dreamtech PRESS

- An all-inclusive book to teach you everything about Oracle PL/SQL Programming
- Easy, Effective, and Reliable
- Quick and Easy learning in Simple Steps
- Most preferred choice worldwide for learning Oracle PL/SQL Programming

# Oracle PL/SQL Programming

## IN SIMPLE STEPS

Premier12

Easy to learn with hundreds of illustrations.

*Do it right, Do it fast!*

```
e 10.1.0.2.0 - Production on Fr

(c) 1982, 2004, Oracle.  All rights r

cted to:
ie Database 10g Enterprise Edition Release
h the Partitioning, OLAP and Data Mining opt

QL> SET SERVEROUTPUT ON
QL> CREATE OR REPLACE PROCEDURE salary (eno I
  2   e_sal NUMBER;
  3  BEGIN
```

ISBN: 10-81-7722-855-2

13-978-81-7722-855-7

Edition: 2008

Printed at: Printman India, Patparganj, Delhi.

# CONTENTS

Database management is very crucial and tedious work for every organization as databases contains all important information's of an organization. Oracle is a database used in almost every organization for managing data. Oracle is a flexible database that eases the problem of managing data stored within your system or on Oracle server. Now, the question arises that how Oracle manages the data. The answer for this question is `PL/SQL`, which is used for managing the database. PL/SQL is a very efficient and easier to learn database programming language designed to manage database. Each Oracle version comes with its corresponding version of PL/SQL. PL/SQL is a procedural database programming language that extends the functionality of Structured Query Language (SQL) and uses them with in its procedural statements. PL/SQL uses SQL DDL and DML statements to perform operations such as creating, altering, deleting the database. These are the basic operations required to manage a database. PL/SQL executes SQL statements within PL/SQL block, which is the basic unit of PL/SQL. PL/SQL is a very flexible database programming language that supports the advanced procedural programming concepts and elements, such as:

- Support for all SQL datatypes along with its PL/SQL defined datatypes. Use of control statements (`IF-ELSE-THEN`), iterative structures (`FOR-LOOP`, `WHILE-LOOP`).
- Support for subprograms such as procedures and functions, triggers, cursors.
- Exception handling that lets you create bug free program and helps in managing errors such as data not found.

SQL can issue only a single statement at a time, which will become more time consuming when need to execute many SQL statements and leads in low database performance but PL/SQL allows you to send multiple SQL statement to database simultaneously and thus reduces the overhead of accessing the database for every single SQL statement. In this way, there are lots of advantages of using Oracle's PL/SQL, to know those let's start working with PL/SQL.

To start working with PL/SQL, you must need to know its need, advantages, architecture, and features. This is all that we cover in this chapter.

## What is PL/SQL?

PL/SQL is procedural language, which is available with Oracle. PL/SQL is not a standalone language because it works with Oracle. It is an extension to the SQL (Structured query language, a database language used to perform various functions such as querying and updating data, on a database). PL in PL/SQL stands for procedural language. PL/SQL extends the functionality of SQL and then combines it with procedural functionality such as loops, procedures, cursors to provide better and more satisfactory result than SQL.

Now, the question arises that if we already have SQL (a flexible and easy to use query language) then what is the need to extend the functionality of SQL by PL/SQL? You must be curious to know that why we need PL/SQL and what are the reasons behind its success and in making it a ubiquitous database programming language. Now, we will depict the need to develop PL/SQL.

### *Need of PL/SQL*

As we know SQL is a very easy and more convenient database query language but besides that it has some limitations that become the need for developing PL/SQL. In SQL, we have to execute a single statement at a time. So, if we need to execute multiple statements then Oracle database

must be called several times to execute all the issued statements that reduce the database performance. To improve the database performance, PL/SQL is developed. There was a problem with the security of database as the code is executed on client-side rather than server. We were not able to handle exceptions that lead in sudden termination of program at runtime.

To overcome all these problems, PL/SQL has been developed. PL/SQL has lots of advantages that have made it a very successful and omnipresent database programming language that is in wide use with Oracle. Here are given the advantages of PL/SQL:

- Easily adaptable and SQL supporting
- Enhanced Performance
- Portability
- Security

We will study all these advantages of PL/SQL in the sequence. Let's start studying all these advantages in detail.

## Easily adaptable and SQL supporting

PL/SQL supports the entire characteristic and statements available in SQL such as select, insert, update, delete. Database can easily be created and manipulated by incorporating the SQL statements in PL/SQL block.

PL/SQL supports all datatypes that are supported by SQL so you do not have to convert SQL datatypes in PL/SQL datatypes. PL/SQL also allows you to use SQL operators, functions, and so on.

PL/SQL is easy to learn and understand as most of the characteristics are extended from SQL and the syntax used in PL/SQL is very simple to learn as it uses lots of keywords that clearly express the purpose of your code. Being familiar with any of the programming language such as c, c++ makes it easier to memorize and use the PL/SQL programming syntax.

## Enhanced Performance

PL/SQL provides better performance than SQL because one SQL statement can be processed at a time by database, which means if you have to execute more than one SQL statement than you have to access database as much as SQL statement you need to execute. Accessing the database several times results in the congestion in network as database are generally stored on server and thus the time to process SQL statement will automatically increased, which leads in low performance.

In PL/SQL, more than one SQL statement can be sent to Oracle database for processing because in PL/SQL, you can collect all the statements in PL/SQL block which is the basic structure of PL/SQL and all the programs in PL/SQL can not be created without PL/SQL block or you can also use PL/SQL subprograms to collect multiple SQL statements. A subprogram is a PL/SQL block that consists the sequence of statements to perform some specific task. A subprogram can be called by other PL/SQL programs. Thus, Oracle database will be accessed only once and there will be no congestion in network, which results in the better performance. See the Fig.PL/SQL-1.1 to understand the working without using PL/SQL and with using PL/SQL.

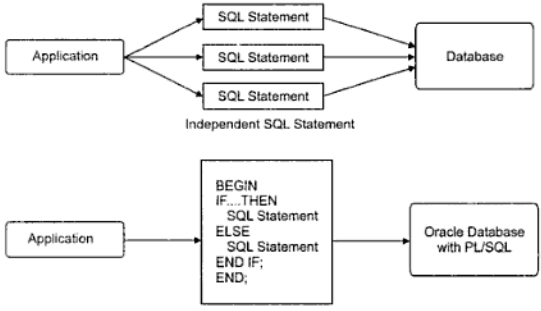



**Fig.PL/SQL-1.1**

In Fig.PL/SQL1.1, it is clearly depicted that before the use of PL/SQL, an application has to communicate with the database with several independent SQL statements that ultimately leads in overhead on the database and reduces its performance, while with the use of PL/SQL, you are passing several SQL statements within single PL/SQL blocks to database at same time and which SQL statements has to be executed depends on the conditions provided within PL/SQL block. In this way, PL/SQL in Oracle provides better performance.

### Portability

PL/SQL is a portable language; that is, programs created in the Oracle environment can also be executed on any operating system that supports Oracle. PL/SQL programs follows Write once and run everywhere (within Oracle environment) slogan.

### Security

PL/SQL provides higher security than SQL. PL/SQL stored procedures places the application code centrally on server rather than placing on client system. By placing the application code centrally, you can hide the code from other users. You can set the permission to access PL/SQL procedure by various users for performing different activities such as updating, deleting a table or data within a table.

These are some advantages of PL/SQL due to which PL/SQL is a ubiquitous database programming language. Let's have an overview of the various PL/SQL versions to know the enhancement made in PL/SQL versions released with every new Oracle version.

## *Versions of PL/SQL*

Till today various versions of PL/SQL has been released along with the versions of Oracle. Every new version of Oracle comes with its own version of PL/SQL and every new version comes with additional characteristics from previous version to provide you better functionality. In this book, we are considering the Oracle 10g and the PL/SQL version that is 10.0 that has been released with it. Table 1.1 depicts all the versions available till the release of Oracle 10g.

Now, we have discussed that how Oracle's PL/SQL has been improving to provide better performance.

Let's study the features of PL/SQL to know all that PL/SQL consists and how it helps in making database programming easy and efficient. Here, we study these features in brief because all features will be discussed in detail further in this book. We also discuss the new features included in the PL/SQL for Oracle 10g.

## Features of PL/SQL

Many of the key features in Oracle 10g PL/SQL are same as the previous versions (PL/SQL 9.2). Still this version has some new improvements due to which the database performance is more enhanced. PL/SQL has changed the way of database programming and becomes the omnipresent database programming language.

PL/SQL allows using SQL statements within PL/SQL block. Program flow can be controlled and multiple SQL statements can be sent to database at the same time. PL/SQL subprograms (procedures and functions) can be used to make the program easier.

We will discuss all features in details in forthcoming chapters but here we can have an overview of all the main features of PL/SQL along with the new features added in this version of PL/SQL in Oracle 10g. Here we discuss the following main features, which are then followed by the new features added in this version:

- PL/SQL Block
- PL/SQL Variables and constants
- PL/SQL control structures
- Using SQL within PL/SQL
- PL/SQL collections
- PL/SQL Records
- PL/SQL Subprograms
- PL/SQL packages
- Exception Handling

Here, we start discussing all these features.

### *PL/SQL Block*

A PL/SQL program consists minimum one PL/SQL block. It is not possible to create a PL/SQL program without PL/SQL block that's why PL/SQL block is very important and basic unit of PL/SQL. A PL/SQL block has different parts and that parts are Declarative part where you declare items such as types, variables; it is optional part, Executed part where you write procedural and SQL statements and is necessary part of PL/SQL block; without it PL/SQL block is incomplete, and final part is Exception handling where you provide the code to handle exceptions; it is also an optional part.

A PL/SQL block has four keywords, `DECLARE`, `BEGIN`, `EXCEPTION`, and `END`. `DECLARE` keyword is used to break the PL/SQL block in declarative part, `BEGIN` is used to define the Execution part, `EXCEPTION` keyword is used to start the exception handling, and `END` keyword

is used to end the PL/SQL block. See the following snippet to understand the PL/SQL block structure:

### Variables and Constants

PL/SQL allows declaring variables and constants so that they can be used in execution part. Variables and constants must have to be declared before using in statements (Execution part). Variables and constants can have any SQL (such as `NUMBER`, `CHAR`, `VARCHAR2`) or PL/SQL datatype (`BOOLEAN`). The only difference between the declaration of variables and constants is that while declaring constants, you need to use `CONSTANT` keyword and value should be assigned at the time of declaration. In case of variables, values can be assigned in execution part.

### Control Structures

It is a very important feature of PL/SQL that lets you allow adding constraints on statements to send multiple statements to database at the same time rather than sending the several independent statements as we ever did in SQL. PL/SQL provides the following control structures:

- **Conditional controls**: This control allows you to execute several statements based on different conditions. You can check those conditions with the help of `IF–THEN–ELSE` statement to execute the proper statement. IF clause checks the condition and if the condition is true then the statement written under `THEN` clause will be executed otherwise the control transfers to `ELSE` clause and that will execute. In this way, conditional control structure works.
- **Iterative controls**: Sometimes you need to execute a sequence of statements multiple times, which was not possible with SQL but PL/SQL provides you iterative controls to complete your requirement. PL/SQL provides you several loops such as `LOOP---END LOOP`, `FOR LOOP`, `WHILE LOOP`. All these iterative structures will be explained in Chapter-4.
- **Sequential control:** This control allows you to transfer the control from one part of program to another. PL/SQL uses `GOTO` keyword to support sequential control.

### within PL/SQL

has extended all the features of SQL. It allows using all SQL dataypes, operators, and so on. Using PL/SQL, all the DDL and DML statements can be executed from within PL/SQL block.

## PL/SQL Collections

Like other programming languages such as c, c++; PL/SQL also allows you to group elements of similar datatypes. In c, c++, group of elements of same datatype is done with the use of arrays, hash table and so on, but here arrays are called as varrays and hash table are known as associative arrays. If you are familiar with any other programming language such as c, c++ then learning PL/SQL collections become very easy but if you are not familiar with any other programming language and it is the first language you are studying then also have no need to worry because PL/SQL is an easy language to understand. The use of PL/SQL collections will be depicted in the Chapter-6 of this book.

## PL/SQL Records

Records are like data structures used in programming languages such as c, c++. As data structures are used to group various elements of different datatypes, in the same way records are used. The only difference between collections and records is that using collections you can group element of similar datatype and using records dissimilar datatypes can be grouped. For example, if you want to collect the information of an employee such as his/her name, age, address, then you can do this by using a single record but if you use collection then you have to make various collection (such as varray) to group the elements of similar datatype. The use of PL/SQL records will be depicted in the Chapter-6 of this book.

## PL/SQL Subprograms

PL/SQL also allows using procedures and functions as other programming languages. In PL/SQL, procedures and functions are collectively known as subprograms. A PL/SQL subprogram is similar to PL/SQL block (anonymous block), except that a subprogram must have some name so that it can be invoked anywhere in the program or in any other application.

Subprograms are very important in large application as it breaks down the big and complex application into easily manageable modules.

Subprograms have promoted the concept of reusability. The same subprogram can be used more than one time in various applications. Once tested successfully then it can be used directly in any application without wasting time on writing it again and again for a new application.

PL/SQL procedures and functions are structurally same but have only one difference in which a function has `RETURN` clause. Subprograms will explain in detail in Chapter-8 of this book.

## PL/SQL Packages

A PL/SQL package is known as database object, which is used to bundle logically related PL/SQL procedures and, functions. PL/SQL comes with lots of predefined packages such as `DBMS_ALERT`, `DBMS_OUTPUT` that can be used directly in your application or new packages with your own specifications which can also be created.

A PL/SQL package has two parts, specification and body. Specification part can be created with the use of `CREATE PACKAGE` SQL statement and use to declare constants, variables, procedures, functions, cursors and exceptions in a package.

Package body is created with the use of CREATE PACKAGE BODY, SQL statement and package body contains procedural and SQL statements. Basic structure to create PL/SQL package is given

This is just an overview of packages in PL/SQL. Packages will be explained in detail in Chapter-9 of this book.

Exceptions are runtime errors that interrupt the execution of PL/SQL program. There are many such as program is not properly designed, dividing any number by zero for occurrence of an exception. To avoid the program interruption, PL/SQL provides the mechanism to handle the exception. To handle exceptions, enclose the program code within BEGIN and END clause with exception handler in PL/SQL block. EXCEPTION keyword is used to start the exception handling in PL/SQL block.

PL/SQL also allows you to create your own exceptions along with handling the predefined exceptions (this type of exception generally occur when any number is tried to be divided by zero, stack overflow and so on). ZERO_DIVIDE exception will occur when you try to divide any number by zero.

To create your own exceptions, RAISE statement needs to be used. How to handle exceptions and how to generate your own exception will be explained in Chapter-11 of this book.

With the end of preceding given feature, you must understand that why PL/SQL is so popular, now we continue with the new features added in PL/SQL for Oracle 10g.

### *Features added in PL/SQL for Oracle 10g*

features added in this version of PL/SQL have made it more convenient, easier, and more automatic to use. New features added in this version are as follows:

- **Improved Performance:** This version of PL/SQL comes with more automated features such as reuse the expression code, introduction of new datatypes for processing scientific operations which has involved in improving its performance.
- **Support for FORALL statement:** FORALL statement lets you allow to process DML statements more efficiently by iterating over non-consecutive indexes. PL/SQL has INDICES OF and VALUE OF clauses to iterate over non-consecutive indexes. FORALL statement uses specified index in a collection to iterate and process the statements.

- **Introducing BINARY_FLOAT and BINARY_DOUBLE floating point datatypes:** These are two new introductions in the datatypes. These datatypes support IEEE 754 format and are of floating point type. These are used for intensive scientific computation where floating datatypes are used in calculation.
- **Enhanced Overloading:** Overloading has improved in this version. Subprograms that have different numeric datatypes as parameter can be overloaded in this version.
- **Improved Nested Tables:** Now you have more enhanced Nested tables. Using this version, nested tables can be compared for equality such as you can check that whether a nested table is a subset of another nested table or not, you can check whether a particular element is a member of a specific nested table or not.
- **Compile Time warnings:** These features helps in making a PL/SQL program more robust and well functioning. By using this feature, Oracle issues warnings during PL/SQL program compilation when found any problem such as passing char value to float column in `INSERT` statement. You can use `PLSQL_WARNINGS` initialization parameter and `DBMS_WARNING` package to manage compile time warnings.
- **Implicit conversion between CLOB and NCLOB:** Before introduction of this feature, user have to use `TO_CLOB` and `TO_NCLOB` to convert `CLOB` into `NCLOB` and `NCLOB` into `CLOB` but now user do not have to convert them explicitly because in this version, Oracle makes implicit conversion between them wherever required.
- **Flashback Query Functions:** This feature is used to know the timestamp associated with a particular `SCN` (System Change Number) and also `SCN` at particular moment of time and for this purpose, you have `SCN_TO_TIMESTAMP` and `TIMESTAMP_TO_SCN` functions to use. `SCN_TO_TIMESTAMP` function takes `SCN` number as parameter and returns the timestamp associated with that `SCN` while `TIMESTAMP_TO_SCN` function takes time value as parameter and returns the `SCN` at that moment.

Here, we have completed discussion on PL/SQL features and the new features added in the PL/SQL for Oracle 10g.

Let's study the basic working structure of PL/SQL in Oracle. We are talking about the architecture of PL/SQL in Oracle. It is necessary to understand the architecture because if you do not know that how PL/SQL process the statements included in PL/SQL block, it becomes difficult to understand the PL/SQL programming.

## PL/SQL Architecture in Oracle

Architecture of PL/SQL represents its basic working in coordination with Oracle. A PL/SQL block or subprogram consists of both procedural statements and SQL statements. PL/SQL architecture describes the process that how PL/SQL in oracle interprets a PL/SQL block or subprograms.

PL/SQL architecture in Oracle 10g consists of PL/SQL block or subprogram, PL/SQL engine, and Oracle server. PL/SQL engine compiles and executes the PL/SQL block or subprogram. Oracle database or application development tools such as Oracle forms, Oracle reports contains PL/SQL engine. PL/SQL engine contains PL/SQL procedural statement executor to execute procedural statements within a PL/SQL block or subprogram and Oracle server contains SQL statement executor to execute SQL statements. PL/SQL engine and Oracle server work in co-ordination to process a PL/SQL block or subprogram. Fig.PL/SQL-1.2 depicts that how PL/SQL engine and Oracle server work together and process a PL/SQL block or subprogram.

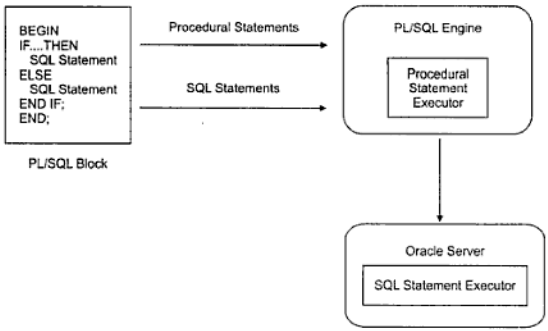



**Fig.PL/SQL-1.2**

PL/SQL engine accepts procedural statements and SQL statements as input. PL/SQL engine then process all procedural statements through procedural statement executor and send the SQL statements to Oracle database server to process. In this way, a PL/SQL block or subprogram is processed.

With this, we have completed discussion on PL/SQL Architecture and the way a PL/SQL block or subprogram processed.

With the end of discussion on PL/SQL architecture, we finish this chapter. By now, you must have enough idea about PL/SQL such as the requirement of PL/SQL and its architecture. Before closing this chapter, let's have a glance on summary.

## Summary

In this chapter, we have studied about:

- The introduction of PL/SQL
- The Need and advantages of PL/SQL
- The various versions of PL/SQL
- Features of the PL/SQL
- The PL/SQL Architecture

In the world of computerization, when we talk about programs then we must also take into consideration concepts, such as the program structure and datatype. A program may contain various variables to store data and those variables must be assigned with some specific datatypes. Thus, datatypes can be defined as a format to store data. PL/SQL provides various datatypes, such as `INTEGER`, `FLOAT` that help Oracle to choose a storage format for internal representation of objects and impose a range of values upon those objects. A PL/SQL developer should have knowledge of the predefined datatypes associated with PL/SQL so that one can choose appropriate datatype for a variable used in an application. For example, you can choose PL/SQL data type `VARCHAR2` for name of a human being. Further knowledge of the basic program structure of PL/SQL also helps us in partitioning the application into easily manageable sections. This partitioning also helps us to catch and fix flaws in different parts of the program.

We begin this chapter with a discussion on the PL/SQL block structure moving on to further concepts, such as datatypes, lexical units, and operators whose prior knowledge is essential before beginning with PL/SQL.

## Describing Block Structure

PL/SQL is also known as a block-structured language as a simple PL/SQL program contains many logical blocks where each block solves a part of the problem. For example, a PL/SQL program may use any function inside itself to perform some operation and this function then helps in solving some part of the problem, which is to be addressed by the same PL/SQL program.

To understand the functioning of a PL/SQL program, we divide its study into blocks and study each block separately. Every program consists of one or more blocks, where each block contains related declarations and statements. A PL/SQL block can consist of the following sections in this sequence:

- Block Header
- Declaration Section
- Execution Section
- Exception Section

### *Block Header*

The block header section contains the name of the block. It is optional and used only when there is a need to assign any name to a PL/SQL program. Name assigned to a PL/SQL program helps in calling it in other PL/SQL programs. The PL/SQL block header in PL/SQL block structure is shown in Fig.PL/SQL-2.1.

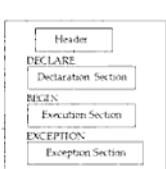



**Fig.PL/SQL-2.1**

After the block header, we need to declare PL/SQL variables (which are further used in a PL/SQL execution section) in the declaration section.

## Declaration Section

We now need to put information in a program block about the variables used in the block, such as associating datatypes with variables, initializing variables. This objective is achieved using the declaration section. However, this section is not compulsory. It is used in a block only when we need to declare variables. If a situation arises where we do not need variables, this section can be avoided.

The declaration section starts with the `DECLARE` keyword. After the `DECLARE` keyword, declarations of variables are provided.

The syntax for writing this block is as follows:

```
DECLARE
Variable name<space> Datatype;
```

The execution section follows the declaration section.

## Execution Section

Execution section follows the declaration section in the conventional program block. Execution section is the most important section of the PL/SQL block structure as it is responsible for the actual execution of the code. In other words, we can say that it is the functional part of the PL/SQL block.

This section starts with the `BEGIN` keyword and ends with the `END` keyword. Between `BEGIN` and `END` keywords, we write the set of instructions or code that we want to execute using PL/SQL.

It is however not necessary that the code we write in this section must produce an action. The code gets compiled even if the instructions given in the execution section do not signify an action.

All DML (Data Manipulation Language) and DDL (Data Definition Language) commands may be used in this section. Listing 2.1 contains an example of the execution section of the PL/SQL block.

**Listing 2.1:** Example of PL/SQL block containing execution section

Listing 2.1 shows the code to retrieve the system date into a variable (`v_date_time`) of type `DATE`.

To retrieve the date, we use `SYSDATE` (In-Built PL/SQL function).

To display the output, we use `PUT_LINE` procedure of `DBMS_OUTPUT` (In-Built PL/SQL package).

To display the result, we use `SET SERVEROUTPUT ON` because by default SQL *Plus does not read what a PL/SQL program has written with `DBMS_OUTPUT` package. We discuss more about these concepts further in the book.

Listing 2.1 can be executed in the following two ways:

- Using SQL*Plus
- Using iSQL*Plus

Let's explain each way in detail to execute a simple PL/SQL program.

### Using SQL*Plus

It is used to execute the PL/SQL programs and statements on a single user mode, that is Oracle PL/SQL installed on a system cannot be shared by multiple users.

Let's see how to execute Listing 2.1 in SQL Plus. To execute the listing, follow the steps given here:

1. *Click* **start→All Programs→Oracle-OraDb10g_home1→Application Development→SQL Plus**, as shown in Fig.PL/SQL-2.2.



**Fig.PL/SQL-2.2**

The Log On window appears, as shown in Fig.PL/SQL-2.3.

2. In the **Log On** window, enter the user as **scott** in the **User Name** text box, as shown in Fig.PL/SQL-2.3.
3. Now, enter the password as **tiger** in **Password** text box (Fig.PL/SQL-2.3).
4. Then, enter the host string in the **Host** text box, as shown in Fig.PL/SQL-2.3.
5. Now, *click* the **OK** button -2.3). The Oracle SQL *Plus window appears, as shown in Fig.PL/SQL-2.4.

6. Now, write the code shown in 2.1 in the Oracle SQL *Plus window, as shown in Fig.PL/SQL-2.4.

```
: Release 10.1.0.2.0 -          on Tue Jan 29 14:31:12 2008
  (c) 1982, 2004,          All rights reserved.

   to:
  Database 10g Enterprise     Release 10.1.0.2.0 - Production
the Partitioning, OLAP and    Mining options

SET SERVEROUTPUT ON
DECLARE
2  v_date_time DATE;
3  BEGIN
4  SELECT          INTO          FROM dual;
5
6  END;
7  /
```

7. Now, press the *ENTER* key to the code. As soon as you press the **ENTER** key, the output will be displayed as shown in PL/SQL-2.5.

```
procedure                completed.
```

This is the process to execute the PL/SQL programs and statements in SQL *Plus.

### Using iSQL*Plus Console

It is used to execute the PL/SQL programs and SQL statements on networked systems. Multiple users in the same network can access Oracle PL/SQL installed on a system by specifying the network path (For example, `http://localhost:5560/isqlplus/`).

Let's follow the steps given here to execute Listing 2.1 in iSQL*Plus console:

1. *Enter* the URL (for example, `http://localhost:5560/isqlplus`) in the **Address** bar of your web browser (Fig.PL/SQL-2.6) to open the **Oracle iSQL*Plus** login console (Fig.PL/SQL-2.7).


**Fig.PL/SQL-2.6**

2. *Click* the **Go** button, as shown in Fig.PL/SQL-2.6. The iSQL*Plus login console appears (Fig.PL/SQL-2.7).
3. In the iSQL*Plus login console, enter the user name as **scott** in the **Username** text field, as shown in Fig.PL/SQL-2.7.
4. Then, enter the password in the **Password** text field (Fig.PL/SQL-2.7). In our case, we have used the default password that is tiger.
5. *Enter* the host string that is orcl in **Connect Identifier** text field, as shown in Fig.PL/SQL-2.7.
6. Now, *click* the **Login** button (Fig.PL/SQL-2.7). The Oracle iSQL*Plus workspace window appears, as shown in Fig.PL/SQL-2.8.

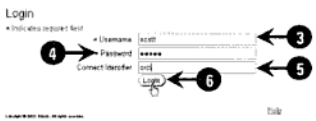



**Fig.PL/SQL-2.7**

7. Write the code written in Listing 2.1 in the workspace window and press the **Execute** button, as shown in Fig.PL/SQL-2.8.


**Fig.PL/SQL-2.8**

The Oracle iSQL*Plus workspace appears once again to display the output, as shown in Fig-PL/SQL-2.9.

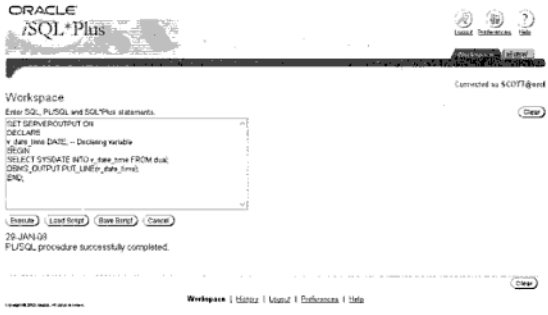



**Fig.PL/SQL-2.9**

This is the process to execute the PL/SQL programs using iSQL*Plus console. Now, if you want to execute some more programs then *click* the **Clear** button, as shown in Fig.PL/SQL-2.10.

This will clear the workspace window (Fig.PL/SQL-2.10) and then you can continue working with other PL/SQL programs.

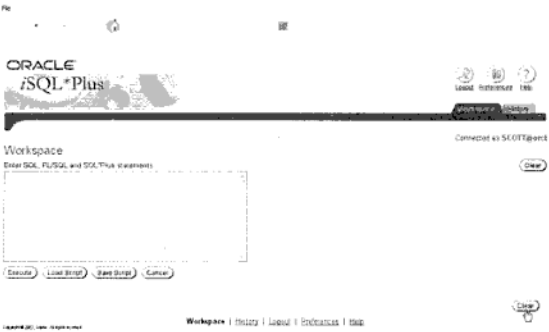



**Fig.PL/SQL-2.10**

So, these are the two processes to execute the PL/SQL programs. Now, let's continue our discussion with the next PL/SQL block structure that is exception section.

### *Section*

Exceptions are certain abnormal conditions, which occur sometimes during execution of a PL/SQL program and cause the program to terminate. Exception section of the program block is an optional one. All sections of the program block except the execution section are optional sections. To make a PL/SQL program free from runtime errors or exceptions, you are suggested to use exception section in PL/SQL programs.

The exception section in a block begins with the keyword `EXCEPTION` and ends with the keyword `END`.

This section basically catches the errors that occur when the program is executed. These errors are caught using the various functions provided specifically for this purpose. Some functions for exception handling are provided under the `STANDARD` or `DBMS_STANDARD` packages. We read more on exceptions in Chapter 11(Handling Exceptions in PL/SQL).

The following code snippet shows a simple `EXCEPTION` section of PL/SQL program.

After understanding the block structure of PL/SQL program, let's study various types of blocks.

## of Blocks

Depending on the header section, there are two main categories of blocks—anonymous blocks and named blocks. A third kind of block known as nested block is also available that can contain one or more anonymous/named blocks. Let's now study all these blocks in detail.

### *Blocks*

Anonymous blocks, as the name suggests, are those blocks that do not have a name. Therefore, we do not have any block header for these blocks; but they have one or more sections, which are declaration, execution, and exception. Since they do not have a name, we cannot call them. Due to this, anonymous blocks are also not stored in the database. However, an anonymous block can call other named blocks.

Uses of anonymous blocks in PL/SQL are:

- Declaring variables that can be used in the execution section.
- Declaring cursors that can be used in the execution section. Cursors are used to access elements stored in a collection. We will study in detail about cursors in Chapter 7(Understanding Cursors in PL/SQL).

- Executing cursor `SELECT` statement.

### *Named Blocks*

Named blocks are the blocks that have some name. It is obvious that if a block is a named block, it would contain a block header because the name of the block is defined in a block header. Therefore, any named block would contain at least two or more block sections. This is because the execution section is a necessary section for any PL/SQL block and if we make header compulsory then there would be two compulsory sections in the block. Overall, a named block can have all the four sections—block header, declaration, execution, and exception.

### *Nested Blocks*

Nested blocks are yet another important kind of PL/SQL blocks. When we have blocks placed inside other blocks, we call this phenomenon as nesting and such blocks are known as nested blocks. These nested blocks might further differ from each other depending on several different criterions. One of such important criterions is the level of nesting. Each Nested block has a certain depth. This depth is the number of levels to which that block has been nested. For example, if we have a block within a block, then it is said to be nested to level one. If the block contained inside the block further contains another block, then the level of nesting increases to two. Nesting is allowed only in two sections—execution and exception—of the PL/SQL block.

Let's now discuss the various data types available in PL/SQL to store real world entities, such as numbers, text, and images.

## Introducing Datatypes

Having knowledge about datatypes is essential before beginning with application development in any programming language because datatypes form the most basic building blocks for developing any program. PL/SQL supports all the predefined datatypes in SQL and also some additional ones. There has to be a datatype for every variable or parameter in PL/SQL. Depending on the need of the application, we can choose the datatype that suits us the best. Various datatypes available in PL/SQL are:

- Number Types
- Character and String types
- National Character Types
- Boolean Types
- LOB Types
- Date and Time Types
- Subtypes

### *Number Types*

Number types are scalar datatypes. The datatypes that hold single value are known as scalar datatypes. As the name suggests, number types are used to store numeric values. These numeric values may include integers, real numbers. Usually, we use number types for fields that need numbers as values, such as age, measurement.

The number types are further divided into following types:

- BINARY_INTEGER
- NUMBER
- PLS_INTEGER
- BINARY_DOUBLE
- BINARY_FLOAT

### BINARY_INTEGER

BINARY_INTEGER stores the signed integer in a two's compliment form. The range of binary types is from –2147483647 to +2147483647. This type is generally used when we need to perform arithmetic operations. It has following sub types:

- **NATURAL:** May store integers in range from 0 to 2147483647.
- **POSITIVE:** May store integers in range from 1 to 2147483647.
- **NATURALN:** Same as NATURAL, but cannot assign null to integer variables.
- **POSITIVEN:** Same as POSITIVE, but cannot assign null to integer variables.
- **SIGNTYPE:** It is used in programming tri-state logic that is an integer variable can only have value either -1, 0, or 1.

## *NUMBER*

The NUMBER type can hold floating-point values or integers in the range from 1.0x10^-130 to 9.99x10^125. The general syntax of NUMBER type is as follows:

```
NUMBER[(precision,scale)]
```

In the preceding syntax:

- **precision:** represents the total number of digits that a number variable can store.
- **scale:** represents the total number of digits that a number variable can contain to the right of the decimal. The scale can range from – 84 to 127. We can only use integer literals for specifying precision and scale.

Now, see the subtypes of the NUMBER type:

- DEC, DECIMAL, and NUMERIC: Used to declare fixed-point numbers with precision up to 38 decimal digits.
- DOUBLE PRECISION and FLOAT: Used to declare floating-point numbers with precision up to 38 decimal digits.
- REAL: Used to declare floating-point numbers with precision up to 18 decimal digits.
- INTEGER, INT and SMALLINT: Used to declare integers with a precision up to 38 decimal digits.

### PLS_INTEGER

PLS_INTEGER type can store signed integers in range from –2147483647 to +2147483647. PLS_INTEGER datatype is more efficient than NUMBER datatype since PLS_INTEGER values

take less storage than NUMBER values and also use hardware arithmetic for operations involving them. Oracle recommends the use of PLS_INTEGER over BINARY_INTEGER for developing new applications.

### BINARY_DOUBLE

BINARY_DOUBLE datatype was released for the first time with Oracle 10g release 1. It is called BINARY_DOUBLE because it is an IEEE-754 double precision floating datatype.

Literals of BINARY_DOUBLE type end with d, such as 1.005634d. It is a high precision datatype, which is generally used to measure system performance. It is used to measure performance because working with system parameters requires performing several complicated calculations correct to several decimal places.

### BINARY_FLOAT

BINARY_FLOAT datatype is newly introduced with the Oracle 10g release 1. It is also used to perform high precision scientific calculations but unlike BINARY_DOUBLE, it is single precision float datatype. However, it finds a similar application to BINARY_DOUBLE datatype in measuring performance gains. Literals of BINARY_FLOAT type end with f, such as 3.87f.

## *Character and String Types*

The character or string types in PL/SQL are used to store everything from single character values to large strings up to 32K in size. We can use these types to store letters, numbers, or binary data. We can also store any character supported by the database character set using this type. Character types are also scalar types. Various different datatypes that are included in this type are as follows:

- CHAR
- LONG
- LONG RAW
- NCHAR
- NVARCHAR2
- RAW
- ROWID
- UROWID
- VARCHAR
- VARCHAR2

### CHAR

CHAR is a fixed length datatype. By default, data is stored in bytes. It is usually used to store the basic character type values. Internal representation of characters depends upon database character set (ASCII, EBCDIC....). The general syntax of CHAR data type is as follows:

```
CHAR[(size[CHAR|BYTE])]
```

The size literal can have value from 1 to 32767. The size can be in terms of characters or bytes. Therefore, upper limit of CHAR variable is 32767 bytes but we cannot insert CHAR values greater than 2000 bytes in CHAR database column. Default value of size is 1.

LONG type is used to store variable-length character strings. It can hold values up to 32760 bytes. Note that the long type in PL/SQL is different from the LONG database column used in Oracle database. Maximum width of LONG column in Oracle database is 2147483648 bytes. we can insert LONG value into LONG column but we cannot do vice versa.

RAW is similar to the LONG type but it can also store binary data in addition to storing strings.

You are already aware of ASCII and EBCDIC character sets, which are used to represent roman These alphabets are internally stored only in one byte but some characters of Asian such as Japanese need more bytes for their internal storage representation. Therefore, provides globalization support to run Oracle applications in many language environments. Two character sets that are used for internationalizing (or globalizing) an application are database character set and national character set.

National character set is used for national language data. National character set represents data two encodings—UTF8 and AL16UTF16. In UTF8 encoding, each character may be stored 1, 2, or 3 bytes depending upon runtime length requirement. In AL16UTF16 encoding, each is stored in 2 bytes whether a string consists of ASCII characters or not. This is more encoding at runtime.

holds fixed-length national character data. Internal storage representation of string depends upon specified national character set with specified encoding. The general syntax of NCHAR data type is as follows:

size in this syntax is an integer literal. The upper limit of size literal is 32767/2 for AL16UTF16 encoding and 32767/3 for UTF8 encoding. The maximum size of NCHAR database is 2000 bytes; therefore, you cannot insert NCHAR values exceeding 2000 bytes into a database column.

can assign a CHAR value into a NCHAR variable but opposite assignment results into loss of bytes.

holds variable-length character data. It is identical to the VARCHAR2 type, but takes character set specified by the National Character Set. The general syntax of NVARCHAR2 as follows:

In this syntax, the size is a literal. The upper limit of size literal is 32767/2 for AL16UTF16 encoding and 32767/3 for UTF8 encoding. As maximum size of NCHAR database column is 4000 bytes, therefore you cannot insert NCHAR values exceeding 4000 bytes into a NCHAR database column.

You can assign VARCHAR2 value into a NVARCHAR2 variable but opposite assignment results into loss of some bytes.

### RAW

The RAW type stores fixed-length binary data, such as a set of graphic characters and can hold up to 32K (32,767 bytes). The general syntax of RAW types is as follows:

```
RAW(size)
```

The size is an integer literal and can have values from 1 to 32767. Since the RAW column of Oracle database can hold only 2 Kilo bytes, therefore, we cannot insert RAW value into RAW column. We can store RAW value into LONG RAW database column. We also cannot retrieve value of the LONG RAW column into LONG RAW variable.

### ROWID

Every record in a database table internally contains a unique binary value called a ROWID. The rowid is storage address of the row. The rowids can be of two types—physical and logical.

Physical rowids provide quick access to specific rows. By default, there is one physical rowid for each row in the database table. Physical row id can be of two formats—10-byte extended rowid format and 6-byte limited rowid format. You can also retrieve the rowid of a particular row. SQL * Plus automatically changes binary rowid into character rowid. Execute the following query to see rowid:

```
select rowid,ename from bonus where empid=102;
```

This query generates the following row:

```
Rowid                  Ename
AAAL+bAAEAAAAAVAAB     Gaurav
```

The general format of received rowid is OOOOOOFFFBBBBBBRRR. This rowid format has following four components:

- **OOOOOO:** These six O's represent database segment. In our case, it is AAAL+b. This number is also called data object number.
- **FFF:** These three F's represent file number that recognizes data file, which consists the row. In our case, it is AAE.
- **BBBBBB:** These six B's represent block number that recognizes data block, which consists the row. In our case, it is AAAAAv.
- **RRR:** These three R's represent the row in data block.

Logical rowids provide quicker accesses to particular rows. Oracle uses these rowids to make secondary indexes on indexed tables. If we change location of the row, its logical rowid remains same.

ROWID type can only store physical rowids not logical ones. You need to be careful when retrieving and storing database rowid into ROWID variable or vice versa. Use ROWIDTOCHAR function to convert binary value into 18 byte character string when fetching database rowid into ROWID variable; otherwise, use CHARTOROWID function.

### UROWID

UROWID provides support for storing both physical and logical rowids. There is no need to use conversion functions when retrieving or storing UROWID variable into UROWID database column. The UROWID column can store maximum 4000 bytes.

### VARCHAR2

VARCHAR2 data type is used to store variable-length character data. The general syntax of VARCHAR2 data type is as follows:

```
VARCHAR2 (size [CHAR|BYTE]
```

The size in this syntax must be integer literal. It can take value from 1 to 32767. VARCHAR2 variables having length shorter than 2000 bytes are known as small VARCHAR2 variables. For these variables, PL/SQL statically allocates memory equal to declared size of variable. For larger variables, PL/SQL allocates memory dynamically whose size is just enough to store runtime value. Note that you cannot insert VARCHAR2 values greater than 4000 bytes into VARCHAR2 database column.

We can insert VARCHAR2 values into LONG database column but we cannot retrieve value of LONG database column into VARCHAR2 variable.

The subtypes of VARCHAR2 are STRING and VARCHAR. These types have the same range as that of VARCHAR2 type.

#### *VARCHAR*

VARCHAR is an ANSI-standard SQL type, synonymous with VARCHAR2. Oracle recommends using VARCHAR2 to protect against future modifications to VARCHAR impacting code.

## *Boolean Types*

PL/SQL supports BOOLEAN data type to represent logical values—TRUE, FALSE, and NULL. You can assign NULL value to a BOOLEAN variable when you do not know either value of the variable or assigning value to a variable does not make any sense for a particular record in table or variable. For example, in record of employee who holds designation of a software engineer, comm_pct variable or field, which represents percentage of commission does not take any value but for a sales person, commission matters and comm_pct variable has some definite value. Therefore, comm_pct variable may be assigned to NULL value for software engineer.

Note that you cannot use BOOLEAN variables in SQL queries since SQL does not support BOOLEAN data type. This note also implies that you neither insert BOOLEAN values into a database column nor retrieve column values into BOOLEAN variable.

You cannot use quotes when assigning TRUE, FALSE, or NULL values to a BOOLEAN variable else you will encounter an error. Listing 2.2 tries to assign TRUE as a character string to a Boolean variable.

**Listing 2.2:** Assigning character variable to boolean variable

On execution on iSQL*Plus, Listing 2.2 generates following output:

The output shows error expression is of wrong type in line 4.

## *LOB Types*

LOB stands for large object. PL/SQL supports various LOB datatypes, such as BFILE, BLOB, CLOB, and NCLOB. These datatypes can store text, graphics, audio, and video clips. Size of BLOB, CLOB, and NCLOB datatypes ranges from 8 to 128 terabytes but size of BFILE is system dependent and cannot exceed than 4 gigabytes.

LOB types are manipulated through lob pointers which can point to large objects stored in external file, inside or outside the row. LOB database columns also store lob locators. When you fetch a LOB column value, you get only lob pointer. This lob pointer is used to manipulate corresponding large object. You can use existing package DBMS_LOB to perform read and write operations on LOBs.

You can also convert CLOBs to CHAR, VARCHAR2 types or BLOBs to RAW and vice versa.

### BFILE Datatype

BFILE datatype is used to store large binary objects in operating system files outside the database. The pointer stored in BFILE variable points to large binary file on server and contains full path of binary file. These BFILE variables are read only. Note that you can limit the number of opened BFILEs by setting Oracle initialization parameter SESSION_MAX_OPEN_FILES. This parameter is also system dependent.

### BLOB Datatype

BLOB data type is used to store large binary objects in the database, inside or outside the row. You can use only BLOBs not BFILEs in transactions .You can also either revert back or commit changes made to BLOBs using DBMS_LOB package.

### CLOB Datatype

CLOB data type is used to store large group of character data in the database, inside or outside the row. CLOBs participate in transactions and you can either revert back or commit changes made to CLOBs using DBMS_LOB package.

### NCLOB Datatype

NCLOB data type is used to store both fixed and variable length NCHAR character blocks. NCLOBs participate in transactions and you can either revert back or commit changes made to NCLOBs using DBMS_LOB package.

## *Date, Time, and Interval Types*

Both date and time are stored in one variable called datetime variables and time periods are stored in another datatype called interval datatype.

### DATE Datatype

DATE data type is used to store fixed length date and time in which time is specified in seconds. Oracle initialization parameter NLS_DATE_FORMAT determines default date format. Usually, default date format is 'DD-MON-YY' where DD is two digit number for day, MON is abbreviation for month, YY includes last two digits of year.

You can extract time from DATE variable in the following two steps:

- Use SYSDATE built-in function to get current date and time and store it in variable say date_variable.
- Extract the current time using date_variable-TRUNC(date_variable).

You can perform other operations, such as comparing two dates, arithmetic operations on DATE variables. For example, SYSDATE+1 returns the same time for tomorrow.

### TIMESTAMP Datatype

The TIMESTAMP data type extends DATE datatype and store day, month, year, hour, minute, and second. The seconds field can store seconds in fractions. The general syntax of TIMESTAMP datatype is as follows:

```
TIMESTAMP[(precision)]
```

Where precision is integer literal between 0 to 9 and specifies number of digits in fractional part of seconds field. Oracle initialization parameter NLS_TIMESTAMP_FORMAT determines the default timestamp format.

### TIMESTAMP WITH TIME ZONE Datatype

The TIMESTAMP WITH TIME ZONE datatype extends TIMESTAMP datatype and also includes difference between local and **Greenwich Mean Time** time zones. The general syntax of this datatype is as follows:

```
TIMESTAMP[(precision)] WITH TIME ZONE
```

Where precision determines number of digits in fractional part of seconds field. Oracle initialization parameter NLS_TIMESTAMP_TZ_FORMAT determines the default timestamp with time zone format. Following are some examples of TIMESTAMP WITH TIME ZONE values:

In this example, time-zone displacement in the first TIMESTAMP is +02:00. Second and third values are same but later specifies time zone by symbolic name, such as US/Pacific. You can see symbolic names of other zones by executing following query on iSQL*Plus.

```
SELECT * FROM V$TIMEZONE_NAMES;
```

### TIMESTAMP WITH LOCAL TIME ZONE Datatype

The TIMESTAMP WITH LOCAL TIME ZONE data type extends TIMESTAMP data type and includes time zone displacement. Difference between TIMSTAMP WITH LOCAL TIME ZONE and TIMSTAMP WITH TIME ZONE is that value of the TIMSTAMP WITH TIME ZONE is not stored in a database column while value of TIMSTAMP WITH LOCAL TIME ZONE is stored in the database time zone and is automatically normalized.

We are explaining you normalization with the help of an example. For example, head department of an organization would like to know about the number of orders placed yesterday in its different departments. All departments are remotely located and they have different time zones. Therefore, yesterday means different date and times in locations of other departments. If head of the department declares ORDER.DATE column of TIMESTAMP WITH LOCAL TIME ZONE data type, then different dates and times of placed orders are automatically converted (normalized) into time zone of the head of department.

### INTERVAL YEAR TO MONTH Datatype

The INTERVAL YEAR TO MONTH datatype is used to store intervals in years and months. The general syntax of this data type is as follows:

```
INTERVAL YEAR[(precision)] TO MONTH
```

Where precision is an integer literal in range 0 to 4 and denotes the number of digits in years filed. See Listing 2.3 which shows the use of INTERVAL YEAR TO MONTH datatype.

**Listing 2.3:** Use of INTERVAL YEAR TO MONTH datatype

First assignment statement assigns an interval of 10 years and 1 month to a timeperiod variable. In second assignment statement, Oracle automatically converts character literal '10-1' to interval type. Third and fourth assignment statements change timeperiod variable to contain interval of years and interval of months respectively.

### INTERVAL DAY TO SECOND Datatype

The INTERVAL DAY TO SECOND datatype is used to store intervals in days, hours, minutes, and seconds. The general syntax of this data type is as follows:

```
INTERVAL DAY[(precision1)]
  TO SECOND[(precision2)]
```

Where `precision1` and `precision2` tell the number of digits in day and second fields respectively. Both `precision1` and `precision2` are integer literals and can take values in the range 0 to 9. Listing 2.4 shows the use of `INTERVAL DAY TO SECOND` datatype.

**Listing 2.4:** Use of `INTERVAL DAY TO SECOND` datatype

This example checks how much extra time an organization spent on a project.

You are already well aware of values that day, month, and year can take. Table 2.1 lists all values that various fields under date, time, and interval types can take.

| | | |
|---|---|---|
| | -4712 to 9999(cannot be 0) | Any nonzero integer |
| | 01 to 12 | 0 to 11 |
| HOUR | 00 to 23 | 0 to 23 |
| MINUTE | 00 to 59 | 0 to 59 |
| SECOND | 00 to 59.9....9 | 0 to 59.9...9 |
| TIMEZONE_HOUR | -12 to 14 | N.A. |
| TIMEZONE_MINUTE | 00 to 59 | N.A. |
| TZNAME | Column of v$TIMEZONE_NAMES | N.A. |
| TZABBREV | Column of V$TIMEZONE_NAMES | N.A. |

### *PL/SQL Subtypes*

PL/SQL subtype defines a range of values and a set of operations on variables of predefined PL/SQL datatype. These restrictions are defined on already existing PL/SQL type. Therefore, subtype is not any new data type and it is the one that is derived from existing PL/SQL data types. PL/SQL has also defined some subtypes in `STANDARD` package. For example, `CHARACTER` and `INTEGER` data types in PL/SQL are defined as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

From these definitions, `CHARACTER` subtype is the same as `CHAR` datatype but an `INTEGER` subtype consists of only 38 digits.

User defined data types are also declared using following PL/SQL syntax:

```
SUBTYPE subtypename IS basetype[(const)] [NOT NULL];
```

Where `subtypename` is name of a new sub type, `basetype` is name of already existing PL/SQL type, and const specifies size of new sub type. Following example declares subtype named `bdate` that can take null value.

```
SUBTYPE bdate IS DATE NOT NULL
```

Let's now discuss syntactic elements of PL/SQL, such as delimiters, identifiers, and literals.

## Introducing Lexical Units

You can choose characters out of alphabets (A...Z, a...z), numbers (0... 9), symbols (+ - * / < > =! ~ ^ ; : .( ) ' @ % , " # $ & _ | { } ? [ ]), tabs, and spaces to write a PL/SQL program. The group of characters in a PL/SQL line of code is called lexical units. Lexical unit in PL/SQL may contain a single character or more than one character. A line of PL/SQL block can contain following lexical units:

- Delimiters
- Identifiers
- Literals
- Comments

### *Delimiters*

A delimiter is a symbol or a set of symbols that has predefined meaning in PL/SQL. For example, + delimiter represents arithmetic operation and : = delimiter represents assignment operator. See Table 2.2 for delimiters containing only one symbol.

**Table 2.2: All one symbol delimiters**

| Delimiter | Name |
|---|---|
| + | Addition operator |

## *Identifiers*

An identifier is a name given to constants, variables, exceptions, cursors, cursor variables, sub programs, and packages. This name should start with a letter that may be followed by more letters, numbers, dollar signs, underscores, and number signs. The name of identifier cannot contain hyphen, slashes, and spaces. The identifier can contain less than or equal to 30 characters. You can also use more than one dollar sign, underscores, and number signs.

Some examples of identifiers are `A`, `temp2`, `name#`, `time_limit`, `FirstName`, and `R`. Try to make identifiers descriptive and meaningful. For example, use variable name as percent instead of pct. Some non-examples of identifiers are `time-limit`, `either/or`, `emp name`, and `i&u`.

### Reserved Words

There exists a long list of identifiers in PL/SQL with each identifier having its specific syntactic meaning to PL/SQL. These identifiers are called reserved words. If you try to redefine them, then they cause compilation error. For example, `BEGIN` and `END` are reserved words indicating the beginning and end of block respectively. Here is an example which uses `BEGIN` word for the name of variable of `DATE` type.



When you execute this code snippet on iSQL*Plus console, an error occurs.

### Predefined Identifiers

Identifiers declared in the `STANDARD` package are called predefined identifiers but these identifiers can be redeclared. Redeclaration of identifier overrides the declaration of the same identifier in `STANDARD` package.

### Quoted Identifiers

PL/SQL supports identifiers enclosed in double quotes. These identifiers used to contain a sequence of characters (excluding double quotes) to be printed. Number of characters in double quotes cannot exceed 30. For example, `"A+B"`, `"**** Write a program to find square root of number ****"`.

## *Literals*

Literal represents numeric, character, or Boolean value. It is not the name of any variable. Example of numeric literal is 135 and Boolean literal is `TRUE`. Literals are divided into following categories:

- Numeric Literal
- Character Literal
- String Literal
- Boolean Literal
- Datetime Literal

### Numeric Literal

The numeric literals are used in arithmetic expressions. These literals are of two types—integer and real. An integer literal is a number (sign not compulsory) without decimal point. Examples of integer literals are 025, 34,-67, and +3456. A real literal is a whole or a fractional number with decimal point. The sign is not compulsory here also. Examples of real literals are 7.45657, 0.0,-23.67, .6, and 24.0.

Scientific notations also contain some numeric literals, such as 3E5 or 3e5, 3.1414e0, and 5e-3. In these literals, E or e is 10 and the number after E or e is power of ten. To write a number using scientific notation, suffix the number with an (E or e) followed by an optionally signed integer.

Note that you cannot assign literal value to a variable greater than the upper limit of the variable's datatype.

### Character Literal

A character literal is a single character enclosed in single quotes. This single character can be any character of PL/SQL character set. Examples of character literals are 'a', 'A', and '('. Note that 'a' literal is different from 'A'.

### String Literal

A string literal is a sequence of zero or more characters enclosed in single quotes. Examples of string literals are 'ABC pvt. Ltd', '$154.99'. If you need to use character literals, such as I 'm ill, then add one more apostrophe before apostrophe within a string as shown here— `'I ''m ill'`.

### Boolean Literal

Boolean literal takes logical values, such as `TRUE`, `FALSE`, and `NULL`. The `NULL` literal stands for missing, unknown, or inapplicable value.

### Datetime Literal

Datetime literals can have different types of date time values depending upon the data type. Listing 2.5 declares various types of datetime literals.

**Listing 2.5:** Declaring different datetime literals

In this example, `d` is `DATE` literal, `ts1` is `TIMESTAMP` literal, `ts2` is `TIMESTAMP WITH TIME ZONE` literal, `iytm1` is `INTERVAL` literal specified in years and months, and `idts2` is `INTERVAL` literal specified in days and seconds.

## *Comments*

The PL/SQL compiler does not compile comments in a PL/SQL program but this does not mean that developers should not add comments. Comments are user friendly and help in better understanding of PL/SQL program. PI/SQL supports two types of comments—single line comments and multi line comments.

### Single Line Comments

Single line comments start with double hyphen (--) and ends at the end of line. Usually, they appear at the end of PL/SQL statements. Here is an example of single line comment.

Two single line comments in this code snippet explain the meaning of interval literals `iytm1` and `idts2`. First comment tells the user that `iytm1` literal stores interval of ten years and one month. Second comment tells the user that `idts2` literal stores interval of time duration of around three days, five hours, three minutes, two and 1/100 seconds.

You may use single line comments to comment out (or disable) any statement during testing or debugging of program.

### Multi-line Comments

Multi-line comment starts with /* symbol and ends with */. In between these symbols, there may be one or more than one lines which explain functionality of the specific portion of PL/SQL program to user. Listing 2.6 is an example of multi-line comment.

**Listing 2.6:** An example of multi-line comment

The multi line comment tells user about author and functionality of the procedure. With the help of multi line comments, you can disable sections of code during debugging of PL/SQL program.

Let's now discuss how to declare variables and constants with restrictions and assigning `DEFAULT` values to them.

## Working with Declarations

You can declare variables and constants which are used to store values in any PL/SQL block, sub program, or package. Each declaration of variable specifies its data type and name of storage space allocated by PL/SQL compiler. This name is further used to access or manipulate it. Following example declares some variables and constants:

This example declares `matchdate` variable of `DATE` type, `noofmatches` constant of `INTEGER` type, and `manofmatch` constant of `REAL` type. Both constants are also initialized since the value of constant does not change during execution of the program.

### *Using DEFAULT Value*

Keyword `DEFAULT` is used in place of assignment operator to initialize variables that have some specific value. For example,

variables have some already predefined values.

You may use `DEFAULT` keyword to initialize cursor and subprogram parameters and fields in a user-defined record.

### *Constraint*

can impose constraints on a variable, such as variable can accept null value. If you assign null value to variable declared as `NOT NULL`, PL/SQL engine generates predefined `VALUE_ERROR` exception. You can declare a variable as `NOT NULL` as follows:

```
DECLARE
Account_NO LONG NOT NULL :=11003533056
```

This code snippet defines `Account_NO` variable as `NOT NULL`.

### *Using Aliases*

Aliases are used when the cursor is defined with the expressions. Aliases can also be used in the case when you are selecting some values from the database table and want to display the set of retrieved values through a single variable. That variable is known as alias. Listing 2.7 illustrates the use of alias in `SELECT` statement.

**Listing 2.7:** Use of alias



This example assigns alias `full_name` to `first_name ||' '|| last_name` expression. In loop, we are accessing `full_name` from record having full names of first four employees.

Let's now move on to understand and use different types of operators available in PL/SQL.

## Introducing Operators

PL/SQL supports a set of operators which are helpful in building a variety of expressions. Expressions consist of operators, which act upon operands and generate a result. All expressions return only one value. We are now discussing all type of operators available in PL/SQL.

### *Assignment Operator*

Assignment operator (:=) in PL/SQL is used to assign a simple value or expression to a variable on its left side. Following example assigns value of expression to Quantity variable.



You can assign a value to a suitable variable in any section, such as declaration, execution, or exception. If you do not assign any value to a variable in declaration section, then it is assumed to store `NULL` value.

You can also assign a function call to variable so that variable will store the value when returned by function. Assignment operator is used to hold Dynamic SQL statements into a string variable.

The EXECUTE IMMEDIATE command takes this string variable and executes the query. Here is an example:



### Arithmetic Operators

PL/SQL supports mathematical operators, such as addition (+), subtraction (-), multiplication (*), division (/) to build and evaluate arithmetic expressions. Usually, multiplication and division have higher priority as compared to addition and subtraction. Parentheses can change the order of evaluation of an expression. For example, when PL/SQL engine evaluates (10+6)/2 expression, first addition is performed, then division is performed and the result is 8 not 13. An arithmetic expression can contain nested parentheses in which deeply nested sub expression is evaluated first. For example, 10+(6/2+(5-3)) generates 15.

### Logical Operators

PL/SQL also supports logical operators, such as AND, OR, and NOT to form logical expressions. The AND and OR operators are binary operators as they require two operands but NOT is a unary operator. Table 2.4 lists truth table for each logical operator.

**Table 2.4: Truth tables of AND, OR, and NOT logical operators**

| A | B | A AND B | A OR B | NOT A |
|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE | FALSE |
| TRUE | NULL | NULL | TRUE | FALSE |
| FALSE | TRUE | FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE | FALSE | TRUE |
| FALSE | NULL | FALSE | NULL | TRUE |
| NULL | TRUE | NULL | TRUE | NULL |
| NULL | FALSE | FALSE | NULL | NULL |
| NULL | NULL | NULL | NULL | NULL |

If we see the truth table of each operator, we find that AND returns TRUE value only if both of its operands are true, OR returns TRUE if any of its operands is true, and NOT operator returns the logical negation of its operand.

Parentheses may change the order of evaluation of logical expression.

## Comparison Operators

Comparison operators are used to compare simple values or expressions. These operators are used in conditional control statements and WHERE clause of SQL DML statements. Table 2.5 lists all the comparison operators.

**Table 2.5: Comparison operators**

| Operator | Purpose |
|---|---|
| = | Checks whether its operands are equal. |
| < | |
| > | Checks whether left side operand is greater than the right side operand. |
| <= | Checks whether left side operand is less than or equal to the right side operand. |
| >= | Checks whether left side operand is greater than or equal to the right side operand. |

## IS NULL Operator

The IS NULL operator is used to check whether its operand is null or not. This operator returns TRUE if the operand is null, otherwise FALSE.

```
IF var IS NULL THEN
```

## Concatenation Operator

Concatenation operator, denoted by double vertical bars (||) concatenates two strings of type, such as CHAR, VARCHAR2, and CLOB. For example,

```
set serveroutput on;
BEGIN
DBMS_OUTPUT.PUT_LINE('I'||'You');
END;
/
```

This example outputs string 'IYou'. Data type of output string depends upon datatypes of the operands. If CHAR is the data type of both operands, then the output string obtained after concatenation will have CHAR data type. If one of the operands has data type CLOB, then result will be of CLOB data type. If one of operands is VARCHAR2, then result will be of VARCHAR2 data type.

## LIKE Operator

LIKE operator is used to compare single character, string, or CLOB value to a specified pattern. Case of characters matters during comparison of string with a pattern. If string matches with the specified pattern, then LIKE operator returns Boolean TRUE; otherwise, it returns FALSE.

Patterns used with LIKE operator can include two important characters called wildcards—underscore (_), percent sign (%). The underscore (_) character checks whether there is only one character present in the actual string between characters surrounding underscore (_) in pattern. The percent sign (%) character finds whether zero or more characters are present in actual string. Here is an example:

```
select * from emp where ename like 'T%N_R';
```

This query results into following record:

```
EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO
7844 TURNER SALSEMAN 7698 08-SEP-81 1500 0 30
```

**Note**

This record already exists in table EMP.

If actual string contains % or_ characters, then you need to define an escape character which is put before the % or _ in pattern. For example:

```
IF discount LIKE '50\% off' ESCAPE '\' THEN ..
```

There may be some real time entities in world that may have % or _ , such as marks, user ids, or email ids.

## Range Operator: BETWEEN

The BETWEEN operator checks whether value of the variable lies in a specified range. It returns TRUE in following cases:

- When value of variable equals to lower bound of range
- When value of variable is greater than lower bound and less than upper bound
- When value of variable equals upper bound

Otherwise, it returns FALSE value.

Let's use BETWEEN operator. See the following query:

```
select * from emp WHERE SAL BETWEEN 3000 AND 5000;
```

This query result fetches following records from table emp.

## List Operator: IN

The IN operator differs from BETWEEN operator as it specifies a set of members or values but BETWEEN operator only specifies lower and upper bounds. This set of values can contain null values. This operator is used when the user knows all set of values and wants to retrieve rows of table corresponding to these values. For example,

```
select * from emp where ename in ('JONES' ,'CLARK','FORD');
```

This query fetches and displays rows of `emp` table having `ename` JONES, CLARK, and FORD.

You have understood all types of operators but there are some PL/SQL expressions containing operators from different categories. To understand evaluation of these expressions, you must be aware of operator precedence of all operators. Table 2.6 lists the Operators precedence.

| | |
|---|---|
| **AND** | **Logical AND** |
| **OR** | **Logical OR** |

Now we describe various attributes used to get information about data type and structure of database entities.

## Introducing Attributes

PL/SQL variables and cursors have standard attributes that are used to access the properties, such as data type, structure of database tables. PL/SQL supports two attributes—`%TYPE` and `%ROWTYPE`. Each has a prefix %.

### *Using the %TYPE Attribute*

The `%TYPE` attribute is used to define variables with data type same as that of database column. Suppose that there is a column named name in table `emp`. To declare a variable `name1` with data type of name column, use `%TYPE` attribute as follows:

```
name1 emp.name%TYPE
```

If database administrator changed the data type of name column, data type of `name1` variable will also change at run time.

### Attribute

attribute is used with records. A record is composed of a set of related fields each field has a data value. The %ROWTYPE attribute declares a record type, which is as the row of a specified table. The declared record can also store the record fetched from

code snippet declares an emp_rec record with fields that have same names and data as columns in EMP table:

further access values of fields of record in some variable as following:

we put all the operators for building PL/SQL expressions.

## PL/SQL Expressions

of operators, which act upon operands and generate a result. An operand can constant, literal, or function call, which results into a value. PL/SQL supports two of operators—unary and binary. Unary operators operate on only one operand and binary operate on two operands. All expressions return only one value. The data type of value upon datatypes of other operands and type of operator.

can use expressions in both SQL and procedural statements. In most cases, Boolean are used. Boolean expressions are made up of simple or complex expressions, which comparisons using relational operators. Logical operators, such as AND, OR, and NOT connect these Boolean expressions to build more complex expressions. These expressions result into TRUE, FALSE, or NULL value.

SQL statements, Boolean expressions are first used to find a particular row of a table and the is executed on that row. In procedural statements, Boolean expressions are used to conditions of conditional control statements, such as IF-THEN, IF-THEN-ELSE.

supports three types of Boolean expressions—arithmetic, character, and date.

### Arithmetic Expressions

arithmetic expressions involve the quantitative comparisons of numbers. For example, num1 and num2 are two integer variables assigned to integer values 45 and 23

This expression results into TRUE value.

### Boolean Character Expressions

Boolean character expressions involve comparisons of character values. These performed on binary representation of each character string. For example,



The expression in this example results into TRUE value. You can make based. This is done by setting NLS_COMP parameter to ANSI. These comparisons using collating sequences. A collating sequence represents characters in a character set numeric codes so that character strings can be compared.

You can make other types of comparisons, such as case-insensitive, accent-insensitive by changing the value of NLS_SORT parameter. If you want to perform case-insensitive comparison, append _CI to end of initial value of NLS_SORT parameter. If you want to perform accent-insensitive comparison, append _AI to the end of initial value of NLS_SORT parameter.

### Boolean Date Expressions

Boolean Date expressions involve chronological comparisons of dates. Chronological comparison of dates means the more recent date would be considered as the larger one between two dates. For example, consider two DATE variables d1 and d2 in the following code snippet:



In this example, since d2 is more recent than d1, therefore d2 is greater than d1 which results (d2>d1) expression into TRUE value.

**Note**



Conversion between different types may happen if it makes sense. PL/SQL supports both implicit and explicit conversion. Let's handle the conversion between different data types of PL/SQL.

## Datatypes conversion in PL/SQL

There is a need of conversion from one data type to another so that you can access and manipulate resultant variable. For example, binary rowid is converted into character rowid by PL/SQL engine so that user can easily understand the location of a particular row. There are two types of conversions—Explicit conversion and Implicit conversion.

### *Explicit Conversion*

To perform conversion from one data type to another data type by user, PL/SQL provides built-in functions. For example, functions `TO_DATE` and `TO_NUMBER` are used to convert `CHAR` value to a `DATE` and `NUMBER` value. You can study section Conversion Functions of Chapter 3 of this book for all types of conversions.

### *Implicit Conversion*

PL/SQL automatically converts one data type into another data type when conversion makes sense and is logical. Listing 2.8 shows a case of implicit conversion.

**Listing 2.8:** Using implicit conversion

This example first stores date value into charvar1 variable of `VARCHAR2` data type and assigns this variable to `date1` variable of `DATE` data type. Then, it assigns `date1` variable to another `charvar2` variable of `VARCHAR2` data type. Both `charvar1` and `charvar2` variables are displayed to see whether they store the same value.

When you execute Listing 2.8 on iSQL*Plus, it shows the following output:

The output shows both the variables containing same date 28-DEC-07. Implicit conversion may also happen when you store values into database columns. See Table 2.7 for all implicit conversions between different data types that PL/SQL supports.


Table 2.7 lists all standard implicit conversions where cross sign(X) indicates conversion between corresponding data type (represented by row) into a data type (represented by column). Implicit conversion also depends on values that are stored in variables. For example, if you stored 'Monday' in `CHAR` variable, PL/SQL cannot convert it into `DATE` value.

## Summary

In this chapter, we learned about:

- Block Structure of PL/SQL program, which consists of four sections—block header, declaration, execution, and exception.
- Various different datatypes available in PL/SQL, such as Number, Character, Boolean, LOB, Date and Time, and sub types with their syntax and uses.
- Lexical units in the program, such as delimiters, Identifiers, Literals, and Comments.
- Different types of operators, such as assignment, arithmetic, logical, comparison, concatenation, LIKE, and IN.
- Various PL/SQL Boolean expressions.
- How to handle conversion between different data types implicitly or explicitly.

Functions are important part of any programming language. A function is a block of code that takes values, such as String, char, number, and date, as argument and gives the output according to its functionality. Using functions in a program enhances the efficiency of the program; since you can simply call the required function instead of writing the entire code to perform a specific operation. For example, if you want to calculate the length of a string, instead of writing long programming code, you can only use the `LENGTH` function provided by PL/SQL.

Functions can be divided into two categories, Built-in and custom or user-defined. Various programming languages, such as C, and C++, support Built-in functions that simplify the work of a programmer by reducing the code size. Similar to these programming languages, PL/SQL also supports Built-in functions that can be used directly in a program for performing a specific task rather than writing a long code for that task.

PL/SQL provides lots of Built-in functions to simplify the PL/SQL programming and provide better functionality. The Built-in functions help reduce the complexity of a program by reducing the size of program code and hiding the logic used to perform any specific task. For example, if you want to convert an upper case string into a lower case string, you need to write the code for performing this conversion. This increases the length of the program code and makes the program more complex to manage. With PL/SQL Built-in functions, you can perform this same conversion by calling the `LOWER` function.

In this chapter, we describe various categories of Built-in functions provided by PL/SQL, which are:

- Character functions
- Date functions
- Numeric functions
- Conversion functions
- LOB and Miscellaneous functions

We will study all these function in detail next. Here we start our discussion with character functions.

## Character Functions

Character functions retrieve information of a string and modify its content. For example, the character functions can be used to calculate the length of a string, convert upper case letters to lowercase letters, replace a character or a sequence of characters of a string with another specified character, concatenate two strings, and so on.

Character functions accept `CHAR` or `VARCHAR2` values as parameters and return either a number or a character value. PL/SQL has various built-in character functions. Some important and widely-used character functions have been summarized in Table 3.1.

**Table 3.1: PL/SQL character functions and their description**

| Function | Description |
|---|---|
| **ASCII** | **Returns the ASCII value the function.** |

After a brief description of some of the character functions, let's understand how these functions work. Let's study all the functions given in Table 3.1.

### ASCII Function

The ASCII function accepts a character as a parameter and returns the ASCII value of that character. You can also pass a string as a parameter but this function only returns the `ASCII` value of the very first character in the string.

Syntax for the ASCII function is:

```
ASCII (Single_character);
```

In this syntax:

- `single_character` is the specified character whose `ASCII` value has to be retrieved.

Listing 3.1 demonstrates how the `ASCII` function works.

**Listing 3.1:** PL/SQL program to use ASCII function

```
SQL>SET SERVEROUTPUT ON
DECLARE
```

In Listing 3.1, we have declared a variable grad of the NUMBER datatype in the Declare section of the PL/SQL block. In the Execution section, we have used SQL SELECT statement to retrieve the value of employee's GRADE from the table salgrade where HISAL=3000. The retrieved value is stored in the grad variable.

Finally, we have passed the grad variable in the ASCII function to know the ASCII equivalent of the value stored in that variable.

The output of Listing 3.1 is as follows:

**Output:**

The output 52 is the ASCII value of 4. If the query executes successfully, then there appears a message PL/SQL procedure successfully completed, as shown in the output.

## *LENGTH Function*

This function accepts a string as an argument. The string can have special characters as well. This function returns the length of a string specified as an argument. This function returns NULL rather than zero if you pass an empty string.

Syntax for the LENGTH function is:

```
LENGTH (string_value);
```

In this syntax:

- string_value is the string whose length has to be calculated.

See Listing 3.2 to understand how the LENGTH function works.

**Listing 3.2:** PL/SQL program to use LENGTH function

In Listing 3.2,first of all, we have declared a variable nam of datatype VARCHAR2 in declare section of the PL/SQL block. Now, in the execution section we have used SQL query to retrieve the ENAME (employee name) from the table EMP where EMPNO=7844. retrieved value is stored in variable nam. Next, the variable nam has been passed in the function to get the length of employee name stored in the variable nam.

The output of Listing 3.2 is as follows:

**Output:**

```
6
PL/SQL procedure successfully completed.
```

In this output 6 shows the length of the employee name retrieved from the table EMP.

## INITCAP Function

The INITCAP function accepts a string as an input and converts the first letter of every word in uppercase and rest of the words in lowercase letters.

Syntax for the INITCAP function is:

```
INITCAP (string_value);
```

In this syntax, string_value is the string in which the first letter of each word has to be converted in uppercase and rest all in lowercase.

Listing 3.3 shows how the INITCAP function works.

**Listing 3.3:** PL/SQL program to use INITCAP function

In Listing 3.3, first we have declared a variable nam of the VARCHAR2 datatype. After declaring the variable, SQL SELECT statement has been used to fetch the employee name into the variable nam from the table EMP where EMPNO = 7876. At last , we have applied the INITCAP function on the variable nam. When we execute the code shown in Listing 3.3, the output we get is as follows:

**Output:**

```
Adams
PL/SQL procedure successfully completed.
```

In this output you can notice that the first letter of the name is in uppercase and rest all are in lowercase letters.

## CONCAT Function

The CONCAT function concatenates two strings passed as parameters. This function appends the second string at the end of first string and then returns the resultant string. If any of the strings passed is NULL, it returns another string as the result. It will return NULL if both the string passed are NULL.

Syntax for the CONCAT function is:

```
CONCAT (string1_vlaue, string2_value);
```

In this syntax: string1_value is the string to which the string2_value has to be appended.

Listing 3.4 shows the PL/SQL program to demonstrate how this function works.

**Listing 3.4:** PL/SQL program to use CONCAT function

In Listing 3.4, we have declared two variables, nam and nam1. The nam1 variable has been initialized with the value 'Welcomes You'. After declaring variables, employee name has been fetched into the nam variable from the EMP table, where employee number is equal to 7844. Finally, we have applied the CONCAT function to add both the strings that are in variables nam and nam1.

***Note***

The CONCAT function appends the strings in the order they are passed as parameters.

The output Listing 3.4 is as follows:

**Output:**

```
TURNER Welcomes You
PL/SQL procedure successfully completed.
```

In the output, TURNER Welcomes You, the first string (TURNER) has been fetched from the table and the second one (Welcomes You) has been declared in the PL/SQL program. In this way, the CONCAT function combines the strings into one.

## *LOWER and UPPER Functions*

Both these functions accept a single character or a string as a parameter and convert it to lowercase or uppercase characters depending on the function being used. If you use the LOWER function, then all uppercase characters will be converted into lowercase characters, and if you use the UPPER function then all lowercase characters will be converted into uppercase characters.

***Note***

Syntax for the LOWER function is:

```
LOWER (char_value or string_value);
```

Syntax for the UPPER function is:

```
UPPER (char_value or string_value);
```

In these syntaxes:

- Char_value represents the single character whose case has to be changed.
- string_value represents the complete string passed to change the case.

The following listing demonstrate how the LOWER and UPPER functions work.

**Listing 3.5:** Using LOWER and UPPER functions

In Listing 3.5, first we have declared two variables nam and nam1 of the VARCHAR2 datatype. After declaring all variables, we have used the SELECT statement to fetch the DNAME (department name) into nam from the table DEPT, where DEPTNO (department number) is equal to 20. Then, we have applied the LOWER function on the string stored in the nam variable. and used the SELECT statement to fetch the DNAME into variable nam1 where DEPTNO is equal to 50. At last, we have applied the UPPER function on the string stored in the nam1 variable.

The output, when we execute the program shown in Listing 3.5, appears as follows:

**Output:**

In this outout, the first output (research) is in lowercase letters because of the LOWER function, whereas the second output (HUMAN RESOURCE) is in uppercase letters, which is the result of the UPPER function.

## INSTR Function

This function is used to find the position of a substring in the main string. This function returns zero in case the substring is not present in the main string.

Syntax for the INSTR function is:

```
INSTR (string1_value, string2_value, starting_position, nth_presence);
```

In this syntax:

- `string1_value` is the main string in which the position of the substring has to be found.
- `string2_value` is the substring whose position has to be searched in the main string.
- `starting_position` represents the position in main string from which the search has to be started. It is an optional argument. If you don't specify the `starting_position`, the search will start from the first character of the main string.
- `nth_presence` represents the nth position of the substring in the main string.

Listing 3.6 shows how the `INSTR` function works.

**Listing 3.6:** PL/SQL program to use `INSTR` function

In Listing 3.6, first we have declared a variable `plac` of the `VARCHAR2` datatype, and then we have fetched the `LOC` (department location) into that variable from the table `DEPT` where `DEPTNO` is equal to 20.

Now, `INSTR` function has been applied on the `plac` variable three times for three results:

- In the first instance, only two parameters—the main string and substring—have been passed. It will show the first occurrence of substring in the main string.
- Secondly, we have passed 4 parameters. This time we have added the position from where a search for the substring has to be started in the main string. Here the third and fourth parameters are 1,1, which means that the search has to be started from the first letter of the string and up to the first occurrence of the substring in the main string.
- At third instance, the parameter passed are same as the in second one except that here we are searching the substring up to its 2nd occurrence in the main string.

The output of Listing 3.6 is as follows:

**Output:**

```
DALLAS
2
2
5
PL/SQL
```

In the output, you can see that there are four results. The first result, `DALLAS`, shows the department location, which we use as the main string in this function. The second result, 2 indicates the first occurrence of the 'A' (substring) in the main string. The third result is same as

the second one because of the fourth parameter due to which the substring has been searched up to its first occurrence. In the fourth result, the position returned by this function is 5, which represents the second occurrence of 'A' in the main string.

### *functions*

are used to trim the string entered as a parameter. The LTRIM function trims the string from the left side by removing the character specified as parameter in this function. The RTRIM function, on the other hand, trims the string from the right side.

Syntax for the LTRIM function is:

```
LTRIM (string_value, string_trim);
```

Syntax for the RTRIM function is:

```
RTRIM (string_value, string_trim);
```

In both of the preceding syntaxes

- string_value represents the main string, which has to be trimmed.
- string_trim specifies the string that will be removed from the left side or right side of the main string depending on the function you use. In both the functions the second argument is optional and if you don't specify the second parameter then all leading (LTRIM function) or trailing (RTRIM function) spaces will be removed from the string.

The following listing shows how this function works.

**Listing 3.7:** Using LTRIM and RTRIM functions

In Listing 3.7, first, we have declared four variables, nam, d_nam, nam2, and d_nam1. The variable nam has been initialized with a string having some trailing blank spaces while the variable nam2 has been initialized with a string, which has leading blank spaces. After this, the DNAME has been fetched into the d_nam and d_nam1 variables from the table DEPT where DEPTNO is equal to 50. At last, we have applied the LTRIM and RTRIM functions on the strings in different variables.

The output of Listing 3.7 is as follows:



The output of Listing 3.7 shows four results. In the first result, you can see that all the trailing spaces have been removed from the string because of the `LTRIM` function.

In the third result you can see that all leading spaces have been removed by the `RTRIM` function.

In the second result, the department name `human resource` has been trimmed from the left side because of the use of the `LTRIM` function; and the fourth result shows that the string `human resource` has been trimmed from the right side.

### REPLACE Function

This function replaces all the occurrences of a substring with the replacement substring from the main string. This function is very useful when you need to search and replace a particular sequence of characters with any other sequence of characters.

Syntax for the `REPLACE` function is:

```
REPLACE (string_value, string_value_rep, rep_string);
```

In this syntax:

- `string_value` represents the main string.
- `string_value_rep` represents the string that has to be replaced in the main string.
- `rep_string` represents the string with which replacement has to be made. If the parameter rep_string is not specified then this function simply removes the `string_value_rep` character sequence from the main string.

Listing 3.8 shows how the `REPLACE` function works:

**Listing 3.8:** PL/SQL program to use `REPLACE` function



In Listing 3.8, we have declared a variable `plac` of the `VARCHAR2` datatype to store the `LOC` (location) of the department fetched from the table `DEPT` where `DEPTNO` is equal to 30. After declaring these variables, we have used the `SELECT` statement to fetch the `LOC`, and finally we have applied the `REPLACE` function on the string fetched from the table.

The output of Listing 3.8 is as follows:

**Output:**

```
CHICzGO
PL/SQL procedure successfully completed.
```

In the output of Listing 3.8, you can see that 'A' in CHICAGO has been replaced by 'z' and the output becomes CHICzGO. CHICAGO is the employee location that has been fetched from the EMP table.

### SUBSTR Function

The SUBSTR function fetches the substring from the main string. This function is the most widely used function in PL/SQL.

Syntax for the SUBSTR function is.

```
SUBSTR (string_value, start_pos, len_substr);
```

In this syntax:

- string_value represents the main string in which a substring has to be searched.
- start_pos represents the position in the main string from which the substring has to be extracted.
- len_substr represents the number of the characters to be fetched from the main string. The len_substr option is optional; if you don't specify it, this function returns the whole string from the starting position of the whole string.

Listing 3.9 shows how this function works.

**Listing 3.9:** PL/SQL program to use SUBSTR function

In Listing 3.9, we have declared a variable nam of the VARCHAR2 datatype, and then the ENAME (employee name) has been fetched into that variable from the table EMP where EMPNO=7844. At last, we have used the SUBSTR function on the fetched string.

The output of Listing 3.9 is as follows:

**Output:**

```
URNE
PL/SQL procedure successfully completed.
```

The string fetched from the table is TURNER, and according to the parameters passed in the function, it has to retrieve a substring starting from the position number 2 to 4 in the main string. In the output, URNE, characters U and E are the second and fourth position characters, respectively, of the string TURNER.

### *TRANSLATE Function*

The TRANSLATE function replaces a single character at a time. This function replaces a character set in a string with another set of characters. For example, if there are two character strings 123 and 456, this function will replace the 1 with 4, 2 with 5 and so on.

Syntax for the TRANSLATE function is:

```
TRANSLATE (string_value, string_value_rep, rep_string);
```

In this syntax:

- string_value shows the main string in which the character has to be searched and replaced.
- string_value_rep represents the string which has to be searched in string_value.
- rep_string represents the string with which the corresponding character of string_value_rep will be replaced.

Listing 3.10 shows how this function works.

**Listing 3.10:** Using TRANSLATE function



The output of Listing 3.10 is as follows:

**Output:**

```
1ABE1MAY
PL/SQL procedure successfully completed.
```

The string fetched from the table is SALESMAN. The strings passed in this function as parameters are 'SLN' and '1BY'. You can see in the output that S has been replaced by 1, L with B, and N with Y and we get the resultant string as 1ABE1MAY.

With this, we have completed discussion on the most commonly used character functions available in PL/SQL. Now, we continue our discussion with Date functions.

## Date Functions

The date functions allow you to manipulate dates. Suppose, you want to know the number of months between any two given dates or need to convert a date string in to a valid date format according to your requirement, then you must have to write a program for achieving the required results.

PL/SQL has made these operations simple with the help of built-in date functions. PL/SQL Built-in date functions accept date as a parameter and return a date or number as a result. PL/SQL provides various built-in date functions that help manage date/time for a database. Table 3.2 shows some important date functions.

**Table 3.2: Date Functions**

| Function | Description |
|---|---|
| TO_DATE | Converts a string into a date. |
| | Converts a date into a string. |
| | Adds n months in the month of the date passed as a parameter. |
| NEXT_DAY | Returns the next date of the day specified in date string. |

Table 3.2 presents some important date functions. Let's continue our discussion by explaining all these functions in detail.

### *TO_DATE Function*

The `TO_DATE` function accepts a string and a format parameter as arguments and converts the string into date according to the format parameter.

Syntax for the `TO-DATE` function is:

```
TO_DATE ('string_value', 'format_para');
```

In this syntax:

- `string_value` shows the string that needs to be converted into a date.
- `Format_para` represents the format parameter according to which the `string_value` will be converted into a date. The following table (Table 3.3) shows some of the format parameters. These parameters can be used in various combinations to get the desired result.

**Table 3.3: Important Format Parameters**

| Parameter | Description |
|---|---|
| YYYY | Represents all 4-digits of the year in the resultant date. |
| YY | Represents last 2-digit of the year in the resultant date. |
| MM | Represents the month in numeric form. For example, Jan as 01. |
| MON | Represents the month in abbreviated form. For example, July as Jul. |
| DD | Represents the day. |

Listing 3.11 shows how the TO_DATE function works:

**Listing 3.11:** PL/SQL program to use TO_DATE function

```
SQL>SET SERVEROUTPUT ON
DECLARE
e_dat VARCHAR2(20):='121588';
BEGIN
DBMS_OUTPUT.PUT_LINE(TO_DATE( e_dat,
END;
/
```

In Listing 3.11, the variable e_dat of the VARCHAR2 datatype has been declared. It is then initialized with the string '121588'. At last, the To_DATE function has been applied on this variable.

The output of Listing 3.11 is as follows:

**Output:**

```
15-DEC-88

PL/SQL procedure successfully completed.
```

In the preceding output, you can see that the date format of the specified string has changed.

### TO_CHAR Function

This function accepts a number or a date along with the format parameter and converts it in a string according to the format parameter specified as an argument.

Syntax for the TO_CHAR function is:

```
TO_CHAR (val1, 'format_para');
```

In this syntax:

- Val1 represents either the number or the date value.
- Format_para represents a format parameter according to which val1 will be converted into a string.

See the following listing to understand how the TO_CHAR function works.

**Listing 3.12:** PL/SQL program to use TO_CHAR function

In Listing 3.12, a variable dat of the DATE datatype and dat1 of NUMBER datatype have been declared. The variable dat1 has been initialized with some numbers. Then DOJ (employee's date of joining) has been fetched into the variable dat from the table EMPLOYEE, where E_NO=7840. Finally, we have used the TO_CHAR function to convert the date and number values into a string according to the specified format passed as a second parameter in this function.

The output of Listing 3.12 is as follows:

**Output**:

In the preceding output, you can see three results. The first results shows the date fetched from the EMPLOYEE table, the second result shows the date converted into a string, and the third result shows the number that has been converted into the string.

## ADD_MONTHS Function

This function returns a new date with the specified number of months added to the date specified as input parameter. This function accepts date and a number as parameters.

Syntax for the ADD_MONTHS function is:

```
ADD_MONTHS (val, n_mon);
```

In this syntax:

- val shows the date whose month has to be modified.
- n_mon represents the number that has to be added in the month of the val.

The following listing shows how this function works.

**Listing 3.13:** PL/SQL program to use ADD_MONTHS function

In Listing 3.13, first we have declared a variable dat of the DATE datatype, and then DOJ has been fetched in to the dat variable from the table EMPLOYEE, where E_NO=7842. At last, we have applied the ADD_MONTHS function on the date fetched from the table.

The output of Listing 3.13 is as follows:

**Output:**

The first result in the output of Listing 3.13 is the date fetched from the table, and the second result is the outcome of the ADD_MONTHS function. According to its functionality four months have been added to the date fetched from the table because the second parameter passed in this function was 4.

### *MONTHS_BETWEEN Function*

The MONTHS_BETWEEN function compares two dates passed as parameters and returns the number of months between the specified dates.

Syntax for the MONTHS_BETWEEN function is:

```
MONTHS_BETWEEN (val1, val2);
```

In the syntax for the MONTHS_BETWEEN function, the val1 parameter represents the first date and the val2 represents the second date. See Listing 3.14 to understand its working:

**Listing 3.14:** PL/SQL program to use MONTHS_BETWEEN function

In Listing 3.14, the variables dat and dat1 of the DATE datatype has been declared; then we have fetched DOJ (employees date of joining) into the variables dat and dat1 from the table EMPLOYEE, where E_NO is equal to 7840 and 7844 for the two variables respectively. At last, we have displayed the dates fetched from the table into those variables, followed by the dates passed as parameters into the MONTHS_BETWEEN function.

The output of Listing 3.14 is as follows:

**Output:**

In the preceding output, the first two values show the dates retrieved from the table and the third value shows the difference of the months between two dates retrieved from the table.

## *LAST_DAY Function*

This function accepts a date as a parameter and returns the last day of the month on the basis that parameter.

Syntax for the LAST_DAY function is:

```
LAST_DAY (val);
```

In this syntax:

- val represents the date value from which the last day of a month will be fetched.

Listing 3.15 shows how the LAST_DAY function works.

**Listing 3.15:** PL/SQL program to use LAST_DAY function

In Listing 3.15, first we have declared a variable dat of the DATE datatype, and then, we have fetched DOJ **(Date Of Joining)** into that variable from the table EMPLOYEE, where E_NO=7844. At last, we have passed the dat as parameter in the LAST_DAY function to know the last day of the month.

The output of Listing 3.15 is as follows:

**Output:**

In the preceding output, the first result shows the date retrieved from the table and the second result shows the date on the last day of that month, that is 31.

### NEXT_DAY Function

This function returns the next day in the date to the day specified in the date string. This function accepts a date and the weekday as parameters.

Syntax for the NEXT_DAY function is:

```
NEXT_DAY (val, w_day);
```

In the preceding syntax:

- Val represents the date string whose day value has to be changed.
- W_day represents the weekdays (Sunday to Saturday).

Listing 3.16 shows how the NEXT_DAY function works.

**Listing 3.16:** PL/SQL program to use NEXT_DAY function

In Listing 3.16, first we have declared a variable dat of the DATE datatype that has been used to store the date fetched from the table employee, where E_NO=7842. Now the date fetched into this variable would be displayed, on which the NEXT_DAY function will be applied.

The output of Listing 3.16 is as follows:

**Output:**

```
15-AUG-07
18-AUG-07

PL/SQL procedure
```

In the preceding output, you can see the date on which the next Saturday (second argument) will fall.

### SYSTIMESTAMP Function

This function returns the time including fractions of seconds and time zone according to your local database.

Syntax for the SYSTIMESTAMP function is:

```
SYSTIMESTAMP;
```

This function does not require any argument and its output includes date, time with fraction of seconds, and time zone.

Listing 3.17 shows how the `SYSTIMESTAMP` function works:

**Listing 3.17:** PL/SQL program to use `SYSTIMESTAMP` function



The output of Listing 3.17 is as follows:

**Output:**



With this we complete discussion on most commonly used date functions. Let's continue our discussion by explaining numeric functions.

## Numeric Functions

Numeric functions help you perform mathematical calculations, such as the total salary of employees, and counting the number of records in a database. These functions accept a number as a parameter and return a number as the output. PL/SQL provides many built-in numeric functions some of which have been explained in Table 3.4.

**Table 3.4: Numeric functions**

| Function | Description |
|---|---|
| ABS | Returns the absolute value of a number. |
|  |  |
|  |  |
| POWER | Used to raise the base number by the nth number specified. |
|  |  |
|  |  |
| | Returns the remainder of one number divided by another number. |

| Table 3.4: Numeric functions | |
|---|---|
| **Function** | **Description** |
| **COUNT** | **Returns the number of rows in a table on the basis of a query.** |
| **SUM** | **Calculates the sum of all the values in an expression.** |
| **SIGN** | **Used to know the sign of a number.** |

Table 3.4 gives a brief description about some of the important numeric functions. Let's study them in detail.

### *ABS Function*

The `ABS` function accepts a single number as a parameter and returns its absolute value. For example, if you want to know the absolute value of -15.45, then this function will return 15.45.

Syntax for the `ABS` function is:

```
ABS (val);
```

In this syntax:

- `val` represents the number whose absolute value has to be calculated.

The following listing shows how the ABS function works.

**Listing 3.18:** PL/SQL program to use `ABS` function



In Listing 3.18, we have passed one positive integer value and one negative integer value in the ABS function.

The output of Listing 3.18 is as follows:

**Output:**



### *CEIL and FLOOR Functions*

The `CEIL` function returns the smallest integer value greater than or equal to the value specified as a parameter; and the `FLOOR` function returns the largest integer value less than or equal to the value specified as a parameter. Both, the `CEIL` and `FLOOR`, functions accept a single parameter of the number datatype and work in the same manner.

Syntax for the CEIL function is:

```
CEIL (val);
```

Syntax for the FLOOR function is:

```
FLOOR (val);
```

In both syntaxes, val represents the number parameter.

Listing 3.19 demonstrates how these functions work.

**Listing 3.19:** Using CEIL and FLOOR functions

The output of Listing 3.19 is as follows:

**Output:**

## *POWER Function*

This function takes two values as parameters—a base number and the other number as the power to the base number. The POWER function returns the base value after calculating its power according to the other argument passed in this function.

Syntax for the POWER function is:

```
POWER (b_num, p_num);
```

In this syntax:

- b_num represents the base number.
- p_num represents the power number with which the base number has to be powered.

Listing 3.20 shows how the POWER function works.

**Listing 3.20:** PL/SQL program to use POWER function

The output of Listing 3.20 is as follows:

**Output:**

```
1000
9924.36543
PL/SQL procedure
```

### ROUND Function

The ROUND function accepts either one or two parameters. First parameter is compulsory and is the number whose value has to be rounded by some specific decimal points specified as the second parameter.

Syntax for the ROUND function is:

```
ROUND (r_num, d_num);
```

In this syntax:

- r_num represents the number which has to be rounded. This parameter is compulsory.
- d_num represents the decimal places according to which r_num will be rounded. It is an optional parameter, and if you don't specify this parameter, then the ROUND function deletes the numbers after decimal point and returns the output.

Listing 3.21 shows how the ROUND function works.

**Listing 3.21:** PL/SQL program to use ROUND function

The output of Listing 3.21 is as follows:

**Output**:

```
16
16.256

PL/SQL procedure
```

In the preceding output, there are two results shown. In the first result, all the numbers after decimal have been removed because we have not specified the second parameter; and therefore, it has deleted numbers after the decimal.

In the second result, there are only three numbers after the decimal point; it is because the function has retained the numbers up to third place after decimal, as specified in the second parameter.

### SQRT Function

The SQRT function accepts a number value as a parameter and returns its value after doing square root of the value passed as the parameter.

**Note**

This finction never accepts a negative value.

Syntax for the SQRT function is:

```
SQRT( s_num);
```

In this syntax:

- s_num is the number value whose square root has to be calculated. This number value should be zero or any positive number.

See the Listing 3.22 to understand how it works.

**Listing 3.22:** PL/SQL program to use SQRT function

The output of Listing 3.22 is as follows:

**Output:**

## MOD Function

The MOD function accepts two parameters—the first number as a dividend and the second number as the divisor. This function then performs calculation on the values passed as parameters and returns the remainder as the result.

Syntax for the MOD function is:

```
MOD (d_num1, d_num2);
```

In this syntax:

- d_num1 represents the dividend.
- d_num2 represents the divisor.

Listing 3.23 shows how this function works.

**Listing 3.23:** PL/SQL program to use MOD function

The output of Listing 3.23 is as follows:

**Output:**

```
1
1.625

PL/SQL procedure
```

## COUNT Function

This function helps you calculate the total number of rows in a table based on a particular query. For example, if you want to know the total number of employees in a table or the number of employees having salary more than 10,000, then this function can be used for retrieving the results easily.

Syntax for the `COUNT` function is:

```
SELECT COUNT (col_name or *) FROM tab_name WHERE condition;
```

In this syntax:

- `COUNT (col_name or *)` is the method with `col_name` as its argument. The `col_name` represents the column name in which the count of the total number of rows filled with some values has to be found out. This function does not count a blank row. The * can be used when you want to calculate the total number of rows in the table with almost one filled entry in every row.

Take a case for example: All the employees in an organization have been asked to submit their cell phone numbers by using their online accounts until a specified date. After the specified date, the HR department wants to know the number of employees that have given there cell phone numbers. In that case, you have to specify the column name (in bracket of `COUNT` function) representing the cell number in the table. This function then counts the number of rows having cell phone numbers and returns the value.

- `tab_name` represents the table name.
- `Condition` represents a specific value on the basis of which the counting will be done.

See Listing 3.24 to understand this function.

**Listing 3.24:** PL/SQL program to use `COUNT` function

In Listing 3.24, we have used * with the count function to retrieve the total number of employees in the table EMP.

output of Listing 3.24 is as follows:

### Function

SUM function allows you to get the sum of the values specified in the expression as a Syntax of this function is almost same as the COUNT function except that here * does work.

Syntax for the SUM function is:

```
SELECT SUM (col_name) FROM tab_name WHERE condition;
```

In this syntax:

- SUM (col_name) specifies the column name (col_name as shown in bracket) whose row's sum you have to calculate.
- tab_name represents the table name.
- condition specifies a specific value on the basis of which the sum will be calculated. For example, if you want to calculate the sum of salaries of those employees who receive 10,000 as salary, then you can specify this condition with WHERE clause.

Listing 3.25 shows the working of the SUM function.

**3.25:** PL/SQL program to use SUM function

In Listing 3.25, we have used the SAL (employee salary) column name with the SUM function to the total salary of the employees in the table EMP.

output of Listing 3.25 is as follows:

### Function

SIGN function accepts a number value as a parameter and returns its sign. For example, it returns 1 if the number is more than zero; returns zero, if the number is equal to zero; and returns -1, if the number is less than zero.

Syntax for the SIGN function is:

```
SIGN (val);
```

In this syntax, val represents the number value.

Listing 3.26 shows how this function works.

**Listing 3.26:** PL/SQL program to use SIGN function



In the Listing 3.26, first we have declared a variable e_sign of the NUMBER datatype, which has been used to store the E_SAL (employee salary) column fetched from the table EMPLOYEE, where E_NO=7844. We have then passed the employee salary retrieved from the table into the SIGN function. At last, we have passed a negative value in this function to check the result.

The output of Listing 3.26 is as follows:

**Output:**



By this we have finished discussion on numeric functions. Now, we continue our discussion with conversion functions.

## Conversion Functions

The conversion functions are very important feature of a programming language. These functions are used to convert one datatype into another datatype.PL/SQL provides lots of conversion functions that can be used explicitly in a PL/SQL program. PL/SQL also performs implicit conversions when you do not perform conversion explicitly. You should perform explicit conversions, wherever possible, to get the desired output, because sometimes implicit conversion does not give the desired output. Table 3.5 shows some important conversion functions.

**Table 3.5: Conversion functions**

| Function | Description |
|---|---|
| CONVERT | It converts a string from one character set to another character set. |
| TO_NUMBER | It converts a string to a number. |

Let's study these functions in detail.

### CONVERT Function

The CONVERT function can accept 2 or 3 parameters, a string value, new character format, and old character format. The third parameter is optional. If the third parameter is not specified than this function uses the default character set.

Syntax for the CONVERT function is:

```
CONVERT (val, n_charset, o_charset);
```

In this syntax:

- val represents the string value which has to converted from one character set to another character set.
- n_charset represents the character set in which the string value has to be converted.
- o_charset represents the character set of the val string value. This is an optional parameter. Table 3.6 shows of the character sets.

**Table 3.6: Character set**

| Character set | Description |
|---|---|
| US7ASCII | US 7 Bit ASCII character set. |
| EL8GCOS7 | Greek 8 Bit Character set. |
| WE8DEC | West European 8 Bit Character Set. |
| UTF 8 | Universal Character Set. |
| WE8HP | West European 8 bit HP Laser jet Character set. |
| NE8ISO8859P10 | North European ISO 8859-10 Character set. |
| WE8PC850 | West European IBM PC code page-500 8 Bit Character set. |
| WE8ISO8859P1 | West European ISO 8859-1 8 Bit Character set. |
| EE8ISO8859P2 | East European ISO 8859-2 Character set. |

Listing 3.27 shows how the CONVERSION function works.

**Listing 3.27:** PL/SQL program to use CONVERT function

```
END;
/
```

In Listing 3.27, the variables `con` and `con1` of the `VARCHAR2` datatype have been declared. The variable `con1` has been initialized with a string, and then the `ENAME` (employee name) has been retrieved into the variable `con` from the table `EMP`, where `EMPNO=7900`.

After declaring variables and retrieving the `ENAME`, we have used the `CONVERT` function to convert the character set of value passed as parameter with the new character set.

The output of Listing 3.27 is as follows:

**Output:**

In this output, you can see two results. First result shows the employee name that has been fetched from the database but it is same as in the database. Employee name has not been converted because the default character set may follow the same convention as the character set specified as parameter.

Second result has been changed according to the new character set passed as third parameter.

## TO_NUMBER Function

The `TO_NUMBER` function accepts a string and a format (according to which the string has to be converted in a number) as a parameter and returns a number value equivalent to the entered string.

Syntax for `TO_NUMBER` function is:

```
TO_NUMBER (val, s_format);
```

In this syntax:

- `val` represents the string that has to converted in to a number.
- `s_format` represents the format according to which the string will be converted in to number.

The following listing shows how the `TO_NUMBER` function works.

**Listing 3.28:** PL/SQL program to use `TO_NUMBER` function

In Listing 3.28, the variables con and con1 of the VARCHAR2 datatype have been declared. The variable con 1 has been initialized with a string. Then the HIREDATE has been retrieved into the variable con from the table EMP, where EMPNO=7900. Finally the TO_NUMBER function has been used on those values.

The output of Listing 3.28 is as follows:

**Output:**



After discussing the important conversion functions, let's continue our discussion with another built-in PL/SQL function, the Large Object (LOB) functions.

## LOB Functions

Sometimes, you may require to store files, such as image files, video files, of very large size, in the database. Now the question arises how you can store such a large file in the database table?. PL/SQL provides you with the function called LOB that allows you to store files up to size of 4GB into the database. LOB has two parts, one is LOB locator and another is LOB value.

A database table stores the LOB locator which is in fact a pointer to the actual location on LOB value. Table 3.7 shows description of some LOB functions.

**Table 3.7: LOB Functions**

| Function | Description |
|---|---|
| BFILENAME | Initializes BFILE large object column in the database table. |
| EMPTY_BLOB | Returns empty locator of BLOB type. |
| EMPTY_CLOB | Returns empty locator of CLOB type. |

Let's study these LOB functions in detail.

### *BFILENAME Function*

The BFILENAME function is used to initialize the column of large object type in a database table to point an external file or it returns the BFILE locator for the LOB binary file. This function accepts two values as parameters namely, the directory name and the file name.

Syntax for the BFILENAME function is:

```
BFILENAME (dir_name, f_name);
```

In this syntax:

- dir_name represents a DIRECTORY object that serves as an alias and stores the location of the file.

- f_name represents the file name that contains the large object.

See the following example to understand how BFILENAME function works:

Before using directory alias in the BFILENAME function, you must be sure that the directory alias exist, or you need to create it before using it. To create a directory alias, you must have DBA privileges. So to create directory alias login as SYSDBA and then follow the syntax given here:

```
CREATE DIRECTORY 'name' AS 'location';
```

In this syntax, the name parameter represents the directory alias name or the object name and the location shows the path where the large object binary files will be stored. Let's see how to create and use directory alias with BFILENAME function:

### Creating a Directory

```
SQL> CREATE DIRECTORY pic AS 'E:\picture';
```

In the preceding statement, we have replaced the attributes name and location with pic and E:\picture respectively. After specifying the name and location, press the ENTER key. You will see the following result.

**Result:**

```
Directory Created.
```

Now, see the Listing 3.29 to use this directory alias with BFILENAME function.

**Listing 3.29:** Using BFILENAME function

In the Listing 3.29, first we have declared and initialized a variable pict of datatype BFILE. After that the INSERT statement has been used to add the employee picture in the table EMPLOYEE. At last, we have used update statement to change the employee's existing picture with the new picture.

The output of Listing 3.29 is as follows:

**Output:**

```
PL/SQL procedure successfully completed.
```

## *EMPTY_BLOB and EMPTY_CLOB Functions*

These functions initialize a BLOB and a CLOB column in the database table to "empty". Column in the table with BLOB or CLOB datatype can not be set as NULL. It must contain a locator that might point to empty or filled LOB. Before you start working with BLOB or CLOB, either to assign it a value in PL/SQL program or to use with SQL INSERT and UPDATE statements, set BLOB or CLOB column to "empty".

Besides initializing BLOB or CLOB, these functions also return empty locator of type BLOB and CLOB respectively. Both of these functions can be used with empty pair or parentheses or without parentheses.

Syntax for the EMPTY_BLOB function is:

```
EMPTY_BLOB or EMPTY_BLOB ();
```

Syntax for the EMPTY_CLOB function is:

```
EMPTY_CLOB or EMPTY_CLOB ();
```

See Listing 3.30 to understand how to set a LOB column to "empty" and how to use these functions with SQL statements.

**Listing 3.30:** Using EMPTY_BLOB and EMPTY_CLOB functions

In Listing 3.30, first we have declared two variables of the datatype BLOB and CLOB respectively, which are also initialized as empty LOB. After declaring the variables, we have inserted these variables in to the database table named EMPLOYEE.

Here you can see the output of Listing 3.30.

**Output:**

```
PL/SQL procedure successfully completed.
```

With this we have completed discussion on LOB functions. let's now discuss miscellaneous functions.

## Miscellaneous Functions

Miscellaneous functions are those functions that do not fall into the category of a particular datatype, for example, character function falls in the category of character data type, and date functions come under the date datatype category. Let's study various types of miscellaneous functions. Table 3.8 shows description of some of the miscellaneous functions.

| Table 3.8: Miscellaneous functions | |
|---|---|
| **Function** | **Description** |
| **LEAST** | |
| **USER** | **Returns the name of the current account.** |
| **UID** | **Returns the id of the current account.** |

Let's discuss these functions in detail and learn how to use them in a PL/SQL program.

### *GREATEST and LEAST functions*

These functions work in the same way but in opposite manner. The `GREATEST` function fetches the highest value from the list passed as an argument, whereas the `LEAST` function fetches the lowest value. Both these functions accept two or more than two values as arguments. There is no limit to pass values in both these functions.

Syntax for the `GREATEST` function is:

```
GREATEST (val1, val2, valn);
```

Syntax for the `LEAST` function is:

```
LEAST (val1, val2, valn);
```

In these syntaxes, `valn` represents the nth value passed. It implies that you can pass as many value as you want in this functions. Now, see the following listing to know how these functions work.

**Listing 3.31:** Using `GREATEST` and `LEAST` functions

In the Listing 3.31, first the `e_dat` variable of the `DATE` datatype has been declared, followed by fetching `HIREDATE` into that variable from the table Employee. After that the `GREATEST` and `LEAST` functions have been applied on the values fetched from the table.

The output of the Listing 3.31 is as follows:

**Output:**

The preceding output displays six results. The first result shows the system date; second result shows the date fetched from the table; third result shows the highest value among the lists of the values passed in the GREATEST function as arguments; fourth result shows the highest date among the dates passed in the GREATEST function; fifth result shows the lowest value among the lists of the values passed in the LEAST function; and the sixth result shows the lowest date among the dates passed in the LEAST function.

## *USER and UID Functions*

Both of these functions work in the same manner. The USER function is used to know the name of the current account, such as scott or sys, whereas the UID function returns the integer value to identify the user. Both these functions do not require any argument; it is for this reason that they look like a variable rather than a function.

Syntax for the USER function is:

```
USER;
```

Syntax for the UID function is:

```
UID;
```

Listing 3.32 shows you how these functions work.

**Listing 3.32:** Using USER and UID functions

In Listing 3.32, we have declared two variables the u_ser, and u_id of the VARCHAR2 and NUMBER datatype respectively. After that the SELECT statement has been used to retrieve the user name and the user id into u_ser and u_id respectively from the table EMP, where EMPNO = 7900.

The output of Listing 3.32 is shown as follows:

**Output:**

With this we conclude our discussion on the built-in functions provided in PL/SQL. You will now be comfortable with using these functions. Let's summarize what we have studied in this chapter.

## Summary

In this chapter, we have studied:

- The character functions, their types, and examples to demonstrate their working.
- The date functions and their various types, supported date formats, and various examples to show how they work.
- The numeric functions and their types along with examples.
- The conversion functions
- The LOB functions
- Miscellaneous functions.

In real life, mostly you need to perform an operation based on a condition. For example, you need to calculate income tax for employees of an organization. For this, you must apply different formulas for calculating income tax based on their different salaries and different filing status. Usually, execution of a PL/SQL program proceeds from top to bottom except no control structure comes in between. Similar to other high level structured languages C, C++ or Java, PL/SQL also provides various control structures to deviate the sequential flow of execution of PL/SQL program depending upon the conditions, execution part of PL/SQL program repeatedly till a specific condition is satisfied, and unconditionally jump to different parts of PL/SQL program. In other words, these control structures organize the flow of execution of a PL/SQL program. This chapter discusses all types of control structures and explains each of them in details with examples. Let us study various control structures available in PL/SQL.

## Describing PL/SQL Control Structures

Control structures are one of main PL/SQL extensions to SQL. Control structures lead to well-structured programming which helps developers to structure the flow of control through a PL/SQL program. Each control structure has only one entry and exit point. Control structures are broadly divided into three main categories:

- Conditional control structures
- Iterative control structures
- Sequential control structures

Let us explain them with the help of flowchart diagrams.

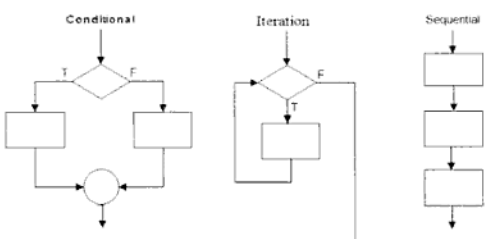



**Fig.PL/SQL- 4.1**

Fig.PL/SQL-4.1 consists of one flowchart corresponding to each control structure. Each rectangular box in these flowcharts represents a set of PL/SQL statements. The selection structure starts from testing a condition and executes only one set of statements according to the value of condition (T or F). Usually, condition is either Boolean variable or an expression which results into Boolean value. The iteration structure executes the set of statements repetitively until condition evaluates to true. The sequential control structure executes the set of statements sequentially.

These structures are represented by the set of PL/SQL statements such as `IF-THEN-ELSE`, `CASE`, `FOR-LOOP`, `WHILE-LOOP`, and `GOTO`. Let us now study Conditional Control Statements.

## Using Conditional Control Statements

Conditional control statements facilitate us to take alternative actions depending upon conditions. Conditional control statements include IF-THEN, IF-THEN-ELSE, IF-THEN-ELSEIF, and CASE statements. Three IF statements execute a different sequence of statements depending upon the value of condition. The CASE statement is nothing but simpler way of writing IF-THEN-ELSEIF statement. The CASE statement also executes efficiently in comparison to IF-THEN-ELSEIF statement. Let us now study each statement in detail.

### *IF-THEN Statement*

The IF-THEN statement is a basic IF statement. This statement executes a set of statements when a condition is true. The general syntax of IF-THEN statement is as follows:

In this syntax, the condition between IF and THEN either evaluates to TRUE or FALSE. If it evaluates to true then only the block of statements between THEN and END will execute.

If condition involves Boolean variable, we can simply place the boolean variable in between IF and THEN keywords as it itself is either TRUE or FALSE.

Let us create an example which calculates how many hours an employee spent during overtime. See Listing 4.1 which shows the use of IF-THEN statement.

**Listing 4.1:** Using IF-THEN statement

Declaration section declares and initializes two variables HoursWorked and OverTime of NUMBER data type. The IF statement checks whether the number of working hours is (denoted by HoursWorked variable) greater than defined standard for number of working hours (8) and then, proceeds to calculate OverTime if condition is TRUE. The PUT_LINE method of DBMS_OUTPUT package is used to display OverTime on *iSQL Plus.

Here you can see the output of the Listing 4.1.

**Output:**

```
Hours spent in overtime = 4
PL/SQL procedure successfully completed.
```

After executing Listing 4.1on *iSQL Plus console, you will get the output as 'Hours spent in overtime = 4'. This message contains number of hours (4) spend in overtime by a person. Upon each successful execution of any PL/SQL program, PL/SQL engine also displays PL/SQL procedure successfully completed message.

### IF-THEN-ELSE Statement

The IF-THEN-ELSE statement's functionality is similar with either/or English language sentence. The IF-THEN-ELSE statement is little complex than IF-THEN statement. It also executes another sequence of statements if condition evaluates to FALSE. The syntax of IF-THEN-ELSE statement is as follows:



keywords are executed based on condition's value (TRUE or FALSE).

We now extend the same example as covered earlier in section IF-THEN statement by including ELSE clause. See Listing 4.2 which shows use of IF-THEN-ELSE statement.

**Listing 4.2:** Using IF-THEN-ELSE statement



Here, ELSE clause means that employee has not worked for overtime in an organization. In the preceding listing, Zero value of OverTime variable represents this.

Here you can see the output of the Listing 4.2.

**Output**:

```
Person does not work for overtime
PL/SQL procedure successfully completed.
```

After execution Listing 4.2 on SQL *plus console, you will get the output as Person does not work for overtime in the console. This output comes since HoursWorked variable initialized to 8 which is standard for number of working hours in a day.

### IF-THEN-ELSEIF Statement

It is the the most complex `IF` statement out of available three IF statements. It is used when a PL/SQL program involves more than two conditions. For example, you can use this statement to create menu which have many options but use of this statement is deprecated. As this statement makes code complex and also does not execute efficiently. The general syntax of `IF-THEN-ELSEIF` statement is as follows:

In this syntax, all conditions such as condition1, `condition2` are mutually exclusive which means more than one condition cannot be true at a time. Here, block of statements associated with `TRUE` condition are executed. You can avoid using this complex `IF` statement since from Oracle 9i Database Release 1, another searched `CASE` statement introduced which performs same purpose.

We are again extending the example to include `ELSE IF` clause. See Listing 4.3 which shows use of `IF-THEN-ELSE IF` statement.

**Listing 4.3:** Using `IF-THEN-ELSE IF` statement

In this listing, we use two different SQL commands, `PROMPT` and `ACCEPT`. The `PROMPT` command here used to print name of program on iSQL*Plus console. The `ACCEPT` command asks user to input the value of variable `TotalHours`. Declaration section assigns value input by user to `HoursWorked` variable and also declares a `LessTime` variable of `NUMBER` type. The

ELSE IF clause of this program will execute and display the value of LessTime variable if an employee works for hours less than 8 and leave office earlier due to some valid reasons.

Here you can see the output of the Listing 4.3.

**Output:**



After executing Listing 4.3 on *iSQL Plus console, it asks user for number of hours. In this example, let's enter value 4 and press Enter key to execute remaining part of Listing 4.3. The output displays the text written in prompt which just shows the purpose of program. The next two lines in output shows that second line of Listing 4.3 now initializes the HoursWorked variable with value entered by user. Main message in output is Person leave office earlier by 4 hours which contains the number of hours (4) by which person leave office earlier.

## *Using CASE Statements*

CASE statements let you to choose one set of statements to be executed out of many sets of statements. Oracle 9i database release 1 and its upper versions to support CASE statements. Let's discuss different CASE statement in detail.

### Simple CASE Statement

A simple CASE statement consists of an expression and blocks of PL/SQL statements; where each block is associated with a different value. The values of the blocks are specified in the WHEN clauses. The expression in the CASE statement may also contain function calls. Expression's value should be of the CHAR, VARCHAR2, or INTEGER types. It cannot be of the BLOB, BFILE, object type, record, or VARRAY data types. When you execute a given expression, the expression's value is compared with the values specified in the WHEN clause. Then, the set of statements associated with the matched value are executed.

The ELSE clause is optional in CASE statements and is executed when no WHEN clause did not execute. If you do not specify the ELSE clause, PL/SQL inserts the ELSE RAISE CASE_NOT_FOUND clause into the CASE statement by default. This clause raises the predefined Exception CASE_NOT_FOUND.

Let's create an example which explains the meaning of grade symbols to student by using CASE statement. See Listing 4.4 which shows the use of CASE statement.

**Listing 4.4:** Using simple CASE statement

The grade variable, which can store single character, stores the uppercase character entered by student. When PL/SQL engine comes across CASE statement, it evaluates grade variable and compares it with values mentioned in WHEN clause till a match is found. Then, statement associated with matched WHEN clause executes and displays the meaning of grade entered by a student.

Here you can see the output of the Listing 4.4.

**Output:**

After executing Listing 4.4 on *iSQL Plus console, it asks student to enter a grade in uppercase. In this example, let's enter grade A and press Enter key to execute remaining part of Listing 4.4. Output displays the text written in command PROMPT which just shows the purpose of program. Main message in output is Grade A means Excellent Performance. This message interprets the meaning of grade A to student in terms of performance.

***Note***

CASE statements can also be labeled and label appears before CASE keyword of CASE statement.

### Searched CASE Statement

The searched CASE statement consists of blocks of PL/SQL statements and each block is associated with one boolean expression. Block of statements associated with expression which results into TRUE value is executed. Note that searched CASE statement does not include any expression, which comes immediately with CASE keyword in case of simple CASE statement.

Let's perform the same example of grading student's performance but here student will enter his marks and corresponding conditions in WHEN clauses are also different. See Listing 4.5 which shows use of searched CASE statement.

**Listing 4.5:** Using searched CASE statement

We assumed that student can get maximum of 100 marks. Note the difference between conditions mentioned in `WHEN` clauses in Listing 4.5 and those in `WHEN` clauses of Listing 4.4. These conditions are Boolean expressions made up of using relational (>=, <=) and logical operators (and). In searched `CASE` statement, condition evaluation also happens in `WHEN` clause.

Here, you can see the output of the Listing 4.5.

**Output:**

After executing Listing 4.5 on *iSQL Plus console, it asks student to enter marks represented by SMarks variable. In this example, let's enter the value 80 and then press Enter key to continue the execution of remaining portion of Listing 4.5. Output displays the text written in command `PROMPT` which just shows the purpose of program. Main message in output is `Student attained Grade A` on the basis of his marks.

***Note***

This section demonstrates the use of searched `CASE` statement in case when there are more than three conditions present in logic of a PL/SQL program.

## Using Sequential Control Statements

We have seen control structures which allow set of statements to execute only when some condition is `TRUE`. PL/SQL also supports two more statements, `GOTO` and `NULL` which do not need any condition to execute themselves. Let us discuss them in detail.

## GOTO Statement

The GOTO statement used to transfer the flow of execution in PL/SQL program from one statement to another statement directly without any condition. The syntax of GOTO statement is as follows:

```
GOTO labelname;
```

labelname is the name of label present inside PL/SQL program. The labels are created by using syntax <<labelname>>. In this syntax, surrounding angle brackets act as label delimiters.

We take an example of checking whether entered number is prime number or not to show you the use of GOTO statement. See Listing 4.6 for use of GOTO statement.

**Listing 4.6:** Using GOTO statement

The ACCEPT command asks user to enter a number of INTEGER data type. Declaration section assigns value entered by user to n variable and also declares a variable msg of VARCHAR2 type. The execution block checks if any of divisions of given number by 2 to integer (less than rounded value of Square root of given number) returns any reminder. If remainder is 0, then the number is not prime number. The MOD function is used for getting the remainder. In case of non-prime number, GOTO statement will execute that transfer control to label named display. Next statement to this label outputs the message msg. Here you can see the output of the Listing 4.6.

**Output:**

After executing Listing 4.6 on *iSQL Plus console, it asks user to enter a number. In this example, let's enter value 991. Output displays the text written in prompt which just shows the purpose of program. Main message in output is written as '991 is a prime number'.

Now, check Listing 4.6 for non-prime number. Here you can see the second output of Listing 4.6.

After executing Listing 4.6 again, let's enter non-prime number such as 990 and press ENTER key. You will get `990 is NOT a prime number` as output.

There are some limitations of GOTO statement which are as follows:

- Cannot transfer control to label present inside the nested block.
- Cannot transfer control from outside an IF statement to a label inside this IF statement.
- Cannot transfer control from an IF statement to a label inside another IF statement.
- Cannot be used in EXCEPTION block to transfer control to any part of PL/SQL program.

### *NULL Statement*

The NULL statement itself does nothing but transfer control to next statement. In case of certain statements such as IF and EXCEPTION block, there should have at least one executable statement for compilation, we can use NULL statement here. Note that NULL statement and boolean value NULL are not similar. They are used in different contexts.

Let's create an example of increasing percentage of commission of an employee having designation Sales Person and employee id equals to 102. For other employees, we need not to perform any action. See Listing 4.7 for the use of NULL statement.

**Listing 4.7:** Using NULL statement

Declaration section declares a Job variable of VARCHAR2 type and EmpId variable of NUMBER type. We initialize EmpId to 102 on which basis the SELECT INTO query is made. After getting designation of employee into Job variable, IF statement checks whether designation is Sales Person. The UPDATE query inside IF statement multiplies commission percentage (denoted by comm) by 2. The ELSE clause meant for other employees, we use NULL statement here to show no action.

This example operates on a bonus table having one following record.

```
ename       Job           sal    comm   Empid
Gaurav      Sales Person  20000  10     102
```

Here you can see the output of the Listing 4.7.

**Output:**

```
PL/SQL procedure successfully completed.
```

After executing Listing 4.7 on *iSQL Plus console, you will get message 'PL/SQL procedure successfully completed'. Execute the following query to see whether the record is updated or not:

```
SELECT * FROM bonus;
```

Here is bonus table after execution of the query. Note that the value of percentage of commission (denoted by comm field) changed from 10 to 20.

The bonus table after executing Listing 4.7 on iSQL*Plus console will be as follows:

```
ename       Job           Sal    comm   empid
Gaurav      Sales Person  20000  20     102
```

Here is another code snippet which uses NULL statement with EXCEPTION block:

In this syntax, when divide by zero exception occurs, rollback operation is performed. The ROLLBACK statement cancels the effect of previous transaction. For any other type of exceptions, we use NULL statement which means do nothing for these exceptions.

## Using Looping Constructs in PL/SQL

PL/SQL supports various types of loops to iterate portion of program for many times based on some condition. However, as a best practice, you must write the type of loop best suited for a specific requirement. Different types of LOOP statements are as follows:

- LOOP
- FOR-LOOP
- WHILE-LOOP

Let us discuss each LOOP statement in detail.

### *LOOP Statement*

The LOOP statement is a simple type of loop and specifies that the specified part of the program is iterated till the specified condition evaluates to true. This statement begins with the LOOP keyword and ends with the END LOOP statement. The general syntax of the LOOP statement is as follows:

In this syntax, set of statements are executed and control again transfers to the beginning of loop.

You can use other clauses with the `LOOP` statement to specify conditional iteration of the program block. For example, you can use the `EXIT` and `EXIT-WHEN` clauses to specify the given condition, as described next.

### EXIT Statement

The `EXIT` statement used to finish loop without any condition. When this statement is executed, loop finishes and control transfers to next statement after the loop. Note that the `EXIT` statement only works inside a loop.

### EXIT-WHEN Statement

The `EXIT-WHEN` statement also finishes the loop but when specific condition is encountered. When this statement is executed, condition inside `WHEN` clause is checked. If this condition is true, loop finishes and control transfers to next statement after the loop. There should be one statement present inside the loop to change the value of condition. We can also use IF statement in place of `EXIT-WHEN` statement as explained in following example:

```
IF num > 100 THEN EXIT; ENDIF;
EXIT WHEN num > 100;
```

The `EXIT-WHEN` statement is simpler than `IF` statement in this example but both perform same operations.

We take an example of inserting complaint ids to `complaints` table using `LOOP` statement. See Listing 4.8 which uses `LOOP` statement.

**Listing 4.8:** Using `LOOP` statement

The `INSERT` query inserts the following in every iteration of the loop:

- `ComplaintId`, which is same as the loop counter
- Description, which represents the string `value to be added later`

If the value of i is greater than the entered value, the `EXIT WHEN` statement completes the loop.

Here you can see the output of Listing 4.8.

**Output:**

After executing Listing 4.8 on *iSQL Plus, the console asks user to enter a number on which LOOP statement completes. In this example, let's enter the value 3 and press Enter key. Output displays the text written in command PROMPT which just shows the purpose of program. You can execute the following query on *iSQL Plus console to determine whether loop adds three rows into complaints table or not:

```
SELECT * FROM complaints;
```

The complaints table after execution has following rows:

This section illustrates the use of LOOP, EXIT and EXIT- WHEN statements.

## FOR LOOP Statement

The FOR LOOP statement iterates the portion of PL/SQL program for known number of times. The general syntax of FOR-LOOP is given as follows:

In this syntax, the FOR-LOOP statement uses double dot operator which is used to specify the range of counter variable between lowerbound and upperbound. This loop iterates the set of statements by upperbound - lowerbound number of times. Both lowerbound and upperbound of loop should be any out of literals, variables and expressions resulting into numbers. Otherwise PL/SQL arise VALUE_ERROR predefined exception. PL/SQL coverts both bounds to the values of PLS_INTEGER data type.

Note that you cannot change the counter inside the body of FOR loop as you are not allowed to reference counter in the body of loop.

We now perform the addition of squares of numbers from 1 to n using FOR loop. See Listing 4.9 which illustrates the use of FOR loop.

**Listing 4.9:** Using FOR loop

The ACCEPT command asks user to enter the value of n. Declaration section assigns the value entered by user to n variable and also declares a msg variable of VARCHAR2 type. The execution block contains a FOR LOOP. In every iteration of this loop, square of number is calculated and added to msum variable. After calculating the sum of squares of integers from 1 to n, we display this sum by using PUT_LINE procedure.

Here you can see the output of the Listing 4.9.

**Output:**

After executing Listing 4.9 on *iSQL Plus console, the console asks the user to enter the value of n. In this example, let's enter the value 20 and press Enter key. First line of output is the written in command PROMPT which just shows the purpose of program. Main message in output is Sum of squares of first 20 numbers = 2870 which shows the sum of 1*1+2*2..... + 20*20 series.

Let's consider an example which explains the scope of loop counter variable. See Listing 4.10 for this purpose.

**Listing 4.10:** Accessing loop counter variable

We used same variable name for loop counter since local declaration supersedes the global declaration. Here, you can see the output of the Listing 4.10.

**Output:**

```
PL/SQL procedure successfully completed.
```

After executing Listing 4.10 on *iSQL Plus console, we concluded that the scope of the loop counter variable `j` is destroyed when the `FOR` loop ends.

You can also access global value of `j` inside the loop. For this, see Listing 4.11.

**Listing 4.11:** Accessing global variables inside the loop

We named entire block of statements with label `global`. Global value of `j` accessed by using syntax `global.j`. The `END global` statement ends the entire block.

Here you can see the output of the Listing 4.11.

**Output:**

After executing Listing 4.11 on *iSQL Plus console, we saw in output that when we access the value of `j` using `global.j`, it prints the value of `i` in global block. If we access value of `j` by just typing `j`, it displays value of `i` local variable.

## *WHILE LOOP Statement*

The `WHILE-LOOP` statement iterates the set of statements present inside its body until condition evaluates to `TRUE`. The general syntax of `WHILE-LOOP` statement is as follows:

The number of times, a set of statements executed, depends upon the condition and cannot be determined till loop finishes. There may be a case in which these statements will not execute for once since condition is examined at the beginning of loop. When condition evaluates to `FALSE` or `NULL`, body of loop is skipped and control transfers to next statement after the loop.

Let's calculate the areas of circles with different radius using `WHILE` loop. See Listing 4.12 for use of `WHILE LOOP` statement.

**Listing 4.12:** Using `WHILE LOOP` statement

The condition inside the `WHILE` and `LOOP` keywords is radius of circle, which should be less than or equal to 10. The expression that calculates the area of circle is `pi* radius* radius`. The pi variable here is mathematical `PI` constant which equals to 3.14. In each iteration, value of `radius` variable is incremented by 1.

Here you can see the output of the Listing 4.12.

**Output:**

When we execute Listing 4.12 on *iSQL Plus console, the console displays the areas of all circles with radius less than 10.

We can also make `WHILE` loop to execute at least once by using following syntax:

As flag variable is set to false, `NOT` operator makes condition present inside `WHILE` and `LOOP` true, therefore, a set of statements will execute for first time. But there must be an assignment statement which assigns a new value to flag Boolean variable so that the loop does not execute for infinite number of times.

## Summary

In this chapter, we have studied about:

- Conditional Control statements such as IF-THEN, IF-THEN-ELSE, IF-THEN-ELSEIF and CASE with their examples.
- Sequential Control statement such as GOTO and NULL with their examples.
- Iterative statements such as LOOP, FOR-LOOP, and WHILE-LOOP with their examples.

In today's scenario, every organization needs to manipulate the data in the database as per the changing requirements of the organization. Data in the database can be manipulated by using the Structured Query Language (SQL), but performing manipulation at large scale is not possible due to limitations of SQL. One of the major limitations of SQL is that it cannot send multiple statements to a database at the same time. With SQL, we cannot manipulate the retrieved data. For example, you want to add 1000 rupees in the salaries of those employees who are earning 3000 rupees monthly. The problem with SQL is that you can only fetch the records of the entire employees earning 3000 rupees but cannot manipulate those records. To overcome this problem, PL/SQL is used to perform SQL operations. PL/SQL extends the entire characteristics and statements available in SQL, such as select, insert, update, drop, and delete. A database can be easily created and manipulated by incorporating SQL statements in a PL/SQL block. PL/SQL supports all the datatypes supported by SQL; therefore, you do not need to convert SQL datatypes to PL/SQL datatypes.

This chapter describes how SQL can be used within PL/SQL and explains transaction management.

## Working with DDL and DML Statements in PL/SQL

SQL has two types of statements, which are used to manage a database. Those statements are `DDL` (Data Definition Language) and `DML` (Data Manipulation Language). DDL statements are used to create, alter, and drop a table in the Oracle database while the DML statements are used to manipulate the records in the database table. Following are the DDL and DML statements that we are going to discuss in this section:

- **CREATE Statement:** used to create a table in the database.
- **INSERT Statement:** used to insert records in the database table.
- **SELECT Statement:** used to retrieve values from the database table.
- **UPDATE Statement:** used to modify the database table.
- **DELETE Statement:** used to delete a record from the database table.
- **DROP Statement:** used to remove the complete table from the database.

You are familiar with using all these statements in SQL. Let's now see how to use these statements in PL/SQL.

### *Using the CREATE statement*

It is a DDL statement and the way to use this statement in PL/SQL is same as in SQL. Let's see the Listing 5.1 where we are creating a table by the name of `E_DETAIL` for storing the details of employees.

**Listing 5.1:** Creating a Database table

On executing the above statement, you will see the following output.

```
Table created.
```

### Using the INSERT statement

It is a DML statement, which is used to add record in the database table. In PL/SQL, we can use this statement multiple times within PL/SQL block for adding multiple records in the database table as compared to SQL where only one record can be added to the database table at a time. Let's see Listing 5.2 to know how to use this statement in PL/SQL.

**Listing 5.2:** Adding record in the `E_DETAIL` table

In the Listing 5.2, we have inserted two records in the database table that were created in the Listing 5.1.

Now let's see the output of the Listing 5.2.

```
PL/SQL procedure successfully completed.
```

### Using the SELECT statement

It is also a DML statement, which is used to retrieve the values from the database. In PL/SQL, you have to use `INTO` clause with `SELECT` while in `SQL SELECT` statement can also be executed without `INTO` clause. Using `INTO` clause helps in retrieving some specific field from the database table rather than retrieving all the fields. Let's see Listing 5.3 to know how to use `SELECT` statement in PL/SQL.

**Listing 5.3:** Selecting values from the `E_DETAIL` table

In Listing 5.3, we have used select statement to retrieve `EMP_NO` and `EMP_BASIC` from the database table `E_DETAIL` where `EMP_NO=100`. Let's see the output of Listing 5.3:

```
EMP_NO = 100 EMP_BASIC = 25000
PL/SQL procedure successfully completed.
```

### Using the UPDATE statement

It is also a DML statement, which is used to modify records in the database table. In PL/SQL, we can use this statement multiple times within PL/SQL block to modify multiple records in the database table while in SQL only one record can be modified at a time. Let's see Listing 5.4 to know how to use this statement in PL/SQL.

**Listing 5.4:** Modifying records in the table E_DETAIL

```
SQL> BEGIN
UPDATE E_DETAIL SET EMP_BASIC
UPDATE E_DETAIL SET EMP_BASIC
END;
/
```

In Listing 5.4, we have executed UPDATE statement twice to modify the basic salary of the employee's having EMP_NO 100 and 101 respectively. Let's see the output of Listing 5.4:

```
PL/SQL procedure successfully completed.
```

### Using the DELETE statement

It is also a DML statement, which is used to delete record from the database table on the basis of the condition specified with WHERE clause. Now, suppose you want to delete all the records in PL/SQL where basic salary is equal to 10000, for this purpose let's see Listing 5.5 to know how to delete record from the database table.

**Listing 5.5:** Removing a record from E_DETAIL table

```
SQL> BEGIN
DELETE E_DETAIL WHERE EMP_BASIC= 10000;
END;
/
```

Output of Listing 5.5 is as follows:

```
PL/SQL procedure successfully completed.
```

In PL/SQL, you can use all the DML statements to perform some transactions which are not possible in SQL. Let's see Listing 5.6 to know how to use all DML statements together.

**Listing 5.6:** Showing the use of all DML statements in a PL/SQL program

```
SQL> DECLAR
ENO NUMBER;
SAL NUMBER
BEGIN
INSERT INTO
INSERT INTO
SELECT EMP_I
UPDATE E_DE
DELETE E_DE
END;
/
```

In Listing 5.6, We have used all the four DML statements in the same PL/SQL block which is not possible to do in SQL.

First, we have inserted two records in the table E_DETAIL.

After that, we have retrieved some values in the variable declared in the declaration part of the PL/SQL block.

Then, we have executed UPDATE statement to modify the salary of an employee whose employee number has been specified with WHERE clause.

Then at last we have executed the DELETE statement to remove all the records from the database table where EMP_BASIC=10000.

On executing the Listing 5.6 you will get the following output:

```
PL/SQL procedure successfully completed.
```

In this way, we can use DML statements in various combinations to perform some meaningful task such as selecting and updating the user record. Using DML statements in different combinations to get the desired result is known as transaction. We will study transactions in detail ahead in this chapter. Now, we discuss the process to remove a table from database.

### *Using the DROP statement*

It is also a DDL statement and the way to use this statement in PL/SQL is same as in SQL. For example, you have created a table in the database but later on felt that there is no need to create that table then what will you do. In that case, you can use DROP statement to remove the unnecessary table from the database. Let's see the following example in which we are deleting a table EMP_DETAIL1.

```
SQL> DROP TABLE EMP_DETAIL1;
```

On executing the above statement, you will get the following output:

```
Table dropped.
```

With this we conclude our discussion upon DDL and DML statements in PL/SQL and SQL support in PL/SQL. You must have observed that under the topic DML statements, you came across a term transaction which has been defined and in a concise manner. Let's now study transaction and its management in detail.

## Transaction Management with PL/SQL

A transaction is a complete unit of work that includes a series of SQL DML (Data Manipulation Language) statements. The properties that describe how transactions should work in a database are collectively known as ACID. The acronym ACID stands for:

- **Atomicity:** It means that the transactions made to database either completes or fails.
- **Consistency:** Transactions should have to maintain data integrity that is no partial transaction should be made when working as a complete unit.
- **Isolation:** Multiple transactions can be executed simultaneously; in that case the changes made by one transaction will be visible to other transactions when the transaction will be committed.

- **Durability:** It means once the collection is committed successfully, the changes made by transaction are permanent and safe from failures such as electricity failure during transaction.

A transaction starts when Oracle encounters the first SQL statement and the transaction ends due to any one of the following reasons:

- User issues either a `COMMIT` or `ROLLBACK` statement.
- User executes statements such as `CREATE`, `DROP`, `RENAME`, or `ALTER`. If user executes any of such statement then Oracle first commits the current transaction and then executes and commits those statements.
- User is disconnected from the Oracle then the current transaction is committed.
- Any failure occurs and the transaction is rolled back.

Let's discuss a banking example to understand transaction. Suppose there are two persons named Adam and Sarah. Adam needs to transfer some amount of money in Sarah's account. For that two operations must have to be performed during transactions that are deduction of the money from Adam's account and addition in the Sarah's account.

To perform these two operations, two update statements should work as a single unit. This complete unit is known as `Transaction` and it needs to be committed to complete the transaction. If you will not commit the transaction then it may lead to database inconsistency because it may happen that during transaction one account is updated correctly but while updating the other account suddenly electricity failure occurs or any other kind of failure occurs such as memory problem, hard disk crash. Without committing, the transaction can not ensure that whether the transaction is a success or a failure but if the transaction is committed then at the time of failure Oracle will role back the transaction and thus maintaining the database consistency.

In the above example, we have talked about committing and rolling back the transactions. Here, you can raise a question about how to commit and rollback the transaction? Answer is, the statements provided by Oracle to manage transaction. Now, we continue discussion by explaining the statements that are provided by Oracle to manage transaction. Those statements are as follows:

- COMMIT
- ROLLBACK
- SAVEPOINT
- SET TRANSACTION
- LOCK TABLE

Let's now continue discussing all the above given statements in detail. We will study all these statements in the same order as given above.

### *Using the COMMIT statement*

This statement is used to make the changes made by SQL statements within the transaction permanent. Whenever a transaction is committed a unique SCN (System Change Number) a specific number in Oracle memory that indicates the location of datafiles is generated, which is then automatically written to the redo logs files (the files that store all the changes made by Oracle database). Syntax to use `COMMIT` is as follows:

```
COMMIT WORK;
```

In the above syntax, the keyword `WORK` is optionai. This can be used for readability of the program.

Let's see an example to use `COMMIT` statement.

In this example, we are considering a consultancy agency that provides recruitment to candidates registered with them. Now, suppose that this consultancy keeps the records of the candidates in two files. In one files it keeps the records of those candidates to whom recruitment has to be provided and in other those has already been recruited.

Now, we have to create a transaction in such a way that if one candidate will get recruitment in any company then his/her record should be deleted from the first table and will add in the another table automatically.

In this transaction, we will use three statements that are `SELECT`, `DELETE` and `INSERT` and if any one of the statement is not executed properly or any failure will occur then it leads into database inconsistency. To ensure that inconsistency will not occur, we have to commit the transaction. Let's see the Listing 5.7 to understand how to commit a transaction:

**Listing 5.7:** Showing how to make the changes permanent

In Listing 5.7, first we have selected a complete record of candidate from the table `RECRUITEMENT` on the basis of CID (candidate ID) when the candidate is recruited.

After that, we have inserted the same record in the table `RECRUITED` with the company name where candidate is recruited.

Then, according to the scenario, we have deleted the same record from the table `RECRUITMENT` that we have inserted into the table `RECRUITED`. After that, we have used `COMMIT` statement to make the changes permanent.

Output of Listing 5.7 is as follows:

```
PL/SQL procedure successfully completed.
```

In this way, COMMIT statement is used to complete a transaction. Let's discuss another statement that helps in managing transaction that is ROLLBACK statement.

### *Using the ROLLBACK statement*

This statement is used to undo the changes that are made during a transaction. This statement is also used to end the current transaction. ROLLBACK undoes all the changes made up to the last commit. It is also used to take the corrective actions for the failures that occur at the time of processing the current transaction. Syntax to use ROLLBACK is as follows:

```
ROLLBACK WORK;
```

In the preceding syntax, the keyword WORK is optional.

Let's see the Listing 5.8 to understand ROLLBACK statement.

**Listing 5.8:** Rolling back the work done by DML statement

```
SQL> BEGIN
INSERT INTO EMP_BONUS ( EMP_NO, EMP_BASIC, BAB) VALUES (100, 6000, 7000);
INSERT INTO EMP_BONUS ( EMP_NO, EMP_BASIC, BAB) VALUES (103, 6000, 7000);
EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
ROLLBACK;
END;
/
```

In Listing 5.8, we are inserting some values in the table EMP_BONUS. We are trying to execute the INSERT statement twice.

Here we want Oracle to check for the duplicate records in the table and for that we have used a pre-defined exception that is DUP_VAL_ON_INDEX. If Oracle finds any duplicate record then the ROLLBACK statements will be executed and the changes made by other statement will also be undone. But if Oracle does not finds any duplicate record then INSERT statement will execute properly.

Here in our case, record with the EMP_NO=100 already exists in the table that's why the Oracle has rolled back the transaction.

*Note*

> Exception will be discussed in detail in chapter number 11 of this book.

Sometimes you may execute many statements in a transaction while specifying ROLLBACK statement for managing database consistency.

And, during execution of those statements some problem may occur in the execution of a single statement, or if the statement is violating any constraint then the Oracle will rollback the whole transaction.

If you want that the whole transaction should not be rolled back and only that statement creating violation should roll back then you can also roll back the same statement. To do that, you have to use SAVEPOINT. Let's study SAVEPOINT in detail.

### statement

INT is known as undeclared identifier, which is used to store intermediate state in a It allows a user to undo a part of transaction. SAVEPOINT is used to rollback only that failed to execute or raised any exception due to any kind of constraints, we use SAVEPOINT then only the work done by failed statement is lost but the work done by successful statements in current transaction will remain safe. Let's see Listing 5.9 to understand its working.

**Listing 5.9:** Using SAVEPOINT in transaction

In Listing 5.9 first we executed the DELETE statement to remove a record from the table EMP_BONUS. Then, we applied a SAVEPOINT to save the changes made by this statement. After that, we executed an insert statement to add a record in the table EMP_BONUS. It may happen that the record we are trying to insert already exists in the table, in this case it violates the constraints and statement will fail.

To check the duplicate records in the table, we have used a pre-defined exception and on the basis of that we have rolled back the transaction to the SAVEPOINT. In the case, Oracle will roll back the transaction to a SAVEPOINT the following occurs:

- Oracle roll back only the statements run after the SAVEPOINT.
- Oracle will undo all the changes made by transaction but retain the changes made up to the SAVEPOINT.

This is all about the SAVEPOINTS in transactions. Let's continue our discussion to know how to set the transaction to read-only or read-write.

## Using the SET TRANSACTION statement

The SET TRANSACTION statement can be used only once in a transaction and this must be the first statement in the transaction. You can set the transaction to read-only or read-write by using SET TRANSACTION statement. Making a transaction read-only is useful where you want to execute multiple queries in the transaction and at the same time allow other users to execute other statements on the same table.

Read-only transactions are beneficial for the queries in the transaction to view only the changes made before the transaction began. Let's see Listing 5.10 to use SET TRANSACTION statement.

**Listing 5.10:** Making the transaction read only

```
SQL> DECLARE
sal NUMBER;
```

In Listing 5.10, we have executed the select statement twice and it will provide the same data irrespective of the changes made to the database table used in this transaction.

Here, we want to convey that if any other user will modify the values in the database table while the first `SELECT` statement has already been executed and the other has not start yet; at that time, modification in the database table will not affect the result of the transaction. This is so because after setting the transaction to read-only this transaction can only read the changes made before the transaction began.

To end the `SET TRANSACTION`, use `COMMIT` or `ROLLBACK` statement. `SET TRANSACTION` also has some restrictions over it, these restrictions are as follows:

- DML statements can not be used with read-only transactions.
- You can not execute queries with the `FOR UPDATE` clause.
- You can only use the `SELECT INTO`, `OPEN`, `FETCH`, `CLOSE`, `LOCK TABLE`, `COMMIT` and `ROLLBACK` statements in the read-only transactions.

With this we have ended discussing `SET TRANSACTION` statement. Here, you have seen that while using `SET TRANSACTION` statement other users are allowed to modify the database table. Modifying the database table while a transaction has already been in execution, may lead in unexpected result, if you will not set the transaction to read-only. If you want that other user can not modify the database while one transaction is already in execution then you can set the `LOCK` on the database table to restrict the modification. Let's study about how to set `LOCK` on the database table.

### *Using the LOCK TABLE statement*

This statement can be used to lock the database table. A database table can be locked in two modes that are `ROW SHARE` mode and `EXCLUSIVE` mode. During row share mode many users are allowed to use the same resource, such as a table while in exclusive mode only one user is allowed to use the resource and that resource locked until the transaction is committed or rolls back. Let's see the syntax to use `LOCK TABLE` statement.

```
LOCK TABLE tab_name IN lock_mode;
```

Here:

- **tab_name**: It is the name of database table that you want to lock.
- **lock_mode**: It is the mode of lock that can either be `ROW SHARE` mode or `EXCLUSIVE` mode. Which lock needs to be applied on the database table depends on your requirement.

Now, see how to use `LOCK TABLE` statement on a table. In the Listing 5.11, we are locking the table `EMP_DETAIL` in `EXCLUSIVE` mode.

**Listing 5.11:** Showing how to lock the table

Output of the Listing 5.11 is as follows:

```
PL/SQL procedure successfully completed.
```

Now, other user can only retrieve the data from the table `EMP_DETAIL` but can not perform operations such as `INSERT`, `UPDATE`, and `DELETE`. Here, we have completed discussion on Transaction management in PL/SQL. With this, we have reached to the end of this chapter. After studying this chapter, you should be able to execute SQL DDL and DML statements within PL/SQL block and can also manage transactions. Let's have a brief summary about all that we have covered in this chapter.

## Summary

In this chapter, we have studied about:

- The SQL supports in PL/SQL
- Using DDL and DML statements in PL/SQL
- Transaction Management

PL/SQL, similar to other programming languages such as C, C++, allows using arrays, list, collections, and records, to group related values. A group of related values, which may be of different datatypes, is called a collection, and a group of values with similar datatypes is called a record. Using collections and records can simplify the work of a programmer, and reduces the length and complexity of a program.

Suppose you want to process the salaries of the employees in the `EMP_DETAIL` table. You can do this easily by using collections, because salary of all the employees must be stored under the same column, implying that you are working with similar datatype. However, you can use records to manage all the details of the employees, since different columns in a table may be of different datatype. Let's now look at how to use collections and records. We start by learning to work with PL/SQL collections.

## Working with PL/SQL Collections

Collections are single dimensional data structures that function as an ordered group of elements having the same datatype. For example, salaries of employees in the table `EMP_DETAIL` is a collection. Every element in a collection has a specific index number, which is used to access that element from the collection for processing. You can use collections as PL/SQL datatype and can also pass them as parameters. Let's discuss about various PL/SQL collection types as well as learn how to define, declare, initialize, and assign collection types, in detail. We describe them under the following topics:

- Selecting PL/SQL Collection Types
- Defining Collection Types
- Declaring Collection Variables
- Initializing Collections
- Referencing Collections
- Assigning Collections
- Comparing Collections
- Using Collection Methods

Let's discuss these topics in detail.

### *Selecting PL/SQL Collection Types*

In programming languages, such as C or C++, a collection type can be a list, an array, or a hash table. Similarly, PL/SQL also offers its own collection types. PL/SQL collection types are varray, nested table and associative array. Selecting a collection type depends on the requirement of your program. Table 6.1 describes various collection types briefly.

**Table 6.1: Collection types in PL/SQL**

| Collection Name | Description |
|---|---|

Now, after learning about collection types, let's learn how to use them in a PL/SQL program.

### *Collection Types in PL/SQL*

To create a collection in PL/SQL, you must have to define its type. Collection type can be defined in the declaration part of PL/SQL block. As we know that there are three different collection types, so the syntaxes used to define them is also different but the scope and instantiation rules for all of them are same as they are for other PL/SQL datatypes. Now, let's study how to define collection types—varray, nested table and associative array— in a program.

#### Varrays

The syntax to define a varray in a PL/SQL program construct is as follows:

```
TYPE varray_type IS VARRAY (size) OF element_type [NOT NULL];
```

In this syntax:

- `varray_type` represents any valid name that will be used while declaring a collection.
- `VARRAY` shows that you are defining a collection of type `VARRAY`. You can also use `VARYING ARRAY` in place of `VARRAY`.
- `(size)` represents a positive integer value, which is used to set the upper bound of the elements that this collection can contain.
- `element_type` represents PL/SQL datatype.

- [ NOT NULL] is optional and sets validation so that the elements in the collection can not have null values.

Listing 6.1 shows how to store ten numeric values:

**Listing 6.1:** PL/SQL program to define VARRAY

```
SQL> DECLARE
TYPE emp_id IS VARRAY (10) OF NUMBER NOT NULL;
BEGIN
NULL;
END;
/
```

Output of the Listing 6.1 is as follows:

```
PL/SQL procedure created successfully
```

The Listing 6.1 shows how you can define a varray in PL/SQL program. You can also define varrays in database tables. To define a varray in a database table, you require to use CREATE TYPE statement in place of the TYPE statement.

Syntax to define a varray in a database table is:

```
CREATE TYPE varray_type AS VARRAY (size) OF element_type;
```

Syntax to define a varray in a database table is same as the syntax to define a varray in PL/SQL program except that we have added the CREATE TYPE statement in the beginning of the syntax, and IS has been replaced by AS.

See the following example to understand how to define a varray in a database table. It can store addresses of ten employees.

```
SQL> CREATE TYPE emp_add1 AS VARRAY (10) OF VARCHAR2 (200);
/
```

Having learned how to define a varray in a database table, let's understand the process to define nested tables.

### Defining Nested Tables

Syntax to define a nested table in a PL/SQL program is:

```
TYPE nested_table_type IS TABLE OF element_type [NOT NULL];
```

In this syntax:

- Nested_table_type represents any valid name, which will be used later while declaring a collection.
- TABLE shows that you are defining a collection of type TABLE.
- element_type represents the PL/SQL datatype.
- [ NOT NULL] is optional and can be used to set the validation so that the elements in the collection can not have null values.

See the Listing 6.2, which can be used to store some numeric values:

**Listing 6.2:** PL/SQL program to define Nested Table

Output of the Listing 6.2 is as follows:

```
PL/SQL procedure created successfully
```

In the Listing 6.2, we have defined a nested table in PL/SQL program. You can also define nested tables in your database. Let's see how to define a nested table in a database.

Syntax to define a nested table in a database table is:

```
CREATE TYPE nested_table_type AS TABLE OF element_type;
```

Syntax to define nested table in a database table is same as the syntax to define nested tables in PL/SQL program except that `CREATE` has been added in the beginning of the syntax and `IS` has been replaced by `AS`.

The following example shows how to define a nested table in a database table which can store addresses of employees.

```
SQL> CREATE TYPE emp_add2 AS TABLE OF VARCHAR2 (200);
/
```

Let's discuss how to define associative arrays.

### Defining Associative arrays

You can define associative arrays in a database. Here's the syntax for the associative array:

```
TYPE associative_array_type IS TABLE OF element_type [NOT NULL] INDEX BY
key_type;
```

In this syntax:

- `associative_array_type` represents any valid name, which will be used later while declaring the collection.
- `element_type` represents the PL/SQL datatype.
- `[ NOT NULL]` is optional and can be used to set the validation so that the elements in the collection can not have null values.
- `INDEX BY` clause is used to define associative array.
- `Key_type` represents a numeric value (`PLS_INTEGER` or `BINARY_INTEGER`) or `VARCHAR2`. The datatype, such as `RAW`, `LONG RAW`, `ROWID`, `CHAR`, and `CHARACTER` are not allowed as keys for an associative array.

Listing 6.3 shows how to define an associative array:

**Listing 6.3:** PL/SQL program to define Associative arrays

Output of the Listing 6.3 is as follows:

```
PL/SQL procedure created successfully
```

With this we conclude discussion on defining collection types. After understanding how to define a collection type, let's learn to declare collection variables.

## *Declaring Collection Variables*

After defining the collection types, you need to declare the collection variable of that collection type. Declaring collection variables is necessary to use the collection type, you have defined. You define collection types in the declaration part of PL/SQL block. Listing 6.4 shows how to declare collection variables. Before declaring the collection variables execute the statements given below to define VARRAY and Nested table in database.

```
SQL> CREATE TYPE emp_add1 AS VARRAY (10) OF VARCHAR2 (200); /* Defining varray
in Database */
```

Output of the preceding statement is as follows:

Output of the preceding statement is as follows:

```
Type created.
```

Let's see the Listing 6.4 to declare collection variables.

**Listing 6.4:** PL/SQL program to declare collection variables

Output of the Listing 6.4 is as follows:

After declaring the variables, you need to initialize them. Let's see how to initialize collection

### Collections

variables can either be initialized in the DECLARE part or in the BEGIN part of block. All collections are initially NULL. Collection variables (varrays and nested tables) initialized using the constructor method (a constructor is a system-defined function having the same name as the collection type). Let's see the Listing 6.5 to understand how to initialize collection variables. Before initializing the collection variables execute the statements given below to define VARRAY and Nested table in database.

of the preceding statement is as follows:

preceding statement is as follows:

```
Type created.
```

**Listing 6.5:** PL/SQL program to Initialize collection

Output of the Listing 6.5 is as follows:

```
PL/SQL procedure created successfully
```

After learning how to initialize collection variables, let's study the process to reference a collection element.

### Referencing Collections

Referencing is used to access an element from a collection. Syntax to reference an element is as follows:

```
Coll_name (subscript);
```

In this syntax:

- `Coll_name` represents the name of a collection.
- `(subscript)` represents the element that has to be processed. There is some specified range for subscript, which varies according to the collection type. Following are the subscript ranges according to their collection types:
  - For varrrays, the range is 1 to upper bound of the varray.
  - For nested table, the range is 1 to 2147483647.
  - For associative arrays with numeric key, the range is -2147483647 to +2147483647.
  - For associative arrays with `VARCHAR2` value as key, the range depends on the length of the value specified during type declaration and database table creation.

Listing 6.6 shows how to perform referencing in a collection.

**Listing 6.6:** PL/SQL program to use referencing in collections

The output of Listing 6.6 is as follows:

In Listing 6.6, we have used a function `COUNT ()`, which counts the number of elements in the nested table and moves the loop up to that count.

this we conclude discussion on referencing the collections. Now, sometimes you may need to assign one collection into another collection, how you can do this is explained next.

### Collections

assign a collection into another collection, the element types and the datatype names of both collections must be same. There are various ways to assign a collection into another collection, such as using `INSERT` statement, `UPDATE` statement, `SELECT` statement, assignment statement, and also the subprogram call. Listing 6.7 shows how to assign one collection into another

**6.7:** PL/SQL program to assign collections

output of Listing 6.7 is as follows:

```
PL/SQL procedure succesfully completed.
```

In Listing 6.7:

- First statement, A :=B, is allowed, because both A and B have the same element type (`NUMBER`) and datatype (`eid` ) name.
- Second statement, B :=C, is not allowed, because although they have the same element type but their datatype names are different, that is, the datatype name of B is `eid` but that of C is `esal`.

of Listing 6.7 shows that the procedure is completed successfully. It is because we given the Illegal assignment as a comment. However, if you try to run Listing 6.7 without the illegal assignment a comment, then the following error message appears:

You can also assign a null value to another collection or can also assign values in a collection by using expressions. See Listing 6.8 to know how to assign an expression and null into another collection.

**Listing 6.8:** PL/SQL program to assign expression and null value

In Listing 6.8, we have assigned an expression to an element present at the first position in the varray, and then displayed the value assigned to the element at first position. After that we have assigned the nested table `B` (initialized as null) to nested table `C`, and at last we have used the `COUNT` method to know the number of elements in the nested table `C` after assigning the expression.

Output of Listing 6.8 is as follows:

This is all about assigning Collections. Let's now learn how to compare collections.

### *Comparing Collections*

Sometimes you may require to know whether a collection is null or not; or whether the elements in two collections are similar or not. In PL/SQL you can compare two collections to find out whether they are null or similar. Listing 6.9 shows how to compare two collections.

**Listing 6.9:** PL/SQL program to compare collections

In Listing 6.9, we have declared two variables of nested tables but initialized only one variable, that is C, and kept B as blank. After that, we have compared B for Null, and then B and C to check whether they have similar elements or not. The output of Listing 6.9 is as follows:

This is all about comparing collections. Let's see how to use collections with database tables by performing operations, such as create, insert, update, on a table in the database.

## Using a VARRAY

To perform operations on a database table using varrays, you must define, declare, and initialize the varray. The process to define, declare, and initialize a varray has already been discussed.

Let's learn how to create a table that has a column of the varray collection type.

```
SQL> CREATE TABLE V_array (ENAME VARCHAR2(30), EADD emp_add1);
```

The output of this statement is as follows:

```
Table created.
```

In the preceding statement, we have created a table named V_array. This table has two columns; one of them is of VARCHAR2 datatype while the other is of VARRAY datatype. Let's now take the table, emp_add1, that we have created earlier (described under the heading 'Defining varrays'), and insert a record in that table. Listing 6.10 shows the code to insert a record in the emp_add1 table.

**Listing 6.10:** PL/SQL program to add records in emp_add1 table

Output of Listing 6.10 is as follows:

```
PL/SQL procedure completed successfully.
```

After understanding how to insert records in a database table, let's understand how to use UPDATE and SELECT statements on a database table having columns of type varray. Listing 6.11 uses UPDATE and SELECT statements on a database table.

**Listing 6.11:** PL/SQL program to use UPDATE and SELECT statement on emp_add1 table

```
SQL> SET SERVEROUTPUT ON
DECLARE
add_varray emp_add1; -- Declaring variable of VARRAY
x emp_add1; -- Declaring variable of VARRAY
BEGIN
add_varray := emp_add1 ('Street no 12', 'H.no 20');-- Initializing varray
variable
-- Updating values in the table V_array
UPDATE V_array set EADD= add_varray WHERE ENAME = 'Amit';
-- selecting the updated field from v_array
SELECT EADD into x from v_array where ENAME = 'Amit';
for i in x.FIRST .. x.LAST LOOP
DBMS_OUTPUT.PUT_LINE (x(i));
END LOOP;
END;
/
```

In Listing 6.11, we have performed two operations: first we have updated the database table V_array, and then we have executed the SELECT statement to view the updated values.

Output of Listing 6.11 is as follows:

```
Street no 12
H.no 20
PL/SQL procedure completed successfully.
```

In the same way as varrays, nested tables can also be used in a database. But there is a difference in creating the database table with column as nested table type. Let's create a database table with the columns of type nested table.

```
SQL> CREATE TABLE nest_demo (ENAME VARCHAR2 (30), EADD emp_add2) NESTED TABLE
EADD STORE AS E_ADD1;
```

In the preceding statement, we have created a table named nest_demo. This table has two columns; one of them is of VARCHAR2 type while the other is of nested table type. The emp_add2 has already been created under the heading '**Defining Nested Table**'. Output of the preceding statement is as follows:

```
Table created.
```

This is all about using varray and nested table with database tables. Now, we continue our discussion by explaining collection methods. In Listing 6.1, we have used a method COUNT to know the number of elements in a nested table. PL/SQL has various other methods to reduce the programming tasks. Let's study PL/SQL built-in methods that can be used with collections.

## Using Collection Methods

PL/SQL provides various built-in methods that make it easy to use collections. These methods can be functions, such as EXISTS and COUNT or a procedure, such as EXTEND, TRIM, and DELETE. There are some limitation with built-in collection methods, such as they can not be called by using SQL statements; EXTEND and TRIM methods can not be used with associative arrays. Table 6.2 gives you a brief description of various built-in collection methods.

| Description |
| --- |
| Returns the number of elements in a collection. |
| Returns the smallest index number in a collection. |
| Returns the largest index number in a collection. |
| Returns the index number that precedes index n in a collection. |
| Returns the index number that succeeds index n in a collection. |
| Increases the size of a collection. |
| Trims or decreases the size of a collection. |
| Deletes the elements from a collection. |

Let's discuss these methods in detail.

### EXISTS

This is a function, which either returns TRUE or FALSE. If any specified element, say nth element, exists in the collection, then it returns 'true'; otherwise, it returns 'false'. When the nth element is out of range, then it returns FALSE rather than raising an exception. This method is used to avoid the referencing of non-existent elements. See Listing 6.12 to understand how it works.

**Listing 6.12:** PL/SQL program to use EXISTS function

In Listing 6.12, we have used the `EXISTS` method in the varray and nested table to check existence of the element whose subscript has been passed as parameter in this method. We have used the `IF... ELSE` control statement to display the output on the basis of the existence and non-existence of the element.

Output of Listing 6.12 is as follows:

## COUNT

The `COUNT` function returns the number of elements that currently exist in a collection. This method is very useful when you do not know the exact number of elements in a collection. See Listing 6.13 to understand its working.

**Listing 6.13:** PL/SQL program to use `COUNT` function

In Listing 6.13, the `COUNT` method is used to calculate the number of elements in `varray` and `nested table`.

Output of Listing 6.13 is as follows:

You can use this method where integer value is expected.

function returns the number of elements that a varray can contain. In other words, returns the upper bound of a varray. In case of nested tables and associative arrays, function returns Null. This method can be used wherever an integer value is expected. 6.14 shows how this function works:

**14:** PL/SQL program to use LIMIT function

the preceding listing, first we have displayed the total number of elements in varray, and then total number of elements that a varray can contain. The output of Listing 6.14 is as follows:

of them are functions. FIRST returns the smallest subscript in the collection, and the LAST returns the last subscript in the collection. If the collection is empty, then these return null.

varrays, FIRST always returns 1, but for Nested table, it may or may not returns 1; if the first has already been deleted from the nested table, then the FIRST function returns 2.

, the LAST function returns the upper bound in case of varrays; but in case of nested it gives the same result as the COUNT function. If you delete some elements from the nested table, then the value returned by the LAST function is more than the value it returns with

.

of associative arrays with VARCHAR2 key values, these functions return the lowest and VARCHAR2 key values. Listing 6.15 shows how these functions work.

**6.15:** PL/SQL program to use FIRST and LAST functions

In Listing 6.15, the FIRST and the LAST methods are used to iterate the loop in and then the smallest and largest subscripts of the nested table are displayed. deleted the first element from the nested table by using the DELETE method; starting subscript is displayed once again.

The output of Listing 6.15 is as follows:

### PRIOR (n) and NEXT (n)

Both these functions return the subscripts of the specified element. PRIOR (n) returns the subscript of the element that precedes index n in that collection; while the NEXT method returns the subscript of the element that succeeds index n of a collection. PRIOR (n) returns null, if n has no predecessor and NEXT (n) returns null when it has no successor.

In case, associative arrays have elements of type VARCHAR2, these functions return the key values corresponding to the specified element. Listing 6.16 shows the use of the PRIOR (n) and the NEXT (n) methods.

**Listing 6.16:** PL/SQL program to use PRIOR and NEXT functions

## EXTEND

This is a procedure, which is used to increase the size of a collection. This method can not be used in case of associative arrays. This method can only be used with collections that have already been initialized. You can use this procedure in the following three ways:

- **EXTEND:** It appends one null element to the collection.
- **EXTEND (n):** It appends n null elements to the collection.
- **EXTEND (n, I):** It appends the ith element n times in the collection.

*Note*

> If a collection has NOT NULL constraint then null values can not be appended in a collection.

Listing 6.18 shows how the `EXTEND` procedure works.

**Listing 6.18:** PL/SQL program to use `EXTEND` function

In Listing 6.18, we have used the `EXTEND` method in all three ways—that is, without any parameter, with single parameter (2), and with two parameters (2, 3).

We have used the `COUNT` method after the `EXTEND` method to display the total number of elements in the collection (here it is a varray) each time the elements are extended.

The output of Listing 6.18 is as follows:

## TRIM

TRIM deletes the elements from the end of the collection. This method can be used in the following ways:

- **TRIM:** It deletes a single element from the end of a collection.
- **TRIM (n):** It deletes n number of elements from the end of a collection. If n is more than the number of elements contained in a collection, then it raises SUBSCRIPT_BEYOND_COUNT exception. Listing 6.19 shows how it works.

**Listing 6.19:** PL/SQL program to use TRIM function

In Listing 6.19, first we have displayed the total number of elements in the nested table by using COUNT method, and then we have trimmed the nested table by one element using the TRIM method. After that we have again applied the COUNT method on the nested table.

At last, we have trimmed the nested table once again by using TRIM method with parameter. The output of Listing 6.19 is as follows:

## DELETE

DELETE is a procedure, and it is used to delete elements from a collection. This method can not be applied on the varray collection type. Using the DELETE procedure, unlike TRIM which deletes elements from the end of the collection, you can delete any specified element from a collection.

This procedure can be used in the following ways:

- **DELETE:** It removes all elements from a collection.
- **DELETE (n):** It removes the nth element from a nested table or associative array with a numeric key. If the associative array has elements of string type, then the element corresponding to that string type is deleted. This method does nothing if n is null.

- **DELETE (m, n):** It removes all elements in the range m to n from a collection. This method will do nothing if m is larger than n or they are null. Listing 6.20 shows how it works.

**Listing 6.20:** PL/SQL program to use DELETE function

```
SQL> SET SERVEROUTPUT ON
DECLARE
TYPE esal IS TABLE OF NUMBER (7) NOT NULL; -- Defining nested table collection
C esal; -- Declaring nested table collection variable
BEGIN
C := esal (5000, 5500,6000, 6500,4500); -- Initializing nested table
DBMS_OUTPUT.PUT_LINE (' Total number of elements in nested table is ' ||
C.COUNT);
C.DELETE (2); -- Using Delete with single parameter
DBMS_OUTPUT.PUT_LINE (' Total number of elements after removing second element
is ' || C.COUNT);
C.DELETE (3, 5); -- Using DELETE with specific range of parameters
DBMS_OUTPUT.PUT_LINE (' Total number of elements after removing a range of
element is ' || C.COUNT);
C.DELETE; -- using DELETE without parameter
DBMS_OUTPUT.PUT_LINE (' Total number of elements after using DELETE without
parameter is ' || C.COUNT);
END;
/
```

In Listing 6.20, first we have displayed the total number of elements in the nested table by using the COUNT method, and then deleted the second element from the table, and again applied the COUNT method to display the count of elements in the nested table.

Now we use the DELETE method to remove the specific range of elements from the table and again use the COUNT method. Similarly, we have again used the DELETE method without any parameter and then the count method to display the count once again.

The output of Listing 6.20 is as follows:

```
Total number of elements in nested table is 5
Total number of elements after removing second element is 4
Total number of elements after removing a range of element is 1
Total number of elements after using DELETE without parameter is 0
PL/SQL procedure successfully completed.
```

With this we complete discussion on collections, and using collection methods. In using collections, we have some limitations such as we can only use the elements of same datatypes. So, in cases when there are different datatypes, you can use Records—a composite data structure in PL/SQL. Let's understand in detail about PL/SQL records.

## Working with PL/SQL Records

Record is a composite data structure, which contains more than one element that may or may not have the same datatype but has its own value. Conceptually, records are similar to the rows in a database table and are used to represent logically related information. Let's study about defining records in detail.

## Defining and Declaring Records

To define a record, first you need to create its RECORD type and then declare record of that type. To declare a record, you can also use %ROWTYPE attribute that represents a row in a database table. Record type is defined and declared in the declaration part of a PL/SQL block, subprogram, or package. You can also specify NOT NULL constraint and default values to fields in record when creating your own record type.

Syntax to define a record is:

```
TYPE rec_name IS RECORD (field1_name DATATYPE NOT NULL := default_value,
field2_name DATATYPE, .....);
```

In this syntax:

- rec_name represents the type name that specifies the record.
- field1_name represents the first field name. You can create as many fields as you wish. In the preceding syntax, we have shown two fields.
- DATATYPE represents any PL/SQL datatype except REF CURSOR.
- NOT NULL is optional, but whenever it is used, it must be initialized with some default value; otherwise, Oracle will generate the error NOT NULL must have an initialization assignment.

Listing 6.21 shows how to define and declare PL/SQL records.

**Listing 6.21:** PL/SQL program to define and declare records

In the Listing 6.21, first we have defined a record, and then we have declared a variable of that record type.

We have also declared a record variable to represent a row in the database table. The table we have used here is DEPT. Before creating a record variable to represent a row in a database table, you must ensure that the table exists in the database, else Oracle will generate an error.

Output of the Listing 6.21 is as follows:

```
Pl/SQL procedure completed successfully
```

After defining and declaring the records variables, you have to assign some values to those variables. Let's discuss the process of assigning values to records variables.

### *Assigning Values to Records*

Syntax to assign values in record variables is:

```
rec_var_name.fieldname:= value;
```

In this syntax:

- `rec_var_name` represents the name of record variable.
- `fieldname` represents the name of the field for which the value has to be assigned. This field can be the one which is used during '**Defining Record**' or can be the name of a column in the database.
- `Value` represents any value that you want to assign to the corresponding field. Before assigning any value to its corresponding field, you must ensure that it is in proper format. For example, if you assign string type value to its corresponding field but the actual datatype of that field is not of string type, then Oracle raises the error `identifier must be declared`.

**Listing 6.22:** shows how to assign the values to records

In the Listing 6.22, we have assigned values to various fields using record variables `r1` and `r2`. Fields name used with the variable `r1` are already defined during 'Defining Record' while the fields name used with variable `r2` are the name of columns in database table.

Output of the Listing 6.22 is as follows:

```
PL/SQL procedure completed successfully
```

After assigning the values, you can perform operations such as inserting and modifying the database table using records. Now, we move ahead to know how to insert a row in the database.

### *Inserting Records into the Database*

Records can help you insert a complete row in database table by using a single variable—the Record variable. With records, you have to specify only the record variable with Value clause instead of specifying all fields with the Value clause. When you use records, it increases readability and maintainability of your PL/SQL program. Before inserting any value in a database table using records, you must ensure that the numbers of fields in the records are same as the number of columns in the database table and that too of compatible datatypes. Listing 6.23 shows how to insert a record into a database.

**Listing 6.23:** PL/SQL program to add records into the database tables

```
r1.DEPTNO := 40;
-- Assiging values to record variable used to represent the rows in database
table
r2.DEPTNO := 80;
r2.DNAME := 'Quality';
r2.LOC := 'Old Delhi';
INSERT INTO EMP VALUES r1; -- -- Inserting record ( a row) in table EMP
INSERT INTO DEPT VALUES r2; -- Inserting record ( a row) in table DEPT
END;
/
```

In Listing 6.23, we have inserted two records in two different tables (EMP and DEPT) by using two different records.

The output of Listing 6.23 is as follows:

```
PL/SQL procedure successfully completed.
```

Now, to check whether these records have been inserted in the table or not, you can execute the SELECT statement.

Let's understand how to update database using records.

### *Updating a Database with Record Values*

Records also allow you to modify a complete row in a database table by using a Record variable. For updating a record you need to specify a record variable with SET clause instead of specifying all fields with SET clause. Before updating any value in a database table using records, you must ensure that the number of fields in the records must be same as the number of columns in the database table and that too of compatible datatypes. You can use ROW keyword to represent an entire row. Listing 6.24 shows how to update a record into the DEPT database table.

**Listing 6.24:** PL/SQL program to update a database with record values

```
SQL> DECLARE
r2 DEPT%ROWTYPE;
BEGIN
r2.DEPTNO := 70;
r2.DNAME := 'Production';
r2.LOC := 'Old Delhi';
UPDATE DEPT SET ROW = r2 where DEPTNO = 70 ; -- Updating the DEPT table
END;
/
```

On executing the code shown in Listing 6.24, you get the following output:

```
PL/SQL procedure successfully completed.
```

So this is how you can update a database table by using records. With this we conclude our discussion on PL/SQL records and collections. After completing this chapter, you should be able to use collections and records in your PL/SQL programs and with database tables.

## Summary

In this chapter, we studied:

- PL/SQL collections types
- Defining and declaring collections
- Initializing and assigning collections
- Using various collection methods
- PL/SQL records
- Defining and declaring records
- Inserting and updating the database rows using PL/SQL records

Cursors are used to refer to a particular SQL transaction, which involves processing of particular SQL statements. Cursors are usually used to execute complex queries on a result set. A cursor individually retrieves the contents of each row in a rowset within the context of SQL statements; and avoids the use of array to load data from a rowset into the array. You can greatly improve the performance of your application by replacing a SELECT statement with a cursor. When you execute a SELECT statement, a portion of memory is allocated to store the parsed representation of the SELECT statement in a PL/SQL block. PL/SQL has a provision to point to this portion of memory by using cursors. This portion of memory is known as context area. The context area contains the rows returned by a SELECT statement.

In this chapter we will discuss about cursor, its types, how cursors are used to fetch and manipulate rows, cursor attributes and cursor variables.

## Introducing Cursors

Cursors are used to manipulate data of many rows and it is also used as temporary work area in which you can store result of the SQL statement. The cursors are used as sequential files. To manipulate cursor, you have to open the cursor using OPEN statement. You can fetch the rows from cursor using FETCH statement. After fetching the rows, the cursor is closed using CLOSE statement. Several queries are processed by opening multiple cursors at a time.

PL/SQL supports two types of cursors: Implicit Cursor and Explicit Cursor. This classification is based on whether the cursor is user defined or not. Let us take a brief look at both types of cursors.

- **Implicit Cursor:** It is the cursor that Oracle internally opens when you issue SELECT ... INTO or any DML statement. PL/SQL uses this cursor for all SQL DML statements and queries which return only one row. Two most common exceptions: NO_DATA_FOUND and TOO_MANY_ROWS raised in case of Implicit cursors.
- **Explicit Cursor:** It is cursor that you can define in your PL/SQL program. This cursor is used when any query returns more than one row. You can declare explicit cursor in declaration part of PL/SQL block, sub program, and package. We are going to use explicit cursors in the rest of this chapter.

After understanding what are cursors, now we are going to cover implicit cursors in detail in next section.

## Understanding Implicit Cursors

PL/SQL automatically provides an implicit cursor when you execute an UPDATE, DELETE or INSERT statement in your program. Upon executing SELECT statement, PL/SQL employs implicit cursor for SELECT statement only when the statement returns a single row. This cursor is called implicit cursor as Oracle itself carries out open, fetch and close operations for this cursor. You cannot programmatically handle these operations of implicit cursor. See following code snippet in which implicit cursor is created.

In this example, the UPDATE statement increments the basic salary of employee with emp_no 100 by 1000. PL/SQL automatically creates an implicit cursor to recognize set of rows in table E_DETAIL changed by update statement.

See Listing 7.1 which shows single-row SELECT INTO statement. Here, implicit cursor is also created.

**Listing 7.1:** Single-row SELECT INTO statement

As shown in above example, SELECT statement is a single-row query. This statement returns the total of salaries of all employees in EMP table.

In case SELECT statement returns more than one row, you need to use explicit cursor.

*Note*

## *Limitations of Implicit Cursors*

You can also use an explicit cursor for a single-row query. Following are the key limitations of implicit cursors:

- Low efficiency
- Error prone
- Low programmatic control

Let's discuss each of these limitation in detail.

- **Low efficiency:** In PL/SQL version 2.2 and earlier, execution of explicit cursor is faster than that of implicit cursor. The reason is that execution of implicit cursor is based on standard execution of a SQL statement. Oracle's SQL follows ANSI standard. This standard enforces single-row query to perform two fetches. The first fetch returns desired row. The second fetch checks whether single-row query has returned more than one row. In case, rowset contains more than one row, TOO_MANY_ROWS exception is raised. The reason of low efficiency of implicit cursor is due to this second fetch. While, if you contain single-row query in explicit cursor, there is need to perform only one fetch.

*Note*

- **Error prone:** We see an implicit SELECT statement may raise TOO_MANY_ROWS exception in case when more than one row is returned. This situation can occur anytime after

In preceding code snippet, eno is formal parameter. You can provide its value at runtime. The eno formal parameter is IN parameter, so you cannot return values in actual parameters. You cannot impose NOT NULL constraint on formal parameters. These formal parameters can be referred only in specified query. Thus, the scope of a parameter is local to cursor. Suppose you open the cursor as:

```
OPEN c1(102);
```

The c1 cursor contains the row corresponding to emp_no 102. If we do not pass any parameter, then it takes default value 101.

## Opening Explicit Cursor

You need to open the cursor before fetching rows from it. Opening the specified cursor executes the query and selects the rows which satisfy query criteria. The syntax of PL/SQL statement used to open the cursor is as follows:

```
OPEN c1;
```

A portion of a program which opens the cursor c1 is shown in Listing 7.2.

**Listing 7.2:** Opening a cursor

Listing 7.2 declares a record Z using %ROWTYPE to contain the rows of cursor c1. We require record variable Z as we are going to use it when we will fetch the rows. Above listing is example of opening cursor without parameters.

**Note**

Cursors can be opened only in EXECUTION or EXCEPTION sections of block.

You can also use named notation for passing parameter. For example, to open cursor with value 101, you can use open statement as following:

```
OPEN c1(eno => 101);
```

If you want to pass more parameters say X, Y to cursor, you can write open statement as following:

```
OPEN c2(X =>5 ,Y=>10);
```

## Obtaining Rows from Explicit Cursor

FETCH statement is used to fetch or obtain rows from cursor. You can manipulate only current row out of total fetched rows. When a FETCH statement is executed, cursor pointer moves to

next row. The FETCH statement should put data in variables having compatible datatypes. Therefore, you should use %TYPE and %ROWTYPE attributes for declaration of these variables.

You can fetch an entire column into a single variable in the following manner:

```
FETCH c1 INTO vrowid;
```

To fetch values of multiple columns of current of a cursor, you need multiple variables. For example:

```
FETCH c2 INTO eno,deptno;
```

In this example, variables eno, deptno must have compatible datatypes with corresponding columns of a table in SELECT statement of cursor c2. These variables must occur in the order that appears in the SELECT statement. The rows, which are fetched in the cursor, are decided on existing conditions such as values of variables used in the SELECT statement at the time of opening the cursor. See Listing 7.3 which uses the FETCH statement.

**Listing 7.3:** Fetching rows from a cursor



In the above example, when cursor is opened, the value of eno (also called bind variable) at that time is 101. Therefore, all the rows are selected having emp_no equal to 101. Even if you change the value of eno later on, still the rows fetched from the cursor correspond to value emp_no equal to 101. This is because rows are defined at the time of opening the cursor. If you want to fetch the rows for emp_no equal to 102, then you have to close and reopen the cursor again.

## *Closing Explicit Cursor*

After manipulating rows with the help of cursor, you must close it. You can close the cursor using CLOSE statement as shown in following syntax:

```
CLOSE c1;
```

When the cursor c1 is closed, any other resource such as memory associated with it is released. Oracle puts limit on maximum number of cursors which can be opened simultaneously. The default MAX_OPEN_CURSOR initialization parameter determines this limit. You can also change the value of this parameter.

You must close the cursor. In some cases, a program may do abnormal exit due to error in statement. Therefore, the cursor may remain open. In such case, the programmer has to ensure that the cursor is closed. You can do so by writing exception handler for exception OTHERS. Example of such a code is shown in Listing 7.4.

**Listing 7.4:** Closing a cursor



In the above example, we used exception handler for OTHERS which is executed in case of error conditions. Closing the cursor in both normal and abnormal termination is the duty of programmer.

***Note***

> You cannot perform any operation such as FETCH on closed cursor as it raises exception INVALID_CURSOR.

Let us understand explicit and implicit cursor attributes to know information about current row of cursor.

## Cursor Attributes

Cursor attributes provide information about current row of the cursor. You can refer attributes using following syntax:

```
cursorname%attribute
```

Cursor attributes referred only in PL/SQL statements not in SQL statements.

### *Explicit Cursor Attributes*

As we know PL/SQL programmer defines explicit cursor. Each explicit cursor is having four attributes %FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT.

For example, if a cursor name is c1 and attribute is %FOUND then it is referred as:

```
c1%FOUND
```

Let's now learn about each attribute of explicit cursor with an example.

#### %FOUND Attribute

The %FOUND Attribute provides information about the recent fetch operation. If the recent fetch statement results into successful fetching of row, it returns TRUE. Otherwise it returns FALSE. If

Using this attribute, you can construct condition for `WHILE ...LOOP` for fetching rows from cursor. See Listing 7.6 for use of `%FOUND` in `WHILE LOOP`.

**Listing 7.6:** Use of `%FOUND` attribute in `WHILE LOOP`

Before using c1%FOUND, at least one fetch statement should be executed. Before WHILE loop, one FETCH statement is required. Inside WHILE loop the FETCH statement is also required to fetch rows one by one from cursor. WHILE loop is terminated when either last fetch operation is unable to fetch a new row or counter variable cnt equals to 5.

Here you can see the output of the Listing 7.6

**Output:**

```
The fifth maximum basic salary is 7000.
PL/SQL procedure successfully completed.
```

On executing Listing 7.6 on *iSQL Plus will display fifth maximum basic salary which is 7000 in our case.

## %NOTFOUND Attribute

The %NOTFOUND attribute is complement of %FOUND attribute. The %NOTFOUND returns FALSE when %FOUND returns TRUE and vice versa. Like FETCH statement, %NOTFOUND returns NULL before first fetch statement.

See Listing 7.7 which calculates number of rows in emp_detail table using LOOP and %NOTFOUND attribute.

**Listing 7.7:** Use of %NOTFOUND attribute

In Listing 7.7, EXIT-WHEN statement will execute when fetch operation is unsuccessful which makes condition c1%NOTFOUND of this statement TRUE and loop terminates. After closing the cursor c1, value of cnt which represents total number of rows displayed using PUT_LINE function.

Here you can see the output of the Listing 7.7.

**Output:**

On executing Listing 7.7 on *iSQL Plus will display total number of rows in `emp_` which is 5 in our case.

### %ISOPEN Attribute

The `%ISOPEN` attribute returns `TRUE` if cursor is open otherwise it returns `FALSE`. It is global cursor when the programmer is not sure whether cursor is opened or not. code snippet which checks whether cursor is opened or not:



The `c1%ISOPEN` condition evaluates to `TRUE` when cursor `c1` is already open or not, if cursor `c1` needs to be opened.

### %ROWCOUNT Attribute

The `%ROWCOUNT` attribute provides number of rows which are currently fetched from the cursor

In Listing 7.8, we access the value of `%ROWCOUNT` using `c1%ROWCOUNT` expression before opening the cursor `c1`, after first and second fetches. We also tried to access it after closing the cursor `c1` that raises the exception `INVALID_CURSOR`, which is handled by exception block.

Here you can see the output of the Listing 7.8

On executing Listing-7.8 on *iSQL Plus, you will get above output. The first line of output will display total number of rows fetched after opening the cursor. Second and third lines total number of rows fetched till first and second fetch operations. Accessing `%ROWCOUNT` attribute after closing the cursor `c1` will raise `INVALID_CURSOR` exception which is handled by exception block. In exception block, we display a message `You cannnot acces %ROWCOUNT attribute after closing the cursor c1` to user.

See table 7.1, which summarizes the values of all explicit cursor attributes in different conditions.

## *Implicit Cursor Attributes*

Implicit cursor attributes are used to test outcome of any DML or SELECT ... INTO statement. The attributes always refer to recent SQL statement. If there is no SQL statement before use of these attributes, the attributes are having NULL values. Implicit cursor attributes include SQL%FOUND, SQL%NOTFOUND, SQL%ISOPEN, and SQL%ROWCOUNT. In all these attributes, the name of cursor is SQL. We are now explaining each implicit attribute.

### SQL%FOUND Attribute

The SQL%FOUND attribute returns TRUE if any statement such as INSERT, DELETE, UPDATE or SELECT affects at least one row. Otherwise this attribute returns FALSE. Before execution of these statements, it returns NULL. See Listing 7.9 which shows the use of SQL%FOUND attribute.

**Listing 7.9:** Use of SQL%FOUND attribute

In Listing 7.9, we increment the basic salary of an employee with emp_no 101 by 5000 using UPDATE statement and then use SQL%FOUND attribute to know whether UPDATE statement is successfully executed.

Following is the output of the Listing 7.9.

```
Required record is affected.
PL/SQL procedure successfully completed.
```

The SQL%FOUND attribute always refers to recent SQL statement. See Listing 7.10 which tells the effect on value of SQL%FOUND attribute in PL/SQL program that involves a procedure call.

**Listing 7.10:** Effect on value of SQL%FOUND attribute

InListing 7.10, calcamt is a procedure call and SQL%FOUND attribute refers to SQL statement in procedure calcamt.

If you want to refer to UPDATE statement, you need to store the value of SQL%FOUND in some variable as shown in the following code snippet:

In the above code snippet, x is Boolean variable used to store the value of SQL%FOUND attribute.

### SQL%NOTFOUND Attribute

It performs opposite operation of SQL%FOUND attribute which means if SQL%FOUND returns TRUE, SQL%NOTFOUND returns FALSE.

When a SELECT ...INTO statement does not return any row, it is useless to use SQL%NOTFOUND because the statement raises exception NO_DATA_FOUND. However, when group function is used in SELECT...INTO statement, then exception NO_DATA_FOUND is never raised as group function always returns a value or a null. The IF condition involving SQL%NOTFOUND attribute is always checked but it is also not TRUE in this case. Listing 7.11 shows the use of SQL%NOTFOUND attribute if you are using only exception handler OTHERS then you can use SQL%FOUND attribute shown in listing 7.10.

**Listing 7.11:** Use of SQL%NOTFOUND attribute

Listing 7.11 uses group function MAX in SELECT INTO statement to find maximum salary from emp_detail table. This statement returns a row whether emp_deatil table does not contain any row. Therefore, IF condition is checked and control does not go to EXCEPTION block.

### SQL%ISOPEN Attribute

Oracle closes the SQL cursor or implicit cursor when execution of a query completes, therefore, SQL%ISOPEN attribute always returns FALSE value.

### SQL%ROWCOUNT Attribute

The SQL%ROWCOUNT attribute returns number of rows affected by INSERT, UPDATE or DELETE statement. If no rows are affected or SELECT ...INTO returns no rows, then the value of SQL%ROWCOUNT is zero. When SELECT...INTO statement returns more than one row, then exception TOO_MANY_ROWS is raised and %ROWCOUNT returns value not the number of rows, which satisfies the condition. Suppose you want to perform an action if more than 20 rows of a table are updated. You can perform this action using SQL%ROWCOUNT attribute as shown in the following code snippet:

If affected number of rows is greater than 20, statement A will execute.

Let us discuss FOR LOOP which derives after concept of cursors.

## Cursor FOR loop

A cursor FOR loop is a FOR loop associated with an explicit cursor. You need to use cursor FOR loop when you want to fetch and process each record of cursor. The concept of cursor FOR loop originated based on the general way a cursor is used. You usually need to perform the following steps for using a cursor.

1. Open the cursor using OPEN statement.
2. Begin any LOOP statement.
3. Fetch a row from cursor using FETCH statement.
4. Check whether another row exist in cursor using %FOUND cursor attribute.
5. Manipulate the data of fetched row.
6. End the opened loop.
7. Finally, close the cursor.

Hence, general usage of cursor enforces to introduce concept of cursor FOR loop. The cursor FOR loop implicitly performs most of these steps and hence reduces the code you need to write for fetching data from cursor. See Listing 7.12 which shows use of cursor FOR loop.

**Listing 7.12:** Use of cursor FOR loop

In Listing 7.12, we declared a cursor `c1` associated with some query. Loop index `z` is automatically declared with data type `c1%ROWTYPE`. The records are fetched one by one. The fields of automatically declared record `z` store the value of the row, which is fetched from the cursor `c1`. Inside the `FOR LOOP`, we accessed `emp_name` field of record `z`.

Here is output of Listing 7.12



On executing Listing 7.12 on iSQL*Plus workspace, you will see names of all employees from `emp_detail` table.

***Note***

> You cannot use the name of cursor e.g c1 in the enclosing FOR LOOP.

You can substitute query associated with cursor `c1` in place of its name in `FOR LOOP`. See Listing 7.13 which substitutes cursor query in `FOR LOOP`.

**Listing 7.13:** Using cursor query in `FOR LOOP`



In Listing 7.13, we added the salaries of all employees one by one in `FOR LOOP`. Then, we use `PUT_LINE` method to display sum of all salaries.

Here is output of Listing 7.13.

```
Sum of basic salaries of all employess in ABC organisation is 73500
PL/SQL procedure successfully completed.
```

When you execute Listing 7.13 on iSQL*Plus workspace, you will get `total` (73500 in our case) printed on iSQL*Plus.

The parameters are passed to cursor after the name of a cursor in `FOR LOOP`. See Listing 7.14 which uses parameterized cursor `c1` in `FOR LOOP`.

**Listing 7.14:** Use of parameterized cursor in `FOR LOOP`

The Listing 7.14 passes parameter deptno equals to 40 to cursor `c1`. The cursor `FOR LOOP` all employees that belong to department number 40.

Here is output of Listing 7.14.

When you execute listing on iSQL*Plus workspace, you will get names that department number 40. In our case, three employees Mandeep, Deepak, and displayed.

After understanding cursor FOR loop, Let's discuss cursor variables which act as pointers to result set of cursor and can be passed to PL/SQL stored programs.

## Cursor Variables

Cursor variables are used when you want to use cursor facility but do not want to bind cursor to a specific query. The cursor variable points to current row in the result set of the query just like the cursor. The difference between cursor and cursor variable is that a cursor is bound to only one specific query while cursor variable can hold many compatible queries. Cursor variables can be passed as formal parameters to subprograms such as procedures or functions. You can also open a cursor variable in a subprogram and process it in another subprogram on server side.

Data type of cursor variable is `REF CURSOR`. `REF CURSOR` type may be strong or weak. In case of strong `REF CURSOR` types, PL/SQL compiler associates cursor variable to queries that generate correct set of columns. In case of weak `REF CURSOR` types, PL/SQL complier can associate cursor variable to any query. If you are using same `REF CURSOR` type in every subprogram of an application, you can declare `REF CURSOR` type in a package body.

To declare a cursor variable, first you need to create `REF CURSOR` type in any PL/SQL block, subprogram, or package. Then you can declare cursor variables of created `REF CURSOR` type. See following code snippet which declares a cursor variable:

In the above code snippet, empcur is strong REF CURSOR type as it specifies RETURN type and gencur is weak REF CURSOR type. Therefore, corresponding cur1 and cur2 are strong and weak cursor variables.

You can control cursor variables using OPEN-FOR, FETCH, and CLOSE statements. The statement OPEN FOR associates query with the cursor variable. It is having following syntax:

```
OPEN cursorvariablename
FOR SELECT statement;
```

You cannot pass the parameters in the OPEN...FOR statement. Similarly, the SELECT statement should not use FOR UPDATE clause and can refer PL/SQL variables, parameters and functions.

The OPEN FOR statement opens a cursor variable for different queries. After opening a cursor variable, if it opens again for a new query, then previous query is lost but if a normal cursor is opened again, it raises predefined exception CURSOR_ALREADY_OPEN.

Just like explicit cursor, FETCH statement is used to fetch data from a cursor variable. Following is the syntax of FETCH statement, which is similar to that in explicit cursor:

```
FETCH <cursorvaraiblename>
INTO <variable list>
```

The variable list should have compatible variables as opened cursor.

The rows in result set are also determined at the time of opening the cursor. If you want to change set of rows, reopen your cursor.

In case of weak type declaration, you should use appropriate types in FETCH statement as error occurs at runtime. Therefore, weak type can generate ROWTYPE_MISMATCH exception which is raised when values of actual and formal parameters are incompatible. You need to make an exception handler block for this exception and use proper data types to ensure no loss of rows. In case of strong type, error occurs at compile time.

If you use procedure for fetching the cursor, then variable should have IN declaration. If procedure both opens and fetches rows, then it should have IN OUT declaration.

When you execute a FETCH statement before opening or after closing the cursor, exception INVALID_CURSOR raised.

Just like explicit cursor, CLOSE statement used to close cursor variable.

See Listing 7.15 which shows the example of strong cursor variable.

**Listing 7.15:** An example of strong cursor variable

Listing 7.15 first declares the `rec_type` record having two fields `ename` and `edes`. We first created strong reference type `cur_num_type` with return type `rec_type` and declare cursor variable `cur_num` of `cur_num_type`. Then we open the cursor variable for specified query, fetch the rows into record `z` and access the name and designation of each employee one by one.

Here is output of Listing 7.15.

When you execute Listing 7.15 on iSQL*Plus workspace, you will see name and designation of each employee in new lines. This example also prints the number of records retrieved into cursor `c1`.

See Listing 7.16 which shows the use of weak cursor variable.

**Listing 7.16:** An example of weak cursor variable

Here is output of Listing 7.16:

When you execute Listing 7.16 on iSQL*Plus workspace, you will see above output. We divide this output into two parts and each output begins with a line of its description.

You can pass cursor variables as a parameter to procedures or functions which manipulate them. See Listing 7.17 which shows the use of cursors and procedures.

**Listing 7.17:** Passing cursors to procedures

The Listing 7.17 declares a strong cursor variable emp of `emp_cur_typ REF CURSOR` type. In execution section, we opened the cursor variable `emp` for two separate SQL queries and pass each opened instance to procedure `process_emp_cv`. The procedure only accesses and prints the names of employees satisfying each SQL query.

Here you can see the output of Listing 7.17.

When you execute Listing 7.17 on iSQL*Plus workspace, you will see first names of employees which belong to technical writing department and then names of employees which joined in ABC organization after 1989 .

Let us understand cursor expressions that are used to execute complex queries.

## Cursor Expressions

The cursor expression returns nested cursor specified by cursor query. The syntax of cursor expression is as follows:

```
CURSOR(subquery)
```

The subquery which generates a result set that contains values (of simple data types such as `NUMBER`, `CHAR`) and cursors generated by cursor expression. If you want details about inner cursor, you need to use nested loops which first fetch data from rows of result set and then fetch data from rows of inner cursor which is present in result set.

You can write cursor expressions in explicit cursor declarations, `REF CURSOR` declarations and `REF CURSOR` variables.

You need not to worry about opening the nested cursor as it is implicitly opened when rows fetched from the parent cursor. A User can close nested cursor explicitly or it is closed automatically in following situations:

- When user reopens the parent cursor
- When user closes the parent cursor
- When parent cursor is cancelled
- If an error arises when executing `FETCH` statement on parent cursor, nested cursor closed during clean up operation.

See Listing 7.18 which uses cursor expression.

**Listing 7.18:** Using cursor expression

The Listing 7.18 finds department number as an `ID` and cursor from which we can fetch all the employees's names in a specific department. You can access other details of a particular employee with the help of inner cursor. The cursor `c1` is parent cursor. Inner cursor `emp_detail` automatically opened when the parent row is fetched. Note that we have closed only parent cursor `c1`.

Here is output of Listing 7.18.

When you execute Listing 7.18 on iSQL*Plus workspace, you will get employees belonging to particular department. In our case, we get employees's names of `OPERATIONAL` department.

***Note***

Let's facilitate locking on rows of cursor using `SELECT...FOR UPDATE` statement so that other users cannot change those rows and perform manipulations on rows using `WHERE...CURRENT OF` clause.

## FOR UPDATE in Cursors

are now implementing row level locking in Cursors. In row level locking, the row which is read or modified is locked. The row level locks are finest locks as compared to table level A locked table or row is known as lock object and stored in lock control block in shared . When you execute a `SELECT` query on the database, there are no locks on retrieved or rows. By default, only those rows are locked which are changed but are still not Sometimes you want to lock some records before you change them in a program. provides the `FOR UPDATE` clause of `SELECT` statement to perform this locking.

... `FOR UPDATE` statement is executed to obtain exclusive locks on rows selected by `SELECT` statement. You can change these records as each row is fetched from the cursor and any one else cannot change them when you perform a `ROLLBACK` or a `COMMIT` operation. This statement can be qualified or not. The qualified `SELECT...FOR UPDATE` statement specifies `OF` list with it. We are first using unqualified `SELECT FOR UPDATE` statement.

Listing 7.19 which uses unqualified `SELECT FOR UPDATE` statement.

**Listing 7.19:** Use of unqualified `SELECT FOR UPDATE` statement

In Listing 7.19, the cursor `c1` uses unqualified `FOR UPDATE` clause. When you execute this code snippet, no other user can change any selected rows.

Let's now use qualified `SELECT FOR UPDATE` statement. The `OF` list of `FOR UPDATE` clause specifies list of name of columns of table. The `OF` list reminds you what you want to change. Still the all rows identified by `SELECT FOR UPDATE` statement are locked. Usually, you need `SELECT FOR UPDATE` statement in case you are retrieving rows from multiple tables.

See Listing 7.20 which uses qualified `SELECT FOR UPDATE` clause.

**Listing 7.20:** Use of qualified `SELECT FOR UPDATE` clause

In Listing 7.20, the cursor `c1` uses qualified `FOR UPDATE` clause. When you execute this code snippet, other users cannot change values of `loc` column that are contained in result set.

You can also associate a NOWAIT keyword to FOR UPDATE clause. This Oracle to return control immediately to your program if a table is locked by another user.

We are now performing update or delete operations on current row. The WHERE clause is used inside UPDATE and DELETE statements to make changes to current from the cursor. The syntax of WHERE CURRNT OF clause in UPDATE statement is

The syntax of WHERE CURRNT OF clause in DELETE statement is as follows:

In both syntaxes, WHERE CURRENT OF clause refers to cursor not the record which is to fetch the current row. The key benefit of using WHERE CURRENT OF clause is that you do not need to repeat criteria in WHERE clause in your cursor declaration and associated UPDATE and DELETE statements. In WHERE CURRENT OF clause, you need to mention criteria only in WHERE clause of SELECT statement. See Listing 7.21 which uses WHERE CURRENT OF clause.

**Listing 7.21:** Using WHERE CURRENT OF clause

In Listing 7.21, you are not referring to particular record in result set .The WHERE CURRENT OF clause with specified cursor enforce increment, or 5000 to basic salary of employee represented as current row fetched in record z.

When you will execute SELECT query on emp_detail table, you will find basic salary of each employee incremented by 5000.

## Summary

In this chapter you learnt about:

- What are cursors and their different types
- How to declare, open, fetch and manipulate rows and close explicit cursors
- Use of Implicit and Explicit cursor attributes
- Cursor FOR LOOP
- Strong and weak cursor variables with examples
- Cursor expressions
- Implementing row level locking in cursors and performing manipulations on rows of cursor

PL/SQL, like other programming languages, also provides subprograms that works as small building blocks and helps in creating and managing large applications very easily. A subprogram consists of a sequence of statements to perform some specific task and also provides modularity. For example, if you want to calculate the length of a string or want to calculate the sum of few numbers then you can create subprograms for that purpose. Once the subprogram is created to perform any specific task and also executed successfully then that subprogram can be called any where in the same application or in any other application. A subprogram can be used many times in an application thus it also provides the reusability. If you will made any changes in the subprograms then that changes will affect only that part of application where that subprogram has been called. Changes made to a particular subprogram will not affect the whole application thus it makes the application easily manageable.

In this chapter, we will discuss about the subprograms in details that is what are subprograms, what are subprogram parameters, how to overload subprograms, how to use recursion with subprograms, etc. Let's start discussing subprograms.

## Overview of PL/SQL Subprograms

In the chapter number 2 of this book, we had described PL/SQL blocks. PL/SQL blocks are broadly divided into two categories that are anonymous block and named block. A PL/SQL named block is also known as PL/SQL subprogram that can be called anywhere in a PL/SQL application. PL/SQL subprogram has following two types:

- Procedures
- Functions

Let's study a PL/SQL subprogram separately as procedure and function.

### *PL/SQL Procedures*

A PL/SQL procedure is a subprogram, which is used to perform some specific action. Statement used to create a procedure is `CREATE PROCEDURE`. A procedure consists of two parts; first part is known as specification part and another part is known as body. Let's see the format of a procedure.

Here in the preceding format the specification part consist of the following:

In this way we create a procedure. Let's see Listing 8.2 to know how to call a procedure PL/SQL program.

**Listing 8.2:** Calling a sample procedure

In Listing 8.2, in the declaration section, we have declared two variables; first to employee number (`eno1`) and the second to retrieve the employee basic salary (`e_`

In the execution part, we have passed an employee number in eno1; corresponding to number the procedure created in Listing 8.1 fetch the employee's basic salary.

Then, we have called the procedure created in the Listing 8.1 and has passed the that procedure.

After that, we have displayed the result.

Let's see the output of the Listing 8.2.

In this way, you can create and call a procedure. Another part of subprogram is function. Now, we depict functions in PL/SQL.

## *PL/SQL Functions*

A PL/SQL function is a subprogram, which is used to perform some computations. used to create a function is `CREATE FUNCTION`. A function is structurally similar to a procedure, except that a function have `RETURN` clause. A function consists of two parts; first part is known as specification part and another part is known as body. Let's see the format of a function.

Function like procedure has two parts-Specification and the body. The specification part consists of the three parts; function name, parameter declaration, and return data type. Let's see the specification part in detail as follows:

- **fun_name:** It is representing the name of the function. You can give any name to the function according to the requirement of your application and ease of remembrance. For example, if you want to create a function to calculate the sum of few numbers then you can name that function as sum.
- **Parameter declaration:** It consists of the following three parts:
  - **param_name1:** It is representing the parameter name. You can specify any number of parameters you want or you can also omit the parameters.
  - **para_mode:** It is representing the parameter mode. Parameter mode can be `IN` or `OUT` mode. If you will not specify any parameter mode then `IN` mode will be the default mode.
  - **para_type:** It is representing the data type of the parameter that can be of any PL/SQL provided data type. Here, you do not have to specify any constraints with data type such as length with `NUMBER` and `VARCHAR2` data type.

*Note*

It is optional, if a procedure does not have any parameter, parenthesis is not used.

  - **return_type:** It is representing the data type of the return value and is specified using `RETURN` clause.

This is all about the specification part of a function. Let's see what the PL/SQL function body consists of:

- **Declaration part:** In this part, local variables are declared. Declaration part starts with `IS` keyword and continues till the start of `BEGIN` keyword.
- **Execution part:** This part can have one or more PL/SQL executable statements. It should have one or more return statements with an expression that is in the form `RETURN` (expr). The expression should return the value of the data type mentioned in the return data type.
- **Exception part:** This is an optional part and can be aborted. This part contains the statements used to handle exceptions which are generated in PL/SQL function.

*Note*

Let's see the Listing 8.3 to create a function.

**Listing 8.3:** Showing a sample function creation



In the Listing 8.3, we have created a function named `maxsal` to retrieve the maximum salary earned by an employee.

To do that first, we have declared a variable of type `NUMBER`, which is used to store the retrieved from the database table.

Then, we have used select statement with `MAX` function to fetch the maximum basic salary from the database table `EMP_DETAIL`.

Then, we have returned the maximum salary retrieved through `RETURN` statement.

Output of the Listing 8.3 is as follows:

```
Function created
```

Let's see Listing 8.4 to know how to call a function.

**Listing 8.4:** Calling a sample function



In the Listing 8.4, we are calling the function created in the Listing 8.3. In the Listing 8.4, have declared two variables; `m_sal` and `disp`.

Then, we have called the function `maxsal` in which we have passed the variable `m_sal` as parameter.

This parameter is receiving the value returned by the function `maxsal` in the Listing 8.3.

Then, we have assigned the received value to variable `disp`, which is then used to display the maximum basic salary fetched through the function `maxsal`.

Let's see the output of the Listing 8.4.

```
Highest salary in the table EMP_DETAIL  is 14000
PL/SQL procedure successfully completed.
```

With this, we have ended discussion on Overview of PL/SQL subprograms. In this section, we have discussed about procedures and functions in which we have talked about parameters. Let's continue our discussion by discussing subprogram parameters.

## Working with Subprograms Parameters

PL/SQL subprograms (procedures and function) used parameters to pass the information between the PL/SQL programs and subprograms. Parameters are equally important as much as the other code in the program. You can use any number of parameters in a subprogram but be assured that the number and type of the parameters must have to be same in both; the subprogram and the program that will call the subprogram. Let's study the types of parameters.

### *Types of Subprogram Parameters*

PL/SQL offers two types of parameters that can be used in subprograms. Those parameters are as follows:

- **Formal Parameters:** The parameter used in the implementation or definition of subprogram are called formal parameters.
- **Actual Parameters:** The parameter that we used in calling the subprograms are known as actual parameters.

Let's consider the Listing 8.1 and 8.2 to understand subprogram parameters. In Listings 8.1&8.2, we have used parameters during creating and calling the procedure. Let's see that which parameters were formal and which were actual parameters in those listings.

In the Listing 8.1, we have used create procedure statement with parameters to create a procedure. Statement used in the Listing 8.1 is as follows:

```
CREATE OR REPLACE PROCEDURE salary (eno IN NUMBER, esal OUT NUMBER)
```

The parameters `eno` and `esal` used in the above statements are formal parameters.

In the Listing 8.2, we have called the procedure created in the Listing 8.1. We have called the procedure as follows:

```
salary (eno1, e_sal1);  -- Caling the procedure
```

We have passed two parameters that are `eno1` and `e_sal1` inside the braces. These parameters are actual parameters.

You must have noticed that during creating and calling a procedure, we have used a different name for parameters. It is not compulsory to use different name; you can also use the same name.

Whenever a subprogram is called the actual parameter is substituted in place of formal parameter. Thus, `eno1` is substituted in place of `eno` and `e_sal1` for `esal`.

## Using Notation for Subprogram Parameters

Notations are the methods that are used to pass the actual parameters. PL/SQL provides three types of notations that can be used to pass the actual parameters in different ways. Those notations are as follows:

- Positional Notation
- Named Notation
- Mixed Notation

Now, we discuss these notations in details.

### Positional Notation

It is the notation in which the actual parameter should have the same position as of formal parameters. If you will change the position then Oracle will raise errors. In Listing 8.1, we have created a procedure named salary in which we have passed two parameters, eno and esal respectively. Then, we have called that procedure in the `Listing 8.2`. The notation, we have followed in the Listing 8.2 is positional notation. Here, you can see the way; we have used to call the procedure.

```
salary (eno1, e_sal1); -- Using positional notation
```

In this case, the formal parameter `eno` is substituted with `eno1` and `esal` with `e_sal1`.

### Named Notation

In this method, the actual parameter is associated with the formal parameter using `arrow (=>)` symbol. If you use this notation then the sequence of parameters does not matter. This notation enhances the readability and maintainability of your PL/SQL subprogram.

Notation used in Listing 8.2 can also be replaced with named notation, if you do not want to follow the sequence. If you replace the notations used in the Listing 8.2 with named notations then, the excerpt having notation in the Listing 8.2 will be as follows:

```
salary (eno=>eno1, esal=>e_sal1); -- Using named notation
```

### Mixed Notation

You can use both positional and named notation in call. The positional notation must precede the named notation otherwise Oracle will raise error. In this case, the notation used in the Listing 8.2 can be as follows:

```
salary (eno1, esal=>e_sal1); -- Using mixed notation
```

## Using Parameter Modes

Parameter modes are used to define the behavior of formal parameters. Parameters can be passed in three modes. The default mode for passing parameter is `IN` mode. If you will not specify any mode then Oracle will use the default mode. The three different parameter modes are as follows:

- IN mode
- OUT mode
- IN OUT mode

Let's study these modes in details. We will study these modes in the same order as given above.

### IN mode

This mode is used to pass the value to a subprogram. Inside the subprogram, IN acts as a constant and any attempt to change its value causes compilation error. The actual parameter corresponding to parameter having IN mode can be initialized to variables, constants, literals, or expressions. Let's reconsider the procedure salary created in Listing 8.1. Statement for that procedure is as follows:

```
CREATE OR REPLACE PROCEDURE salary (eno IN NUMBER, esal OUT NUMBER)
```

In this procedure, we have passed the parameter eno in the IN mode and esal in the OUT mode.

**Note**

OUT mode will be depicted in the next topic.

Let's see Listing 8.5 to know how the actual parameter corresponding to parameter having IN mode (eno) can be initialized in various ways.

**Listing 8.5:** Showing the use of actual parameter in context to IN mode

Output of Listing 8.5 is:

```
PL/SQL procedure successfully completed.
```

In the above topic we have also talked about the OUT mode. Let's study what is that OUT mode, why it is used, and how to use this mode?

### Out mode

This mode is used to return a value to the PL/SQL program that calls a subprogram. Inside a subprogram, the parameter having OUT mode acts as an uninitialized variable. Let's reconsider the procedure salary created in Listing 8.1. Statement for that procedure is as follows:

```
CREATE OR REPLACE PROCEDURE salary (eno IN NUMBER, esal OUT NUMBER)
```

In the above statement, the parameter esal is of OUT mode. The actual parameter for OUT mode must be a variable. Let's see Listing 8.6 to know what will happen if the parameter for OUT mode is not a variable.

**Listing 8.6:** Showing the use of actual parameter in context of `OUT` mode

In the Listing 8.6, e_sal1 is the actual parameter and is a variable assigned with When you execute this listing then the following output will be displayed.

But, if you try to execute this listing after removing the comment beside `salary e_sal1+2); -- Not allowed` statement then you will get the following error:

This error is trying to convey that the actual parameter for `OUT` mode must be a variable not expression or any other value such as literal.

### IN OUT Mode

The `IN OUT` parameter is used to pass value to a subprogram and for taking the value out of a subprogram. Inside the subprogram, the parameters used with `IN OUT` acts like initialized variable. Thus, you can refer value of variable or assign it some other value.

The actual parameter for `IN OUT` formal parameter must be a variable. If any expression or constant is tried to assign to actual parameter then Oracle will generate the error. Let's see the Listing 8.7 to know how to use this mode.

In Listing 8.7, we have created a procedure sum to calculate the sum of two numbers. We have used a parameter `a` with `IN OUT` mode and is of type number. On executing the Listing 8.7, output will be as follows:

```
Procedure created.
```

Now, we call the procedure created above in Listing 8.8.

**Listing 8.8:** Calling the preceding procedure



In the Listing 8.8, we have declared a variable `B` of type number and have assigned a value to this variable.

Then, we have called the procedure created in Listing 8.7. On executing the Listing 8.8, output will be.

```
Sum of the value passed is 20
PL/SQL procedure successfully completed.
```

This listing is executed properly because the actual parameter used is of type variable but if you will try to execute the listing after removing the comment beside `sum (20);` then you will get the following error:



## *Using Subprogram Aliasing*

Aliasing occurs when a global variable is referred in the procedure and is also passed as parameter to a subprogram. The result of aliasing depends on the type of parameter passing method being used by the compiler.

The parameters are passed using two methods- pass by-value and pass by- reference. When a parameter is passed by-value method, the value of an actual parameter is passed to the subprogram. But, with by-reference method, address attached with the value of actual parameter is passed. `OUT` and `IN OUT` parameters are passed by value while `IN` parameters are passed by reference. Let's see Listing 8.9 to understand subprogram aliasing by the method pass by-value.

**Listing 8.9:** Showing subprogram aliasing in context to pass by-value

In Listing 8.9, first we have declared a global variable, which is then followed by creating a procedure named **demo**.

Then, we have assigned a value to the procedure parameter. Mode of the parameter used is `IN OUT` that means in this listing, we use the method pass by-value. After that, we have called the procedure and have passed the global variable in that procedure as actual parameter.

Let's see the output of the Listing 8.9.

```
10
PL/SQL procedure successfully completed.
```

Let's see Listing 8.10 to understand subprogram aliasing by the method pass by-reference.

**Listing 8.10:** Showing the subprogram aliasing in context to pass by-reference

In the Listing 8.10, first we have declared two global variables.

Then, a procedure named `demo1` has been created in which we have passed two parameters; first with the `IN` mode and second with the `OUT` mode.

After that, we have called the procedure and have passed the global parameters in that procedure.

output of this listing.

we end out discussion on subprogram aliasing and on working with Subprogram is also complete. Now, you should be able to understand and use subprogram very easily.

study about the subprogram overloading that allows you to use same name for more

## Subprograms

overloading means using the same subprogram name for different subprograms. the various subprograms can be same but there should be some difference between can be set by varying the number of subprograms formal parameter, order, or

Listing 8.11 to understand subprogram overloading.

**1:** Showing subprogram overloading

In Listing 8.11, we have created two functions; first to fetch the employee name corresponding to the number passed when the function will be called and second fetch the employee number corresponding to the employee name. Both of the function have same name that is ename but varies in the parameter name and data type. Let's see the output of the Listing 8.11.

In this way, subprograms can be overloaded but they also have some restrictions in overloading. Let's study about those restrictions.

### *Restriction while applying Overloading*

1. Standalone subprograms can not be overloaded. For example, if you will try to overload the function created in Listing 8.3 then you will receive the following error because the function created in that Listing is a standalone function:

But this problem can be solved if you will create packaged subprograms or by creating subprograms as we have created in the Listing 8.11.

2. Subprograms having the same name but vary in formal parameter name or mode can not be overloaded. In that case the following procedures can not be overloaded:

3. Subprograms having the parameters of same data type family that is char and `VARCHAR2` can not be overloaded. In that case, following two subprograms can not be overloaded.

Here, we have completed discussion on overloading subprograms. Sometimes, it may happen that you need to call a subprogram by itself then what will you do. Solution to call a subprogram by itself is recursion. Let's continue discussing recursion in subprograms.

## Using Recursion with Subprograms

Recursion is the process to call a subprogram by itself. When a subprogram is called then a new instance of items (such as parameters, variable, SQL statements) declared in the subprogram is created. Let's see Listing 8.12 to generate the Factorial series.

**Listing 8.12:** Showing the Factorial series in context to recursion

Output of the Listing 8.12 is as follows:

```
Function created.
```

Now, we call the procedure created to calculate the factorial of a specific number in the Listing 8.13.

**Listing 8.13:** Calling the fac function

Let's see the output of the Listing 8.13.

```
Factorial of five is 120
PL/SQL procedure successfully completed.
```

Recursive programs are simple to write, but are inherently slow. In recursion, the information is put in stack for each subprogram call. So it requires a large stack. If a subprogram call is made within cursor, `LOOP–END LOOP` or between `OPEN CURSOR` and `CLOSE CURSOR` statement, then for each call a new cursor is opened and the cursor limit may exceed the maximum limit of `OPEN_CURSOR`.

With this we have completed discussion on recursions. Now, we discuss about the definer and invoker rights related to subprograms.

## Using AUTHID Clause

This clause is used to set the privileges to execute the subprograms. By default the subprograms have definers or owner rights, which mean that any reference to database tables are resolved at compile time. You can set the rights to owner or invoker by using `AUTHID` clause.

Benefits of using invoker rights is that you can create only one instance of subprogram and many users can use that subprogram.

`AUTHID` clause is allowed to use in the header of a subprogram, packages, etc. `AUTHID` clause must be used just before the `IS` or `AS` keyword in the subprogram declaration.

You can specify `AUTHID CURRENT_USER` to provide the invoker rights and `AUTHID DEFINER` to set the definer rights. Let's see the following snippet to know that how to use `AUTHID` clause.

Packages consist of a set of procedures, functions, and variables grouped as a single unit. They help in building well structured PL/SQL applications. Specifically, you need to define packages in two cases. First case is when you need to create a context or an environment for many related program units of a specific domain. Second case is that your application has both private and public variables and its design is top-bottom approach oriented.

Oracle Corporation has increased the number of built-in packages to extend the functionality of PL/SQL. For example, the built-in package `STANDARD` includes the definition of basic operators and built-in functions. You can also build your own package and include domain specific functionality inside this package. Some third parties can also build a library of packages which needs to be installed so that the PL/SQL environment can use them.

This chapter makes you understand what is a package, how to create package specification and its body, how to use some built-in packages from Oracle and create user defined package.

## Overview of PL/SQL Packages

A package is a construct that lets programmers to logically group the application components of one domain. Similar to subprograms, packages are also used to organize PL/SQL programs or applications but there are some differences between subprograms and packages. You can only store packages but subprograms can be both declared locally in a block or stored in a database. Packages impose fewer restrictions as compared to stored subprograms when using dependencies. Let's learn how to create user defined packages.

### *Package Specification and Package Body*

Oracle defines the top-bottom approach to create a user defined package which means that you first need to create its specification and then its body. Package specification is considered as a prototype of the entire package. This prototype enlists the resources of the package available to an application. The package body contains the implementation of that prototype. As earlier discussed, the data dictionary is used to store definition of schema objects. Therefore, the definition of the user defined package is also stored in data dictionary. Let's learn how to create a package specification and its body.

#### Package Specification

Package specification is also known as the package header. You can declare new types, variables, procedures, functions, cursors, and exceptions in a package specification and can also refer all these members from other PL/SQL blocks. See Listing 9.1 for a sample package specification.

**Listing 9.1:** Creating a sample package specification

In Listing 9.1, the package specification of `enq_analysis` package defines the data type `address_rec`, variable `lastenqno`, exception `invalid_date`, function `findname`, and procedure `updateenrollstatus`. Each object is an element of the package. These declarations can appear in any order but object declaration must precede its reference. There is also no need to specify all types of elements, such as exceptions or types. Declarations of procedures and functions should be forward declarations. A forward declaration describes the type and name of the subprogram along with its arguments.

### Body

The package body contains the implementation of package elements given in the package specification. The only elements that require the implementation in package body are cursor and subprograms. The declaration of subprograms in a package specification must match with the declaration of the same subprograms in the package body. The package body can contain additional variables and subprograms, which can be used for implementation of package elements in the specification. These are called private variables or private subprograms and are not accessible from outside. Package body becomes optional when package header does not contain forward declarations of subprograms and cursors. Still, we may need to define package header as all the variables and types in it are visible outside the package. Notice that the package body does not compile without successful compilation of package specification of same package. See Listing 9.2 for sample package body of `enq_analysis` package.

**Listing 9.2:** Creating body of `enq_analysis` package

In Listing 9.2, the variable `enqdate1` is a private variable, which can be used only inside the package body. Thus, the following reference in any PL/SQL block or program results in an error:

```
enq_analysis.enqdate1
```

The body of `enq_analysis` package defines cursor `c1`, `checkenqno`, `findname`, functions, and `updateenrollstatus` procedure. The package initialization part is executed when any package element is referred for the first time. Function `checkenqno` is private to package; therefore, it cannot be referred in any other PL/SQL program.

Any public member declared in a package header is accessed by qualifying the member name with the package name. In general, the public package elements are referred in other PL/SQL blocks using the following syntax:

```
packagename.elementname;
```

With the help of this syntax, you can now declare the variable `mailing_address` of `address_rec` record type in the program. The syntax to declare variable of type defined in `enq_analysis` package is as follows:

```
mailing_address enq_analysis.address_rec;
```

Let's learn how to access other members, such as procedures and functions of `enq_analysis` package. Suppose you want to use or access function `find name` in a PL/SQL program, you can do it using the following syntax:

```
x VARCHAR2(4);
x:=enq_analysis.findname(100);
```

You can also call packaged procedures using positional or named notation like stand alone stored procedures but packaged procedures should be prefixed by a package name.

In this section, you learned how to create your own package and how to access its members. Now, let's go through the advantages of packages.

## Advantages of Packages

Package is not a new concept in software engineering. Many software packages are continuously being built by software firms to solve the problems of developers. Packages help in creating reliable, maintainable, and reusable code, which help in developing software. These packages are now also used to build high scalable Oracle PL/SQL based software systems. Packages offer the following advantages for PL/SQL application or software design:

- **Modularity:** As packages group all the logically related items, such as data types, variables, and subprograms, they are modular in nature. All data types, variables, and subprograms have the same context. The user can easily understand a package as it can be logically broken into small program units. The interfaces between packages are also clear and well defined.
- **Information Hiding:** Package consists of specification part and implementation part. Only the specification part is available to users. The implementation part is hidden from the users for security reasons. Each user can access different functionalities of a package depending

upon access rights. The specification part is called public part while the implementation part is called private part. If the user needs to use packaged subprograms in a calling program, the user requires only knowledge about declaration of packaged subprograms. The declaration of subprograms is included in the package specification. This makes the developers and users more independent as developer can change the subprogram definition and user can invoke the changed functionality with the same syntax. For example, the calling program for a package math and function fact is as follows:

```
x:=4;
y:=math.fact(x);
```

Even if the definition of function fact is changed from recursion to iteration, the calling program is not changed.

- **Global Variables:** The variables declared in packages persist across the session. These variables, which include cursors, can also be used to pass information from one subprogram to another. Thus, you can assign a value to packaged variable in one subprogram and use it in another subprogram.
- **Simple Application Design:** The prototype of the application can be built very fast by only writing package specification. You can even compile package specification without its body and write other subprograms and compile them without the package body.
- **Better Performance:** When a package is referred for the first time, then all the package elements are loaded in memory so that the further reference to the package does not require disk I/O. This speeds up the application execution.

Let's study about the most commonly used built-in PL/SQL packages from Oracle 10g.

## Understanding PL/SQL Packages with Oracle

Each built-in package has a set of PL/SQL objects and they are automatically installed during the installation of Oracle database. Each package extends the functionality of PL/SQL. For example, `DBMS_SQL` package is used to execute dynamic SQL statements. All the Oracle built-in packages are classified into following three categories:

- **Application Development Packages:** The packages under this category are used by application developers to build Oracle applications.
- **Server Management Packages:** The packages under this category are used by database administrators for management of database servers.
- **Distributed Database Packages:** The packages under this category are used by both database administrators and application developers for managing distributed data.

See Table 9.1 which lists all the built-in packages used in application development with description

| Name |
| --- |
| DBMS_RANDOM |

Now, let's discuss some of the most commonly used packages.

## The STANDARD Package

The STANDARD package defines the predefined data types, exceptions, and subprograms, which are available to every PL/SQL program. For example, exception NO_DATA_FOUND is defined in the STANDARD package. This exception arises when either you execute a query, which do not identify any rows or you reference an uninitialized row during fetching rows from cursor. Most of the predefined subprograms, such as SUBSTR (to find substring from a string), TO_CHAR (convert a number to character value) are specified in this package. You can also overload most of the subprograms from package STANDARD. For example, you can overload TO_CHAR function in the following ways:

PL/SQL identifies that a call is made to a particular subprogram, such as TO_CHAR by matching a number of arguments and their data types. If the user has defined subprograms having same name, then the original subprograms can be called by using package name STANDARD as STANDARD.SUBSTR. Package STANDARD is available in pre compilers, forms, reports, and various Oracle products.

## The DBMS_PIPE Package

The DBMS_PIPE package is used to transmit information from one session to another. The first session makes use of DBMS_PIPE package to wrap a message into a message buffer and send it to memory area inside the pipe while the second session receives and unwraps the same message into variables. Before the introduction of the DBMS_PIPE package, the database used to act as a communication medium for sending and receiving of messages. With the help of DBMS_PIPE package, your application can also use external routine which do not reside in the database. All the service routines need to connect to Oracle instance and wait for their requests on a particular database pipe. Database pipes can be public or private. A public pipe can be accessed by any user for sending or receiving messages; whereas in case of private pipes, only the authorized users can send or receive messages during a session.

You must logon as a SYSDBA to execute the following command to give access to normal users:

```
grant execute on dbms_pipe to scott;
```

Table 9.2 lists the commonly used subprograms in this package.

**Table 9.2: Subprograms in DBMS_PIPE package**

| Name | Purpose |
|---|---|
| PACK_MESSAGE | Builds message into local message buffer. |

**Table 9.4: Commonly used subprograms of UTL_HTTP package**

| Name | Purpose |
|---|---|
| SET_PROXY | Sets URL of proxy server of main server of website to be read. |
| BEGIN_REQUEST | Initiates HTTP request to the specified URL. |
| SET_HEADER | Sets a specific response header with a value. |
| GET_RESPONSE | Gets response from server. |
| READ_LINE | Reads a line of text from specified URL of web page. |
| END_RESPONSE | Ends the response. |
| GET_HEADER_COUNT | Gets total number of headers. |
| GET_HEADER | Gets value of specific response header. |

## The DBMS_SQL Package

The DBMS_SQL package is used to execute SQL, DDL, and DML statements dynamically at runtime. A DDL statement can be used inside the PL/SQL block only with the help of a DBMS_SQL package; but to issue this statement, you further need some privileges from the administrator. DBMS_SQL package allows the developer to specify only conditions or portions of SELECT statement. From these conditions, this package builds the entire SELECT query.

DBMS_SQL package executes the dynamic PL/SQL programs in steps, which are as follows:

1. Store the names of procedures in a database table.
2. Build a UI for this table so that user provides the inputs to execute a particular procedure.
3. Execute the procedure.

You can hide DBMS_SQL package from all the users by issuing following command from SYS account:

```
REVOKE EXECUTE ON DBMS_SQL FROM PUBLIC;
```

The database administrator can provide access of DBMS_SQL package to a particular user by issuing following command:

```
GRANT EXECUTE ON DBMS_SQL TO username;
```

DBMS_SQL consists of many standard subprograms and data structures. Table 9.5 lists the commonly used subprograms inside this package.

**Table 9.5: Commonly used subprograms of DBMS_SQL package**

| Name | Purpose |
|---|---|
| BIND_ARRAY | Associates a value to host array |

**Table 9.5: Commonly used subprograms of DBMS_SQL package**

| Name | Purpose |
|---|---|
| BIND_VARIABLE | Associates a value to host variable |
| CLOSE_CURSOR | Closes the cursor |
| EXECUTE | Executes the SQL statement |
| FETCH_ROWS | Fetches the rows from cursor |
| OPEN_CURSOR | Opens the cursor |
| PARSE | Parse the passed SQL statement |
| COLUMN_VALUE | Retrieves a value from the cursor into a local variable |

Following are some of the unique advantages of DBMS_SQL package:

- Currently, the DBMS_SQL package is supported in client-side programs, but native dynamic SQL is not. Every call to the DBMS_SQL package from the client-side program translates to a PL/SQL remote procedure call. These calls occur when you need to bind a variable, define a variable, or execute a statement.
- The DESCRIBE_COLUMNS procedure in the DBMS_SQL package can be used to describe the columns for a cursor which is opened and parsed through DBMS_SQL.
- The DBMS_SQL package supports statements greater than 32 kilobytes.
- The DBMS_SQL package supports statements with a RETURNING clause that updates or deletes multiple rows.
- The PARSE procedure parses the SQL statement only once. Therefore, the statement can be used multiple times with different binding variables.

You can also execute dynamic SQL statements using statements, such as EXECUTE IMMEDIATE without using API of DBMS_SQL package.

## *Native Dynamic SQL vs. DBMS_SQL Package*

Native dynamic SQL enables you to place dynamic SQL statements directly into PL/SQL code. The DBMS_SQL package is a PL/SQL library that offers a programmatic API to execute SQL statements dynamically. Following list describes some of differences between them:

- Native dynamic SQL is much simpler to use than the DBMS_SQL package because it is integrated with SQL. You can use it in the same way as you currently use static SQL within PL/SQL code. The DBMS_SQL package is not as easy to use as native dynamic SQL. It involves many procedures and functions that must be used in a strict sequence.
- Native dynamic SQL is comparable with static SQL because PL/SQL interpreter has built-in support for native dynamic SQL. Therefore, using native dynamic SQL greatly improves the performance of programs as compared to the programs that use DBMS_SQL package.

- DBMS_SQL package is based on a procedural API and therefore, there are large procedure calls and data copy overhead. For example, when you bind a variable, DBMS_SQL package copies the bind variable into its space for later use. Similarly, when FETCH statement executes, the data copies into the space managed by DBMS_SQL package and then fetched data is copied one column at a time.

However, native dynamic SQL bundles the statement preparation, binding, and execution steps into a single operation that minimizes the data copying and procedure call overhead and improves performance.

- Native dynamic SQL provides support for the user defined types, such as user defined objects, collections, and REF. The DBMS_SQL package does not support these user defined types.
- Native dynamic SQL and static SQL both support fetching into records, but the DBMS_SQL package does not.

## The DBMS_ALERT Package

The DBMS_ALERT package is used to alert the application when a specific database value is changed by providing notification. These notifications are called alerts. Alerts are usually signaled using database triggers on specific tables. Alerts are asynchronous and only committed transactions can signal an alert. You do not need to check frequently now whether any change has taken place or not. The process of receiving an alert consists of two steps. Initially, the application registers itself to get an alert on modification of specific data. Then, the application waits for signaling of alert which tells developer to check again the same data. This package is also used to give signals to other sessions. It operates independently of any time mechanism.

Table 9.6 lists the commonly used procedures and functions used in this package.

**Table 9.6: Commonly used subprograms of DBMS_ALERT package**

| | Purpose |
|---|---|
| | Deregisters the named alert passed as an argument. |
| WAITANY | Waits for any alert to occur for which the current session is |

## Using PL/SQL Packages

In the last section, we discussed the functionality of packages, such as DBMS_SQL, DBMS_ALERT, UTL_HTTP, UTL_FILE, and DBMS_PIPE. Now we are going to use these packages using some examples.

### *The DBMS_SQL Package*

The DDL and DML statements of SQL can be executed dynamically at runtime with the help of DBMS_SQL package. Under this section, we create two examples in Listing 9.3 and Listing 9.4, respectively. First example increments the salaries of employees who belong to department number 10 of table emp. Second example finds ids and names of employees from table emp whose salary is greater than 3000.

Listing 9.3 executes the dynamic UPDATE statement using DBMS_SQL package on table emp.

**Listing 9.3:** Executing dynamic UPDATE statement



Perform the following steps to execute dynamic SQL statement with the help of DBMS_SQL package:

1. Create a cursor to get cursor handle for a dynamic SQL statement.

2. Build the SQL statement and parse it in Oracle version 7 mode. Our SQL statement contains two bind variables acting as placeholders.
3. Associate bind variables to actual variables passed in procedure.
4. Execute the SQL statement.
5. Close the cursor.

Here is the output of Listing 9.3.

```
Number of rows updated 3
PL/SQL procedure successfully completed.
```

When you execute Listing 9.3 on iSQL*Plus console, it will increment the salaries of three employees and shows the total number of rows updated.

Second example finds ids and names of employees from table `emp` whose salary is greater than 3000. This example also uses `FETCH_ROWS`, `DEFINE_COLUMN`, and `COLUMN_VALUE` subprograms of `DBMS_SQL` package. Now, let's create the second example. Listing 9.4 shows the use of API of `DBMS_SQL` package.

**Listing 9.4:** Using API of `DBMS_SQL` package

greater than 3000. In our case, two records are displayed.

You can also execute Dynamic SQL statements using native dynamic SQL. The `EXECUTE IMMEDIATE` command is used to execute dynamic SQL statement from inside the PL/SQL block. See Listing 9.5 which uses `EXECUTE IMMEDIATE` command.

**Listing 9.5:** Using `EXECUTE IMMEDIATE` command

Listing 9.5 counts the number of employees from `emp_detail` table who work in a particular department. The Using clause is used to associate department number to be passed to `countemployees` function with placeholder `:dept` inside the string `str` of `EXECUTE IMMEDIATE` statement.

Here is the output of Listing 9.5:

```
Function created
```

After executing Listing 9.5 on iSQL*Plus console, you will get the message `Function created` which indicates that `countemployees` function is successfully created inside the Oracle database.

See Listing 9.6 which executes `countemployees` function.

**Listing 9.6:** Executing `countemployees` function

Listing 9.6 declares `x` variable of `NUMBER` type to accept the return value after making call to `countemployees` function.

Here is output of Listing 9.6:

```
The number of employees who belong to department no. 40 are: 3
PL/SQL procedure successfully completed.
```

After executing Listing 9.6 on iSQL*Plus console, the console displays the total number of employees in `emp_detail` table who work in department number 40. In our case, the total number of such employees is 3.

## The DBMS_PIPE Package

In this section, we are creating four examples. First example creates a named database pipe and second example removes that pipe. Third and fourth examples send and read messages to and from a particular database pipe.

## Creating and Removing Pipes

You can create and remove both public and private database pipes for a database session using `CREATE_PIPE` and `REMOVE_PIPE` subprograms of `DBMS_PIPE` package.

See Listing 9.7 which creates public database pipe `PIPE_MSG`.

### Sending and Reading Messages

Database pipes are usually used to store as well as retrieve messages from a specific session. As we know, database pipes can be public or private. The public pipes are asynchronous. Any schema object can write the public pipe if it has the execute privilege and it knows the name of public pipe. Irrespective of the type of database pipe, you need two sessions to send messages and then receive same messages. The first session (sending session) uses `PACK_MEASSGE` procedure to build a message. This procedure adds the message to session's local message buffer. The information in this buffer is sent by calling the function `send_message` along with the pipe name to be used to send the message. The second session (receiving session) uses `receive_message` function. This function receives the message along with the pipe name from which the message is to be received.This session then calls the `unpack_message` function to access each item in the message. Once the receiving session reads the buffer information, it is emptied from the buffer and then it is not available to reader of the same pipe.

See Listing 9.9 which sends messages to database pipe `PLSQL$MESSAGE_INBOX`.

**Listing 9.9:** Sending messages to database pipe `PLSQL$MESSAGE_INBOX`

pipe. The FOR loop prints messages to be stored in a local buffer. `PACK_MESSAGE` subprogram puts each message in the local buffer. The `SEND_MESSAGE` function sends all the three messages

Listing 9.10 gets messages from PLSQL$MESSAGE_INBOX database. As this pipe is public and uses shared buffer, RESET_BUFFER procedure is invoked to reset the contents of shared buffer. The RECEIVE_MESSAGE procedure reads messages from PLSQL$MESSAGE_INBOX pipe and puts them in shared local buffer. The UNPACK_MESSAGE procedure retrieves messages from the local buffer. Then all the messages are displayed.

Here is the output of Listing 9.10.

After executing Listing 9.10 on iSQL*Plus console, the console displays retrieved messages, such as Message [ 1] ,Message [ 2] ,Message [ 3] and the message Messages received from PLSQL$MESSAGE_INBOX.

## The UTL_FILE Package

The UTL_FILE package is used to read and write to and from server-side files. Here, we create an example which writes some lines of text to a specific text file. Listing 9.11 shows the use of UTL_FILE package.

**Listing 9.11:** Using UTL_FILE package

Listing 9.11 creates a directory OUT_DIR get_nth_line function, which accepts three parameters—loc_in, file_in, and line_num_in. The loc_in represents the location of file with the name specified by file_in paramter. The line_num_in represents number of the line to be read. The f variable of UTL_FILE.FILE_TYPE opens only if the arguments to FOPEN subprogram are valid. This f variable acts as a file handle. The FOPEN method assigns a file handle of file something.txt present in OUT_DIR directory to f variable. Third parameter w passed to FOPEN method opens the something.txt file to read and write in replace mode. This f variable is used in UTL_FILE.PUT_LINE method to write two lines, such as line one: Santosh Jena, line two: Mandeep Singh on something.txt file. Finally, we close the file handle.

When you execute Listing 9.11 on iSQL*Plus workspace, you can see following output.

```
line one: Santosh Jena
line two: Mandeep Singh
```

This listing shows something.txt file which has two lines of text, such as line one: Santosh Jena, line two: Mandeep Singh.

### The UTL_HTTP Package

The UTL_HTTP package is used to read web pages and information, such as response headers and cookies associated with them. In this section, we create two examples to explain the usage of UTL_HTTP package. First example reads and displays lines of text from a web page.

See Listing 9.12 which reads a web page using API of UTL_HTTP package.

**Listing 9.12:** Reading a web page using UTL_HTTP package

Listing 9.12 declares two req and resp variables of UTL_HTTP.REQ and UTL_HTTP.RESP types. The SET_PROXY procedure sets proxy server of web page. The BEGIN_REQUEST function initiates request to a web page and returns the request object. The GET_RESPONSE function is used to get response from the server. Inside the loop, the READ_LINE procedure retrieves a line of text and writes it into value character string. When the entire response is read, END_OF_BODY exception is generated. This suggests developer to end the response using END_RESPONSE procedure to free memory.

Here is the output of Listing 9.12.

Here is the output of Listing 9.13.



### The DBMS_ALERT Package

DBMS_ALERT package is used to give alerts in a session using triggers. In this section, we are creating an example which gives a different alert when an INSERT, UPDATE, or DELETE statement is executed on messages table. See Listing 9.14 in which signal_trig trigger raises a signal or alert.

**Listing 9.14:** Raising an alert using signal_trig trigger

To use the trigger you have to register your interest in an alert so that every time the alert fires, you receive a message for insert, update, or delete operation.

See Listing 9.15 which registers interest in the raised alert `EVENT_MSG_QUEUE` in a session.

**Listing 9.15:** Registering for raised alert in a session

Listing 9.15 subscribes you to the alert.

After you have registered your interest in an alert, you may or may not receive an alert. Therefore, you need to wait to receive a message. Listing 9.16 shows you how to wait on a single alert.

**Listing 9.16:** Waiting on a single alert

In Listing 9.16, we define a message, status variables of `VARCHAR2`, and `INTEGER` data types respectively. We use `DBMS_ALERT.WAITONE` procedure to create a polling loop for 30 seconds. The default value for timeout period is five seconds. Finally, we use an if-then-else statement to check if the status was due to a time-out. A time-out occurs when no alert is received. If the time-out does not occur before an alert is received, it will print the alert. You can see the list of sent messages in `alerts_table` table by executing `SELECT` query.

### Triggering an alert

To trigger an alert, you have to execute the code as shown in Listing 9.16 in one session, In another session, you need to run the following listing within thirty seconds. See Listing 9.17, which raises the trigger `signal_trig`.

In Listing 9.19, the `salary` procedure gets the salary of employee based on passed `empno` and writes it into `esal OUT` variable. The `fire_employee` procedure deletes employee with specified `empno` from `emp_detail` table.

The `hireemployee` procedure inserts a new record into `emp_detail` table.

The `nth_highest_salary` function returns nth highest salary, such as highest salary in an organization or 2nd highest salary in the same organization. This function is already explained in Chapter 7 of this book.

The `raise_salary` function raises the salary of employee with specified `empno` and returns that employee's new salary.

Here is the output of Listing 9.19.

```
Package body created.
```

After executing Listing 9.19 on iSQL*Plus console, the console displays message `Package body created` which indicates that body of `emp_manipulate` package is successfully created.

See Listing 9.20 which executes various procedures and functions in `emp_manipulate` package.

**Listing 9.20:** Using functionality of `emp_manipulate` package



Listing 9.20 invokes `salary`, `fire_employee`, `hireemployee` procedures and `nth_highest_salary`, `raise_salary` functions with respective values.

Here is the output of Listing 9.20.

```
Basic salary of the emplyoee having employee number 102 is17500
Highest salary is 17500
```

After executing Listing 9.20 on iSQL*Plus console, the console displays the employee with empno 102, highest salary, 3rd highest salary, new salary of empno 101.

## Summary

In this chapter, we learned about:

- A package and its advantages
- Syntax of creating package specification and its body.
- Use of commonly used built-in PL/SQL packages, such as DBMS_SQL, DBMS DBMS_PIPE, UTL_FILE, and UTL_HTTP from Oracle 10g.
- Creating a user defined package having various procedures and functions.

Oracle automatically executes (or fires) certain procedures whenever a database event (such as creating or updating a table) occurs. These procedures are known as triggers. Triggers can be created on various types of Oracle events for a database table, database schema, or the database itself. Triggers are used to maintain integrity constraints in database tables, auditing information in a data table, and keep information about various events, such as database, system events (logon, logoff). For example, suppose you want to insert a row in a database table as well as display a primary key (for example, the employee number) when a record is added. To do this, you can create a trigger, which will automatically fire when any row is inserted in the database table. Triggers also help handle autonomous transactions. Autonomous transactions are independent transactions initiated by other transactions.

In this chapter, you learn about triggers, their types, using triggers on views, maintaining triggers, and autonomous transactions. Let's begin by a detailed description of triggers.

## Describing Triggers

A trigger is a code, executed for certain database events, such as DDL, DML operations on a database table. Triggers can be written on the database table or view. Structurally, triggers are similar to procedures and functions. Triggers cannot be stored locally as a program or package; instead, they have to be stored as standalone objects. Triggers do not accept any argument like procedures and functions and are fired implicitly.

Triggers help in maintaining database atomicity. For example, if a DML statement fires a trigger or a sequence of triggers, and if an error is generated during their execution, the effects produced by the DML statement and the trigger code are rolled back. In addition, the state of the system is reverted to the state it was before the statement fired the trigger.

A trigger can be fired due to DML (`INSERT`, `UPDATE`, `DELETE`), DDL (`CREATE`, `ALTER`, `DROP`), and other database operations (such as logon, logoff).

Based on the events performed on the database, there can be several types of triggers. Let's look at some of the common types.

## Types of Triggers

Triggers are of several types and depend on the event performed on the database. Considering the event, Oracle automatically fires the trigger suitable to that event. In this section, we discuss the following triggers:

- DML Triggers
- DDL Triggers
- Database Event Triggers
- INSTEAD OF Triggers
- AFTER SUSPEND Triggers

### *DML Triggers*

DML triggers are those triggers that are fired by DML statements. A DML trigger can be fired before or after the execution of the DML statement. These triggers can be fired before or after every row or multiple rows that are being affected by the DML statement.

DML triggers can be fired for a single statement or for a combination of DML statements. The following is the syntax of common types of triggers:

The parameters used in the preceding syntax are:

- **CREATE or REPLACE TRIGGER:** Specifies the statement used to create a new trigger or replace an already created trigger.
- **trig_name:** Specifies the name of the trigger. You can give any name to trigger according to your requirement.
- **BEFORE or AFTER:** Specifies whether the trigger is to be fired before or after executing the statement.
- **INSERT or UPDATE or DELETE:** Specifies the DML statement for which the trigger is to be fired.
- **tab_name:** The table name on which a trigger is to be fired.
- **FOR EACH ROW:** Specified when you want triggers to be activated for each row processed by a DML statement. If you do not specify this clause, the trigger is activated only once for the DML statement.
- **WHEN (...):** Specified to avoid unnecessary execution of a trigger.
- **DECLARE:** This section is used to specify a local variable.
- **BEGIN:** Specifies the start of the execution section. Here, you specify SQL statements and other execution code.
- **EXCEPTION:** In this section, you can use exception handlers to catch exceptions that have occurred during program execution.
- **END trig_name:** Used to end a trigger.

Let's now view Listing 10.1 to find out how DML triggers work.

**Listing 10.1:** Creating a sample (`ins_val`) DML trigger on `INSERT` statement

In Listing 10.1, we created a DML trigger named `ins_val`, which fires when a record is inserted into the `DEPT` database table.

In the `BEGIN` section, we used `:NEW.DEPTNO` to store the department number for any new record inserted in the table. The stored value is displayed by using `DBMS_OUTPUT.PUT_LINE()`. `:NEW` and `:OLD` store new and old values respectively.

However, we have to follow certain rules when using `:NEW` and `:OLD`. These rules are:

- `:NEW` and `:OLD` can only be used for row-level triggers. That is, if you do not specify the `FOR EACH ROW` statement in a trigger, you cannot use `:NEW` and `:OLD`.
- `:OLD` should not be used when a trigger is fired on the `INSERT` statement because there is no old value to store. For example, in `Listing 10.1`, we used `:OLD`; In this case, the trigger is created normally but won't display any value when it is fired because it does not have old data to display corresponding to the column number specified with `:OLD`.
- `:NEW` should not be used when a trigger is fired for the `DELETE` statement.
- You can use both `:NEW` and `:OLD` in triggers when executing the `UPDATE` statement.

The output of Listing 10.1 will be:

```
Trigger created.
```

Now, to find out how a trigger works, let's insert a record in the `DEPT` database table. To do this, we have created Listing 10.2.

**Listing 10.2:** Inserting a record in `DEPT` table to check the working of `ins_val` trigger

The following is the output of Listing 10.2:

```
Record is entered for employee number 92
PL/SQL procedure successfully completed.
```

In the output, the statement written in the trigger is displayed when the `INSERT` statement is executed on the `DEPT` table. In this way, you can see that triggers are fired according to the DML operation performed on the corresponding database table.

In Listing 10.1, we have only used the `INSERT` statement. However, we can use all the DML statements in the same trigger. To do this, PL/SQL provides some functions to determine which DML operation is responsible for firing which trigger. These functions are:

- **INSERTING:** This function returns true when a trigger is fired by the `INSERT` statement.
- **UPDATING:** This function returns true when a trigger is fired by the `UPDATE` statement.
- **DELETING:** This function returns true when a trigger is fired by the `DELETE` statement.

In Listing 10.3, we create a trigger that will use all DML statements.

**Listing 10.3:** Creating trigger (`iud_val`) to use on all DML statements



In Listing 10.3, we created a trigger named `iud_val` in which we have used all three functions, that is `INSERTING`, `UPDATING` and `DELETING`.

Depending on the type of DML statement, a trigger will be fired and the function in the `BEGIN` section will be executed.

Let's now see what happens when we update the `DEPT` table. To do this, we have created Listing 10.4.

**Listing 10.4:** Modifying a record in `DEPT` table to check the working of `iud_val` trigger



On executing Listing 10.4, the `UPDATING` function in the trigger is executed and the following output is displayed:



With this, we come to the end of our discussion on DML triggers. Let's look at DDL triggers next.

## *DDL Triggers*

Like DML triggers, you can create DDL triggers that fire when DDL statements are executed. DDL triggers cannot be applied on individual tables; you must create DDL triggers either on the database or on the current schema.

The syntax to create DDL triggers is the same as DML triggers, the only difference being that DML events are replaced by DDL events (for example, in place of `INSERT` you write `CREATE` or any other DDL event). Moreover, in DDL triggers, you cannot use the `FOR EACH ROW` statement, and the table name is replaced by the database name or the current schema.

Listing 10.5 shows you how to create a DDL trigger.

**Listing 10.5:** Creating a sample (`cre_ate`) DDL trigger



In Listing 10.5, we created a DDL trigger named `cre_ate` on the current schema. This trigger will be fired when a current user executes the `CREATE` statement.

The output of the Listing 10.5 is:

```
Trigger Created
```

Now, we create a table to check whether the trigger is fired when we execute the `CREATE` statement. To do this, we created a table named `STUDENT` with two fields, `sname` and `age`. The statement to create the table is:

```
SQL> CREATE TABLE STUDENT (sname VARCHAR2(50),
age NUMBER(2));
```

Executing the statement fires the trigger created in Listing 10.5 and gives the following output:

```
Execution successful
Table Created
```

In the same way, you can create triggers for other DDL statements as well.

### *Database Event Triggers*

These triggers are fired by Oracle when any database-level or system-level event occurs, such as when the database is started or shut down. To create database event triggers, you must have `ADMINISTER DATABASE TRIGGER` privileges, which we will shortly explain in this chapter. Database event triggers are of the following types:

- **STARTUP:** This trigger is fired when the database is opened.
- **SHUTDOWN:** This trigger is fired just before the shutdown of a database instance. This trigger will not be fired if the database instance is shut down suddenly.
- **SERVERERROR:** This trigger is fired when any server related error occurs.
- **LOGON:** This trigger is fired when a user logs on to Oracle.
- **LOGOFF:** This trigger is fired when a user logs off from Oracle.

Now, Deepak is able to create database event triggers. Before creating an event trigger, we have to create a database table to store the action of the trigger when it is fired by Oracle. We know that DBMS_OUTPUT.PUT_LINE does not display actions generated by database event triggers. To view these actions, we create a database table named D_EVENT.

In this table, we create two fields: (USER_NUM of type NUMBER), to store the user number and (OUT_STR of type VARCHAR2), to store messages that we want to add in the table when user logs on.

Listing 10.6 shows you how to create a database event trigger.

**Listing 10.6:** Creating a sample trigger (u_connect) on LOGON event

In Listing 10.6, we created a trigger named U_Connect. In this trigger, we used the AFTER attribute with the LOGON event.

In the execution section, we used the INSERT statement to add the action performed by the U_Connect trigger in the D_EVENT database table for the user Deepak. We also used CURRENT_TIMESTSAMP to know the user's logon date and time.

On executing Listing 10.6, the following output is displayed:

```
Trigger created.
```

Now that you have created the trigger, you want to check if it is working properly. To do so, we log on as Deepak and execute the SELECT statement on the D_EVENT table to retrieve all the values from that table. The output of the SELECT statement is:

```
USER_NUM      OUT_STR
1             Successfully logged on at 19-FEB-08 02.20.59.484000 PM +05:30
```

The output of the SELECT statement shows that the trigger has been fired successfully.

In this way, you can also create database event triggers for all database events.

With this, we move on to the next topic where we discuss how to fire triggers on views.

## *INSTEAD OF Triggers*

Triggers used to modify views are known INSTEAD OF triggers. They can only work for DML events on views because views cannot be modified directly through DML statements. You cannot create INSTEAD OF triggers on DDL and database events. This is because there are some internal restrictions set by Oracle that do not allow the use of INSTEAD OF triggers on DDL and database events. These restrictions mark the triggers as invalid.

If a view is inherently updateable (a view that allows you to perform DML operations on an underlying table) and also has the INSTEAD OF trigger, Oracle fires the trigger instead of

executing the UPDATE statement on the view. The BEFORE and AFTER attributes are not used with INSTEAD OF triggers.

The syntax to create INSTEAD OF triggers is the same as DML triggers. Before defining the INSTEAD OF trigger on a view, be sure that view exists in the Oracle database. In Listing 10.7, we create a view named demo_ins to select some values from the DEPT database table.

**Listing 10.7:** Creating a view (demo_ins)

In Listing 10.7, we create a view named demo_ins to query the DEPT table, where DEPTNO is equal to 21.

Executing Listing 10.7 displays the following output:

```
View created.
```

Listing 10.8 shows you how to create a trigger on a view:

**Listing 10.8:** Creating a sample trigger (inst_of) on view

In Listing 10.8, we create the INSTEAD OF trigger named inst_of on the view created in Listing 10.7.

In the execution section, we used the INSERT statement to add a record in the DEPT database table.

In the exception section, we used the built-in DUP_VAL_ON_INDEX exception to check for duplicate values in the database table. If a user enters a duplicate value, the statement 'Record already exists, please enter another record' in the exception handler is executed.

On executing Listing 10.8, the following output is generated:

```
Trigger created.
```

Let's now insert a new record in the DEPT database table, as shown in Listing 10.9.

**Listing 10.9:** inserting a record in table to check the working of inst_of trigger

On executing Listing 10.9, the record will be added to the DEPT database table and the following message is generated:

```
PL/SQL procedure successfully completed.
```

To check if the record has been added in the DEPT database table, execute the SELECT statement on the table.

However, if you enter a record of an already existing DEPTNO, the exception specified in the trigger will be executed. For example, suppose you enter a record with DEPTNO=22, which already exists in the database table. In such a case, the following output will be displayed:

```
Record already exists, please enter another record
PL/SQL procedure successfully completed.
```

Let's take up AFTER SUSPEND triggers next.

## AFTER SUSPEND Triggers

AFTER SUSPEND triggers are fired when SQL statements don't have sufficient space to complete their action. As a result, these statements are suspended. For example, suppose you are importing a large file but the server you are importing the file to fails to allocate sufficient memory to the file. In such a case, the process is suspended.

Once the process is suspended, it will remain in that state for two hours (which is default timeout). However, if you want to resume the session sooner, you can use DBMS_RESUMABLE PACKAGE with the SET_TIMEOUT method or execute the ALTER SESSION ENABLE RESUMABLE TIMEOUT (time in seconds) statement.

AFTER SUSPEND triggers can be created for database schema or for the whole database. To create these triggers for the whole database, you must have ADMINISTER DATABASE TRIGGER privileges.

Listing 10.10 shows you how to create an AFTER SUSPEND trigger.

**Listing 10.10:** Creating a sample (resume_after) AFTER SUSPEND trigger

```
SQL> SET SERVEROUTPUT ON
CREATE OR REPLACE TRIGGER resume_after
AFTER SUSPEND ON SCHEMA
BEGIN
DBMS_RESUMABLE.SET_TIMEOUT (60);
EXCEPTION
WHEN STORAGE_ERROR THEN
DBMS_OUTPUT.PUT_LINE ('more space required completing the processes');
END resume_after;
/
```

In Listing 10.10, we created an AFTER SUSPEND trigger named resume_after on the database schema that fires when a session is held up due to any reason, such as insufficient memory.

We know that the default time for any process to be in suspension is two hours, which is a long period for anyone to stop his or her work.

```
SQL> ALTER TABLE DEPT DISABLE ALL TRIGGERS;
```

Executing this statement displays the following result:

```
Table altered.
```

Now, performing any operation on the DEPT table will not fire triggers since the triggers are disabled. To check whether the triggers have been disabled, we execute Listing 10.11.

**Listing 10.11**: Inserting record in DEPT to check the whether trigger on DEPT is disabled or not

In Listing 10.11, we insert a record in the database table DEPT. Executing the listing displays the following output:

```
PL/SQL procedure successfully completed.
```

This message confirms that all triggers on the database have been disabled. Now, to know what the result will be when we enable all the triggers created on the DEPT table, we enable all the triggers once again on the table and then insert a record in the table.

To enable the triggers on the DEPT table, execute the following statement:

```
SQL> ALTER TABLE DEPT ENABLE ALL TRIGGERS;
```

Executing this statement displays the following result:

```
Table altered.
```

Now, to check whether the triggers have been enabled on the table, see Listing 10.12.

**Listing 10.12:** Inserting record in DEPT to check the whether trigger on DEPT is enabled or not

On executing Listing 10.12, all the triggers created on the DEPT table for the INSERT event will be fired and you get the following output:

This output is shows that the triggers have been enabled and fired successfully.

### *Dropping Triggers*

Oracle also allows you to drop triggers. Suppose, you have created a trigger for a specific task and after completing that task you want to drop the trigger. In that case, use the following syntax to drop the trigger:

```
DROP TRIGGER trig_name;
```

In this syntax, `trig_name` is the name of the trigger you want to drop. Now, suppose a trigger named `ins_val` created in Listing 10.1 needs to be dropped. You can do this easily by the following statement:

```
SQL> DROP TRIGGER ins_val;
```

On executing this statement, the following output is displayed:

```
Trigger dropped.
```

### Renaming Triggers

You can also rename triggers in Oracle. Suppose you have created a trigger for a specific task but the name of the trigger is confusing and you want to change it. In that case, use the following syntax to rename the trigger:

```
ALTER TRIGGER old_trig_name RENAME TO new_trig_name;
```

Where `old_trig_name` is the name of the trigger you want to change and `new_trig_name` is the new name you want for the trigger.

Now, suppose you want to rename the trigger `iud_val` created in the Listing-10.3. You can do so easily by the following statement:

```
SQL> ALTER TRIGGER iud_val RENAME TO all_chk;
```

On executing this statement, you get the following output:

```
Trigger altered.
```

With this, we have completed discussion on managing triggers. Next, we discuss how triggers are used for handling the autonomous transactions.

## Handling Autonomous Transactions using Triggers

A transaction is a logical unit of work that contains one or more SQL statements while an autonomous transaction is an independent transaction started by another transaction. Transaction that start autonomous transactions are known as parent or main transactions. Once an autonomous transaction has started, it is fully independent, meaning that it shares no locks, resources, or commit dependencies with the main transaction. An autonomous transaction is executed without affecting the parent transaction. With an autonomous transaction you can suspend a parent transaction, perform some SQL operations, commit or rollback the operation, and then resume the parent transaction. For example, let's suppose you are inserting some records in a database table through parent and autonomous transactions. Now, suppose the autonomous transactions have been performed and committed correctly but some errors have been encountered in the parent transaction. This in no way affects the working of the autonomous transactions and the changes (that is the records inserted) made in them are saved since autonomous transactions are independent of the parent transaction.

Let's now create an autonomous transaction. The PL/SQL compiler directive `PRAGMA AUTONOMOUS_TRANSACTION` is used to define an autonomous transaction. Listing 10.13 shows you how to create an autonomous transaction.

**Listing 10.13:** Creating an autonomous procedure



In Listing 10.13, we created an autonomous procedure named chk to perform some

In the execution section, we executed the `UPDATE` statement to update the based on the employee number received through the formal parameter. Next, we `SELECT` statement to retrieve the updated name. Then, we assigned the retrieved formal parameter of the procedure and displayed the name of the employee. In the section, we used the exception handler to catch an exception if data is employee number received through formal parameter.

On executing Listing 10.13, you get the following output:

```
Procedure created.
```

In Listing 10.14, we create another procedure to perform some transactions. In this procedure, we use the autonomous transaction.

**Listing 10.14:** Creating a procedure to use chk (autonomous procedure)

In Listing 10.17, we inserted two records in the TEST table. We committed the record inserted by the first statement. Then, we inserted another record. However, this time we rolled back the record inserted in the table.

The output of Listing 10.17 is:

```
PL/SQL procedure successfully completed.
```

Now, to check whether the record, which was rolled back has been stored in the TEST or in TEST1 database table, we executed the SELECT statement for both the tables as follows:

```
SQL> SELECT * FROM TEST;
```

The output of this statement is:

Output of this statement is:

According to the output of the first statement, while only the committed record is inserted in the TEST database table and the rolled back record is not inserted in the table, the TEST1 database table keeps both the records.

With this, we complete our discussion on handling autonomous transactions using triggers. This also concludes the chapter. We hope that the chapter has helped you gain enough knowledge to use triggers. A brief summary the chapter follows.

## Summary

In this chapter, we have studied about:

- Triggers and their types.
- Maintaining triggers.
- How to handle autonomous transactions using triggers.

Exceptions are certain abnormal conditions that occur at runtime and can cause the abrupt termination of a program. In PL/SQL, the errors are treated as exceptions. A program should be written in the way that it can handle exceptions very efficiently and thus can overcome the problem of sudden program termination. For example, you have created a PL/SQL program or a subprogram to fetch some value from the database table corresponding to a primary key. Now, suppose that data is not available corresponding to that primary key, then Oracle will raise `NO_DATA_FOUND` exception (an in-built PL/SQL exception) or the exception defined by you. Now, the question arises that how to handle an exception. Answer for this question is Exception handler. Now, you definitely want to know that what exception handler is. Let's continue discussing exception handling in detail to know more about the facts related with exceptions.

In this chapter, we discuss about the exceptions, their types, and the process to raise and handle them.

## Understanding PL/SQL Exceptions

In PL/SQL, an exception is raised every time an error occurs. The error can occur due to various problems, such as bad coding, hardware failure, memory faults. For example, if you are trying to divide any number or expression by zero, then `ZERO_DIVIDE` exception will raise.

Oracle has provided exception handler to handle errors. Whenever any exception is raised, normal execution of program stops and the control transfers to the exception-handler.

Exception handling helps to manage both expected and unexpected exceptions and thus helps in making your PL/SQL program robust. You can use either user-defined exception or predefined exception or both to handle the exceptions.

While writing a PL/SQL program, you are recommended to do the following to take the advantage of exception handling:

- Always add exception handler in a PL/SQL program.
- Always try to use the named exceptions, such as `NO_DATA_FOUND`, `TOO_MANY_ROWS` rather than using `WHEN OTHERS` in exception handler. Always test PL/SQL program with various combination of data to check the program for errors.

Here, we study about the two types of exceptions in detail—the user-defined exceptions and predefined or inbuilt exceptions.

### *In-Built Exceptions*

When a PL/SQL program violates certain Oracle rules, then exceptions are raised. For example, if you try to store some values in the database table corresponding to a primary key but values corresponding to that key already exist in the database table, then Oracle will generate the `DUP_VAL_ON_INDEX` exception.

In Oracle, for every error there is an error number and error message. You can use exception names to handle exception but exception numbers are not allowed to use. You can use the functions `SQLCODE` and `SQLERRM` to set the actual error numbers. You can also use `EXCEPTION_INIT` pragma to associate exception with Oracle error number. In Table 11.1, we have summarized some of the In-Built exceptions.

Table 11.1: In-Built Exceptions

| EXCEPTION NAME | ORACLE ERROR NUMBER | SQL CODE VALUE | Exception Overview |
|---|---|---|---|
| | | | other cursor inside cursor FOR...LOOP, exception is raised. |
| INVALID_CURSOR | ORA-01001 | -1001 | This exception occurs when you try to perform any cursor related operation, which is not allowed by Oracle. For example, you try to close a cursor which is not opened. |
| DUP_VAL_ON_INDEX | ORA-00001 | -1 | This exception is raised when you try to store any duplicate value in the database table on which you had set primary key constraint. For example, in the table EMP_DETAIL, you have applied primary key constraint on EMP_NO; want to enter a record with the EMP_NO=101 but a record with this number already exists in the table then Oracle will raise this exception. |
| INVALID_NUMBER | ORA-01722 | -1722 | This exception occurs in SQL statements when conversion of a string into a number fails. In procedural statements, Oracle raised VALUE_ERROR when conversion fails. |
| NOT_LOGGED_ON | ORA-01012 | -1012 | This exception is raised when you try to execute any PL/SQL program without being connected to Oracle. |
| PROGRAM_ERROR | ORA-06501 | -6501 | This exception occurs due to some internal problems in PL/SQL and the execution of program fails. |

**Table 11.1: In-Built Exceptions**



This is all about the In-Built exceptions. It is recommended to remember all the In-Built exceptions so that you can take care of their cause while writing any PL/SQL program. It will reduce the chance of the occurrence of In-Built exceptions. Do you know that Oracle allows you to define your own exceptions to handle the errors in your own way? The exceptions defined by you are known as user-defined exceptions.

## *User-Defined Exceptions*

The exceptions that you yourself can declare to handle unwanted or abnormal conditions are known as user-defined exceptions. These exceptions are application specific. User-defined exceptions must have to be declared and raised explicitly. The keyword `RAISE` is used to raise the exceptions explicitly. Let's see how to declare a user-defined excep...

### Declaring user-defined exceptions

User-defined exceptions are declared in the `DECLARE` section of a PL/SQL block. Syntax to declare an exception is as follows:

```
exec_name EXCEPTION;
```

In the above syntax, `exec_name` can be any valid name that suits to program requirement. This name must be followed by the keyword `EXCEPTION`. Exceptions are declared just like variables and also follow the same scope rule as variable.

### Scope rules for user-defined PL/SQL exceptions

The same exception cannot be declared twice in the same block. When an exception is declared in the block, it has its scope in that block and in all the blocks, which are enclosed by that block. See the Listing 11.1 to understand exception scope.

### The RAISE Statement

This statement is used to raise the user-defined or In-Built exception. A PL/SQL program or subprogram should raise an exception when it is impossible to complete the processing because of any abnormal condition. To raise an exception, `RAISE` statement can be used any where in your PL/SQL program within the scope of that exception.

Let's see Listing 11.2 to know how to use `RAISE` statement.

**Listing 11.2:** Creating procedure to use `RAISE` statement

In Listing 11.2, we have created a procedure to `add_record` in the database table named `EMP_DETAIL` in which we have used user-defined exception.

To use, user defined exception, first we have declared an exception named `sal_les` in between the `IS` and `BEGIN` keyword.

Then, in the execution part, we have used `IF...ELSE` control statement. We have set control on the employee basic salary `(ebasic)` that is if the `ebasic` entered by the user is less than `8000`, then the exception `sal_les` is raised, and the control transfers to the exception handling part of the procedure.

In case, the salary entered is more than 8000, then the else part of the control statement executes.

In the else part, we have used `INSERT` statement to add the record in the database table.

On execution of Listing 11.2, you will receive the following output:

```
Procedure Created.
```

Let's see Listing 11.3 in which we have called the procedure `add_record`.

**Listing 11.3:** Calling procedure `add_record`

On executing Listing 11.3, the output can come in the following two ways:

- If the employee basic salary entered is more than 8000, then the output is as follows:

```
PL/SQL procedure successfully completed.
```

- If the salary entered is less than 8000, then the exception will raise and the output is as follows:

```
Employee basic salary can not be less then 8000
```

This is all about the use of `RAISE` statement. Besides allowing raising user-defined exceptions, Oracle also provides the procedure to generate the user-defined errors. Let's see how to generate user-defined errors.

## The RAISE_APPLICATION_ERROR Procedure

This procedure is used to generate the user-defined and application specific errors. This procedure is defined in the `DBMS_STANDARD` package. In this procedure, we have to pass three parameters to generate the user-defined errors. Let's see the syntax to know about those parameters.

```
raise_application_error(err_code, err_msg,{TRUE/FALSE});
```

In this syntax:

- `err_code`: It is the error code, which is a negative integer and must be in the range of -20000 to -20999.
- `err_msg`: It is the error message and is of string type. Maximum size for an error message can be 2048 bytes.
- `TRUE/FALSE`: It is optional. If this parameter is set to `TRUE`, then the error is added in the stack of previous errors and if it is set to false, then it replaces all previous errors. Listing 11.4 shows the working of `RAISE_APPLICATION_ERROR` Procedure.

**Listing 11.4:** Creating procedure to use `RAISE_APPLICATION_ERROR`

In Listing 11.4, we have created a procedure named `up_record` in which we have passed parameter, `ebasic` (Employee basic salary).

In the execution section, we have used IF...ELSE control statement to check `ebasic`. If the `ebasic` passed by the calling program is less than 11000, then the

raise_application_error procedure will execute and the user-defined error with error code will be displayed. But, if the ebasic is not less than 11000, then the else part will execute.

Let's see Listing 11.5 in which we have called the procedure up_record.

**Listing 11.5:** Calling procedure up_record

In Listing 11.5, we have executed SELECT statement to fetch the employee basic salary (EMP_BASIC) from the table EMP_DETAIL corresponding to the EMP_NO=100.

EMP_BASIC fetched from the EMP_DETAIL has been stored in the variable e_basic.

Then, we have called the procedure up_record and passed e_basic as a parameter in it.

On executing Listing 11.5, the output retrieved is as follows:

When the raise_application_error is executed, the subprogram ends and the error code and error message returns to the calling program. The code and message in the calling program can be obtained using the functions SQLCODE and SQLERRM. We discuss these functions (SQLCODE and SQLERRM) in the next section.

With this, we have completed discussion on Raising Exceptions in PL/SQL. Now, you know what a PL/SQL exception is and also how to raise exceptions. Now is the time to learn handling PL/SQL exceptions.

## Handling PL/SQL Exceptions

In PL/SQL program or subprogram, when an exception is raised, the execution of that program or subprogram stops and the control transfers to the exception handling part, where the exception handler then handles the exception. See Listing 11.6 to know the format to handle exception.

**Listing 11.6:** PL/SQL program structure to handle exception

In the exception handling part, you can specify one or more exception handlers depending on your requirement.

An exception handler consists of the `WHEN` keyword, which is then followed by exception (user-defined or In-Built). After that `THEN` keyword is used, which is then followed by the statement that you want to display when an exception is handled.

Whenever any exception is raised, the control is transferred to the required exception handler and the handler gets executed. After executing the handler, the current block terminates and the control returns to the executing block.

If the exception part does not have any exception then the exception handler `OTHERS` is executed. It is recommended to use `OTHERS` exception handler in the last of all exception handlers, so that handling all possible exceptions is guaranteed.

You can also execute the same statement for two or more exceptions in the same handler by separating them by using `or` clause, as shown here:

**Note**

Let's now discuss the various methods of handling exceptions. PL/SQL provides the following methods to handle exceptions:

- Handling Exceptions Raised in Declarations
- Handling Exceptions Raised in Handlers
- Using SQLCODE and SQLERRM
- Catching Unhandled Exception

## Handling Exceptions Raised in Declarations

Exceptions can also be raised in the declaration part of the PL/SQL block. Exception in the declaration part occurs when you try to assign some abnormal values in the variable declared. For example, you have declared a variable `X` of type `NUMBER(2)` that is `X` variable cannot accept any numeric value whose length is more than 2.

If you try to assign any value whose length is more than 2, then the Oracle will raise the exception. When exceptions are raised in declaration block then they are not handled in the current block, and the control transfers to the outer block where those exceptions are handled. Let's see Listing 11.7 to know how to handle exceptions that occur in declaration part of PL/SQL block.

In the output, you can see that exception has been handled by the outer block. Besides that, we also write the same exception handler inside the inner block. Output is also trying to convey that the record we are trying to add in the database table `EMP_Detail` already exists in the table.

With this, we complete discussion on handling exceptions raised within exception handler. Now, we continue discussion by depicting the functions used to retrieve the error code and error message.

## *Using SQLCODE and SQLERRM*

Oracle provides `SQLCODE` and `SQLERRM` functions to know the code and message associated with the error respectively. `SQLCODE` returns error code as a negative number but for the Exception `NO_DATA_FOUND`, it returns number +100.

For user-defined exceptions, it returns +1. The function `SQLERRM` returns the error message that can be 512 characters long. This message includes code, message, table name, and column name.

If no exception is raised, then `SQLCODE` returns zero and `SQLERRM` returns `ORA-0000-` normal, successful, completion. In case of user-defined exceptions, `SQLERRM` returns `'User-defined Exception'`.

`SQLCODE` and `SQLERRM` functions are used in exception handler. You can also pass an error number to `SQLERRM`; in that case, `SQLERRM` returns the messages associated with that error number. But, before passing a number into `SQLERRM`, ensure that the number should be negative; otherwise, it will return the message user-defined exception. Listing 11.9 shows how to use `SQLCODE` and `SQLERRM`.

*Note*

**Listing 11.9:** PL/SQL program to use `SQLCODE` and `SQLERRM`

In Listing 11.9, first we have declared three variables named code, `num` of type `NUMBER`, and `msg` of type `VARCHAR2` respectively.

We have limited the length of num variable to 5 that is we cannot assign it any number whose length is more than 5.

In the execution part, we have assigned some value to num and the length of that value has exceeded the length declared for num.

Then, in the exception handler, we have used VALUE_ERROR and have initialized the variable code with SQLCODE and msg with SQLERRM associated with VALUE_ERROR.

After that, we have displayed those variables. The output of Listing 11.9 is as follows:

The function SQLCODE and SQLERRM are used specially in exception handler for OTHERS, because they help in finding the error code and error messages associated with unspecified errors.

Here, we have completed discussion on using SQLCODE and SQLERRM. Now, we continue depicting how to catch unhandled exceptions.

### Catching Unhandled Exceptions

In a PL/SQL program or subprogram, if you have not used exception handler and exceptions are raised then Oracle returns unhandled exceptions. If Oracle returns unhandled exceptions, then the actual outcome depends upon the host environment. In case, PL/SQL programs or subprograms fail due to unhandled exceptions, then the changes made by them are not rolled back. If an unhandled exception occurs in PL/SQL block in the precompiler environment, then all the changes made by the PL/SQL block are rolled back.

PL/SQL does not assign values to OUT parameter, if subprograms fail due to unhandled exceptions.

To catch unhandled exceptions, use OTHERS handler.

Here, we discussed handling PL/SQL exceptions. Now, you should be able to use exception handling in PL/SQL programs and subprograms to make them robust. Before closing this chapter, let's have a look on all that has been discussed in this chapter.

## Summary

In this chapter, we have studied about:

- Overview of exceptions and their types.
- Process to raise exceptions.
- Handling exceptions occurring in different parts of a PL/SQL block, such as declaration, exception handler.

In all preceding chapters of this book, variables (objects) of any predefined PL/SQL type are known as embedded objects. An embedded object completely contains within another object and represented as a column of a table. For example, nested table is an embedded object contained in another table. PL/SQL supports also another type of objects known as row objects to take advantages of Object oriented Programming. The row objects represented using rows not columns and are referenced objects. These row objects allow you to represent real world in PL/SQL. For example, you need to define bank account type which has account customer name, address, identity variables of simple PL/SQL types. As each bank associated with deposit, withdraw and balance enquiry operations in real life. Creating type support in PL/SQL allows user to define own methods within them.

This chapter focuses only on row objects. The chapter makes you understand what are how to create an object type and its body, comparing two objects with each other, implement inheritance and overriding and how to perform various operations on object tables.

## Introducing Object Types

Object-oriented programming helps in creating reusable entities. These entities are objects. These Objects maps real life discrete units such as bank account, student and so PL/SQL facilitates object-oriented programming with concept of object types. Users specify both structure of data and ways of operating on this data inside objects. Each declared inside object type is called attribute. The ways are procedures and functions operate on these attributes. These ways are also called methods.

Object types help in breaking real world entity or unit into other logical entities.

Let us move on next section which creates new object types, initializes and manipulate them.

## Using Object Types

In this section, we are creating simple object types and some object types with different types of methods. We have created an object type `complex1` with procedure `print` and function `add1`. The procedure `print` displays real and imaginary parts of complex number and `add1` adds two complex numbers. We also used `MAP` and `ORDER` methods to compare complex object types.

In declaring and initializing object types section, we build an example which declares `customer_typ` object type and initializes it with the help of constructor method. We manipulate objects in various ways such as accessing attributes of object using dot operator, inserting rows in object table, updating and deleting rows from object table.

### *Creating Object Types*

You can use the object types both as variable in PL/SQL block and type of column in a database table. For using the object type, you need to create the object type. The general syntax of creating an object type without sub programs is as follows:

```
    var3 type3;
    )
```

In this syntax, `name` is name of object type. The `name` object type consists of three variables or attributes of type1, type2, type3 respectively. The type1, type2, and type3 can be of `VARCHAR2`, `NUMBER` or other already defined object type. In last case, object type becomes nested object type.

See listing 12.1 which creates object type `address`.

**Listing 12.1:** Creating object type `address`

When you will execute listing on iSQL*Plus, message `Type created` will be displayed.

```
Type created
```

See listing 12.2 which uses `address` object type as one of attribute of `customer` type.

**Listing 12.2:** Creating nested object type `customer`

You will get same message `type created` when you will execute listing 12.2. This listing will execute successfully if address object type already exits in user's database such as scott.

Defining data types does not involve any storage allocation. These object types are used in the similar way as built-in types in SQL statement.

## Declaring and Initializing Object Types

An object type can describe the discrete units that exist in real life. These discrete units can be an employee, organization, bank account and so on. An object type can be data type of an attribute, variable, bind variable, field of record, formal parameter, or return value of function.

See listing 12.3 which creates `customer_typ` object type.

**Listing 12.3:** Creating `customer_typ` object type

In listing 12.3, we create customer_typ object type with custid, custname attributes and custadr attribute of address object type. The customer_typ object type has one procedure dis_cust_address with SELF built-in parameter. You can declare this procedure without parameter SELF as this parameter is implicitly passed upon invocation. The NOCOPY mode makes temporary storage of nested object types efficient.

No

See listing 12.4 which creates body of customer_typ object type.

**Listing 12.4:** Creating body of customer_typ object type

In listing 12.4, we define dis_cust_address procedure which accesses the attributes plot, street, city, state and pincode of nested object type variable custadr.

See listing 12.5 which declares and initializes variable of customer_typ object type.

**Listing 12.5:** Declaring and initializing variable of customer_typ object type

In listing 12.5, we declare a customer variable of customer_typ object type and initialize it using constructor method. Please refer to Using methods in Object Types section for use of constructor method. The procedure dis_cust_address is invoked using following syntax:

```
customer.dis_cust_address();
```

Note that we invoked the dis_cust_address procedure without passing SELF parameter as it is implicitly passed.

After executing listings 12.3, 12.4 and 12.5 one by one and in the same order as given in the chapter, you finally get the following output:

The output contains Id and name, address of customer.

## Using methods in Object Types

Methods are sub programs which return a value. Inside object type's definition you can define or implement various types of methods. These methods are given as follows:

- **Simple Method:** Suppose `X` is PL/SQL variable having type `complex1` has a method `add1`. It is declared in `complex1` type as follows:

```
MEMBER FUNCTION add1(x complex1) RETURN NUMBER;
```

You can call its `add1` function using following statement:

```
n :=x.add1();
```

In above line of code, type of `n` is same as that of `add1` function.

- **Constructor Method:** The constructor method is a function. It returns new object as its value. Every object type has a system defined constructor method. The name of constructor method is same as object type. Following expression is a call to constructor method of `address` object type:

```
address('71','MGRoad','Nagpur','440010');
```

- **Comparison Method:** Comparison methods are used to compare objects. They are of two types: `Map` method and `Order` method.
- **Map Method:** `Map` method is used to maintain order between instances of object types. It implicitly accepts built-in parameter `SELF` and returns any of types such as `DATE`, `NUMBER`, and `VARCHAR2`. The built-in parameter `SELF` is an instance of object type. If two objects `x` and `y` are having type `complex1`, then order decides on basis of value of each complex number. An example of `MAP` function inside object type is:

```
MAP MEMBER FUNCTION val RETURN NUMBER;
```

PL/SQL implicitly uses `MAP` method when either Boolean expressions or `DISTINCT`, `GROUP BY` and `ORDER BY` clauses occurred in PL/SQL blocks. An object type may contain only one `MAP` method.

- **Order Method:** The `ORDER` method is a function which returns a numeric value. It takes built-in parameter `SELF` and an object of same type. For example, `x` and `y` are two complex1 objects, any comparison between them such as `x>y` invokes `ORDER` method. This method returns a –ve, 0 or +ve number which indicate `SELF` is less than, equal to or greater than other object. The object type can have one of two comparison methods.

### Adding two instances of an object type using methods

You are now aware of various different methods which can be used in object type. This example creates two instances of `complex1` object type and then adds them using methods.

See listing 12.6 which creates `complex1` object type.

**Listing 12.6:** Creating `complex1` object type

In listing 12.6, we created `complex1` object type which has two `r1` and `im` attributes. The real part and `im` is imaginary part of complex number. The `complex1` object type has member methods `print`, `add1`, and `val`.

See listing 12.7 which represents the body of `complex1` object type.

**Listing 12.7:** Creating body of `complex1` object type

In listing 12.7, we define function `add1`, procedure `print`, and `MAP` function `val`. The `add1` function adds one complex object passed as an argument with another complex object on which `add1` function is to be invoked. The `print` procedure displays the real and imaginary complex object on which this procedure is to be invoked. The `MAP` function `val` value of complex object using formula `sqrt(r1*r1+im*im)`.

See listing 12.8 which creates two `complex1` objects and adds them.

**Listing 12.8:** Adding two `complex1` objects

In listing 12.8, we declare three variables x, y, and z of complex1 object type. In execution section, two complex objects are created using constructor method and assigned to variables x and y respectively. Then, we invoke print procedure on variables x and y to display real and imaginary parts of created complex objects. The add1 function accepts y variable, invoked on variable x adds complex objects represented by x and y. Finally, we display the real and imaginary parts of complex object represented by z.

After executing listings 12.6, 12.7, and 12.8, you will get following output:

The output shows real and imaginary parts of created complex objects and also shows these parts after addition.

### two instances of an object type

Let's now use MAP and ORDER methods to compare two instances of complex1 object type. This section contains two examples. First example compares two instances of complex1 object type using MAP method. Second example compares two instances of complex2 object types using ORDER method.

See listing 12.9 which compares two objects of complex1 object type using MAP method.

**Listing 12.9:** Comparing two objects using MAP method

In listing 12.9, when you use x>y in IF condition, it converts this condition into x.val> y.val. The MAP function on both variables x and y is invoked and returned values are compared with each other. Depending upon result of IF condition, print procedure on one of variables is invoked.

On executing listing 12.9 on iSQL*Plus, you will get following output:

The output shows first real and imaginary parts of two complex objects and then again these parts of greater complex object.

You can also compare two instances of an object type using `ORDER` method.

See listing 12.10 which shows the usage of `ORDER` method to compare two objects.

**Listing 12.10:** Creating `complex2` object type

In listing 12.10, we created `complex2` object type similar to `complex1` object type uses `ORDER` method for comparison.

See listing 12.11 which creates body of `complex2` object type.

**Listing 12.11:** Creating body of `complex2` object type

Observe the difference between use of `ORDER` and `MAP` methods. The `MAP` method used only evaluates the value of complex object but `ORDER` method compares passed complex with complex object on which `ORDER` method is invoked inside its body.

See listing 12.12 which compares two objects using `ORDER` method.

**Listing 12.12:** Comparing two objects using `ORDER` method

Listing 12.12 is totally similar to listing 12.9 but when PL/SQL engine encounters expression `x>y` in `IF` condition, the `ORDER` method is invoked.

After executing listings 12.10, 12.11, and 12.12, you will get the following output:

The output shows first real and imaginary parts of two complex objects and then again displays these parts of greater complex object.

## *Manipulating Object Types*

In this section, we manipulate attributes and methods of objects in PL/SQL. We perform operations such as accessing attributes and methods of object types, creating object table and manipulating its rows.

### Accessing Attributes of an object using dot operator

You can access and modify values of attributes of object using dot operator. To access the attributes of nested objects, you need to use more than one dot operator according to hierarchy.

See listing 12.13 which accesses attributes of `customer` object.

**Listing 12.13:** Accessing attributes of `customer` object

12.15, VALUE function accepts correlation variable which can be row variable or table with particular row in an object table. The SELECT INTO retrieves the return of this function into variable of same object type. The customer variable can be further to invoke its methods and attributes.

listings 12.14 and 12.15 in this sequence, you will get following output:

shows the address of customer with ID equals to 2.

### deleting rows from an object table

you can use UPDATE and DELETE statements to update and delete rows from an table. See listing 12.16 which updates and deletes rows of customer_tab object table.

**12.16:** Updating and deleting rows of customer_tab object table

12.16, you need to use table alias for specifying changed values of attributes in WHERE clauses.

### through REF Modifiers

use REF and DEREF modifiers to get pointer to particular object in an object table and pointer into an object type variable. See listing 12.17 which accesses objects modifiers.

**7:** Accessing objects using REF modifiers

In listing 12.17, we use `REF` modifier to get pointer to object type record of equals to1. We comment lines where attribute name is being accessed from `REF` variable in this listing. This is because you cannot access value of attribute from Therefore, we dereference `customer_ref` variable into `customer` variable `INTO` statement. As partial `SELECT INTO` statement does not execute, we, thus, table `DUAL` inside it.

On executing listing 12.17 on iSQL*Plus, you will get following output:

The output shows name of customer with id equals to 2.

Let us create sub types which use the inheritance feature of object oriented programming.

## Inheritance in PL/SQL

You can create an object from existing base objects. By default, base objects are declared as `FINAL`. To allow inheritance, the base objects should be declared as `NOT FINAL`.

We are creating an example which implements inheritance in PL/SQL. In this example, `complex1` is a base object while `complex3` is an inherited object. In `complex3` object, additional method `sub1` is added. Therefore, `complex3` inherit attributes `r1, im` and three methods of base object `complex1`.

See listing 12.18 which creates `complex1` object type.

**Listing 12.18:** Creating inheritable `complex1` object type

Listing 12.18 is similar to listing 12.6 as described earlier, except here we append `NOT` keyword to listing 12.18. This keyword causes this object type to be inherited from other types.

See listing 12.19 which defines the body of `complex1` object type.

**Listing 12.19:** Defining body of `complex1` object type

Listing 12.19 is exactly same as listing 12.7. We are again showing its code here as it is now used to implement inheritance.

See listing 12.20 which creates inherited object type `complex3`.

**Listing 12.20:** Creating sub type `complex3`

In listing 12.20, we created object type `complex3` which inherits attributes and methods of `complex1` using `UNDER` keyword. The `complex3` object type has its own `sub1` member function.

See listing 12.21 which creates body of `complex3` object type.

**Listing 12.21:** Creating body of `complex3` object type

In listing 12.21, sub type `complex3` needs to define unique methods inside itself. The `sub1` function returns `complex3` object type after subtracting two complex objects.

See listing 12.22 which shows the use of `complex3` sub type:

**Listing 12.22:** Using `complex3` sub type

In listing 12.22, we accessed inherited `print` procedure on `complex3` in addition to own `sub1` function.

After executing listings 12.18, 12.19, 12.20, 12.21, and 12.22 in same sequence, you will the following output:

The output shows real and imaginary parts of created complex objects and also parts after subtraction.

Let us implement overriding after implementing inheritance between object types. This also explains dynamic method dispatch.

## Method Overriding

You can override a method having the same signature as method of super or base type. signature means that both methods have same name, same number and types of arguments, same return type. You can override a method in sub type using `OVERRIDING` keyword.

See listing 12.23 which creates object type `complex4`.

**Listing 12.23:** Creating object type `complex4`

In listing 12.23, `complex4` object type is exactly similar to `complex2` object type.

See listing 12.24 which defines body of `complex4` object type.

**Listing 12.24:** Defining body of `complex4` object type

The functionality of listing 12.24 is similar to listing 12.11 except the name of object type is
4 in listing 12.24.

12.25 which creates `complex5` object type.

**12.25:** Creating `complex5` object type

12.25 is similar to listing 12.20 except it uses `OVERRIDING` keyword before procedure
You cannot declare procedure with same name as that of base type inside sub type
using `OVERRIDING` keyword.

12.26 which defines body of `complex5` sub type.

**Listing 12.26:** Defining body of `complex5` sub type

listing 12.26, we override the procedure `print` and it only uses different sentence in output.
See listing 12.27 which uses dynamic method dispatch technique.

Using dynamic method dispatch technique

```
--z:=x;
END;
/
```

In listing 12.27, we first invoke procedure `print` on variables `x` and `y` of `complex4` object types and then overridden procedure `print` invoked on variable `z` of `complex5` object type. Finally, we assign variable `z` of sub type `complex5` to variable `x` of super type `complex4`. This type casting is also called dynamic method dispatch. Using this type casting, you can assign sub types to super types and call methods of sub objects on base object.

After executing listings 12.23, 12.24, 12.25, 12.26, and 12.27, you will get the following output:

```
real part 3 imag part 4
real part 5 imag part 6
The real part is 8 the imaginary part is 9
The real part is 8 the imaginary part is 9
PL/SQL procedure successfully completed.
```

The output shows real and imaginary parts of base and sub objects using base procedure print and overridden procedure print.

If you neither want to put any code inside methods nor define methods in super type, then declare super type as not instantiable. Consider an example which involves three object types shape, circle, and rectangle. The shape is super type and has one area function which calculates the area of particular shape. The circle and rectangle are sub types of shape super type. The bodies of circle and rectangle sub types have one area function which calculates the area of circle and rectangle. Therefore, it does not make any sense to define area function inside shape super type as this object type has no dimensions. You can do it using `NOT INSTANTIABLE` key word. When the super type is not instantiable, you cannot create object of that super type. You have to create sub type to use the functionality.

Let us learn how to define SQL object types equivalent to PL/SQL collection types, manipulate them and finally store them in the form of object tables.

## PL/SQL Collections and Object Types

You can define SQL object types equivalent to PL/SQL collections and then use these collections with SQL object types.

### *Defining SQL Types equivalent to PL/SQL Collection Types*

With the help of `CREATE TYPE` statement, you can store nested tables and varrays in database tables. The created types may be further used as attributes of SQL object types. You may declare equivalent types in PL/SQL using SQL type inside variable declaration.

See the following statements to create a nested table.

```
CREATE TYPE listofcourses AS TABLE OF VARCHAR2(10);
```

In the preceding syntax, we created a collection type (nested table) `listofcourses` which acts as table of course names.

See listing 12.28 which uses VARRAY constructor within a SQL statement.

**Listing 12.28:** Using VARRAY constructor within a SQL statement.

In listing 12.28, we use listofprojects constructor to store one record company_branch_projects table.

See listing 12.29 which uses nested table constructor inside SQL statement.

**Listing 12.29:** Using nested table constructor inside SQL statement

In listing 12.29, we insert one record into student_tbl table using listofcourses table constructor.

### *Using PL/SQL Collections with SQL Object Types*



With the help of collections, you can work with complex data types in PL/SQL. After and modifying particular elements in instances of these types, you can use SQL statements store them in database table.

Now, see the following syntax and code snippets to inserts some records into table.

In the preceding syntax, we created a collection type dnames_tbl which acts as department names.

Output of the preceding syntax is as follows:

Let's create table named departments which uses dnames_tbl collection type.

```
CREATE TABLE departments(loc VARCHAR2(20),deptnames dnames_tbl)
NESTED TABLE deptnames STORE AS int_dnames;
```

In the preceding code snippet we have created nested table as an attribute of departments object type.

Output of the preceding syntax is as follows:

```
Table Created
```

Let's see the Listing 12.30 to insert some records into departments table.

**Listing 12.30:** Inserting records into departments table



In the preceding we have inserted three records into departments table using `dnames_tbl` constructor. Each record represents types of departments in a particular location.

Output of the Listing 12.30 is as follows:

```
PL/SQL procedure successfully completed.
```

Now execute the following query to check whether the records have been inserted into departments table or not:

```
SELECT * FROM departments;
```

The `SELECT` query generates following output:



The output shows rows currently inserted in `departments` table. Note that each value in nested table column is a nested table constructor.

See listing 12.31 which uses `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements with `dnames_tbl` nested table.

**Listing 12.31:** Using `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements with `dnames_tbl` nested table

```
INSERT INTO TABLE(SELECT deptnames FROM departments WHERE loc='Banglore')
VALUES('Sales');
DELETE FROM TABLE(SELECT deptnames FROM departments WHERE loc='Hyderabad')
WHERE column_value='Payroll';
UPDATE TABLE(SELECT deptnames FROM departments WHERE loc='New Delhi')
SET column_value='Finance' WHERE column_value='Production';
COMMIT;
END;
/
```

In listing 12.31, we declare variable `vdnames, depts` and `chng_dnames`.The `vdnames` variable can store a set of department names. The `depts` variable declares a record that can store row of `departments` table. The `chng_names` variable used to assign the new set of department names to be stored in `departments` table corresponding to a location Banglore.

First `SELECT INTO` query retrieves department names into `vdnames` variable. The `FOR LOOP` used to display each of department name using syntax `vdnames(i)`. Second `SELECT INTO` query retrieves entire row in `departments` table corresponding to location New Delhi. Second `FOR LOOP` used to display department names in New Delhi using syntax `depts.deptnames(i)`.

We used `UPDATE` statement to change set of department names for Hyderabad using collection `chng_dnames`. We also can extend the collection `deptnames` corresponding to location Banglore using `EXTEND` procedure and add one more name in it using `COUNT` function. Then, this collection is used to update the department names in `departments` table.

You can also use `INSERT, UPDATE,` and `DELETE` statements with nested table represented by set of `deptnames`. For this, you need to use `TABLE` operator to have effect of `INSERT, UPDATE,` and `DELETE` statements on nested table produced by sub query. The `column_value` refers to particular name in `deptnames` collection. You cannot apply DML SQL statements to nested table directly.

On executing listing 12.31 on iSQL*Plus, you will get following output:

```
Department names:Technical Writing
Department names:Software Development
New Delhideptnames=Human Resources
New Delhideptnames=Sales
New Delhideptnames=Finance
PL/SQL procedure successfully completed.
```

The output shows types of departments in banglore and New Delhi after using many manipulation operations.

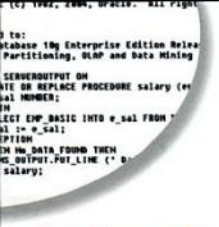
Now we are manipulating SQL `VARRAY` objects with PL/SQL statements. The departments table now contains a `VARRAY` column. See listing 12.32 to insert some records into departments table.

***Note***

> Now we have dropped the table named departments because we are creating this table once again in the coming section.

Let's see the following syntax and code snippets to insert records into departments table having `VARRAY` column:

```
CREATE TYPE dnames_varray IS VARRAY(7) OF VARCHAR2(30);
```




# Oracle PL/SQL Programming

Oracle PL/SQL Programming in Simple Steps book effectively explains the concepts of PL/SQL programming. It gives short-yet-complete description of the PL/SQL programming concepts and explains them step by step. This book provides core information that every PL/SQL developer should know to write PL/SQL programs, interact with Oracle databases, perform complex calculations, and handle exceptions. Loaded with lots of examples and illustrations to explain concepts, this book would help you learn PL/SQL programming with minimal effort.

The book covers:

- Features of PL/SQL language
- PL/SQL architecture
- PL/SQL language elements, such as block structure, datatypes, declarations, and operators
- PL/SQL expressions and datatype conversions
- Built-in PL/SQL functions
- PL/SQL control structures
- SQL operations in PL/SQL
- Transaction management in PL/SQL
- PL/SQL collections, records and objects
- Cursors in PL/SQL
- Procedures and functions in PL/SQL
- Packages in PL/SQL
- Triggers in PL/SQL
- Exceptions and exception handlers in PL/SQL
- Writing object-oriented PL/SQL applications

IN SIMPLE STEPS



Published by:
dreamtech PRESS
Dreamtech Press
19-A, Ansari Road, Daryaganj,
New Delhi-110002
Tel.: 91-11-23284212, 23243075
Fax: 91-11-23243078
Email: feedback@dreamtechpress.com
http://www.dreamtechpress.com

ISBN 10: 81-7722-855-2
ISBN 13: 978-81-7722-855-7


SPECIAL INDIAN PRICE

Rs. 169/-

International Price $9.99