have no way of reading between the lines and assuming what you Intended.

*Procedural* refers to a series of ordered steps that the computer should follow to produce a result. This type of language also includes data structures that hold information that can be used multiple times. The individual statements could be expressed as a flow chart (although flow charts are out of fashion these days). Programs written in such a language use its sequential, conditional, and Iterative constructs to express *algorithms*. So this part of the PL/SQL's definition is just saying that it is in the same family of languages as BASIC, COBOL, FORTRAN, Pascal, and C.

## Language Categories

Saying that PL/SQL is a procedural language makes more sense when you understand some other types of programming languages. There are at least four ways to categorize popular languages.

### Procedural programming languages

Allow the programmer to define an ordered series of steps to follow in order to produce a result. Examples:  PL/SQL, C, Visual Basic, Perl, Ada.

### Object-oriented programming languages

Based on the concept of an *object*, which is a data structure encapsulated with a set of routines, called *methods* that operate on the data. Examples: Java, C++, JavaScript, and sometimes Perl and Ada 95.

## Declarative programming languages

Allow the programmer to describe relationships between variables in terms of functions or rules; the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result. Examples: Logo, LISP, and Prolog.

## Markup languages

Define how to add information into a document that indicates its logical components or that provides layout instructions. Examples: HTML, XML.

*Structured Query Language* is a language based on set theory, so it is all about manipulating sets of data. SQL consists of a relatively small number of main commands such as SELECT, INSERT, CREATE, and GRANT; in fact, each statement accomplishes what might take hundreds of lines of procedural code to accomplish. That's one reason SQL-based databases are so widely used.

## Learning Oracle PL/SQL

PL/SQL, Oracle's programming language for stored procedures, delivers a world of possibilities for your database programs. PL/SQL supplements the standard relational database language, SQL, with a wide range of procedural features, including loops, IF-THEN statements, advanced data structures, and rich transactional control--all closely integrated with the Oracle database server.

PL/SQL is a procedural structured Query Language, is an extension to SQL where we can write programs using all the SQL statements and procedural statements.

### Various procedural statements we can use in PL/SQL are

- Assignment statements
- Conditional statements
- Loops
- Transactional processing statements

### Assignment statements

In C like programming language, we use = as assignment operator and == as comparison operator. Where as in PL/SQL we use = as comparison operator and := as assignment operator and the statement used with this operator is called as assignment statement. An example of assignment is shown below.

```
c := a + b;
```

## Conditional statements

Generally, in any programming language, we use If statement as conditional statement.

Various formats of ***IF*** statement can be used in programming.

### Simple If:

```
IF <CONDITION>  THEN
    ST1;
    ST2;
ELSE
    ST3;
END IF;
```

### ElseIf Ladder:

```
IF <CONDITION1> THEN
    Statements;
ELSIF <CONDITION2> THEN
    Statements;
ELSIF <CONDITION3> THEN
    Statements;
ELSE
    Statements;
END IF;
```

In addition to these IF statements, we can also use SQL functions DECODE () and CASE.

## Loops

In any programming language, we use tow different types of Loops

- Iterative Loops
- Conditional Loops

## Iterative Loop

### *For Loop*

For is an iterative loop. This loop performs operation for a range of values.

```
FOR loop_counter IN [REVERSE] lower_bound .. upper_bound
LOOP
    Statements
END LOOP;
```

| | |
|---|---|
| loop_counter | An identifier that has not been declared in the program, this variable gives you a way of detecting the "trip number" through the loop. |
| lower_bound | A numeric expression that Oracle uses to compute the smallest value assigned to loop_counter. Often, this will just be the number 1. You should make this an integer, but if you don't, PL/SQL automatically rounds it to an integer. If the lower bound is greater than the upper bound, the loop will not execute; if it is null, your program will end in a runtime error. |
| upper_bound | A numeric expression that Oracle uses as the maximum value for the loop_counter. |

***REVERSE***

Without this keyword, the loop counter increases by one with every trip through the loop, from the lower to the upper bound. With REVERSE, though, the loop will *decrease* by one instead, going from the upper to the lower bound.

```
FOR loop_counter IN REVERSE 1..3
LOOP
    Statements;
END LOOP;
```

Also, the low and high values in the FOR loop range do not have to be literals, as you can see in the next example:

```
FOR month_num IN 1 .. TO_NUMBER(TO_CHAR(SYSDATE, 'MM'))
LOOP
    Statements;
END LOOP;
```

As you're working with loops, it's important to know that PL/SQL declares the loop counter variable for you automatically and there can be problems if it has the same name as a variable you've declared in the usual place (the program's declaration section). The scope (the only part of the code in which it can be referenced) of the loop counter is between the LOOP and END LOOP keywords.

Guidelines to use FOR Loop

- Use a FOR loop to shortcut the test for the number of iterations
- Do not declare the counter; it is declared implicitly
- lower_bound .. upper_bound is required
- Reference the counter within the loop only; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.

## Conditional Loops

### *Simple Loop*

The simple loop is the, well, simplest loop structure. It has the following syntax:

```
LOOP
   EXIT WHEN <condition>;
   Statements;
END LOOP;
```

We can use this exit condition any where within the loop.

### *While Loop*

```
WHILE <condition>
LOOP
    statements;
End LOOP;
```

## Transactional Processing Statements

We use COMMIT and ROLLBACK as transactional processing statements. Once the transaction is over then you decide whether you save the data or not to save the data.

**Note**: PL/SQL supports 4GL (object oriented programming language) features.

## Programming structure

Every language uses some structure in writing programs. PL/SQL also uses two different types of block structures in writing programs.

- Unnamed PL/SQL block structure
- Named PL/SQL block structure

## Unnamed PL/SQL block structure

They are also called as anonymous blocks; they do not have names in the database. Its code looks as follows

```
DECLARE
    Declaration section  (optional)
BEGIN
    Executable section   (mandatory)
EXCEPTION
    Exception section    (optional)
END;
```

- Between DECLARE and BEGIN we can declare all the variables, constants which are going to be used in the program. It is called as DECLARATIONSECTION (optional).
- The BEGIN and END keywords are mandatory and enclose the body of actions to be performed. This section is referred to as the EXECUTABLE SECTION (mandatory) where we write actual logic of the program.
- The section between EXCEPTION and END is referred to as the exception section. The exception section traps error conditions.

## Errors in PL/SQL Programming

Errors are of two types.

- Primitive errors
- Logical or runtime errors

– Primitive (syntax) errors are handled at the time of compiling the program itself.
– Logical or runtime errors are handled only at runtime.

– These logical errors are handled in the EXCEPTION SECTION.

– This section is optional.

– The keywords DECLARE, semicolons do not follow BEGIN and EXCEPTION, but END and all other PL/SQL statements do require statement terminators.

## Summary

| Section | Description | Inclusion |
|---|---|---|
| Declarative | Contains all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and declarative sections | Optional |
| Executable | Contains SQL statements in manipulate data in the database and PL/SQL statements to manipulate data in the block | Mandatory |
| Exception | Specifies the sections to perform when errors and abnormal conditions arise in the executable section | Optional |

## Identifiers/Variables

### Naming Rules for Identifiers

Identifiers are the names given to PL/SQL elements such as variables, procedures, variables, and user-defined types. These names must follow certain rules of the road; namely, they:

- Are no more than 30 characters in length. Start with a letter.
- Consist of any of the following: letters, numerals, the dollar sign ($), the hash sign (#), and the underscore (_).
- Cannot be the same as a PL/SQL reserved word.
- Are unique within its scope. You cannot, in other words, declare two variables with the same name in the same block of code.

These are valid names for variables:

Birthdate
vote_total
sales_year7
contribution$
item#

These names, on the other hand, will cause compilation errors:

| | |
|---|---|
| the_date_of_birth_of_my_grandchildren | TOO LONG |
| 1st_choice | STARTS WITH DIGIT |
| Myemail@stevenfeuerstein.com | CONTAINS INVALID CHARACTER |

## Scope of Variables

The *scope* of a variable is the portion of PL/SQL code in which that variable can be referenced (i.e., its value read or modified). Most of the variables you define will have as their scope the block in which they were defined. Consider the following block:

```
DECLARE
    v_book_title VARCHAR2 (100);
BEGIN
    v_book_title := 'Learning Oracle PL/SQL';
END;
/
```

The variable **v_*book_title*** can be referenced within this block of code, but nowhere else. So if it happens to write another separate block of code, any attempt to read or change the value of **v_*book_title*** will result in a compilation error. Consider the block below that is assumed to be the continuation of the above block.

```
SQL> BEGIN
2      IF v_book_title LIKE '%PL/SQL%'
3      THEN
4          buy_it;
5      END IF;
6 END;
```

In this case the above block will produce a compilation because it has a variable **v_book_title**, that is not defined in that block. We can not access the variables in another block here.

## Types of Variables

### *PL/SQL variables:*

All PL/SQL variables have a data type, which specifies a storage format, constants, and valid range of values. Pl/SQL supports four data type categories

- Scalar
- Composite
- Reference
- LOB(large objects)

### *Non-PL/SQL variables:*

They include host language variables declared in precompiler programs, screen fields in Forms applications, and SQL*Plus or iSQL*Plus host variables.

## Using SQL or iSQL*plus variables within PL/SQL Blocks

- PL/SQL does not have input or output capability of its own.
- You can reference substitution variables within a PL/SQL block with a preceding ampersand.
- ISQL*Plus host (or bind) variables can be used to pass run time values out of the PL/SQL block back to the iSQL*Plus environment.

## Declaration of variables

A variable in PL/SQL can be declared in two different ways

1) By using the predefined/user defined data types and sizes
2) By referencing to a predefined column of a database object

## Declaring variable by giving its data type and size

**Syntax**: Identifier [CONSTANT] DATATYPE [NOT NULL] [ := | DEFAULT expr];

Example:

```
v_joindate    DATE;
v_empno       NUMBER(3) NOT NULL := 1001;
v_location    VARCHAR2(10)       := 'sec-bad';
c_comm.       CONSTANT NUMBER    := 1400;
```

**Note:**

1. **=** Used as comparison operator and **:=** as assignment operator.
2. Naming convention for variable, variable name starts with "v" and constant start with "c".
3. We can also give integrity constraints at the time of declaring variableVename varchar2 (10) not null.
4. Only one variable is allowed to declare in each line.
5. Follow naming conventions.

## Declaring variable by using *%TYPE* Attribute

A variable declared by referring column of a table. We can also use %TYPE attribute to declare a variable according to another previously declared variable.

Example

```
v_empno       emp.empno%TYPE;
v_salary      NUMBER(8,2);
v_comm.       v_salary%TYPE;
```

**Note**: A NOT NULL database column constraint does not apply to variables that are declared using %TYPE. Therefore, if you declare a variable using the %TYPE

attribute that uses a database column defined as NOT NULL, you can assign the NULL value to the variable.

Only one variable is allowed to declare in each line.

### Naming convention for variable

Variable name starts with “v”, constant start with “c” and parameters with “p”.

We can also give integrity constraints at the time of declaring variable

```
v_ename VARCHAR2(10) NOT NULL;
```

### Scalar Data type variables

- Hold a single value
- Have no internal components

### *Base Scalar Data Types*

| Data Type | Description |
|---|---|
| ***CHAR [(max-Len)]*** | Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1. |
| ***VARCHAR2 (max-Len)*** | Base type for variable-length character data up o 32,767 bytes. There is no default size for VARCHAR2 variables and constants. |
| ***LONG*** | Base type for variable-length character data up to 32,760 bytes. Use the LONG data type to store variable-length character strings. You can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2 ** 31 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG column into a LONG variable. |
| ***LONG RAW*** | Base type for binary data and byte strings up to 32,760 bytes. |
| ***NUMBER(precision, scale)*** | Number having precision p and scale s. the precision p can range from 1 to 38. |
| ***BINARY_INTEGER*** | Base type for integers between –2,147,483,647 and 2,147,483,647 |
| ***PLS_INTEGER*** | BASE type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER |

| | |
|---|---|
| | values require less storage and are faster than NUMBER and BINARY_INTEGER values. |
| ***BOOLEAN*** | Base type that stores one of three possible values |
| ***DATE*** | Base type for date and time |

## *Declaring Boolean Variables*

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. The variables are compared by the logical operators AND, OR and NOT.

Arithmetic, character, and date expressions can be used to return a Boolean value.

```
v_s1   := 5000;
v_s2   := 5500;
v_s1 < v_s2

ANALYSIS:
Results TRUE
```

## Composite Data type variables

A scalar type has no internal components. A composite type has internal components that can be manipulated individually. Composite data types (Also known as collections) are TABLE, RECORD, NESTED.

**Syntax**: <variable> <tablename>%ROWTYPE;

```
v_rec   emp%ROWTYPE;
```

## LOB Data Type Variables

With LOB (large object) data types you can store blocks of unstructured data (such as text, graphic, images, video clips and sound wave forms) up to 4GB in size. LOB data types allow efficient, random, piecewise access to the data and can also be used as attributes of an object type. LOBs also support random access to data.

**CLOB** (Character large object) data type is used to store large blocks of single-byte character data in the database.

**BLOB** (Binary large object) data type is used to store large binary objects in the database.

**BFILE** (Binary file) data type is used to store large binary objects in operating system files outside the database.

**NCLOB** (National language character large object) data type is used to store large blocks of single-byte or fixed-width multi-byte NCHAR Unicode data in the database, in line or out of line.

## Use of variables

Variables can be used for:

**Temporary storage of data:** Data can be temporarily stored in one or more variables for use when validating data input and for processing later in the data flow process.

**Manipulation of stored values:** Variables can be used for calculation and other data manipulations without accessing the database.

**Reusability:** After they declared, they can be used repeatedly in an application simply by referencing them in other statements, including other statements, including other declarative statements.

**Ease of maintenance:** When we declare variables using %TYPE and %ROWTYPE, if any underlying definition changes, the variable definition changes accordingly at run time. This provides data independence, reduces maintenance costs.

Program

Write a program to maintain students information with Rollno, Student_name, sub1, sub2, total, average where sub1, sub2 are the marks in different subjects.

Solution

Step 1 create student table with above structure

Step 2 In SQL*plus environment, open the system editor using a command

SQL> EDIT student

Or

In iSQL*plus environment, write the code in Executable window and save the script into a file.

```
DECLARE
    v_stu student%ROWTYPE;
BEGIN
    -- generate rollno automatically
    SELECT NVL(MAX(rollno),0) + 1 INTO v_stu.rollno FROM student;
    -- input name and marks in two subjects
    v_stu.sname    := '&name';
    v_stu.sub1     := &marks1;
    v_stu.sub2     := &marks2;
    v_stu.total    := v_stu.sub1 + v_stu.sub2;
    v_stu.average  := v_stu.total/2;
    INSERT INTO STUDENT VALUES v_stu;
    COMMIT;
END;
/
```

To run PL/SQL program From SQL*Plus Environment

```
SQL> START student
       Or
SQL> @student
```

From iSQL*Plus Environment

```
Use Execute button
```

## Bind Variables

A bind variable is a variable that you declare in a host environment. Bind variables can be used to pass run-time values, either number or character, into or out of one or more PL/SQL programs.

The PL/SQL program uses bind variables as they would use any other variable. You can reference variables declared in the host or calling environment in PL/SQL statements, unless the statement is in a procedure, function, or package.

### *Creating Bind Variables*

To declare a bind variable in the iSQL*Plus environment, use the command VARIABLE.

For example, we can declare a variable of type number and VARCHAR2 as follows:

```
VARIABLE v_sal NUMBER
VARIABLE v_getmsg AS VARCHAR2(30)
```

Both SQL and iSQL*Plus environments can reference the bind variable, and iSQL*Plus can display its value through PRINT command

### *Using Bind Variables*

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:)

```
VARIABLE g_salary number
BEGIN
    SELECT sal INTO :g_salary FROM EMP WHERE empno = 7521;
END;
/
PRINT g_salary
```

## Referencing Non-PL/SQL Variables

To reference host-variables, you must prefix the references with an ampersand (&) to distinguish them from declared PL/SQL variables

```
VARIABLE g_month_sal NUMBER
DEFINE   p_annual_sal=50000
SET VERIFY OFF

DECLARE
    v_sal  NUMBER(9,2) := &p_annual_sal;  -- reads the value from host variable
BEGIN
     :g_monthly_sal := v_sal/12;
END;
/
PRINT  g_monthly_sal
```

**Note:** The DEFINE command specifies a user variable and assigns it a CHAR value. Even though you enter the number 50000, iSQL*Plus assigns a CHAR value to p_annual_sal consisting of the characters 5,0,0,0 and 0.

## Embedding Single Quotes inside a String

The trickiest part of working with string literals comes when you need to include a single quote inside a string literal (as part of the literal itself). Generally, the rule is that you write two single quotes next to each other inside a string if you want the literal to contain a single quote in that position.

The following table shows the literal in one column and the resulting "internal" string in the second column:

| Literal | Actual Value |
|---|---|
| ‘There’ ‘s no business like show business’ | There’s no business like show business |
| ‘ “INSTITUTE FOR TECHNOLOGY” ‘ | “INSTITUTE FOR TECHNOLOGY” |
| ‘NLS LANGUAGE= ‘ ‘ENGLISH’’’ | NLS LANGUAGE = ‘ENGLISH’ |
| ‘ ‘ ‘ ‘ | ‘ |
| ‘ ’ ‘HELLO’ ‘ ‘ | ‘HELLO’ |

## Programming Guidelines

Make code maintenance easier by:

- Documenting code with comments
- Developing a case convention for the code
- Developing naming conventions for identifiers and other objects
- Enhancing readability by Indenting

### *Code Conventions*

The following table provides guidelines for writing code in uppercase or lowercase to help you distinguish keywords from named objects.

| **Category** | **Case Convention** | **Examples** |
|---|---|---|
| SQL statements | Uppercase | SELECT, INSERT |
| PL/SQL keywords | Uppercase | DECLARE,BEGIN,IF |
| Data types | Uppercase | VARCHAR2, BOOLEAN |
| Identifiers and parameters | Lowercase | v_sal_emp_cursor, p_sal |
| Database tables and columns | Lowercase | emp, empno |

Exercise

1) Evaluate each of the following declarations. Determine which of them are not legal and explain why.

a) v_id NUMBER(4);
b) v_x,v_y,v_z VARCHAR2(10);
c) V_birthdate DATE NOT NULL;
d) v_in_stock BOOLEAN :=1;

2) Identify the values at the arrow positions

```
DECLARE
        v_weight   NUMBER(3) := 600;
        v_message  VARCHAR2(255) := 'Product 10012';
BEGIN
        DECLARE
            v_weight    NUMBER(3) := 1;
            v_message   VARCHAR2(255):='Product 11001';
            v_new_locn  VARCHAR2(50) := 'Europe';
         BEGIN
            v_weight    := v_weight  + 1;
            v_new_locn  :=  'Western ' || v_new_locn;
            1 <----------
         END;
         v_weight    := v_weight + 1;
         v_message   := v_message || ' is in stock ' ;
         v_new_locn  := 'Western ' || v_new_locn;
         2 <----------
END;
/
```

Evaluate the PL/SQL block above and determine the data type and value of each of the following variables according to the rules of scooping.

a) The value of V_WEIGHT at position 1 is :
b) The value of V_NEW_LOCN at position 1 is :
c) The value of V_WEIGHT at position 2 is
d) The value of V_MESSAGE at position 2 is
e) The value of V_NEW_LOCN at position 2 is

3)

a) Create and execute a PL/SQL block that accepts two numbers through iSQL*Plus substitution variables

Use the DEFINE command to provide the two values

DEFINE p_num1 := 2;

DEFINE p_num2 := 4

b) Pass the two values defined in step a above, to the PL/SQL block through iSQL*Plus substitution variables. The first number should be divided by the second number and have the second number added to the result. The result should be stored in a PL/SQL variable and printed on the screen

## Summary

A Pl/SQL block is a basic, unnamed unit of a Pl/SQL program. It consists of a set of SQL or PL/SQL statements and it performs a single logical function. The declarative part is the first part of a PL/SQL block and is used for declaring objects such as variables, constants, cursors and exceptions. The executable part is the mandatory part of a PL/SQL block, and contains SQL and PL/SQL statements for querying and manipulating data. The exception-handling part is embedded inside the executable part of a block and is placed at the end of the executable part.

An anonymous PL/SQL block is the basic, unnamed unit of a PL/SQL program. Procedures and functions can be compiled separately and stored permanently in an Oracle database, ready to be executed.

## Exception Handling

Objectives

After completing this lesion, you should be able to do the following:

- Define PL/SQL exceptions
- Recognize unhandled exceptions
- List and use different types of PL/SQL exception handlers
- Trap unanticipated errors
- Describe the effect of exception propagation in nested blocks
- Customize PL/SQL exception messages

## Introduction

In PL/SQL, errors and warnings are called as exceptions. Whenever a predefined error occurs in the program, PL/SQL raises an exception. For example, if you try to divide a number by zero then PL/SQL raises an exception called ZERO_DIVIDE and if SELECT can not find a record then PL/SQL raises exception No_DATA_FOUND.

PL/SQL has a collection of predefined exceptions. Each exception has a name. These exceptions are automatically raised by PL/SQL whenever the corresponding error occurs.

In addition to predefined exceptions, user can also create his own exceptions to deal with errors in the application.

An exception is an identifier in PL/SQL that is raised during the execution of a block that terminates its main body of actions. A block always terminates when PL/SQL raises an exception, but can you specify an exception handler to perform final actions.

PL/SQL allows developers to raise and handle errors (exceptions) in a very flexible and powerful way. Each PL/SQL block can have its own exception section, in which exceptions can be trapped and handled (resolved or passed on to the enclosing block).

When an exception occurs (is raised) in a PL/SQL block, its execution section immediately terminates. Control is passed to the exception section.

Every exception in PL/SQL has an error number and error message; some exceptions also have names.

## Handling Exceptions with PL/SQL

An exception is an identifier in PL/SQL that is raised during execution.

How is it raised?

- An Oracle error occurs.
- You raise it explicitly.

How do you handle it?

- Trap it with a handler.
- Propagate it to the calling environment.

### *Declaring Exceptions*

Some exceptions (see the following table) have been pre-defined by Oracle in the STANDARD package.

You can also declare your own exceptions as follows:

```
DECLARE
   exception_name EXCEPTION;
```

An exception can be declared only once in a block, but nested blocks can declare an exception with the same name as an outer block. If this multiple declaration occurs, scope takes precedence over name when handling the exception. The inner block's declaration takes precedence over a global declaration.

When you declare your own exception, you must RAISE it explicitly. All declared exceptions have an error code of 1 and the error message "User-defined exception," unless you use the EXCEPTION_INIT pragma.

You can associate an error number with a declared exception with the PRAGMA EXCEPTION_INIT statement:

```
DECLARE
   Exception_name EXCEPTION;
   PRAGMA EXCEPTION_INIT (exception_name,error_number);
```

Where *error_number* is a literal value (variable references are not allowed). This number can be an Oracle error, such as -1855, or an error in the user-definable -20000 to -20999 range.

## *Raising Exceptions*

An exception can be raised in three ways:

- By the PL/SQL runtime engine
- By an explicit RAISE statement in your code
- By a call to the built-in function RAISE_APPLICATION_ERROR

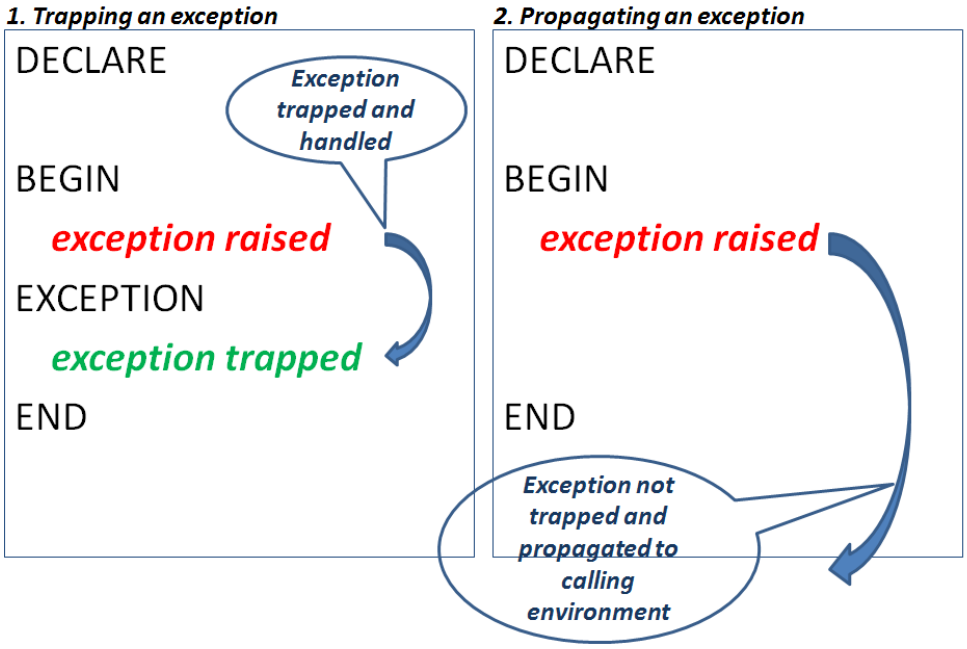
The syntax for the RAISE statement is:

```
RAISE exception_name;
```

Where *exception_name* is the name of an exception that you have declared, or that is declared in the STANDARD package.

If you use the RAISE statement inside an exception handler, you can leave off an exception name to re-raise

## Handling exceptions




## Trapping an Exception

If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagates to the enclosing block or environment. The PL/SQL block terminates successfully.

## Propagating an Exception

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to the calling environment.

## Exception Types

| | |
|---|---|
| Predefined Oracle Server | Implicitly Raised |
| Non-Predefined Oracle Server | |
| User Defined | Explicitly Raised |

You can program for exceptions to avoid disruption at run time. There are three types of exceptions.

| Exception | Description | Directions for Handling |
|---|---|---|

| | | |
|---|---|---|
| Predefined Oracle Server error | One of approximately 20 errors that occur most often in PL/SQL code | Do not declare and allow the Oracle server to raise them implicitly |
| No predefined Oracle Server error | Any other standard Oracle Server error | Declare within the declarative section and allow the Oracle Server to raise them implicitly. |
| User-defined error | A condition that the developer determines is abnormal | Declare within the declarative section, and raise explicitly. |

## *Trapping Exceptions*

**Syntax**: EXCEPTION
WHEN exception1 [ OR exception2 … ] THEN
Statement1;
Statement2;
WHEN exception3 [ OR exception4 …] THEN
Statement3;
Statement4;
WHEN OTHERS THEN
Statement5;
Statement6;

The following PL/SQL block attempts to select information from the employee and includes an exception handler for the case in which no data is found:

```
DECLARE
   vempno NUMBER;
BEGIN
   SELECT empno INTO vempno FROM EMP WHERE ename = 'RAM';
EXCEPTION
   WHEN NO_DATA_FOUND THEN
   INSERT INTO emp (empno, ename, job, deptno) VALUES (101,'RAM', 'EXECUTIVE', 10);
END;
```

In other words, if I am not already an employee in the company, the SELECT statement fails and control is transferred to the exception section (which starts with the keyword EXCEPTION). PL/SQL matches up the exception raised with the exception in the WHEN clause (NO_DATA_FOUND is a named, internal exception that represents ORA-01403-no data found). It then executes the statements in that exception handler, so I am promptly inserted into the employee table.

## *The WHEN OTHERS clause*

```
EXCEPTION WHEN OTHERS THEN
...
End;
/
```

Use the WHEN OTHERS clause in the exception handler as a catch all to trap any exceptions that are not handled by specific WHEN clauses in the exception section. If present, this clause must be the last exception handler in the exception section.

The OTHERS handler traps all exceptions not already trapped. Some Oracle tools have their own predefined exceptions that you can raise to cause events in the application. The OTHERS handler also traps these exceptions.

## *Trapping Exceptions Guidelines*

- The EXCEPTION keyword starts exception-handling section.
- several exception handlers are allowed.
- only one handler is processed before leaving the block.
- WHEN OTHERS is the last clause.
- Exceptions cannot appear in assignment statements or SQL statements.
- You can have only one OTHERS clause.

## *Trapping Predefined Oracle Server Errors*

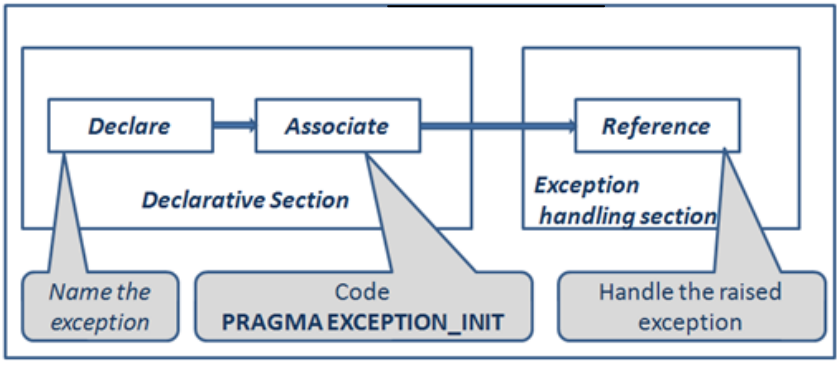
- Reference the standard name in the Exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

For a complete list of predefined exceptions, see PL/SQL User's Guide and Reference "Error Handling"

**Note**: PL/SQL declares predefined exceptions in the STANDARD package.

It is good idea to always handle the NO_DATA_FOUND and TOO_MANY_ROWS exceptions, which are the most common.

## *Trapping Non-predefined Oracle Server Errors*




Example

```
declare
    deptno_in number;
    still_have_employees EXCEPTION;
    PRAGMA EXCEPTION_INIT(still_have_employees,  -2292);
BEGIN
    deptno_in := &deptno;
    DELETE FROM d1 WHERE deptno = deptno_in;
EXCEPTION WHEN still_have_employees THEN
    DBMS_OUTPUT.PUT_LINE('Please delete employees in dept first');
    ROLLBACK;
    -- RAISE; /* Re-raise the current exception. */
END;
```

You trap a nonpredefined Oracle server error by declaring it first, or by using the OTHERS handler. The declared exception is raised implicitly. In PL/SQL, the PRAGMA_EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

**Note**: PRAGMA (also called pseudoinstructions) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the Pl/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle server error number.

## *Associating a PL/SQL Exception with a Number:*

## *Pragma EXCEPTION_INIT*

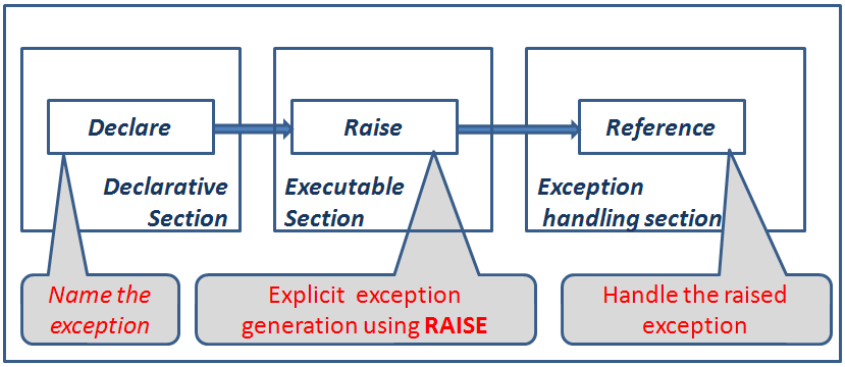
To handle error conditions (typically ORA- messages) that have no predefined name, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A pragma is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
DECLARE
   duplicate_value EXCEPTION;
   PRAGMA EXCEPTION_INIT(duplicate_value,-00001);
BEGIN
   INSERT INTO e VALUES(1);
EXCEPTION
   WHEN duplicate_value THEN
      dbms_output.put_line('duplicate');
END;
/
```

## *Trapping User-Defined Exceptions*




PL/SQL allows you to define your own exceptions. User-defined PL/SQL exceptions must be:

- Declared in the declare section of a PL/SQL block
- Raised explicitly with RAISE statements

Example

```
DEFINE p_department_desc = 'Accounts'
DEFINE p_department_number = 50
DECLARE
    E_invalid_department EXCEPTION;
BEGIN
    UPDATE dept SET dname = '&p_department_desc' WHERE deptno=&p_department_number;
    IF  SQL%NOTFOUND THEN
        RAISE e_invalid_department;
    END IF;
    COMMIT;
EXCEPTION
    WHEN e_invalid_department THEN
    DBMS_OUTPUT.PUT_LINE('No such department id');
END;
```

Program

Write a program to maintain students information with Rollno, Student_name, sub1,sub2, Total,average Where sub1,sub2 are the marks in different subjects.

Solution

Step 1 create student table with above structure

Step 2 In SQL*plus environment, open the system editor using a command

```
DECLARE
    v_sturec       STUDENT%rowtype;
    e_Negative     EXCEPTION;
BEGIN
    -- generate rollno automatically
    SELECT NVL(MAX(rollno),0) + 1 INTO v_sturec.rollno FROM student;
    -- input name and marks in two subjects
    v_sturec.sname  := '&name';
    v_sturec.sub1   := &marks1;
    v_sturec.sub2   := &marks2;
    IF v_sturec.sub1 < 0 or v_sturec..sub2  < 0 THEN
        RAISE e_Negative;
    END IF;
    v_sturec..total       := v_sturec..sub1 + v_sturec..sub2;
    v_sturec..average:= v_sturec..total/2;
    INSERT INTO STUDENT VALUES stu;
    COMMIT;
EXCEPTION
    WHEN e_negative THEN
       DBMS_OUTPUT.PUT_LINE(' -VE MARKS');
END;
/
```

To run PL/SQL program

```
From SQL*Plus Environment
SQL> start student
       Or
SQL> @student
From iSQL*Plus Environment
Use Execute button
```

**Program**

Write a program to get the salary of an employee

```
DECLARE
    v_empno     emp.empno%TYPE;
    v_sal       emp.sal%TYPE;
BEGIN
    v_empno    :=  &empno;
    SELECT sal into v_sal FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('SALARY = ' || v_sal);
EXCEPTION
    WHEN NO_DATA_FOUND THEN                    -- we are handling exception here
        DBMS_OUTPUT.PUT_LINE('Employee not found…');
END;
/
```

Please verify the above program for the following

1. Check what happened when we input a 5 digit number
2. Check what happens when we input employee number in single quotes
3. Check what happens when we input alpha-numeric information
4. Check what happens when there is a duplicate empno

To handle all the above errors, we can modify the above program as

```
DECLARE
    v_empno     emp.empno%TYPE;
    v_sal       emp.sal%TYPE;
BEGIN
    v_empno    :=  &empno;
    SELECT sal INTO v_sal FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('SALARY = ' || v_sal);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee not found…');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM || SQLCODE);
END;
/
```

Note :- SQLCODE and SQLERRM are called as Error Trapping functions.
SQLCODE displays error number (ORA-00001)
SQLERRM displays SQL error message

**Program**

Write a PL/SQL block that prints the number of employees who make plus or minus $100 of the salary value entered.

1. If there is no employee within that salary range, print a message to the user indicating that is the case. Use an exception for this case.
2. If there is one or more employees within that range, the message should indicate how many employees are in that salary range.
3. Handle any other exception with an appropriate exception handler. The message should indicate that some other error occurred.

```
VARIABLE g_message VARCHAR2(100)
SET VERIFY OFF
ACCEPT p_sal PROMPT 'Please enter the salary : '
DECLARE
    V_sal               emp.sal%type       := &p_sal;
    V_low_sal           emp.sal%type      := v_sal - 100;
    V_high_sal          emp.sal%type     :=  v_sal + 100;
    V_No_Emp            NUMBER(7);
    E_no_emp_returned   EXCEPTION;
    E_more_than_one_emp EXCEPTION;
BEGIN
    SELECT count(enme) INTO v_no_emp FROM emp WHERE sal between v_low_sal and
    v_high_sal;
    IF v_no_emp = 0 THEN
        RAISE e_no_emp_returned;
    ELSIF v_no_emp > 0 THEN
       RAISE e_more_than_one_emp;
   END IF;
EXCEPTION
WHEN e_no_emp_returned THEN
    :g_message := 'THERE is no employee salary between ' || v_lowsal || '  and '  ||
    v_high_sal;
WHEN e_more_than_one_emp THEN
    :g_message := ' There is/are ' || v_no_emp || ' employee(s) with a salary
    between ' || v_low_sal || ' and ' || v_high_sal;
END;
/
SET VERIFY ON
PRINT g_message
```

## Nested Blocks and Variable Scope

One of the advantages that PL/SQL has over SQL is the ability to nest statements. You can nest blocks whenever an executable statement is allowed, thus making the nested block a statement. Therefore, you can break down the executable part of a block into smaller blocks. The exception section can also contain nested blocks.

```
DECLARE
    v_empno  NUMBER(4) := 7521;
    v_sal    NUMBER;
BEGIN
    DECLARE
        v_empno  NUMBER(4) := 7843;
        v_sal    NUMBER;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = v_empno;
        DBMS_OUTPUT.PUT_LINE(v_sal);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee 7843 not found..');
    END;
    SELECT sal INTO v_sal FROM emp WHERE empno=vempno;
    DBMS_OUTPUT.PUT_LINE('salary = '|| v_sal);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee 7521 not found..');
END;
/
```

Exercise

1) Write a program to maintain employee details with

a) Empno, Ename, basic, da, hra, gross, pf, net
b) Generate Empno automatically
c) Input name and basic salary
d) Calculate da, hra, gross, pf and net
e) Assume da and hra some % on basic
f) Gross = basic + da + hra
g) Pf is 10% on gross
h) Net is gross – pf

2) Write a PL/SQL block to select the name of the employee with a given salary value.

a) Use the DEFINE command to provide salary
b) Pass the value in the PL/SQL block through a iSQL*Plus substitution variable. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the MESSAGES table the message “More than one employee with a salary of <salary>”
c) If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the MESSAGES table the message “No Employee with a salary of <salary>”.
d) If the salary entered returns only one row, insert into the MESSAGES table the employee’s name and the salary amount.
e) Handle any other exception with an appropriate exception handler and insert into the MESSAGES table the message “Some other error occurred”.
f) Test the block for a variety of text cases, Display the rows from the MESSAGES table to check whether the PL/SQL block has executed successfully.

3) Assume that there are two tables (LOWSAL and HISAL) with same structure

EMPNO
ENAME
SALARY

a) If the salary is < 10000 then handle the exception with an exception handler and insert into the MESSAGES table the message “This salary <salary> is not a valid amount”
b) If the salary is < 30000 then handle the exception with an exception handler and insert the details into LOWSAL table
c) Otherwise insert the details into HIGHSAL table.
d) Make sure that same employee should not appear in both tables.
e) Handle any other exception with an appropriate exception handler.

4)Write a program to maintain the passbook

| sno | vchdate | trntype | amount | balance |
|---|---|---|---|---|
| 1 | sysdate | d | 20000 | 20000 |
| 2 | sysdate | w | 5000 | 15000 |
| 3 | sysdate | w | 3000 | 12000 |
| 4 | sysdate | d | 10000 | 22000 |

a) generate sno automatically
b) get the opening balance for each transaction
c) Input (D)eposit / (W)ithdraw for trntype
d) Input amount and calculate the balance
e) Amount is always a +ve number
f) Withdrawal amount is always within the balance

## Summary

In this lesion, you should have learned that

- Exception types:
  - Predefined Oracle server error
  - Nonpredefined Oracle server error
  - User-defined error
- Exception trapping
- Exception handling
  - Trap the exception within the PL/SQL block.
  - Propagate the exception

***Exercise on Loops***

Write a program to print the numbers in the format

1 2 3

4 5 6

7 8 9

Write a program to accept a string only with lowercase letters

1) Solution

```
BEGIN
    DBMS_OUTPUT.NEW_LINE;
    FOR loop_counter1 IN 1 .. 8
    LOOP
        FOR loop_counter2 IN 1 .. 8
        LOOP
            DBMS_OUTPUT.PUT(loop_counter2||' ');
        END LOOP;
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;
/
```

2) Solution

?

# Cursors

## What is a Cursor?

Oracle uses work area to execute SQL commands and store processing information. PL/SQL allows you to access this area through a name using a cursor.

Whenever you issue a SQL statement, the Oracle Server opens an area of memory in which the command is parsed and executed. This area is called a cursor.

When you execute a SQL statement from PL/SQL, the Oracle RDBMS assigns a private work area for that statement. This work area contains information about the SQL statement and the set of data returned or affected by that statement.

The PL/SQL cursor is a mechanism by which you can name that work area and manipulate the information within it. In its simplest form, you can think of a cursor as a pointer into a table in the database.

Cursors are of two types in PL/SQL

- Implicit cursor
- Explicit cursor

## Implicit cursor

PL/SQL declares a cursor implicitly for all SQL data manipulation statements, including queries that return only one row. However, for queries that return more than one row, you must declare an explicit cursor or use a cursor FOR loop. The name of the implicit cursor is SQL. You can directly use the cursor without any declaration.

## Explicit Cursor

The set of rows returned by a query can consist of zero, one or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly declare a cursor to process the rows.

The set of rows returned by a multiple-row query is called the active set. *It is manipulated just like a file in programming languages.*

Explicit cursor functions

- Can process beyond the first row returned by the query, row by row.
- Keep track of which row is currently being processed.
- Allow the programmer to manually control them in the PL/SQL block.

Note: The fetch for an implicit cursor is an array fetch, and the existence of a second row still raises the TOO_MANY_ROWS exception. Furthermore, you can use explicit cursors to perform multiple fetches and to re-execute parsed queries in the work area.

## Handling Explicit Cursor

Explicit cursor is a name used to refer to an area where you can place multiple rows retrieved by SELECT

**STEPS**

The following are the required steps to process an explicit cursor.

- Declare the cursor in declare section
- Open the cursor using OPEN
- Fetch rows from the cursor FETCH
- Close the cursor after the process is over using CLOSE

## How does a cursor work?

Let us understand the cursor, with C File. To make any changes in the file first we have to open it (FOPEN () method). Once it is opened, it is placed in the memory and a file pointer identifies this memory area (FP). To read a particular line of information, we use a method (FREAD ()). The read information is stored into a variable. The variable size is equal to the size of the information that you are reading. To read each and every line from the file, we have to use a loop.

This loop is performed until it identifies EOF. Then we have to close the file (FCLOSE () method), to free the memory space.

Assume that the file is there in the common share folder(in the server). Reading each line from the server takes more time as well as we have to interact with the server repeatedly, which increases the network traffic and reduces the efficiency.

Similarly, assume that your server is a Oracle server and the file is a database. Every time we have to interact with server to get one row from the database. For

getting multiple records, each time we have to interact with database, which reduces the efficiency. To overcome this situation, we are defining a cursor, through which we can get the required information into the memory and that memory area is identified with cursor name (Like a file pointer).

In the place of fopen() method we are using OPEN() method. It places all the required data into the memory.

We are using fread() method for reading each line of information from the file. In the similar way we use FETCH method to get each record from memory.

After fetching one record, it should be placed into a variable. So, we have to create a variable whose size is equal to the size of the information that we are fetching.

Then we have to use a loop to read each and every record until nothing is there to fetch.

Then close the cursor using CLOSE () method.

In more technical terms, a cursor is the name for a structure in memory, called a *private SQL area*, which the server allocates at runtime for each SQL statement. This memory area contains, among other things, a parsed version of the original SQL statement.

If the host program uses any variables in the SQL statement, the cursor also contains the memory addresses of these variables.

When you put SELECT statements into your PL/SQL, there are two ways to deal with the data. You can use the SELECT INTO, as seen in the previous section, in which case you are using an *implicit cursor* ("implicit" because you don't refer to it specifically in your code; Oracle manages implicit cursors automatically). The second way gives you more direct control over the creation, naming, and use of cursors associated with your SELECTs. These cursors are called explicit cursors.

## How to Code an Explicit Cursor

An explicit cursor in PL/SQL is one with which you "explicitly" code each of the following steps:

1. Declare the cursor in declare section
2. Open the cursor using OPEN
3. Fetch rows from the cursor using FETCH
4. Close the cursor after the process is over using CLOSE.

### *Declare a Cursor*

A cursor is declared in DECLARE section using CURSOR statement.

**Syntax**: CURSOR <cursorname> IS <SELECT command>;

Example

```
CURSOR emp_cur IS
   SELECT empno, ename, job, sal FROM emp WHERE empno >= 7521;
```

**Note**: No data is actually retrieved at the time of cursor declaration. Data is placed in the cursor when cursor is opened.

### *Opening a cursor using OPEN command*

OPEN statement is used to execute the query associated with the cursor and place the result into cursor.

**Syntax**: OPEN cursor_name;

Example

```
Open emp_cur;
```

When a cursor is opened, all the rows retrieved by SELECT, given at the time of cursor declaration, are placed in the cursor.

## *Fetching rows from a cursor using FETCH command*

Once cursor is opened using OPEN, cursor has a set of rows, which can be fetched using FETCH statement.

**Syntax**: FETCH cursor_name INTO variable1, variable2, …

For each column in the cursor there should be a corresponding variable in FETCH statement.

FETCH statement is to be repeatedly executed to fetch all the rows of the cursor.

## *Closing a Cursor using CLOSE command*

CLOSE statement is used to close after the cursor is processed.

**Syntax**: CLOSE cursor_name

Example

```
CLOSE emp_cur;
```

Write a program to test the cursor

```
SET SERVEROUTPUT ON   -- SQL*plus Environment command
DECLARE
    CURSOR emp_cur IS SELECT empno, ename, job, sal FROM emp WHERE empno >= 7521;
    v_emp_rec  emp_cur%ROWTYPE;
BEGIN
    /* open the cursor */
    OPEN emp_cur;
    /* fetch a record from cursor */
    FETCH emp_cur INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || v_emp_rec.ename|| v_emp_rec.sal);
    -- closing the cursor
    CLOSE emp_cur;
END;
/
Analysis:
This program reads and prints only one record from cursor
```

Write a program to read each and every record from the cursor

```
SET SERVEROUTPUT ON
DECLARE
    Cursor emp_cur is Select empno, ename, job, sal FROM emp WHERE empno >= 7521;
    v_emp_rec  emp_cur%ROWTYPE;
BEGIN
    /* open the cursor */
    OPEN emp_cur;
    /* fetch all the records of the cursor one by one */
    LOOP
        FETCH emp_cur INTO v_emp_rec;
        /* Exit loop if reached end of cursor
        NOTFOUND is the cursor attribute */
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (v_emp_rec.empno || v_emp_rec.ename|| v_emp_rec.sal);
    END LOOP;
    CLOSE emp_cur; -- closing the cursor
END;
/
```

## Passing parameters to cursors

Program

```
SET SERVEROUTPUT ON
DECLARE
    -- p_empno is the formal parameter
    CURSOR emp_cur(p_empno NUMBER)
    IS SELECT empno, ename, job, sal FROM emp WHERE empno >= p_empno;
    v_emp_rec  emp_cur%ROWTYPE;
    v_eno      emp.empno%TYPE;
BEGIN
    /* input the employee number */
    v_eno  := &empno;
    /* open the cursor */
    OPEN emp_cur(v_eno);  -- Actual argument
    /* fetch all the records of the cursor one by one */
    LOOP
        FETCH emp_cur INTO v_emp_rec;
        --Exit loop if reached end of cursor NOTFOUND is the cursor attribute
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (emp_rec.empno || emp_rec.ename|| emp_rec.sal);
    END LOOP;
    -- closing the cursor
    CLOSE emp_cur;
END;
/
```

**Note**: We can pass any number of arguments

Program

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cur(p_empno_from number, p_empno_to number)
    IS   -- formal parameter and called as input arguments
    SELECT empno, ename, job, sal FROM emp
    WHERE empno BETWEEN p_empno_from AND p_empno_to;
    emp_rec      emp_cur%ROWTYPE;
    v_empno_from emp.empno%TYPE;
    v_empno_to   emp.empno%TYPE;
BEGIN
    /* input the employee number */
    v_empno_from  := &start_empno;
    v_empno_to    := &end_empno;
    /* open the cursor */
    OPEN emp_cur(v_empno_from,v_empno_to);  -- Actual argument
    /* fetch all the records of the cursor one by one */
    LOOP
        FETCH emp_cur INTO emp_rec;
        -- Exit loop if reached end of cursor NOTFOUND is the cursor attribute
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (emp_rec.empno ||
          emp_rec.ename|| emp_rec.sal);
    END LOOP;
    -- closing the cursor
    CLOSE emp_cur;
END;
/
```

## Declaring Multiple Cursors

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR dept_cur IS SELECT * FROM dept;
    CURSOR emp_cur(p_deptno IN NUMBER) IS
    SELECT empno,ename,sal FROM emp WHERE deptno=p_deptno;
    dept_rec   dept_cur%ROWTYPE;
    emp_rec    emp_cur%ROWTYPE;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO dept_rec;
        EXIT WHEN dept_cur%NOTFOUND:
        DBMS_OUTPUT.PUT_LINE('Employees working Under ' || dept_rec.deptno);
        OPEN emp_cur(dept_rec.deptno);
        LOOP
            FETCH emp_cur INTO emp_rec;
            EXIT WHEN emp_cur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(emp_rec.empno || emp_rec.ename||emp_rec.job);
        END LOOP;
        CLOSE emp_cur;
    END LOOP:
    CLOSE dept_cur;
END;
/
```

## Cursor Attributes

Cursor attributes allow getting information regarding cursor. For example, you can get the number of rows fetched so far from a cursor using ROWCOUNT

Use the following syntax to access cursor attributes Cursor_name%Attribute

The following is the list of available cursor attributes:

| Attribute | Data type | Significance | Recommended time to use |
|---|---|---|---|
| FOUND | BOOLEAN | TRUE if most recent fetch found a row in the table; otherwise FALSE | After opening and fetching from the cursor but before closing it (will be NULL before first fetch) |
| NOTFOUND | BOOLEAN | This the just logical inverse of FOUND | Same as above |
| ROWCOUNT | BOOLEAN | Number of Rows fetched so far | Same as above (except it will be zero before the first fetch) |

In addition to those cursor attributes, there are some less-commonly used cursor attributes that you might see from time to time. They include:

ISOPEN Returns TRUE or FALSE depending on whether cursor_name is open. For the implicit cursor ISOPEN is always FALSE.

Program

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cur IS SELECT empno, ename, job, sal FROM emp WHERE empno >= 7521;
    emp_rec  emp_cur%ROWTYPE;
BEGIN
    If NOT emp_cur%ISOPEN THEN
        OPEN emp_cur;
    END IF;
    LOOP
        FETCH emp_cur INTO emp_rec;
        IF emp_cur%FOUND THEN
            DBMS_OUTPUT.PUT_LINE('No of rows effected ' || emp_cur%ROWCOUNT);
        ELSE
            DBMS_OUTPT.PUT_LINE('END Of file ');
        END IF;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (emp_rec.empno || emp_rec.ename|| emp_rec.sal);
    END LOOP;
    CLOSE emp_cur;
END;
```

Cursor attributes can be used with both implicit and explicit cursors.

For example, the following shows how to use cursor attribute with implicit cursor:

```
BEGIN
    UPDATE emp SET sal = 1200 WHERE ename LIKE '%E%';
    /* If more than 5 rows are effected then rollback updation */
    IF SQL%ROWCOUNT > 5 THEN
        ROLLBACK;
    ELSE
       COMMIT;
    END IF;
End;
/
```

When %ROWCOUNT is used with implicit cursor, it returns number of rows affected by UPDATE, INSERT and DELETE commands. And when used with explicit cursors, it returns the number of rows fetched so far from the cursor.

## Cursor For Loop

In order to process a cursor, you can use cursor FOR loop to automate the following steps.

- Opening cursor
- Fetching rows from the cursor
- Terminating loop when all rows in the cursor are processed
- Closing cursor

**Syntax**: FOR rowtype_variable IN cursor_name
LOOP
Statements;
END LOOP;

Example

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cursor IS SELECT empno,ename FROM emp;
BEGIN
     FOR record_variable IN emp_cursor
     LOOP
         DBMS_OUTPUT.PUT_LINE(record_variable.empno || record_variable.ename);
     END LOOP;
END;
/
```

The following are the important steps in the above program:

- Cursor **emp_cursor** is automatically opened by cursor for loop
- **record_variable** is declared automatically as:
  **record_variable** **emp_cursor**%ROWTYPE;

But **record_variable** is available only inside the cursor for loop. It contains the same columns as the cursor. In order to access a column of the current row of the cursor, in the cursor for loop, use the format:

***record_variable.columnname***

- Statements in the cursor for loop are executed once for each row of the cursor. And for each row of the cursor, row is copied into **record_variable.**
- Loop is terminated once end of cursor is reached and cursor is closed.

**Note**: If parameters are to be passed to cursor, give the values after the name of the cursor.

```
DECLARE
    CURSOR emp_cursor(p_empno NUMBER)
    IS SELECT empno, ename FROM emp WHERE empno >= p_empno;
BEGIN
    FOR record_variable IN emp_cursor(7521)
    LOOP
        DBMS_OUTPUT.PUT_LINE(record_variable.empno || record_variable.ename);
    END LOOP;
END;
/
```

## *For Update Of and Current Of*

By default, Oracle locks rows while updating and deleting rows. But it is possible to override default locking by using FOR UPDATE.

FOR UPDATE clause can be used with SELECT while declaring cursor to lock all records retrieved by cursor to make sure they are not changed before we update or delete them. As Oracle automatically locks rows for you, FOR UPDATE clause is required only when you want to lock rows before the update or delete – at the time of opening cursor.

CURRENT OF clause can be used to refer to the current row in the cursor.

**Note**: FOR UPDATE must be given if you want to use CURRENT OF clause to refer to current row in the cursor.

Exercise

Write a program to Modify a particular entry in the passbook and update the corresponding balances.( Handle suitable exceptions).

PASSBOOK

| SNO | TRNDATE | TRNTYPE | AMOUNT | BALANCE |
|---|---|---|---|---|
| 1 | SYSDATE | D | 30000 | 30000 |
| 2 | SYSDATE | W | 10000 | 20000 |
| 3 | SYSDATE | W | 2000 | 18000 |
| 4 | SYSDATE | D | 5000 | 23000 |
| 5 | SYSDATE | W | 3000 | 20000 |
| 6 | SYSDATE | D | 8000 | 28000 |

In the above table, if we modify the amount in one particular transaction, Write a program to modify the corresponding balances.

Handle suitable exceptions, to accept only a positive amount and in no case, balance should become negative.

## Summary

A cursor is always used by PL/SQL to execute single-row queries and DML command. But in order to use multi-row query, you have to use an explicit cursor. An explicit cursor contains a row-set, which is retrieved by multi-row query.

Specially designed cursor FOR loop for cursors make cursor handling very easy and automatic. FOR UPDATE clause is used to override default locking and CURRENT OF is used to refer to current record in the cursor.

## Sub-Programs

Subprograms are named PL/SQL blocks that can accept parameters and be invoked from a calling environment.

PL/SQL has two types of subprograms, ***procedures*** and ***functions***.

– A procedure that performs an action
– A function that computes a value

A Subprogram

- is based on standard PL/SQL block structure
- provide modularity, reusability, extensibility and maintainability.
- provide easy maintenance, improved data security and integrity, improved performance, and improved code clarity

A subprogram is based on standard PL/SQL structure that contains a declarative section, an executable section, and an optional exception-handling section.

A subprogram can be compiled and stored in the database. It provides modularity, extensibility, reusability, and maintainability.

Modularization is the process of breaking up large blocks of code into smaller groups of code called modules. After code is modularized, the modules can be re-used by the same program or shared by other programs. It is easier to maintain and debug code of smaller modules than a single large program. Also, the modules can be easily extended for customization by incorporating more functionality, if required, without affecting the remaining modules of the program.

Subprograms provide easy maintenance because the code is located in one place and hence any modifications required to the subprogram can be performed in this single location. Subprograms provide improved data integrity and security. The data objects are accessed through the subprogram and a user can invoke the subprogram only if appropriate access privilege is granted to the user.

## Benefits of Sub Programs

- Easy maintenance
- Improved data security and Integrity
- Improved performance
- Improved code clarity

Procedures and functions have many benefits in addition to modularizing application development:

### *Easy maintenance*

Sub-programs are located in one location and hence it is easy to:

- Modify routines online without interfering with other users
- Modify one routine to affect multiple applications
- Modify one routine to eliminate duplicate testing

### *Improved data security and integrity*

They help controlling indirect access to database objects from non-privileged users with the help of security privileges. As a subprogram is executed with its definer's right by default, it is easy to restrict the access privilege by granting a privilege only to execute the subprogram to a user. And also ensures that related actions are performed together, or not at all.

### *Improved performance*

After a subprogram is compiled, the parsed code is available in the shared SQL area of the server and subsequent calls to the subprogram use this parsed code. This avoids reparsing for multiple users.

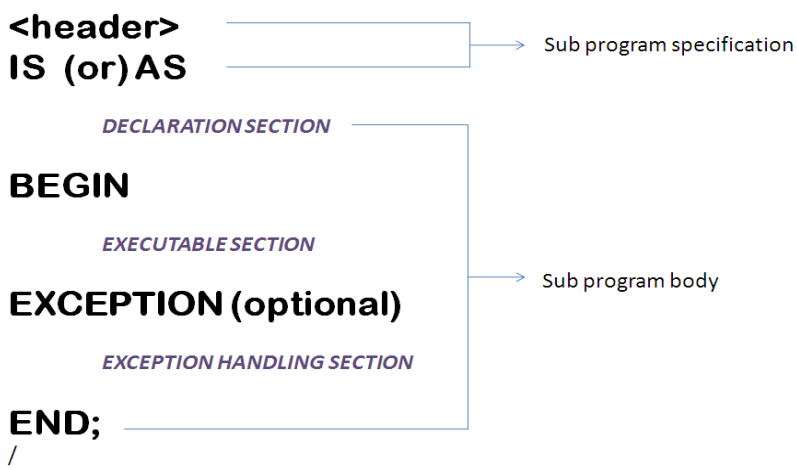
Avoids PL/SQL parsing at run time by parsing at compile time

Reduces the number of calls to the database and decreases network traffic by bundling commands.

### *Improves code clarity*

Using appropriate identifier names to describe the action of the routines reduces the need for comments and enhances the clarity of the code.

## Block structure for PL/SQL Subprograms




### *Sub-program Specification*

The header is relevant for named blocks only and determines the way that the program unit is called or invoked.

The header determines:

- The PL/SQL subprogram type, that is, either a procedure or a function
- The name of the subprogram
- The parameter list, if one exists
- The RETURN clause, which applies only to functions
- The IS or AS keyword is mandatory.

### *Sub-program Body*

The declaration section of the block between IS | AS and BEGIN. The keyword, DECLARE that is used to indicate the start of the declaration section in anonymous blocks is not used here.

The executable section between the BEGIN and END keywords is mandatory, enclosing the body of actions to be performed. There must be at least one statement existing in this section. There should be at least one NULL; statement, which is considered an executable statement.

The exception section between EXCEPTION and END is optional. This section traps predefined error conditions. In this section, you define actions to take if the specified error condition arises.

# Procedures

## Objectives

After completing this lesson, you should be able to do the following:

- Describe PL/SQL blocks and subprograms
- Describe the uses of procedures
- Create procedures
- Differentiate between formal and actual parameters
- List the features of different parameter modes
- Create procedures with parameters
- Invoke a procedure
- Handle exceptions in procedures
- Remove a procedure

## Lesson Aim

In this lesson, you learn the difference between anonymous PL/SQL blocks and subprograms. You also learn to create, execute, and remove procedures.

## What is a Procedure?

- A procedure is a type of subprogram that performs an action.
- A procedure can be stored in the database, as a schema object, for repeated execution.

## Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments), and be invoked. Generally speaking, you use a procedure to perform an action. A procedure has a header, a declaration section, an executable section, and an optional exception-handling section.

A procedure can be compiled and stored in the database as a schema object. Procedures promote reusability and maintainability. When validated, they can be used in any number of applications. If the requirements change, only the procedure needs to be updated.

## Creating Procedures

**Syntax**:
```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter1 [mode1] datatype1,
parameter2 [mode2] datatype2,. . .)]
IS|AS
PL/SQL Block;
```

## *Syntax Definitions*

You create new procedures with the CREATE PROCEDURE statement, which may declare a list of parameters and must define the actions to be performed by the standard PL/SQL block. The CREATE clause enables you to create stand-alone procedures, which are stored in an Oracle database.

PL/SQL blocks start with either BEGIN or the declaration of local variables and end with either END or END procedure_name. You cannot reference host or bind variables in the PL/SQL block of a stored procedure.

The REPLACE option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.

***You can not restrict the size of the data type in the parameters.***

| Parameter | Description |
|---|---|
| Procedure_Name | Name of the procedure |
| Parameter | Name of PL/SQL variable whose value is passed to or populated by the calling environment, or both, depending on the mode being used |
| Mode | Type of argument<br>IN (default)<br>OUT<br>IN OUT |
| Data type | Data type of argument – can be any SQL/ PLSQL data type. Can be of %TYPE, %ROWTYPE, or any scalar or composite data type. **You can not specify size of the data type in the parameters**. |
| PL/SQL block | Procedural body that defines the action performed by the procedure. |

## Formal versus Actual Parameters

***Formal parameters***: variables declared in the parameter list of a subprogram specification.

Example:

```
CREATE PROCEDURE raise_sal(p_id NUMBER, p_amount NUMBER)
    ...
END raise_sal;
```

***Actual parameters***: variables or expressions referenced in the parameter list of a subprogram call.

Example:

```
raise_sal(v_id, 2000)
```

## Procedural Parameter Modes

You can transfer values to and from the calling environment through parameters. Select one of the three modes for each parameter: IN, OUT, or IN OUT.

Attempts to change the value of an IN parameter will result in an error.

**Note**: DATATYPE can be only the %TYPE definition, the %ROWTYPE definition or an explicit data type with no size specification.

### *Type of Parameter Description*

IN (default) passes a constant value from the calling environment into the procedure.

OUT passes a value from the procedure to the calling environment.

IN OUT Passes a value from the calling environment into the procedure and a possibly different value from the procedure back to the calling environment using the same parameter

## Creating Procedures with Parameters

| IN | OUT | IN OUT |
|---|---|---|
| Default Mode | Must be specified | Must be specified |
| Value is passed into subprogram | Returned to Calling Environment | Passed into subprogram; returned to calling environment. |
| Formal parameters acts as a constant | Uninitialized variable | Initialized variable |
| Actual parameter can be a literal, expression, constant, or initialized variable | Must be a variable | Must be a variable |
| Can be assigned a default value | Cannot be assigned a default value | Cannot be assigned a default value |

## *IN Parameters example*

```
CREATE OR REPLACE PROCEDURE raise_salary(p_id IN emp.empno%TYPE)
IS
    v_sal  emp.sal%type;
BEGIN
    SELECT sal INTO v_sal FROM emp WHERE empno = p_id;
    DBMS_OUTPUT.PUT_LINE(v_sal);
END raise_salary;
/
```

The example shows a procedure with one IN parameter. Running this statement in *i*SQL*Plus creates the RAISE_SALARY procedure. When invoked, RAISE_SALARY accepts the parameter for the employee ID and gets the Salary of that employee.

To invoke a procedure in *i*SQL*Plus, use the EXECUTE command.

```
EXECUTE raise_salary (176)
```

To invoke a procedure from another procedure, use a direct call. At the location of calling the new procedure, enter the procedure name and actual parameters.

```
raise_salary (176);
```

IN parameters are passed as constants from the calling environment into the procedure. Any attempt to change the value of an IN parameter result in an error.

## *OUT Parameters example*

```
CREATE OR REPLACE PROCEDURE query_emp(p_id  IN emp.empno%TYPE,
p_name OUT emp.ename%TYPE, p_salary OUT emp.sal%TYPE, p_comm OUT emp.comm%TYPE)
IS
BEGIN
    SELECT ename, sal, comm. INTO p_name, p_salary, p_comm
    FROM emp WHERE empno = p_id;
END query_emp;
/
```

Run the script file shown in the slide to create the QUERY_EMP procedure. This procedure has four formal parameters. Three of them are OUT parameters that return values to the calling environment.

The procedure accepts an EMPNO value for the parameter P_ID. The name, salary, and commission values corresponding to the employee ID are retrieved into the three OUT parameters whose values are returned to the calling environment.

## *Viewing OUT Parameters*

```
VARIABLE g_name VARCHAR2(25)
VARIABLE g_sal NUMBER
VARIABLE g_comm NUMBER
EXECUTE query_emp(171, :g_name, :g_sal, :g_comm)
PRINT g_name
```

How to View the Value of OUT Parameters with *i*SQL*Plus

1. Run the SQL script file to generate and compile the source code.
2. Create host variables in *i*SQL*Plus, using the VARIABLE command.
3. Invoke the QUERY_EMP procedure, supplying these host variables as the OUT parameters.
   ***Note the use of the colon (:) to reference the host variables in the EXECUTE command***
4. Variables which are used with **:** are called as binded variables. The meaning of binded is, getting the value that we are getting from Pl/SQL environment to SQL environment.
5. To view the values passed from the procedure to the calling environment, use the PRINT command.

The example shows the value of the G_NAME variable passed back to the calling environment. The other variables can be viewed, either individually, as above, or with a single PRINT command.

```
PRINT g_name g_sal g_comm
```

Do not specify a size for a host variable of data type NUMBER when using the VARIABLE command. A host variable of data type CHAR or VARCHAR2 defaults to a length of one, unless a value is supplied in parentheses.

## IN OUT Parameters example

Create a procedure with an IN OUT parameter to accept a character string containing 10 digits and return a phone number formatted as (800) 633-0575.

```
CREATE OR REPLACE PROCEDURE format_phone (p_phone_no IN OUT VARCHAR2)
IS
BEGIN
    p_phone_no := '(' || SUBSTR(p_phone_no,1,3) || ')' || SUBSTR(p_phone_no,4,3) ||
    '-' || SUBSTR(p_phone_no,7);
END format_phone;
/
```

## Using IN OUT Parameters

With an IN OUT parameter, you can pass values into a procedure and return a value to the calling environment. The value that is returned is the original, an unchanged value, or a new value set within the procedure.

An IN OUT parameter acts as an initialized variable.

Run the statement to create the FORMAT_PHONE procedure.

## Viewing IN OUT Parameters

```
SQL> VARIABLE g_phone_no VARCHAR2(15)
SQL> BEGIN
2 :g_phone_no := '8006330575';
3 END;
4 /
SQL> PRINT g_phone_no
SQL> EXECUTE format_phone (:g_phone_no)
SQL> PRINT g_phone_no
```

## Methods for Passing Parameters

**Positional**: List actual parameters in the same order as formal parameters.

**Named**: List actual parameters in arbitrary order by associating each with its corresponding formal parameter.

**Combination**: List some of the actual parameters as positional and some as named.

Example

```
CREATE OR REPLACE PROCEDURE add_dept
   (p_name IN departments.department_name%TYPE DEFAULT 'unknown',
    p_loc IN departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
    INSERT INTO departments(department_id, department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
/
```

Examples of Passing Parameters

```
BEGIN
    add_dept;
    --above is true because the formal parameters have
    --default values associated with them
    add_dept ('TRAINING', 2500);
    --above method call is using positional parameters
    add_dept ( p_loc => 2400, p_name =>'EDUCATION');
    --above method call is using named parameter passing
    add_dept ( p_loc => 1200) ;
    --above method call using named parameter passing
    --it works fine because the second parameter has
    --default value associated with it
END;
/
```

Exercise

Write a procedure to get the salary of an employee

Write a procedure to get the name and hiredate of an employee

Write a procedure to delete particular employee

Write a procedure to encrypt and decrypt your password

Write a procedure to get the total salary of all the managers

Write a procedure to get the name and salary of the highest paid employee

## Functions

### Objectives

After completing this lesson, you should be able to do the following:

- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Differentiate between a procedure and a function

Lesson Aim

In this lesson, you will learn how to create and invoke functions.

## Overview of Stored Functions

- A function is a named PL/SQL block that returns a value.
- A function can be stored in the database as a schema object for repeated execution.
- A function is called as part of an expression.

## Stored Functions

A function is a named PL/SQL block that can accept parameters and be invoked.

Generally speaking, you use a function to compute a value. Functions and procedures are structured alike. A function must return a value to the calling environment, whereas a procedure returns zero or more values to its calling environment. Like a procedure, a function has a header, a declarative part, an executable part, and an optional exception-handling part. A function must have a RETURN clause in the header and at least one RETURN statement in the executable section.

Functions can be stored in the database as a schema object for repeated execution. A function stored in the database is referred to as a stored function.

**Syntax:**
```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter1 [mode1] datatype1,
parameter2 [mode2] datatype2,. . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

The PL/SQL block must have at least one RETURN statement.

Example

```
CREATE OR REPLACE FUNCTION get_sal(p_id  emp.empno%TYPE) RETURN NUMBER
IS
    v_salary emp.sal%TYPE ;
BEGIN
    SELECT sal INTO v_salary FROM emp WHERE empno = p_id;
    RETURN v_salary;
END get_sal;
/
```

## Executing Functions

- Invoke a function as part of a PL/SQL expression.
- Create a variable to hold the returned value.
- Execute the function. The variable will be populated by the value returned through a RETURN statement.

Example

```
VARIABLE g_salary NUMBER
EXECUTE :g_salary := get_sal(117)
PRINT g_salary
```

Exercise

Write a function to get the retirement date of a particular employee

Write a function to calculate income tax

Write a function to get the details of a particular employee

Write a program to maintain cashbook

Vchno vchdate trntype amount balance

## How Procedures and Functions Differ

1. You create a procedure to store a series of actions for later execution. A procedure can contain zero or more parameters that can be transferred to and from the calling environment, but a procedure does not have to return a value.

2. You create a function when you want to compute a value, which must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions should return only a single value, and the value is returned through a RETURN statement.

3. Functions used in SQL statements cannot have OUT or IN OUT mode parameters.

## Procedure to Encrypt a password

```
CREATE OR REPLACE PROCEDURE encry(p_user VARCHAR2,p_passwd VARCHAR2)
IS
    v_pass_encrypted VARCHAR2(100);
BEGIN
    FOR char_index IN 1 .. LENGTH(p_passwd)
    LOOP
        v_pass_encrypted := v_pass_encrypted ||
        CHR(ASCII(SUBSTR(UPPER(p_passwd),char_index,1))- 42) ;
    END LOOP;
    INSERT INTO logram VALUES(p_user, v_pass_encrypted);
    --DBMS_OUTPUT.PUT_LINE(v_pass_encrypted);
END;
/
```

## Procedure to Decrypt a password

```
CREATE OR REPLACE PROCEDURE decry(p_user VARCHAR2)
IS
   v_pass_encrypted VARCHAR2(100);
   v_pass_decrypted VARCHAR2(100);
BEGIN
   SELECT p INTO v_pass_encrypted FROM logram WHERE u=p_user;
   FOR char_index IN 1 .. length(v_pass_encrypted)
   LOOP
       v_pass_decrypted := v_pass_decrypted ||
       CHR(ASCII(SUBSTR(v_pass_encrypted,char_index,1))+42);
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(v_pass_decrypted);
end;
/
```

# PACKAGES

## What is A Package?

Packages are containers of related procedures, functions and variables.

When we create different procedures and functions, they are stored as independent objects. To group these objects together, we are creating packages.

Each package contains the following parts:

- Package specification
- Package body

### *Package Specification*

Contains all declarations for variable, cursor, procedures and functions that are to be made public. All public objects of package are visible outside the package.

**Syntax**:
```
CREATE OR REPLACE PACKAGE  package_name
IS
  /* declare public objects of package */
END;
```

### *Package Body*

Defines all the objects of the package. Objects declared in the specification are called as public objects and the objects directly defined in the body without declaring in the specification, are called as PRIVATE members.

**Syntax**:
```
CREATE or REPLACE PACKAGE BODY package_name
 IS
 [ declarations of variables and types ]
 [ specification and SELECT statement of cursors ]
 [ specification and body of modules ]
 [ BEGIN
 executable statements ]
 [ EXCEPTION
 exception handlers ]
 END [ package_name ];
```

In the body you can declare other variables, but you do not repeat the declarations in the specification.

The body contains the full implementation of cursors and modules. In the case of a cursor, the package body contains both specification and SQL statement for the cursor. In the case of a module the package body contains both the specification and body of the module.

The BEGIN keyword indicates the presence of an execution or initialization section for the package. This section can also optionally include an exception section.

As with a procedure or function, you can add the name of the package, as a label, after the END keyword in both the specification and package

The body of a package can contain

- Procedures declared in the package specification
- Functions declared in the package specification
- Definitions of cursors declared in the package specification
- Local procedures and functions, not declared in the package specification.
- Local variables

Program

```
CREATE OR REPLACE PACKAGE samplepack
IS
    PROCEDURE proc1(p_n  NUMBER, p_n1 OUT NUMBER);
    FUNCTION fun1(p_n NUMBER) RETURN NUMBER;
END;
/
CREATE OR REPLACE PACKAGE BODY samplepack
IS
    PROCEDURE proc1(p_n NUMBER, p_n1 OUT NUMBER)
    IS
    BEGIN
       p_n1 := p_n * 5;
    END proc1;
    FUNCTION fun1(p_n NUMBER) RETURN NUMBER
    IS
       v_n1 NUMBER;
    BEGIN
       v_n1  := p_n * 2;
       RETURN v_n1;
    END fun1;
END;
/
```

Execution

```
VARIABLE N NUMBER
EXECUTE SAMPLEPAK.PROC1(5,:N)
 PRINT N
EXECUTE :N := SAMPLEPAK.FUN1(4)
PRINT N
```

Program to define private member

```
CREATE OR REPLACE PACKAGE samplepack
IS
    PROCEDURE proc1(p_n  NUMBER, p_n1 OUT NUMBER);
    FUNCTION  fun1(p_n NUMBER) RETURN NUMBER;
 END;
/
CREATE OR REPLACE PACKAGE BODY samplepack
IS
    PROCEDURE test     -- private member definition
    IS
    BEGIN
       DBMS_OUTPUT.PUT_LINE('I AM A PRIVATE MEMBER');
    END test;
   PROCEDURE proc1(p_n NUMBER, p_n1 OUT NUMBER)
   IS
   BEGIN
       TEST;     -- private member called
       p_n1 := p_n * 5;
    END proc1;
    FUNCTION fun1(p_n NUMBER) RETURN NUMBER
    IS
       v_n1 NUMBER;
    BEGIN
       v_n1  := p_n * 2;
       RETURN v_n1;
    END fun1;
END samplepack;
/
```

Execution

```
VARIABLE N NUMBER
EXECUTE samplepack.proc1(5,:N)
PRINT N
EXECUTE :N := samplepack.fun1(4)
PRINT N
```

**Note**: A private member cannot be accessed by referring package object. They are called only through public members of the package object.

## *Program to test Polymorphism*

Program

```
CREATE OR REPLACE PACKAGE polypack
IS
    PROCEDURE proc1( v_n  NUMBER, v_n1 OUT NUMBER);
    PROCEDURE proc1(v_x VARCHAR2,v_y VARCHAR2,v_z OUT VARCHAR2);
END;
/
CREATE OR REPLACE PACKAGE BODY polypack
IS
    PROCEDURE proc1(v_n NUMBER, v_n1 OUT NUMBER)
    IS
    BEGIN
       v_n1 := v_n * 5;
    END proc1;
    PROCEDURE proc1(v_x VARCHAR2,v_y VARCHAR2,v_z OUT VARCHAR2)
    IS
    BEGIN
       Z :=  CONCAT(v_x,v_y);
    END proc1;
END polypack;
/
```

Execution

```
VARIABLE N NUMBER
VARIABLE  ST VARCHAR2(100)
EXECUTE polypack.proc1(5,:N)
PRINT N
EXECUTE  polypack.proc1('RAVI',' KIRAN',:ST)
PRINT ST
```

## *Refcursor With Packages*

```
CREATE OR REPLACE PACKAGE Employee_RefCur_pkg
AS
    TYPE empcur IS REF CURSOR;
    FUNCTION empsearch(p_ename VARCHAR2) RETURN empcur;
END Employee_RefCur_pkg;
/
CREATE OR REPLACE PACKAGE BODY Employee_RefCur_pkg
AS
    FUNCTION empsearch(i_ename VARCHAR2) RETIRM empcur
    IS
        v_empcursor empcur;
    BEGIN
        OPEN v_EmpCursor FOR SELECT empno, ename, job, sal FROM emp WHERE ename LIKE
        '%' || i_EName || '%' ORDER BY UPPER(emp.ename);
        RETURN v_empcursor;
    END EmpSearch;
END Employee_RefCur_pkg;
/
SELECT Employee_RefCur_pkg.empsearch('E') FROM dual
```

# Triggers

## Objectives

After completing this lesion, you should be able to do the following:

- Describe different types of triggers
- Describe database triggers and their use
- Create Database triggers
- Describe database trigger firing rules
- Remove database triggers

## Trigger - Levels

These Triggers are written at three different levels

- Schema
- Table
- Row

A Schema level trigger is a trigger which is written at database level or user level.. These triggers can be written by DBA only.

Any user can write table and ROW level triggers. Table level triggers are ment for providing security at object (Table) level.

Row level triggers are ment for validations

## Types of Triggers

A trigger:

- is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or the database.
- Executes implicitly whenever a particular event takes place
- Can be either
    - Application trigger: Fires whenever an event occurs with a particular application.
    - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database.

## *Types of Triggers*

- ROW Triggers and Statement Triggers
- BEFORE or AFTER Triggers
- INSTRAD-OF triggers
- Triggers on System events and User Events

**Row Triggers**: A row trigger is fired each time the table is affected by triggering statement. For example, if an UPDATE statement updates multiple rows of a table,

a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not executed at all.

A Row trigger is fired once for each row affected by the command. These triggers are used to check for the validity of the data in the triggering statements and rows affected.

**Statement Triggers:** A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects (even no rows are affected). 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.

**BEFORE and AFTER Triggers**: When defining a trigger, you can specify the trigger timing- whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers.

BEFORE and AFTER triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the on the base table(s) of a view fired if an INSERT, UPDATE, or DELETE statement is issued against the view.

BEFORE and AFTER triggers fired by DDL statements can be defined only on the database or a schema, not on particular tables.

**INSTEAD-OF Triggers:** These triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements. These triggers are called INSTEAD-OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.

## *Guidelines for Designing Triggers*

- Design triggers to:
  - Perform related actions
  - Centralize global operations
- Do not design triggers :
  - Where functionality is already built into the Oracle server
  - That duplicates other triggers
- Create stored procedures and invoke them in a trigger, if the PL/SQL code is very length.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

## How triggers are Used?

Triggers can supplement the standard capabilities of Oracle to provide a highly customized database management system.

For example,

- A trigger can restrict DML operations against a table to those issued during regular business hours.
- A trigger could also restrict DML operations to occur only at certain times during weekdays.

Other uses of triggers are to

- Prevent invalid transaction
- Enforce complex security authorizations
- Enforce complex business rules
- Gather statistics on table access
- Modify table data when DML statements are issued against views

## *Table Level Trigger*

Syntax:
```
CREATE OR REPLCE TRIGGER <triggername>
BEFORE | AFTER INSERT OR UPDATE OR DELETE ON  <table>
[DECLARE]
BEGIN
      ST1;
      ST2;
[EXCEPTION]
```

END;

### *ROW Level Trigger*

Syntax: CREATE OR REPLCE TRIGGER <triggername>
BEFORE | AFTER INSERT OR UPDATE OR DELETE ON <table>
FOR EACH ROW
[DECLARE]
BEGIN
ST1;
ST2;
[EXCEPTION]
END;

Example to create Table level or Statement triggers

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON emp
BEGIN
   IF TO_CHAR(SYSDATE,'DY') IN('SAT','SUN') OR TO_CHAR(SYSDATE,'HH24:MI') NOT
   BETWEEN '08:00' AND '18:00' THEN
      RAISE_APPLICATION_ERROR(-20001,'You may Insert into EMP table only during
      business hours..');
   END IF;
END;
/
```

**Note**: RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the Oracle server automatically rolls the triggering statement back.

Testing secure_emp

```
INSERT INTO EMP (empno, ename) VALUES (101,'ravi');
ERROR at line 1
ORA-20001 You may insert into EMP table only during business hours
ORA-06512  at PLSQL SECURE_emp, LINE 4
Ora-04088 error during execution of trigger "PLSQL SECURE_EMP"
```

Note: The row might be inserted if you are in a different time zone from the database server. The trigger fires even if your system clock is within these business hours.

### *Combining Triggering Events*

You can combine several triggering events into one by taking advantage of the special conditional predicates INSERTING, UPDATING and DELETEING within the trigger body.

Create one trigger to restrict all the data manipulation events on the EMP table to certain business hours, Monday through Friday.

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE  ON emp
BEGIN
    IF TO_CHAR(SYSDATE,'DY') IN('SAT','SUN')  OR
      TO_CHAR(SYSDATE,'HH24:MI') NOT BETWEEN '08:00' AND '18:00' THEN
          IF INSERTING THEN
             RAISE_APPLICATION_ERROR(-20001,'You may Insert into EMP table
                   only   during business hours..');
           ELSIF UPDATING THEN
             RAISE_APPLICATION_ERROR(-20001,'You may UPDATE  EMP table
                   only   during business hours..');
           ELSIF DELETING THEN
             RAISE_APPLICATION_ERROR(-20001,'You may DELETE from  EMP
                 table only   during business hours..');
           END IF;
    END IF;
END;
/
```

## Creating ROW Trigger

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

Create a trigger to allow only certain employees to be able to earn a salary of more than 15,000.

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF  sal ON emp
FOR EACH ROW
BEGIN
    IF (:NEW.job  NOT IN ('CLERK', 'SALESMAN')  AND :NEW sal > 15000 THEN
       RAISE_APPLICATION_ERROR(-20002,'Employees cannot earn this amount');
    END IF:
END;
/
```

### *Using OLD and NEW Qualifiers*

Within a ROW trigger, reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifiers.

| Data Operation | Old Value | New Value |
|---|---|---|
| INSERT | Null | Inserted value |
| UPDATE | Value before update | Value after update |
| DELETE | Value before delete | NULL |

- The OLD and NEW qualifiers are available only in ROW  triggers
- Prefix these qualifiers a colon ( : ) in every SQL and PL/SQL statement
- There is no colon ( : ) prefix if the qualifiers are referenced in the WHEN restricting condition

**Note:** Row triggers can decrease the performance if you do a lot of updates on larger tables.

Example using OLD and NEW Qualifiers

```
CREATE OR REPLACE TRIGGER AUDIT_EMP_VALUES
AFTER INSERT ON emp
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp_table
    (user_name,timestamp,id,old_name,new_name,old_salary,new_salary)
    VALUES(user,sysdate, :OLD.empno, :OLD.ename, :NEW.ename, :OLD.sal, :NEW.sal);
END;
/
```

Exercise

1. Write a trigger to access the table in a specified time
2. Write a trigger to accept only valid updations (Example, fire the trigger if the employee salary is modified with lesser value than his original salary)
3. Write a trigger to accept unique values in to a particular column of a table, when we are placing a new record.
4. Write a trigger to update stock automatically when customer makes a transaction using below tables

Itemmaster ( Itemno, itemname, stock)

Itemtran(transaction_no,itemno,transaction_date,transaction_type, quantity)

## PL/SQL Records

Records in PL/SQL programs are very similar in concept and structure to the rows of a database table. A record is a composite data structure, which means that it is composed of more than one element or component, each with its own value. The record as a whole does not have value of its own; instead, each individual component or *field* has a value. The record gives you a way to store and access these values as a group.

If you are not familiar with using records in your programs, you might initially find them complicated. When used properly, however, records will greatly simplify your life as a programmer. You will often need to transfer data from the database into PL/SQL structures and then use the procedural language to further manipulate, change, or display that data. When you use a cursor to read information from the database, for example, you can pass that table's record directly into a single PL/SQL record. When you do this you preserve the relationship between all the attributes from the table.

## Different Types of Records

PL/SQL supports three different kinds of records: table-based, cursor-based, and programmer-defined. These different types of records are used in different ways and for different purposes, but all three share the same internal structure: every record is composed of one or more fields. However, the way these fields are defined in the record depend on the record type. Table shows this information about each record type.

| Record-type | Description | Fields in Record |
|---|---|---|
| Table-based | A record based on a table's column structure | Each field corresponds to -- and has the same name as -- a column in a table. |
| Cursor-based | A record based on the cursor's SELECT statement | Each field corresponds to a column or expression in the cursor SELECT statement. |
| Programmer-defined | A record whose structure you, the programmer, get to define with a declaration statement. | Each field is defined explicitly (its name and data type) in the TYPE statement for that record; a field in a programmer-defined record can even be another record. |

## Benefits of Using Records

The record data structure provides a high-level way of addressing and manipulating program-based data. This approach offers the following benefits:

The following sections describe each of these benefits.

### *Data abstraction*

When you abstract something, you generalize it. You distance yourself from the nitty-gritty details and concentrate on the big picture. When you create modules, you abstract the individual actions of the module into a name. The name (and program specification) represents those actions.

When you create a record, you abstract all the different attributes or fields of the subject of that record. You establish a relationship between all those different attributes and you give that relationship a name by defining a record.

### *Aggregate operations*

Once you have stored information in records, you can perform operations on whole blocks of data at a time, rather than on each individual attribute. This kind of aggregate operation reinforces the abstraction of the record. Very often you are not really interested in making changes to individual components of a record, but instead to the object which represents all of those different components.

Suppose that in my job I need to work with companies, but I don't really care about whether a company has two lines of address information or three. I want to work at the level of the company itself, making changes to, deleting, or analyzing the status of a company. In all these cases I am talking about a whole row in the database, not any specific column. The company record hides all that information from me, yet makes it accessible when and if I need it. This orientation brings you closer to viewing your data as a collection of objects with rules applied to those objects.

### *Leaner, cleaner code*

Using records also helps you to write clearer code and less of it. When I use records, I invariably produce programs which have fewer lines of code, are less vulnerable to change, and need fewer comments. Records also cut down on variable sprawl; instead of declaring many individual variables, I declare a single record. This lack of clutter creates aesthetically attractive code which requires fewer resources to maintain.

## Guidelines for Using Records

Use of PL/SQL records can have a dramatic impact on your programs, both in initial development and in ongoing maintenance. To ensure that I personally get the most out of record structures, I have set the following guidelines for my development:

- Create corresponding cursors and records. Whenever I create a cursor in my programs, I also create a corresponding record (except in the case of cursor FOR loops). I always FETCH into a record, rather than into individual variables. In those few instances when it might involve a little extra work over simply fetching into a single variable, I marvel at the elegance of this approach and compliment myself on my commitment to principle.

- Create table-based records. Whenever I need to store table-based data within my programs, I create a new (or use a predefined) table-based record to store that data. I keep my variable use to a minimum and dynamically link my program data structures to my RDBMS data structures with the %ROWTYPE attribute.

- Pass records as parameters. Whenever appropriate, I pass records rather than individual variables as parameters in my procedural interfaces. This way, my procedure calls are less likely to change over time, making my code more stable. There is a downside to this technique, however: if a record is passed as an OUT or IN OUT parameter, its field values are saved by the PL/SQL program in case of the need for a rollback. This can use up memory and consume unnecessary CPU cycles.

## PL/SQL tables

A PL/SQL table is a one-dimensional, unbounded, sparse collection of homogeneous elements, indexed by integers. In technical terms, it is like an array; it is like a SQL table; yet it is not precisely the same as either of those data structures.

Like PL/SQL records, PL/SQL tables are composite data structures. Figure shows a PL/SQL table composed of a single column named emp_name, with names saved to rows 100, 225, 226, 300, and 340.

### *The single-column, one-dimensional PL/SQL table*

| Row # | Emp_Name |
|---|---|
| 100 | Smith |
| 225 | Feuerstein |
| 226 | Silva |
| 300 | Russell |
| 340 | Ziemba |

PL/SQL tables are available only in releases of PL/SQL Version 2. PL/SQL tables reside in the private PL/SQL area of the Oracle Server database instance; they are not available as client-side structures at this time. As a result, you cannot declare and manipulate PL/SQL tables in your Oracle Developer/2000 environment.

You can, on the other hand, build stored procedures and packages which work with PL/SQL tables, but hide these structures behind their interface. You can then call this stored code from within Oracle Developer/2000 to take advantage of Version 2 features like PL/SQL tables.

### *Nested Tables and VARRAYs*

Like PL/SQL tables, the two collection types introduced under Oracle8 -- nested tables and variable arrays (VARRAYs) -- can be used in PL/SQL programs. But these collection types offer something new: they can be used as the data types of fields in conventional tables and attributes of objects.

Both nested tables and VARRAYs can be used in PL/SQL and in the database (for example, as a column).

## Characteristics of PL/SQL Tables

A definition worth repeating: A PL/SQL table is a one-dimensional, unbounded, sparse collection of homogenous elements, indexed by integers.
Let's examine each of these characteristics in detail:

***One-dimensional***

A PL/SQL table can have only one column. It is, in this way, similar to a one-dimensional array. You cannot define a PL/SQL table so that it can be referenced as follows:

my_table (10, 44)

This is a two-dimensional structure and not currently supported.

***Unbounded or Unconstrained***

There is no predefined limit to the number of rows in a PL/SQL table. The PL/SQL table grows dynamically as you add more rows to the table. The PL/SQL table is, in this way, very different from an array.

Related to this definition, no rows for PL/SQL tables are allocated for this structure when it is defined

***Sparse***

In a PL/SQL table, a row exists in the table only when a value is assigned to that row. Rows do not have to be defined sequentially. Instead you can assign a value to any row in the table. So row 15 could have a value of `Fox' and row 15446 a value of `Red', with no other rows defined in between.

In contrast, an array is a *dense* data structure. When you declare an array, all cells in the array are allocated in memory and are ready to use.

***Homogeneous elements***

Because a PL/SQL table can have only a single column, all rows in a PL/SQL table contain values of the same data type. It is, therefore, homogeneous.

***Indexed by integers***

PL/SQL tables currently support a single indexing mode: by BINARY_INTEGER. This number acts as the "primary key" of the PL/SQL table. The range of a BINARY_INTEGER is from $-2^{31}-1$ to $2^{31}-1$, so you have an awful lot of rows with which to work.

Because the row number does not have to be used sequentially and has such enormous range, you can use this integer index in interesting ways. For example, the row number for a PL/SQL table could be the primary key of your company table, so that:

company_name (14055)

Contains the name of the company whose company_id = 14055.

**Restrictions in mind when you work with PL/SQL tables:**

- There is no concept of transaction integrity with PL/SQL tables. You cannot commit information to a PL/SQL table or roll back changes from the table.
- You cannot SELECT from PL/SQL tables. There is no way to perform set-at-a-time processing to retrieve data from a PL/SQL table. This is a programmatic construct in a programmatic language. Instead you can use PL/SQL loops to move through the contents of a PL/SQL table, one row at a time.
- You cannot issue DML statements (INSERTs, UPDATEs, and DELETEs) against PL/SQL tables (though PL/SQL Release 2.3 does offer a DELETE operator).

## Declaring a PL/SQL Table

As with a record, a PL/SQL table is declared in two stages:

- Define a particular PL/SQL table structure (made up of strings, dates, etc.) using the table TYPE statement. The result of this statement is a datatype you can use in declaration statements.
- Declare the actual table based on that table type. The declaration of a PL/SQL table is a specific instance of a generic data type

### *Defining the Table TYPE*

The TYPE statement for a PL/SQL table has the following format:

```
TYPE <table_type_name> IS TABLE OF <datatype> [ NOT NULL ]
 INDEX BY BINARY_INTEGER;
```

Where

- <table_type_name> is the name of the table structure you are creating and
- <datatype> is the datatype of the single column in the table. You can optionally specify that the table be NOT NULL, meaning that every row in the table that has been created must have a value.

You must always specify INDEX BY BINARY_INTEGER at the end of the TYPE...TABLE statement, even though it is the only type of index you can have currently in a PL/SQL table.

PL/SQL uses BINARY_INTEGER indexes because they allow for the fastest retrieval of data. (In this case, the primary key is already in the internal binary format, so it does not have to be converted before it can be used by the runtime environment.)

The rules for the table type name are the same as for any identifier in PL/SQL: the name can be up to 30 characters in length, it must start with a letter, and it can include some special characters such as underscore ( _ ) and dollar sign ($).

The datatype of the table type's column can be any of the following:

***Scalar datatype***

Any PL/SQL-supported scalar data type, such as VARCHAR2, POSITIVE, DATE, or BOOLEAN.

***Anchored data type***

A data type inferred from a column, previously defined variable, or cursor expression using the %TYPE attribute.

Here are some examples of table type declarations:

```
TYPE company_keys_tabtype IS TABLE OF company.company_id%TYPE NOT NULL INDEX BY
BINARY_INTEGER;
TYPE reports_requested_tabtype IS TABLE OF VARCHAR2 (100) INDEX BY BINARY_INTEGER;
```

**NOTE:** Prior to PL/SQL Release 2.3, you may not use composite data types declaring a PL/SQL table's column. With Release 2.3, you can create PL/SQL tables of records.

## Declaring the PL/SQL Table

Once you have created your table type, you can reference that table type to declare the actual table. The general format for a table declaration is:

```
<table_name> <table_type>;
```

Where <table_name> is the name of the table and <table_type> is the name of a previously declared table type. In the following example I create a general table type for primary keys in PACKAGE and then use that table type to create two tables of primary keys:

```
PACKAGE company_pkg
IS
   /* Create a generic table type for primary keys */
   TYPE primary_keys_tabtype IS TABLE OF NUMBER NOT NULL
      INDEX BY BINARY_INTEGER;
   /* declare two tables based on this table type */
   company_keys_tab primary_keys_tabtype;
   emp_keys_tab primary_keys_tabtype;
END company_pkg;
```

## Referencing and Modifying PL/SQL Table Rows

You refer to a particular row in a PL/SQL table by specifying the name of the table and the primary key value for that row. The syntax is very similar to standard, one-dimensional arrays:

```
<table_name> ( <primary_key_value> )
```

where <table_name> is the name of the table and <primary_key_value> is a literal, variable, or expression whose datatype is compatible with the BINARY_INTEGER datatype. You assign values to a row in a table with the standard assignment operator (:=).

All of the following table references and assignments are valid:

```
company_names_tab (15) := 'Fabricators Anonymous';
company_keys_tab (-2000) := new_company_id;
header_string := 'Sales for ' || company_names_tab (25);
```

## *Automatic Conversion of Row Number Expressions*

PL/SQL will go to great lengths to convert an expression or string to BINARY_INTEGER for use as a row number. Here are some examples:

- Store the string in row 16:

  requests_table (15.566) := 'Totals by Category';

- Store the string in row 255:

  company_keys_table ('25'||'5') := 1000;

- The expression will be evaluated and used as the row number:

  keyword_list_table (last_row + 15) := 'ELSE';

## *Filling the Rows of a PL/SQL Table*

You can assign values to rows of a table in several ways:

- Direct assignment
- Iterative assignment
- Aggregate assignment

These methods are described in the following sections.

### *Direct Assignment*

As shown in previous examples, you can simply assign a value to a row with the assignment operator:

```
countdown_test_list (43) := 'Internal pressure';
company_names_table (last_name_row) := 'Johnstone Clingers';
```

***Iterative Assignment***

In order to fill up multiple rows of a table, I recommend taking advantage of a PL/SQL loop. Within the loop you will still perform direct assignments to set the values of each row, but the primary key value will be set by the loop rather than hardcoded into the assignment itself.

Example

```
CREATE OR REPLACE PROCEDURE show_bizdays
   (start_date_in IN DATE := SYSDATE, ndays_in IN INTEGER := 30)
IS
   TYPE date_tabtype IS TABLE OF DATE INDEX BY BINARY_INTEGER;
   bizdays date_tabtype;
   /* The row in the table containing the nth_day */
   nth_day  BINARY_INTEGER := 1;
   v_date DATE := start_date_in;
BEGIN
   /* Loop through the calendar until enough biz days are found */
   WHILE nth_day <= ndays_in
   LOOP
      /* If the day is not on the weekend, add to the table. */
      IF TO_CHAR (v_date, 'DY') NOT IN ('SAT', 'SUN')
      THEN
         bizdays (nth_day) := v_date;
         DBMS_OUTPUT.PUT_LINE (v_date);
         nth_day := nth_day + 1;
      END IF;
     v_date := v_date + 1;
   END LOOP;
END show_bizdays;
/
```

As you can see from this example, using the WHILE loop produces a neat, sequential load of the PL/SQL table.

*Aggregate Assignment*

Just as you can assign one entire record to another record of the same type and structure, you can perform aggregate assignments with tables as well. In order to transfer the values of one table to another, the datatype of the two tables must be compatible. Beyond that you simply use the assignment operator (:=) to transfer the values of one table to the other. The following example contains an example of an aggregate table assignment:

```
DECLARE
   TYPE name_table IS TABLE OF VARCHAR2(100) INDEX BY BINARY_INTEGER;
   old_names name_table;
   new_names name_table;
BEGIN
   /* Assign values to old_names table */
   old_names(1) := 'Smith';
   old_names(2) := 'Harrison';
   /* Assign values to new_names table */
   new_names(111) := 'Hanrahan';
   new_names(342) := 'Blimey';
   /* Transfer values from new to old */
   old_names := new_names;
   /* This assignment will raise NO_DATA_FOUND */
   DBMS_OUTPUT.PUT_LINE (old_names (1));
END;
```

A table-level assignment completely replaces the previously defined rows in the table. In the preceding example, rows 1 and 2 in old_names are defined before the last, aggregate assignment.

After the assignment, only rows 111 and 342 in the old_names table have values.

## *Clearing the PL/SQL Table*

What happens when you are done with a PL/SQL table and want to remove it from memory? If a PL/SQL table is like a table, we should be able to DELETE the rows of that table or DROP it entirely, right? It's a nice idea, but you can't perform a SQL DELETE statement on a PL/SQL table because it is not stored in the database. You also cannot DROP a PL/SQL table.

You can set a single row to NULL with the following kind of assignment:

```
company_names_table (num_rows) := NULL;
```

But this assignment doesn't actually remove the row or make it undefined; it just sets the value of the row to NULL.

The only way to actually empty a PL/SQL table of all rows is to perform an aggregate assignment with a table that is empty -- a table, that is, with no rows defined.

With this approach, for every PL/SQL table you want to be able to empty, you declare a parallel, empty table of the same table type. When you are finished working with your table, simply assign the empty table to the actual table. This will unassign all the rows you have used. The following example demonstrates this technique:

```
DECLARE
   TYPE company_names_tabtype IS TABLE OF company.name%TYPE
      INDEX BY BINARY_INTEGER;
   company_names_tab company_names_tabtype;
   /* Here is the empty table declaration */
   empty_company_names_tab company_names_tabtype;
BEGIN
   ... set values in company names table ...
   /* The closest you can come to "dropping" a PL/SQL table */
   company_names_tab := empty_company_names_tab;

END;
```

## *Transferring Database Information to PL/SQL Tables*

You cannot use a SQL SELECT statement to transfer data directly from a database table to a PL/SQL table. You need to take a programmatic approach. A cursor FOR loop usually makes the most sense for this process, which requires the following steps:

1. Define a PL/SQL table TYPE for each datatype found in the columns of the database table.
2. Declare PL/SQL tables which will each receive the contents of a single column.
3. Declare the cursor against the database table.
4. Execute the FOR loop. The body of the loop will contain a distinct assignment of one column into one PL/SQL table.

Tech Mahindra

# Oracle Packages

## DBMS_LOB Package

By the end of this lesson we should learn

- Using Large Objects
- Working with LOBs
- Large Object Types
- Internal LOBs
- External LOBs
- Temporary LOBs

## The DBMS_LOB Package

We can use LONG data type to store character data up to 2GB in length per row. In place of LONG and LONG RAW, you can also use the LOB data types (BLOB, CLOB, NCLOB and BFILE) for storage of long data up to 4GB in length.

### What are CLOBs?

Basically, LOBs (Large Objects) are designed to support large unstructured data such as text, graphic images, still video clips, full motion video, and sound waveforms. A typical employee record may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger. Oracle supports the following two types of LOBs:

- Those stored in the database either in-line in the table or in a separate segment or tablespace, such as BLOB (Binary LOB), CLOB (Character LOB) and, NCLOB (National Character LOB). As the name signifies, BLOB holds binary data while the CLOB holds textual data and the NCLOB holds, character data that corresponds to the national character set defined for the Oracle database.

- Those stored as operating system files, such as BFILEs

CLOBs can store large amounts of character data and are useful for storing unstructured XML documents. Also useful for storing multimedia data, BFILEs which are external file references can also be used. In this case the XML is stored and managed outside the RDBMS, but can be used in queries on the server.

While LOBs provide the infrastructure in the database to store multimedia data, Oracle8*i* and Oracle9*i* also provide developers with additional functionality for the most commonly used multimedia types. The multimedia types include text, image, locator, audio, and video data.

## Why Not Use LONGs?

In Oracle7, most applications storing large amounts of unstructured data used the LONG or LONG RAW data type.

Oracle*8*i and Oracle9i's support for LOB data types is preferred over support for LONG and LONG RAWs in Oracle7 in the following ways:

***LOB Capacity:*** With Oracle8 and Oracle*8i*, LOBs can store up to 8TB of data. This is more than the 2GB of data that LONG and LONG RAW data types could store.

## The LOB Data type

Oracle*9i* regards LOBs as being of two kinds depending on their location with regard to the database -- **internal LOBs** and **external LOBs**, also referred to as **BFILEs** (binary files). Note that when we discuss some aspect of working with LOBs without specifying whether the LOB is internal or external, the characteristic under discussion pertains to both internal and external LOBs.

### *Internal LOBs*

*Internal* LOBs, as their name suggests, are stored inside database tablespaces in a way that optimizes space and provides efficient access. Internal LOBs use copy semantics and participate in the transactional model of the server. You can recover internal `LOB`s in the event of transaction or media failure, and any changes to a internal `LOB` value can be committed or rolled back. In other words, all the ACID properties that pertain to using database objects pertain to using internal LOBs

- ***Number of LOB columns in a table:*** An Oracle8, Oracle*8i,* or Oracle9*i* table can have multiple LOB columns. Each LOB column in the same table can be of a different type. In Oracle7 Release 7.3 and higher, tables are limited to a single LONG or LONG RAW column.
- ***Random piece-wise access:*** LOBs support random access to data, but LONGs support only sequential access.

## *Internal LOB Datatypes*

There are three SQL datatypes for defining instances of internal LOBs:

- **BLOB**, a LOB whose value is composed of unstructured binary ("raw") data.
- **CLOB**, a LOB whose value is composed of character data that corresponds to the database character set defined for the Oracle*9i* database.
- **NCLOB**, a LOB whose value is composed of character data that corresponds to the national character set defined for the Oracle*9i* database.

Internal LOBs are divided into **persistent** and **temporary** LOBs.

## *External LOBs (BFILEs)*

*External* LOBs (BFILES) are large binary data objects stored in operating system files outside database tablespaces. These files use reference semantics. Apart from conventional secondary storage devices such as hard disks, BFILEs may also be located on tertiary block storage devices such as CD-ROMs, PhotoCDs and DVDs.

The BFILE *datatype* allows *read-only* byte stream access to large files on the file system of the database server.

Oracle can access BFILEs provided the underlying server operating system supports stream-mode access to these operating system (OS) files.

## *External LOB Datatypes*

There is one datatype, BFILE, for declaring instances of external SQL LOBs.

- **BFILE**, a LOB whose value is composed of binary ("raw") data, and is stored outside the database tablespaces in a server-side operating system file.

## *Internal LOBs Use Copy Semantics, External LOBs Use Reference Semantics*

- Copy semantics: Both LOB locator and value are copied
- Reference semantics: Only LOB locator is copied

**Example:**

Assuming lob_table is a table and b_lob,key_value are the columns in the table

```
DBMS_LOG.APPEND(dest_lob IN OUT NOCOPY BLOB, src_lob IN BLOB);
```

**Example 2**

```
DBMS_LOG.APPEND(dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS, src_lob  IN CLOB
CHARACTER SET dest_lob%CHARSET);
```

```
CREATE OR REPLACE PROCEDURE Example_1b IS
    dest_lob  BLOB;
    src_lob BLOB;
BEGIN
  -- get the LOB locators
    SELECT b_lob INTO dest_lob FROM lob_table WHERE key_value = 12 FOR UPDATE;
    SELECT b_lob INTO src_lob FROM lob_table WHERE key_value = 12;
    DBMS_LOG.APPEND(dest_lob, src_lob);
    COMMIT;
END;
/
```

## The Datatypes

When talking about manipulating documents within a database, there are only a few choices for a datatype that can handle a large document. These large objects (LOBs) can use any one of the four datatypes depending on the characteristics of the object you are storing. These large objects can be in the form of text, graphics, video or audio.

| Datatype | Description |
|---|---|
| BLOB | Used to store unstructured binary data up to 4G. This datatype stores the full binary object in the database. |
| CLOB/NCLOB | Used to store up to 4G of character data. This datatype stores the full character data in the database. |
| BFILE | Used to point at large objects that are external to the database and in operating system files. The BFILE column also contains binary data and cannot be selected. |

## Benefits of LOBs

It use to be that the largest object you could store in the database was of the datatype LONG. Oracle has for the last few releases kept telling us to convert our LONG datatypes to a LOB datatype (maybe they will too). The reason for converting our LONGs to LOBs can be seen in this short list of benefits.

1. LOB columns can reach the size of 4G.
2. You can store LOB data internally within a table or externally.
3. You can perform random access to the LOB data.
4. It is easier to do transformations on LOB columns.
5. You can replicate the tables that contain LOB columns.

Now let us understand various predefined procedures to process LOBs in oracle

## *Append*

```
CREATE TABLE test(t1 CLOB, t2 CLOB, s NUMBER PRIMARY KEY);
INSERT INTO test VALUES('this is my source ' ,'this is my destination',1);
COMMIT;
```

***Program***

```
DECLARE
    b1 CLOB;
    b2 CLOB;
 BEGIN
    SELECT t1 INTO b1 FROM test FOR UPDATE WHERE s = 1;
    SELECT t2 INTO b2 FROM test;
    DBMS_LOB.APPEND(b1,b2);
 COMMIT;
 END;
```

## *SUBSTR*

The method DBMS_LOB.SUBSTR() can return only 4000 characters which is the limit of a VARCHAR in Oracle.

Is there any other way to fetch more than 4000 characters in a single stroke?

One way, we can read 4000 characters at a time and store them into a VARCHAR2 variable and finally combine them together.

This results in improper output because of the carriage returns and other non printable characters in the script.

Example:

```
SELECT s, DBMS_LOB.GETLENGTH(t1) len, DBMS_LOB.SUBSTR(t1,40,10) raw_data FROM test
WHERE s=1
```

To read all the inform ation at a time, We can a program using our C program logic as

Write a procedure to read character by character from CLOB column

```
CREATE OR REPLACE PROCEDURE print_clob( p_clob in CLOB)
AS
       l_offset NUMBER DEFAULT 1;
BAGIN
       DBMS_OUTPUT.NEW_LINE;
       LOOP
              EXIT WHEN l_offset > DBMS_LOB.GETLENGTH(p_clob);
              DBMS_OUTPUT.PUT( DBMS_LOB.SUBSTR( p_clob, 1, l_offset ) );
              DBMS_OUTPUT.NEW_LINE;
              l_offset := l_offset + 1;
       END LOOP;
END;
/
```

or

```
CREATE OR REPLACE PROCEDURE print_clob( p_clob IN CLOB )
AS
       l_offset NUMBER DEFAULT 1;
BEGIN
       LOOP
              EXIT WHEN l_offset > DBMS_LOB.GETLENGTH(p_clob);
              DBMS_OUTPUT.PUT_LINE( DBMS_LOB.SUBSTR( p_clob, 1, l_offset ) );
              l_offset := l_offset + 1;
       END LOOP;
END;
```

Write PLSQL code to execute above procedure

```
declare
   y clob;
begin
  select t1 into y from test;
  print_clob(y);
end;
/
```

## DBMS_LOB.READ

```
DECLARE
      v_amount NUMBER:=1000
      v_offset NUMBER:=10;
      v_buffer CLOB;
      clobvalue1 CLOB;
 BEGIN
    SELECT t1 INTO clobvalue1 FROM test WHERe s = 1;
    DBMS_LOB.READ(clobvalue1,v_amount,v_offset,v_buffer);
 DBMS_OUTPUT.PUT_LINE(v_buffer);
 END;
```

## COPY

```
DECLARE
    v_amount NUMBEr:=3000;
    v_offset NUMBER:=1;
    v_buffer CLOb;
    clobvalue1 CLOB;
    clobvalue2 CLOB;
 BEGIN
       SELECT t1 INTO clobvalue1 FROM test WHERE s = 3 FOR UPDATE;
       SELECT t2 INTO clobvalue2 FROM test WHERE s = 3;
       DBMS_LOB.READ(clobvalue1,v_amount,v_offset,v_buffer);
       DBMS_OUTPUT.PUT_LINE(v_buffer);
       DBMS_LOB.COPY (clobValue1,
              clobValue2,2000,DBMS_LOB.GETLENGTH(clobvalue1)+2,1);
       DBMS_OUTPUT.PUT_LINE(clobvalue2);
END;
/
```

## *ERASE Procedure*

This procedure erases an entire internal LOB or part of an internal LOB.

When data is erased from the middle of a LOB, zero-byte fillers or spaces are written for BLOBs or CLOBs respectively. The actual number of bytes or characters erased can differ from the number you specified in the amount parameter if the end of the LOB value is reached before erasing the specified number. The actual number of characters or bytes erased is returned in the amount parameter.

**Syntax1:**
```
DBMS_LOB.ERASE (
   lob_loc      IN OUT  NOCOPY  BLOB,
   amount       IN OUT  NOCOPY  INTEGER,
   offset       IN              INTEGER := 1);
```

**Syntax2:**
```
DBMS_LOB.ERASE (
   lob_loc      IN OUT  NOCOPY  CLOB CHARACTER SET ANY_CS,
   amount       IN OUT  NOCOPY  INTEGER,
   offset       IN              INTEGER := 1);
```

***Parameters***

| Parameter | Description |
|---|---|
| lob_loc | Locator for the LOB to be erased. |
| amount | Number of bytes (for BLOBs or BFILES) or characters (for CLOBs or NCLOBs) to be erased. |
| Offset | Absolute offset (origin: 1) from the beginning of the LOB in bytes (for BLOBs) or characters (CLOBs). |

***Exceptions***

| Exception | Description |
|---|---|
| VALUE_ERROR | Any input parameter is NULL. |
| INVALID_ARGVAL | Either:<br>- amount < 1 or amount > LOBMAXSIZE<br>- offset < 1 or offset > LOBMAXSIZE |

***Usage Notes***

It is not mandatory that you wrap the LOB operation inside the Open/Close APIs. If you did not open the LOB before performing the operation, the functional and domain indexes on the LOB column are updated during the call. However, if you opened the LOB before performing the operation, you must close it before you commit or rollback the transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap the LOB operation inside the Open/Close API, the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. Therefore, it is recommended that you enclose write operations to the LOB within the `OPEN` or `CLOSE` statement.

***Example***

```
CREATE OR REPLACE PROCEDURE Example_4
IS
    lobd        BLOB;
    amt         INTEGER := 3000;
BEGIN
    SELECT b_col INTO lobd FROM lob_table WHERE key_value = 12 FOR UPDATE;
    DBMS_LOG.ERASE(dest_lob, amt, 2000);
    COMMIT;
END;
```

## *TRIM Procedure*

This procedure trims the value of the internal LOB to the length you specify in the newlen parameter. Specify the length in bytes for BLOBs, and specify the length in characters for CLOBs.

If you attempt to TRIM an empty LOB, then nothing occurs, and TRIM returns no error. If the new length that you specify in newlen is greater than the size of the LOB, then an exception is raised.

**Syntax1:** DBMS_LOB.TRIM(lob_loc IN OUT NOCOPY BLOB,
newlen IN INTEGER);
**Syntax2:** DBMS_LOB.TRIM(lob_loc IN OUT NOCOPY CLOB
CHARACTER SET ANY_CS, newlen IN INTEGER);

***Parameters***

| Parameter | Description |
|---|---|
| lob_loc | Locator for the internal LOB whose length is to be trimmed. |
| newlen | New, trimmed length of the LOB value in bytes for BLOBs or characters for CLOBs. |

***Exceptions***

| Exception | Description |
|---|---|
| VALUE_ERROR | lob_loc is NULL. |
| INVALID_ARGVAL | Either:<br>- new_len < 0<br>- new_len > LOBMAXSIZE |

***Usage Notes***

It is not mandatory that you wrap the LOB operation inside the Open/Close APIs. If you did not open the LOB before performing the operation, the functional and domain indexes on the LOB column are updated during the call. However, if you opened the LOB before performing the operation, you must close it before you commit or rollback the transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap the LOB operation inside the Open/Close API, the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. Therefore, it is recommended that you enclose write operations to the LOB within the `OPEN` or `CLOSE` statement.

***Example***

```
CREATE OR REPLACE PROCEDURE Example_15
IS
    lob_loc         BLOB;
BEGIN
     -- get the LOB locator
    SELECT b_col INTO lob_loc FROM lob_table WHERE key_value = 42 FOR UPDATE;
    DBMS_LOB.TRIM(lob_loc, 4000);
    COMMIT;
END;
```

## *WRITEAPPEND Procedure*

This procedure writes a specified amount of data to the end of an internal `LOB`. The data is written from the `buffer` parameter.

There is an error if the input amount is more than the data in the buffer. If the input amount is less than the data in the buffer, then only amount bytes or characters from the buffer are written to the end of the `LOB`.

**Syntax1:**
```
DBMS_LOB.WRITEAPPEND (
lob_loc IN OUT NOCOPY BLOB,
amount  IN        BINARY_INTEGER,
buffer  IN        RAW);
```
**Syntax2:**
```
DBMS_LOB.WRITEAPPEND (
lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
amount  IN        BINARY_INTEGER,
buffer  IN        VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

***Parameters***

| Parameter | Description |
|---|---|
| lob_loc | Locator for the internal `LOB` to be written to. |
| amount | Number of bytes (for `BLOBs`) or characters (for `CLOBs`) to write, or number that were written. |
| Buffer | Input buffer for the write. |

***Exceptions***

| Exception | Description |
|---|---|
| VALUE_ERROR | Any of `lob_loc`, `amount`, or `offset` parameters are `NULL`, out of range, or `INVALID`. |
| INVALID_ARGVAL | Either:<br>- `amount` < 1<br>- `amount` > `MAXBUFSIZE` |

***Usage Notes***

The form of the VARCHAR2 buffer must match the form of the CLOB parameter. In other words, if the input LOB parameter is of type NCLOB, then the buffer must contain NCHAR data. Conversely, if the input LOB parameter is of type CLOB, then the buffer must contain CHAR data.

When calling DBMS_LOB.WRITEAPPEND from the client (for example, in a BEGIN/END block from within SQL*Plus), the buffer must contain data in the client's character set. Oracle converts the client-side buffer to the server's character set before it writes the buffer data to the LOB.

It is not mandatory that you wrap the LOB operation inside the Open/Close APIs. If you did not open the LOB before performing the operation, the functional and domain indexes on the LOB column are updated during the call. However, if you opened the LOB before performing the operation, you must close it before you commit or rollback the transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

***Example***

```
CREATE OR REPLACE PROCEDURE Example_17 IS
    lob_loc    BLOB;
    buffer     RAW;
    amt        BINARY_INTEGER := 32767;
    i          INTEGER;
BEGIN
    SELECT b_col INTO lob_loc FROM lob_table WHERE key_value = 12 FOR UPDATE;
    FOR i IN 1..3 LOOP
        -- fill the buffer with data to be written to the lob
        DBMS_LOB.WRITEAPPEND(lob_loc, amt, buffer);
    END LOOP;
END;
```

## COMPARE

```
CREATE OR REPLACE PROCEDURE Example_2b IS
    fil_1, fil_2       BFILE;
    retval             INTEGER;
BEGIN
    SELECT f_lob INTO fil_1 FROM lob_table WHERE key_value = 45;
    SELECT f_lob INTO fil_2 FROM lob_table WHERE key_value = 54;
    DBMS_LOB.FILEOPEN(fil_1, dbms_lob.file_readonly);
    DBMS_LOB.FILEOPEN (fil_2, dbms_lob.file_readonly);
    retval := DBMS_LOB.COMPARE(fil_1, fil_2, 5600, 3348276, 2765612);
    IF (retval = 0)
    THEN
        ; -- process compared code
    ELSE
        ; -- process not compared code
    END IF;
    DBMS_LOB.FILECLOSE (fil_1);
    DBMS_LOB.FILECLOSE(fil_2);
END;
```

## *How to Convert Long data type to Lob*

It is relatively easy if you want to convert long datatype to lob data type. This section describes the following techniques for migrating existing tables from LONG to LOB datatypes:

A. Using ALTER TABLE to Convert LONG Columns to LOB Columns

B. Copying a LONG to a LOB Column Using the TO_LOB Operator

C. Online Redefinition of Tables with LONG Columns where high availability is critical

D. Using Oracle Data Pump to Migrate a Database when you can convert using this utility

Example to demonstrate these procedures.

### A) Using ALTER TABLE to Convert LONG Columns to LOB Columns

```
SQL> CREATE TABLE TEST_LONG_LOB(A NUMBER PRIMARY KEY, B LONG);
Table created.
SQL> INSERT INTO TEST_LONG_LOB VALUES (1,'This is the first entered row');
1 row created.
SQL> INSERT INTO TEST_LONG_LOB VALUES (2,'This is the second entered row');
1 row created.
SQL> COMMIT;
Commit complete.
SQL> ALTER TABLE TEST_LONG_LOB MODIFY B CLOB;
Table altered.
SQL> DESC TEST_LONG_LOB;
Name                    Null?        Type
-----------------------------------------------
A                       NOT NULL     NUMBER
B                                    CLOB
```

## B) Copying a LONG to a LOB Column Using the TO_LOB Operator

```
SQL> CREATE TABLE TEST_LONG_LOB(A NUMBER PRIMARY KEY, B LONG);
Table created.
SQL> INSERT INTO TEST_LONG_LOB VALUES (1,'This is the first entered row');
1 row created.
SQL> COMMIT;
Commit complete.
SQL> CREATE TABLE TEST_CLOB AS SELECT A, TO_LOB(B) B FROM TEST_LONG_LOB;
Table created.
```

After you ensure that the data is accurately copied, you can drop the original table and create a view or synonym for the new table using one of the following sequences:

```
SQL> DROP TABLE TEST_LONG_LOB;
Table dropped.
SQL> CREATE VIEW TEST_LONG_LOB AS SELECT * FROM TEST_CLOB;
View created.
or
SQL> DROP TABLE TEST_LONG_LOB;
Table dropped.
SQL> CREATE SYNONYM TEST_LONG_LOB FOR TEST_CLOB;
Synonym created.
or rename the table
SQL> RENAME TEST_CLOB TO TEST_LONG_LOB;
Table renamed.
SQL> DESC TEST_LONG_LOB;
Name              Null?    Type
----------------- -------- ----------------------------
A                          NUMBER
B                          CLOB
```

## C) Online Redefinition of Tables with LONG Columns where high availability is critical

1. This is the table that need to change LONG data.

```
SQL> CREATE TABLE TEST_LONG_LOB(A NUMBER PRIMARY KEY, B LONG);
Table created.
SQL> INSERT INTO TEST_LONG_LOB VALUES (1,'This is the first entered row');
1 row created.
SQL> COMMIT;
Commit complete.
```

2. Determine if the table is a candidate for online re-organization

```
DBMS_redefinition is a sys object
SQL> exec dbms_redefinition.can_redef_table('ARJU','TEST_LONG_LOB');
PL/SQL procedure successfully completed.
```

A primary key is mandatory since materialized views and logs are created during the start of redefinition.

3. Create an Interim Table.

```
SQL> CREATE TABLE TEST_LONG_LOB_INT(A NUMBER NOT NULL, B CLOB);
Table created.
```

Note that interim table has no primary key.

4. Start the re-organization process

```
SQL>declare
col_mapping varchar2(1000);
BEGIN
-- map all the columns in the interim table to the original table
col_mapping:='a a,'||'to_lob(b) b';
dbms_redefinition.start_redef_table('ARJU','TEST_LONG_LOB','TEST_LONG_LOB_INT',col_m
apping);
END;
/
PL/SQL procedure successfully completed.
Here Arju is username.
```

5. Run dbms_redefinition.copy_table_dependents

```
SQL>declare
error_count pls_integer := 0;
BEGIN
dbms_redefinition.copy_table_dependents('ARJU',
'TEST_LONG_LOB','TEST_LONG_LOB_INT',1, true, true, true, false,error_count);
dbms_output.put_line('errors := ' || to_char(error_count));
END;
/
PL/SQL procedure successfully completed.
```

6. Execute dbms_redefinition.finish_redef_table procedure.

```
EXEC DBMS_REDEFINITION.FINISH_REDEF_TABLE('ARJU',
'TEST_LONG_LOB','TEST_LONG_LOB_INT');
PL/SQL procedure successfully completed.
SQL> DROP TABLE TEST_LONG_LOB_INT;
Table dropped.
SQL> DESC TEST_LONG_LOB;
Name            Null?        Type
--------------- --------     ----------------------------
A               NOT NULL     NUMBER
B                            CLOB
```

## D) Using Oracle Data Pump to Migrate a Database when you can convert using this utility

If you are exporting data as part of a migration to a new database, create a table on the destination database with LOB columns and Data Pump will call the LONG-to-LOB function implicitly.

Serach within my blog about data pump export or import.

## DBMS_SQL Package – Dynamic SQL

This topic shows you how to use native dynamic SQL (dynamic SQL for short), a PL/SQL interface that makes your applications more flexible and versatile. You learn simple ways to write programs that can build and process SQL statements "on the fly" at run time.

Within PL/SQL, you can execute any kind of SQL statement (even data definition and data control statements) without resorting to cumbersome programmatic approaches. Dynamic SQL blends seamlessly into your programs, making them more efficient, readable, and concise.

### What Is Dynamic SQL?

Most PL/SQL programs do a specific, predictable job. For example, a stored procedure might accept an employee number and salary increase, then update the `sal` column in the `emp` table. In this case, the full text of the `UPDATE` statement is known at compile time. Such statements do not change from execution to execution. So, they are called *static* SQL statements.

However, some programs must build and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different `SELECT` statements for the various reports it generates. In this case, the full text of the statement is unknown until run time. Such statements can, and probably will, change from execution to execution. So, they are called *dynamic* SQL statements.

Dynamic SQL statements are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments. A *placeholder* is an undeclared identifier, so its name, to which you must prefix a colon, does not matter.

## Dynamic SQL

- It is an SQL statement that contains variables which can change at runtime.
- It is a SQL statement and is stored as a character string
- It enables to write general purpose code to be written.
- It Enables data definition, data-control to be written and executed from PL/SQL.
- It is written using either DBMS_SQL package or native dynamic SQL.
- We can use Dynamic SQL to create a procedure that operates on a table whose name is not known until runtime
- In Oracle 8, and earlier, we have to use DBMS_SQL to write dynamic SQL
- In Oracle 8i , we can use DBMS_SQL or native dynamic SQL .
- The EXECUTE IMMEDIATE statement can perform dynamic single row queries

## The Need for Dynamic SQL

You need dynamic SQL in the following situations:

- You want to execute a SQL data definition statement (such as `CREATE`), a data control statement (such as `GRANT`), or a session control statement (such as `ALTER SESSION`). In PL/SQL, such statements cannot be executed statically.
- You want more flexibility. For example, you might want to defer your choice of schema objects until run time. Or, you might want your program to build different search conditions for the `WHERE` clause of a `SELECT` statement. A more complex program might choose from various SQL operations, clauses, etc.

You use package `DBMS_SQL` to execute SQL statements dynamically, but you want better performance, something easier to use, or functionality that `DBMS_SQL` lacks such as support for objects and collections.

## Using the EXECUTE IMMEDIATE Statement

**The `EXECUTE IMMEDIATE` statement prepares (parses) and immediately executes a dynamic SQL statement or an anonymous PL/SQL block. The syntax is**

**Syntax:**
```
EXECUTE IMMEDIATE dynamic string
    [INTO {define_variable [, define_variable]... | record}]
    [USING [IN | OUT | IN OUT] bind_argument
    [, [IN | OUT | IN OUT] bind_argument]...]
    [{RETURNING | RETURN} INTO bind_argument[, bind_argument]...];
```

Where

- **Dynamic string** is a string expression that represents a SQL statement or PL/SQL block
- `define_variable` is a variable that stores a selected column value
- `record` is a user-defined or `%ROWTYPE` record that stores a selected row.
- input `bind_argument` is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.
- An output `bind_argument` is a variable that stores a value returned by the dynamic SQL statement or PL/SQL block.

Except for multi-row queries, the dynamic string can contain any SQL statement (*without* the terminator) or any PL/SQL block (with the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement.

Used only for single-row queries, the `INTO` clause specifies the variables or record into which column values are retrieved. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the `INTO` clause.

Used only for DML statements that have a `RETURNING` clause, the `RETURNING INTO` clause specifies the variables into which column values are returned. For each value returned by the DML statement, there must be a corresponding, type-compatible variable in the `RETURNING INTO` clause.

You can place all bind arguments in the `USING` clause. The default parameter mode is `IN`. For DML statements that have a `RETURNING` clause, you can place `OUT`

arguments in the RETURNING INTO clause without specifying the parameter mode, which, by definition, is OUT. If you use both the USING clause and the RETURNING INTO clause, the USING clause can contain only IN arguments.

At run time, bind arguments replace corresponding placeholders in the dynamic string. So, every placeholder must be associated with a bind argument in the USING clause and/or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL).

Dynamic SQL supports all the SQL datatypes. So, for example, define variables and bind arguments can be collections, LOBs, instances of an object type, and refs. As a rule, dynamic SQL does not support PL/SQL-specific types. So, for example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the INTO clause.

You can execute a dynamic SQL statement repeatedly using new values for the bind arguments. However, you incur some overhead because EXECUTE IMMEDIATE re-prepares the dynamic string before every execution.

Example

```
create or replace procedure del_all_rows(
                p_tab_name in varchar2, p_rows_del out number)
                is
                begin
                execute immediate 'delete from ' || p_tab_name;
                P_ROWS_DEL := SQL%ROWCOUNT;
               END;
```

Execution

```
Step1    :    SQL> variable    p_rows number
Step2    : SQL> execute del_all_rows('EMP',:p_rows);
Step3   :    SQL> print p_rows
```

Example 2

Procedure to update a column value in a table

```
create or replace procedure UPDATE_rows(
```

```
                p_tab_name in varchar2, COL1 IN VARCHAR2,
                VAL1 IN NUMBER,COL2 IN VARCHAR2,VAL2 IN NUMBER)
                is
                begin
        execute immediate 'UPDATE ' || P_TAB_NAME || ' SET ' || COL1 || ' = ' ||
VAL1 ||
        ' WHERE ' || COL2 || ' = ' ||   VAL2 ;
              END;
```

Example 3

Procedure to update a string value

```
create or replace procedure update_str(tname varchar2,col1 varchar2,val1 varchar2)
 is
 begin
    execute immediate ' update ' || tname || ' set ' || col1 || ' = ' || ''''||val1
|| '''' || ' where empno = 7521'
 end;
```

Execute

```
SQL> exec update_str('emp','ename','ravi');
```

Example 4

```
CREATE PROCEDURE delete_rows (
   table_name IN VARCHAR2,
   condition IN VARCHAR2 DEFAULT NULL) AS
   where_clause VARCHAR2(100) := ' WHERE ' || condition;
BEGIN
   IF condition IS NULL THEN where_clause := NULL; END IF;
   EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || where_clause;
EXCEPTION
   ...
END;
```

Example 5

The following PL/SQL block contains several examples of dynamic SQL:

```
DECLARE
   sql_stmt    VARCHAR2(200);
   plsql_block VARCHAR2(500);
```

```
   emp_id      NUMBER(4) := 7566;
   salary      NUMBER(7,2);
   dept_id     NUMBER(2) := 50;
   dept_name   VARCHAR2(14) := 'PERSONNEL';
   location    VARCHAR2(13) := 'DALLAS';
   emp_rec     emp%ROWTYPE;
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';
   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;
   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;
   plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
   EXECUTE IMMEDIATE plsql_block USING 7788, 500;
   sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
      RETURNING sal INTO :2';
   EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
      USING dept_id;
   EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
END;
/
```

## Backward Compatibility of the USING Clause

When a dynamic `INSERT`, `UPDATE`, or `DELETE` statement has a `RETURNING` clause, output bind arguments can go in the `RETURNING INTO` clause or the `USING` clause. In new applications, use the `RETURNING INTO` clause. In old applications, you can continue to use the `USING` clause. For example, both of the following `EXECUTE IMMEDIATE` statements are allowed:

```
Set serveroutput on
DECLARE
    sql_stmt VARCHAR2(200);
    my_empno NUMBER(4) := 7902;
    my_ename VARCHAR2(10);
    my_job   VARCHAR2(9);
    my_sal   NUMBER(7,2) := 3250.00;
 BEGIN
    sql_stmt := 'UPDATE emp SET sal = :1 WHERE empno = :2
       RETURNING ename, job INTO :3, :4';
    /* Bind returned values through USING clause. */
    EXECUTE IMMEDIATE sql_stmt
       USING my_sal, my_empno, OUT my_ename, OUT my_job;
```

```
    /* Bind returned values through RETURNING INTO clause. */
    EXECUTE IMMEDIATE sql_stmt
       USING my_sal, my_empno RETURNING INTO my_ename, my_job;
dbms_output.put_line(my_ename);
dbms_output.put_line(my_job);
 END;
OUTPUT
SQL> set serveroutput on
SQL> /
FORD
ANALYST
```

Test the above code by writing a procedure with empno,sal as IN parameters And my_ename and my_job as OUT parameters

## Specifying Parameter Modes

With the `USING` clause, you need not specify a parameter mode for input bind arguments because the mode defaults to `IN`. With the `RETURNING INTO` clause, you cannot specify a parameter mode for output bind arguments because, by definition, the mode is `OUT`. An example follows:

```
DECLARE
   sql_stmt VARCHAR2(200);
   dept_id  NUMBER(2) := 30;
   old_loc  VARCHAR2(13);
BEGIN
   sql_stmt :=
      'DELETE FROM dept WHERE deptno = :1 RETURNING loc INTO :2';
   EXECUTE IMMEDIATE sql_stmt USING dept_id RETURNING INTO old_loc;
Dbms_output.put_line('Location = ' || old_loc);
END;
```

When appropriate, you must specify the `OUT` or `IN OUT` mode for bind arguments passed as parameters. For example, suppose you want to call the following standalone procedure:

```
CREATE PROCEDURE create_dept (
   deptno IN OUT NUMBER,
   dname  IN VARCHAR2,
   loc    IN VARCHAR2) AS
BEGIN
   SELECT deptno_seq.NEXTVAL INTO deptno FROM dual;
```

```
   INSERT INTO dept VALUES (deptno, dname, loc);
END;
```

To call the procedure from a dynamic PL/SQL block, you must specify the `IN OUT` mode for the bind argument associated with formal parameter `deptno`, as follows:

```
DECLARE
   plsql_block VARCHAR2 (500);
   New_deptno NUMBER (2);
   new_dname  VARCHAR2 (14) := 'ADVERTISING';
   new_loc    VARCHAR2(13) := 'NEW YORK';
BEGIN
   plsql_block := 'BEGIN create_dept(:a, :b, :c); END;';
   EXECUTE IMMEDIATE plsql_block
      USING IN OUT new_deptno, new_dname, new_loc;
   IF new_deptno > 90 THEN ...
END;
```

Execute Immediate statement can also perform DDL operations

```
CREATE OR REPLACE PROCEDURE ddl_add_table(TNAME VARCHAR2) is
BEGIN
  EXECUTE IMMEDIATE
    'CREATE TABLE ' || TNAME ||
       '(EMPNO NUMBER(3) PRIMARY KEY,
        ENAME VARCHAR2(30),
        SAL NUMBER(8,2)
       )' ;
END;
/
```

Dynamic Single Row Query

```
CREATE OR REPLACE FUNCTION GET_COUNT(DESIG VARCHAR2)
RETURN NUMBER IS
Str varchar2(300);
Num_of_emp number;
BEGIN
   str := 'SELECT COUNT(*) FROM EMP WHERE JOB = :DESIG';
EXECUTE IMMEDIATE str INTO num_of_emp USING desig;
RETURN num_of_emp;
END;
/
```

## Dynamic Insert Statement

```
CREATE OR REPLACE PROCEDURE INSERT_INTO_TABLE(
     table_name VARCHAR2,
     empno NUMBER, Name VARCHAR2, saL number)
IS
    str VARCHAR2(200);
BEGIN
   str := 'INSERT INTO ' || table_name || ' values (:empno,
            :ename,:sal)';
EXECUTE IMMEDIATE str USING empno,ename,sal;
END;
/
```

## Psuedo Columns

### ROWID Pseudocolumn

Each table and nonjoined view has a pseudocolumn called ROWID. For example:

```
CREATE TABLE T_tab (col1 Rowid);
INSERT INTO T_tab SELECT Rowid FROM Emp_tab WHERE Empno = 7499;
```

This command returns the ROWID pseudocolumn of the row of the EMP_TAB table that satisfies the query, and inserts it into the T1 table.

### External Character ROWID

The extended ROWID pseudo column is returned to the client in the form of an 18-character string (for example, "AAAA8mAALAAAAQkAAA"), which represents a base 64 encoding of the components of the extended ROWID in a four-piece format, OOOOOOFFFBBBBBBRRR:

- OOOOOO: The **data object number** identifies the database segment (AAAA8m in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- FFF: The **datafile** that contains the row (file AAL in the example). File numbers are unique within a database.
- BBBBBB: The **data block** that contains the row (block AAAAQk in the example). Block numbers are relative to their datafile, *not* tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- RRR: The **row** in the block (row AAA in the example).

There is no need to decode the external ROWID; you can use the functions in the DBMS_ROWID package to obtain the individual components of the extended ROWID.