SQL and PL/SQL Tutorial

S.RAFEEQ AHMED, M.Sc. & M.RIYAZ MOHAMMED M.C.A.,M.Phil.,

Table of Contents

[Introduction to SQL 5](#_Toc250091655)

[A Brief History of SQL 5](#_Toc250091656)

[Relational DataBase Management System (RDBMS). 5](#_Toc250091657)

[An Overview of SQL 6](#_Toc250091658)

[Introduction to the Query: The SELECT Statement 6](#_Toc250091659)

[General Rules of Syntax 7](#_Toc250091660)

[Terminating an SQL Statement 9](#_Toc250091661)

[Changing the Order of the Columns 9](#_Toc250091662)

[Selecting Individual Columns 10](#_Toc250091663)

[Selecting Different Tables 10](#_Toc250091664)

[Queries with Distinction 10](#_Toc250091665)

[CREATE TABLE 11](#_Toc250091666)

[ALTER TABLE 12](#_Toc250091667)

[Creating a View 15](#_Toc250091668)

[Introduction to PL/SQL 16](#_Toc250091669)

[Introduction 16](#_Toc250091670)

[Features of PL/SQL 16](#_Toc250091671)

[The PL/SQL Identifiers 17](#_Toc250091672)

[The PL/SQL Delimiters 17](#_Toc250091673)

[The PL/SQL Comments 17](#_Toc250091674)

[Data Types in PL/SQL 18](#_Toc250091675)

[Character String Data Types 18](#_Toc250091676)

[Numeric Data Types 18](#_Toc250091677)

[Binary Data Types 18](#_Toc250091678)

[The DATE Data Type 18](#_Toc250091679)

[BOOLEAN 19](#_Toc250091680)

[ROWID 19](#_Toc250091681)

[Variable Assignment 19](#_Toc250091682)

[Constant Assignment 19](#_Toc250091683)

[Cursors 19](#_Toc250091684)

[Implicit Cursors 20](#_Toc250091685)

[The %TYPE Attribute 21](#_Toc250091686)

[The %ROWTYPE Attribute 21](#_Toc250091687)

[The %ROWCOUNT Attribute 22](#_Toc250091688)

[Explicit Cursors 22](#_Toc250091689)

[The PROCEDURE Section 22](#_Toc250091690)

[Output Statements 23](#_Toc250091691)

[Displaying Output to the User 23](#_Toc250091692)

[Control Statements 23](#_Toc250091693)

[Cursor Control Commands 23](#_Toc250091694)

[DECLARE 23](#_Toc250091695)

[OPEN 23](#_Toc250091696)

[FETCH 24](#_Toc250091697)

[CLOSE 24](#_Toc250091698)

[Transactional Control commands in PL/SQL 25](#_Toc250091699)

[COMMIT 25](#_Toc250091700)

[ROLLBACK 25](#_Toc250091701)

[SAVEPOINT 25](#_Toc250091702)

[Statements Control Commands 25](#_Toc250091703)

[Conditional Statements 25](#_Toc250091704)

[IF...THEN 25](#_Toc250091705)

[LOOPS 26](#_Toc250091706)

[The EXCEPTION Section 27](#_Toc250091707)

[Raising Exceptions 27](#_Toc250091708)

[Handling Exceptions 28](#_Toc250091709)

[Procedures 28](#_Toc250091710)

[Parameter types and passing technics 28](#_Toc250091711)

[IN only parameter: 29](#_Toc250091712)

[OUT only parameter 29](#_Toc250091713)

[IN OUT Parameter 29](#_Toc250091714)

[Methods for Passing Parameters 30](#_Toc250091715)

[POSITIONAL NOTATION 30](#_Toc250091716)

[NAMED NOTATION 30](#_Toc250091717)

[MIXED NOTATION 30](#_Toc250091718)

[Creating a Procedure 30](#_Toc250091719)

[Deleting a Standalone Procedure 31](#_Toc250091720)

[Functions 31](#_Toc250091721)

[Library Functions 31](#_Toc250091722)

[Stored Functions (User Defined) 33](#_Toc250091723)

[Creating a Function 33](#_Toc250091724)

[Calling a Function 34](#_Toc250091725)

[PL/SQL Recursive Functions 35](#_Toc250091726)

[Factorial 6 is 720 35](#_Toc250091727)

[Triggers 36](#_Toc250091728)

[Creating Triggers 36](#_Toc250091729)

[Triggering a Trigger 37](#_Toc250091730)

[Packages 38](#_Toc250091731)

[Reasons to Use Packages 38](#_Toc250091732)

[Package Specification 39](#_Toc250091733)

[Creating Package Specifications 39](#_Toc250091734)

[Package Body 40](#_Toc250091735)

[Package Instantiation and Initialization 42](#_Toc250091736)

[SERIALLY\_REUSABLE Packages 43](#_Toc250091737)

# Introduction to SQL

## A Brief History of SQL

The history of SQL begins in an IBM laboratory in San Jose, California, where SQL was developed in the late 1970s. The initials stand for Structured Query Language, and the language itself is often referred to as "sequel." It was originally developed for IBM's DB2 product (a relational database management system, or RDBMS, that can still be bought today for various platforms and environments). In fact, SQL makes an RDBMS possible. SQL is a nonprocedural language, in contrast to the procedural or third generation languages (3GLs) such as COBOL and C that had been created up to that time.

The *S,* for Structured, and the *L,* for Language, are straightforward enough, but the *Q* is a little misleading. *Q*,

of course, stands for "Query," which--if taken literally--would restrict you to asking the database questions. But SQL does much more than ask questions. With SQL you can also create tables, add data, delete data, splice data together, trigger actions based on changes to the database, and store your queries within your program or database.

The characteristic that differentiates a DBMS from an RDBMS is that the RDBMS provides a set-oriented database language. For most RDBMSs, this set-oriented database language is SQL. *Set oriented* means that SQL processes sets of data in groups.

### Relational DataBase Management System (RDBMS).

This method has several advantages and many disadvantages. In its favor is the fact that the physical structure of data on a disk becomes unimportant. The programmer simply stores pointers to the next location, so data can be accessed in this manner. Also, data can be added and deleted easily. However, different groups of information could not be easily joined to form new information. The format of the data on the disk could not be arbitrarily changed after the database was created. Doing so would require the creation of a new database structure.

The idea for an RDBMS uses the mathematical concepts of relational algebra to break down data into sets and related common subsets. Because information can naturally be grouped into distinct sets, We organize the database system around this concept. Under the relational model, data is separated into sets that resemble a table structure. This table structure consists of individual data elements called columns or fields. A single set of a group of fields is known as a record or row. For instance, to create a relational database consisting of employee data, you might start with a table called **EMPLOYEE** that contains the following pieces of information: **Name**, **Age**, and **Occupation**. These three pieces of data make up the fields in the **EMPLOYEE** table, shown in Table 1.1.

**Table 1.1. The EMPLOYEE table.**

|  |  |  |
| --- | --- | --- |
| **Name** | **Age** | **Occupation** |
| Williams | 25 | Electrical engineer |
| Davidson | 34 | Museum curator |
| Janis | 42 | Chef |
| Jackson | 19 | Student |
| DeMarco | 32 | Game programmer |
| Boudreaux | 25 | Model |

The six rows are the records in the **EMPLOYEE** table. To retrieve a specific record from this table, for example, Dave Davidson, a user would instruct the database management system to retrieve the records where the **NAME** field was equal to Dave Davidson. If the DBMS had been instructed to retrieve all the fields in the record, the employee's name, age, and occupation would be returned to the user. SQL is the language that tells the database to retrieve this data. A sample SQL statement that makes this query is

SELECT \* FROM EMPLOYEE

Remember that the exact syntax is not important at this point. We cover this topic in much greater detail beginning tomorrow. Because the various data items can be grouped according to obvious relationships (such

as the relationship of **Employee Name** to **Employee Age**), the relational database model gives the database designer a great deal of flexibility to describe the relationships between the data elements. Through the mathematical concepts of join and union, relational databases can quickly retrieve pieces of data from different sets (tables) and return them to the user or program as one "joined" collection of data. The join feature enables the designer to store sets of information in separate tables to reduce repetition.

## An Overview of SQL

SQL is the de facto standard language used to manipulate and retrieve data from these relational databases. SQL enables a programmer or database administrator to do the following:

* Modify a database's structure
* Change system security settings
* Add user permissions on databases or tables
* Query a database for information
* Update the contents of a database

The first step to learning SQL is to understand the different types of commands.

* **Data Manipulation Language (DML)** statements manage data within schema objects.
* **Data Definition Language (DDL)** statements create or alter structures (not data) in the database.
* **Data Control Language (DCL)** statements manage security in the database.
* **Transaction Control Language (TCL)** statements manage transactions in the database.

You can see that SQL can be broken down into different areas.

### Introduction to the Query: The SELECT Statement

"Introduction to SQL," you need to communicate with it. The ultimate communication would be to turn to your computer and say, in a clear, distinct voice, "Show me all the left-handed, brown-eyed bean counters who have worked for this company for at least 10 years." A few of you may already be doing so (talking to your computer, not listing bean counters). Everyone else needs a more conventional way of retrieving information from the database.

#### Objectives

* Write an SQL query
* Select and list all rows and columns from a table
* Select and list selected columns from a table
* Select and list columns from multiple tables

An SQL query is not necessarily a question to the database. It can be a command to do one of the following:

* Build or delete a table
* Insert, modify, or delete rows or fields
* Search several tables for specific information and return the results in a specific order
* Modify security information

A query can also be a simple question to the database. To use this powerful tool, you need to learn how to write an SQL query. Your SQL statements can be broken into sections called clauses. Not all statements contain all the clauses. It depends on what you’re trying to do.

#### Basic architecture of the SQL SELECT clauses.

**SELECT** Contains the columns and operators to display the data

**FROM** Contains one or more tables from which the data originates

**WHERE** Determines what data will be returned or restricted

**GROUP BY** Groups the data according to certain values

**ORDER BY** Orders the output of the data based on specified columns

### General Rules of Syntax

As you will find, syntax in SQL is quite flexible, although there are rules to follow as in any programming language. A simple query illustrates the basic syntax of an SQL select statement. Pay close attention to the case, spacing, and logical separation of the components of each query by SQL keywords.

**SELECT**

**NAME, STARTTERM, ENDTERM**

**FROM**

**PRESIDENTS**

**WHERE**

**NAME = 'LINCOLN';**

In this example everything is capitalized, but it doesn't have to be. The preceding query would work just as well if it were written like this:

**select**

**name, startterm, endterm**

**from**

**presidents**

**where**

**name = 'LINCOLN';**

Notice that **LINCOLN** appears in capital letters in both examples. Although actual SQL statements are not case sensitive, references to data in a database are. For instance, many companies store their data in uppercase. In the preceding example, assume that the column **name** stores its contents in uppercase. Therefore, a query searching for 'Lincoln' in the **name** column would not find any data to return. Check your implementation and/or company policies for any case requirements.

**NOTE:** Commands in SQL are not case sensitive.

Take another look at the sample query. Is there something magical in the spacing? Again the answer is no. The following code would work as well:

**select name, startterm, endterm from presidents where name ='LINCOLN';**

However, some regard for spacing and capitalization makes your statements much easier to read. It also makes your statements much easier to maintain when they become a part of your project. Another important feature of ; (semicolon)semicolon (;)the sample query is the semicolon at the end of the expression. This punctuation mark tells the command-line SQL program that your query is complete.If the magic isn't in the capitalization or the format, then just which elements are important? The answer is keywords, or the words in SQL that are reserved as a part of syntax. (Depending on the SQL statement, a keyword can be either a mandatory element of the statement or optional.) The keywords in the current example are

**SELECT**

**FROM**

**WHERE**

Check the table of contents to see some of the SQL keywords.

**The Building Blocks of Data Retrieval: SELECT and FROM**

As your experience with SQL grows, you will notice that you are typing the words **SELECT** and **FROM** more than any other words in the SQL vocabulary. They aren't as glamorous as **CREATE** or as ruthless as **DROP**, but they are indispensable to any conversation you hope to have with the computer concerning data retrieval. And isn't data retrieval the reason that you entered mountains of information into your very expensive database in the first place?

This discussion starts with **SELECT** because most of your statements will also start with

**SELECT**:

**SYNTAX:**

SELECT <COLUMN NAMES>

The commands, see also statementsbasic **SELECT** statement couldn't be simpler. However, **SELECT** does not work alone. If you typed just **SELECT** into your system, you might get the following response:

**INPUT:**

SQL> **SELECT;**

**OUTPUT:**

SELECT \* ERROR at line 1:

ORA-00936: missing expression

The asterisk under the offending line indicates where Oracle7 thinks the offense occurred. The error message tells you that something is missing. That something is the **FROM** clause:

**SYNTAX:**

FROM <TABLE>

Together, the statements **SELECT** and **FROM** begin to unlock the power behind your database.

**NOTE:** keywordsclausesAt this point you may be wondering what the difference is between a keyword, a statement, and a clause. SQL keywords refer to individual SQL elements, such as **SELECT** and **FROM**. A clause is a part of an SQL statement; for example, **SELECT** column1, column2, ... is a clause. SQL clauses combine to form a complete SQL statement. For example, you can combine a **SELECT** clause and a **FROM** clause to write an SQL statement.

**NOTE:** Each implementation of SQL has a unique way of indicating errors. Microsoft Query, for example, says it can't show the query, leaving you to find the problem. Personal Oracle7, the engine used in the preceding example, gives you an error number (so you can look up the detailed explanation in your manuals)

and a short explanation of the problem.

**Examples**

Before going any further, look at the sample database that is the basis for the following examples. This database illustrates the basic functions of **SELECT** and **FROM**. In the real world you would use the techniques described as "Manipulating Data," to build this database, but for the purpose of describing how to use **SELECT** and **FROM**, assume it already exists. This example uses the **CHECKS** table to retrieve information about checks that an individual has written.

The **CHECKS** table:

CHECK# PAYEE AMOUNT REMARKS

--------- -------------------- ------ ---------------------

1 Ma Bell 150 Have sons next time

2 Reading R.R. 245.34 Train to Chicago

3 Ma Bell 200.32 Cellular Phone

4 Local Utilities 98 Gas

5 Joes Stale $ Dent 150 Groceries

6 Cash 25 Wild Night Out

7 Joans Gas 25.1 Gas

#### Your First Query

**INPUT:**

SQL> **select \* from checks;**

**OUTPUT:**

Queries CHECK# PAYEE AMOUNT REMARKS

------ -------------------- ------- ---------------------

1 Ma Bell 150 Have sons next time

2 Reading R.R. 245.34 Train to Chicago

3 Ma Bell 200.32 Cellular Phone

4 Local Utilities 98 Gas

5 Joes Stale$ Dent 150 Groceries

6 Cash 25 Wild Night Out

7 Joans Gas 25.1 Gas

7 rows selected.

**ANALYSIS:**

This output looks just like the code in the example. Notice that columns 1 and 3 in the output statement are right-justified and that columns 2 and 4 are left-justified. This format follows the alignment convention in which numeric data types are right justified and character data types are left-justified. Data types are discussed as "Creating and Maintaining Tables." The asterisk (**\***) in **select \*** tells the database to return all the columns associated with the given table described in the **FROM** clause. The database determines the order in which to return the columns.

### Terminating an SQL Statement

In some implementations of SQL, the semicolon at the end of the statement tells the interpreter that you are finished writing the query. For example, Oracle's SQL\*PLUS won't execute the query until it finds a semicolon (or a slash). On the other hand, some implementations of SQL do not use the semicolon as a terminator. For example, Microsoft Query and Borland's ISQL don't require a terminator, because your query is typed in an edit box and executed when you push a button.

### Changing the Order of the Columns

The preceding example of an SQL statement used the **\*** to select all columns from a table, the order of their appearance in the output being determined by the database. To specify the order of the columns, you could type something like:

**INPUT:**

SQL> **SELECT payee, remarks, amount, check# from checks;**

Notice that each column name is listed in the **SELECT** clause. The order in which the columns are listed is the order in which they will appear in the output. Notice both the commas that separate the column names and the space between the final column name and the subsequent clause (in this case **FROM**). The output would look like this:

**OUTPUT:**

PAYEE REMARKS AMOUNT CHECK#

-------------------- ------------------ --------- ---------

Ma Bell Have sons next time 150 1

Reading R.R. Train to Chicago 245.34 2

Ma Bell Cellular Phone 200.32 3

Local Utilities Gas 98 4

Joes Stale $ Dent Groceries 150 5

Cash Wild Night Out 25 6

Joans Gas Gas 25.1 7

7 rows selected.

Another way to write the same statement follows.

**INPUT:**

**SELECT payee, remarks, amount, check#**

**FROM checks;**

Notice that the **FROM** clause has been carried over to the second line. This convention is a matter of personal taste when writing SQL code. The output would look like this:

**OUTPUT:**

PAYEE REMARKS AMOUNT CHECK#

-------------------- -------------------- --------- --------

Ma Bell Have sons next time 150 1

Reading R.R. Train to Chicago 245.34 2

Ma Bell Cellular Phone 200.32 3

Local Utilities Gas 98 4

Joes Stale $ Dent Groceries 150 5

Cash Wild Night Out 25 6

Joans Gas Gas 25.1 7

7 rows selected.

**ANALYSIS:**

The output is identical because only the format of the statement changed. Now that you have established control over the order of the columns, you will be able to specify which columns you want to see.

### Selecting Individual Columns

Suppose you do not want to see every column in the database. You used **SELECT \*** to find out what information was available, and now you want to concentrate on the check number and the amount. You type

**INPUT:**

SQL> **SELECT CHECK#, amount from checks;**

which returns

**OUTPUT:**

CHECK# AMOUNT

--------- ---------

1 150

2 245.34

3 200.32

4 98

5 150

6 25

7 25.1

7 rows selected.

**ANALYSIS:**

Now you have the columns you want to see. Notice the use of upper- and lowercase in the query. It did not affect the result. What if you need information from a different table?

### Selecting Different Tables

Suppose you had a table called **DEPOSITS** with this structure:

DEPOSIT# WHOPAID AMOUNT REMARKS

-------- ---------------------- ------ -------------------

1 Rich Uncle 200 Take off Xmas list

2 Employer 1000 15 June Payday

3 Credit Union 500 Loan

You would simply change the **FROM** clause to the desired table and type the following statement:

**INPUT:**

SQL> **select \* from deposits**

The result is

**OUTPUT:**

DEPOSIT# WHOPAID AMOUNT REMARKS

-------- ---------------------- ------ -------------------

1 Rich Uncle 200 Take off Xmas list

2 Employer 1000 15 June Payday

3 Credit Union 500 Loan

**ANALYSIS:** With a single change you have a new data source.

### Queries with Distinction

If you look at the original table, **CHECKS**, you see that some of the data repeats. For example, if you looked at the **AMOUNT** column using

**INPUT:**

SQL> **select amount from checks;**

**OUTPUT:**

AMOUNT

---------

150

245.34

200.32

98

150

25

25.1

Notice that the amount **150** is repeated. What if you wanted to see how may different amounts were in this column? Try this:

**INPUT:**

SQL> **select DISTINCT amount from checks;**

The result would be

**OUTPUT:**

AMOUNT

---------

25

25.1

98

150

200.32

245.34

6 rows selected.

**ANALYSIS:**

Notice that only six rows are selected. Because you specified **DISTINCT**, only one instance of the duplicated data is shown, that means that one less row is returned. **ALL** is a keyword that is implied in the basic **SELECT** statement. You almost never see **ALL** because **SELECT <Table>** and **SELECT ALL <Table>** have the same result. Try this example--for the first (and only!) time in your SQL career:

**INPUT:**

SQL> **SELECT ALL AMOUNT**

2 **FROM CHECKS;**

**OUTPUT:**

AMOUNT

---------

150

245.34

200.32

98

150

25

25.1

7 rows selected.

It is the same as a **SELECT <Column>**.

## CREATE TABLE

##### Creating a Table from a Table

Oracle lets you create a new table on-the-fly, based on a **select** statement on an existing table:

CREATE TABLE RAIN\_TABLE AS

SELECT CITY, PRECIPITATION FROM TROUBLE

Table created.

**NOTE**

*The* ***create table … as select …*** *command will not work if one of the selected columns uses the LONG datatype.*

When the new table is described, it reveals that it has “inherited” its column definitions from the TROUBLE table. A table created in this fashion can include all columns, using an asterisk if you like, or a subset of columns from another table. It also can include “invented” columns, which are the product of functions or the combination of other columns, just as in a view. The character column definitions will adjust to the size necessary to contain the data in the invented columns. NUMBER columns that had specified precision in the source table but undergo computation in inventing a new column, will simply be NUMBER columns, with no specified precision, in the new table.

Each table is created via the **create table** command, which specifies the names of the columns in the table, as well as the characteristics of those columns. Here is the **create table** command for the NEWSPAPER table,

CREATE TABLE NEWSPAPER

(

FEATURE VARCHAR2(15) NOT NULL,

SECTION CHAR(1),

PAGE NUMBER

);

## ALTER TABLE

**ALTER** **TABLE** <table\_name>

**ADD** <column\_name data\_type> **constraint**;

In this statement:

* First, you specify the name of the table, which you want to add the new column, after the ALTER TABLE clause.
* Second, you specify the column name, data type, and its constraint.

Note that you cannot add a column that already exists in the table; trying to do so will cause an error. In addition, the ALTER TABLE ADD column statement adds the new column at the end of the table. Oracle provides no direct way to allow you to specify the position of the new column like other database systems such as [MySQL](http://www.mysqltutorial.org/mysql-add-column/).

In case you want to add more than one column, you use the following syntax:

**ALTER** **TABLE** table\_name

**ADD** (

column\_name\_1 data\_type **constraint**,

column\_name\_2 data\_type **constraint**,

...

);

Code language: SQL (Structured Query Language) In this syntax, you separate two columns by a comma.

Oracle **ALTER TABLE ADD** column

The following statement adds a new column named birth\_date to the members table:

**ALTER** **TABLE** MEMBERS **ADD** BIRTH\_DATE DATE **NOT** NULL;

Code language: SQL (Structured Query Language). In this example, the birth\_date column is a DATE column and it does not accept null.

Suppose, you want to record the time at which a row is created and updated. To do so, you need to add two columns created at and updated at as follows:

**ALTER** **TABLE** MEMBERS **ADD**(

CREATED\_ATt TIMESTAMP **WITH** TIME ZONE **NOT** NULL,

UPDATED\_DT TIMESTAMP **WITH** TIME ZONE **NOT** NULL

);

Code language: SQL (Structured Query Language).

The data types of the CREATED\_AT and UPDATED\_AT columns are TIMESTAMP WITH TIME ZONE. These columns also do not accept null.

To change the definition of a column in a table, you use the ALTER TABLE MODIFY column syntax as follows:

**ALTER** **TABLE** <table\_name> **MODIFY** <column\_name> **action**;

Code language: SQL (Structured Query Language)

The statement is straightforward. To modify a column of a table, you need to specify the column name, table name, and action that you want to perform.

Oracle allows you to perform many actions but the following are the main ones:

* Modify the column’s visibility
* Allow or not allow null values
* Shorten or widen the size of the column
* Change the default value of a column
* Modify expression of the virtual columns

To modify multiple columns, you use the following syntax:

**ALTER** **TABLE** table\_name

**MODIFY** (

column\_name\_1 **action**,

column\_name\_2 **action**,

...

)

To make a **NOT NULL** column nullable, use the **alter table** command with the **NULL** clause, as follows:

ALTER TABLE TROUBLE MODIFY(CONDITION NULL);

**The Rules for Adding or Modifying a Column**

These are the rules for modifying a column:

* You can increase a character column’s width at any time.
* You can increase the number of digits in a NUMBER column at any time.
* You can increase or decrease the number of decimal places in a NUMBER column at any time.

In addition, if a column is **NULL** for every row of the table, you can make any of these changes:

* You can change the column’s datatype.
* You can decrease a character column’s width.
* You can decrease the number of digits in a NUMBER column.
* You can only change the datatype of a column if the column is empty (**NULL**) in all rows of the table.

There is one notable exception to the restrictions on datatype changes. Oracle supports the changing of LONG datatype columns to the LOB datatype, even if there is data already in the LONG column. The following listing illustrates this functionality:

CREATE TABLE LONGTEST

(

COL1 NUMBER,

LONGCOL LONG

);

Table created.

INSERT INTO LONGTEST VALUES(1,'This is a LONG value');

1 row created.

ALER TABLE LONGTEST MODIFY(LONGCOL CLOB);

Table altered.

desc LONGTEST

Name Null? Type

----------------------------------------- -------- --------

COL1 NUMBER

LONGCOL CLOB

#### Creating Read-Only Tables

As of Oracle 11*g,* you can alter tables to be in read-only or read-write state. This allows you to prevent **insert**, **update**, and **delete** operations at the table level (in prior releases this capability exists at the tablespace level, or requires the use of views). To make a table read-only, use the **alter table** command:

ALTER TABLE LONGTEST READ ONLY;

Attempts to change the records within LONGTEST will result in an error. To return the table to its prior state, use the **read write** clause of the **alter table** command, as shown here:

ALTER TABLE LONGTEST READ WRITE;

To see the state of a table, query the Read\_Only column of the USER\_TABLES data dictionary view. While a table is in read-only mode, you can continue to perform DML operations against it.

#### Altering Actively Used Tables

When you issue the **alter table** command, Oracle attempts to acquire a DDL lock on the table. If anyone else is accessing the table at that time, your command will fail—you need to have exclusive access to the table while you are changing its structure. You may need to repeatedly try to execute your command in order to acquire the lock you need.

As of Oracle 11*g,* you can use the DDL lock timeout options to work around this problem. You can execute the **alter session** command to set a value for the **ddl\_lock\_timeout** parameter, specifying the number of seconds during which Oracle should continually retry your command. The retry attempts will continue until the command is successful or the timeout limit is reached, whichever comes first. To try your command for 60 seconds, issue the following command:

ALTER SESSTION ddl\_lock\_timeout=60;

DBAs can enable this at the database level via the **alter system** command, as shown next:

alter system set ddl\_lock\_timeout=60;

#### Dropping a Column

You can drop a column from a table. Dropping a column is more complicated than adding or modifying a column because of the additional work that Oracle has to do. Just removing the column from the list of columns in the table—so it doesn’t show up when you **select** \* from the table—is easy. It’s recovering the space that was actually taken up by the column values that is more complex, and potentially very time-consuming for the database. For this reason, you can drop a column immediately or you can mark it as “unused,” to be dropped at a later time. If the column is dropped immediately, the action may impact performance. If the column is marked as unused, there will be no impact on performance. The column can actually be dropped at a later time when the database is less heavily used.

To drop a column, use either the **set unused** clause or the **drop** clause of the **alter table** command. You cannot drop a pseudo-column, a column of a nested table, or a partition key column. In the following example, column Wind is dropped from the TROUBLE table:

ALTER TABLE TROUBLE DROP COLUMN WIND;

Alternatively, you can mark the Wind column as unused:

ALTER TABLE TROUBLE SET UNUSED COLUMN WND;

## Creating a View

There is even more here than meets the eye. Not only does this look like a new table, but you can give it a name and treat it like one. This is called “creating a view.” A *view* provides a way of hiding the logic that created the joined table just displayed. It works this way:

CREATE VIEW INVASION AS

SELECT CITY, CONDITION, TEMPERATURE, LATTITUDE, NORTHSOUTH, LONGITURE,

EASTWEST

FROM

WEATHER,

View created.

Now you can act as if INVASION were a real table with its own rows and columns. You can even ask Oracle to describe it to you:

DESCRIBE INVASION

Name Null? Type

------------------------------- -------- ------------

CITY VARCHAR2(11)

CONDITION VARCHAR2(9)

TEMPERATURE NUMBER

LATITUDE NUMBER

NORTHSOUTH CHAR(1)

LONGITUDE NUMBER

EASTWEST CHAR(1)

You can query it, too (note that you will not have to specify which table the City columns were from, because that logic is hidden inside the view):’

SELECT CITY, CONDITION, TEMPERATURE, LATTITUDE, NORTHWIND,LONGITUDE, EASTWEST

FROM INVASION;

CITY CONDITION TEMPERATURE LATITUDE N LONGITUDE E

---- --------- ----------- -------- - ---------- -

ATHENS SUNNY 97 37. 58 N 23.43 E

CHICAGO RAIN 66 41.53 N 87.38 W

LIMA RAIN 45 12.03 S 77.03 W

There will be some Oracle functions you won’t be able to use on a view that you can use on a plain table, but they are few and mostly involve modifying rows and indexing tables, which will be discussed in later chapters. For the most part, a view behaves and can be manipulated just like any other table.

# Introduction to PL/SQL

## Introduction

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. PL/SQL is not a stand-alone language. Following are notable facts about PL/SQL:

### Features of PL/SQL

* PL/SQL is a completely portable, high-performance transaction-processing language.
* PL/SQL provides a built-in interpreted and OS independent programming environment.
* PL/SQL can also directly be called from the command-line SQL\*Plus interface.
* Direct call can also be made from external programming language calls to database.
* PL/SQL's general syntax is based on that of ADA and Pascal programming language.
* PL/SQL is tightly integrated with SQL.
* It offers extensive error checking.
* It offers numerous data types.
* It offers a variety of programming structures.
* It supports structured programming through functions and procedures.
* It supports object-oriented programming.

SQL\* Plus is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt. There are other tools also available to type PL/SQL and execute, such as Oracle Developer in the later versions of Oracle, Toad etc. These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen.

To run PL/SQL programs, you should have Oracle RBDMS Server installed in your machine which will take care of executing SQL commands.

#### Basic Structure Of PL/SQL Code

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

1. **Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2. **Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
3. **Exception Handling** This section starts with the keyword **EXCEPTION**. This section is again optional and contains exception(s) that handle errors in the program.

Below is an example for a typical PL/SQL code block.

**DECLARE**

<declarations section>

**BEGIN**

<executable command(s)>

**EXCEPTION**

<exception handling>

**END;**

Below is the sample PL/SQL program for displaying “Hello World”:

**DECLARE**

message varchar2(20):= 'Hello, World!';

**BEGIN**

dbms\_output.put\_line(message);

**END;**

/

### The PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

### The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL:

|  |  |
| --- | --- |
| **Delimiter** | **Description** |
| **+, -, \*, /** | Addition, subtraction/negation, multiplication, division |
| **%** | Attribute indicator |
| **'** | Character string delimiter |
| **.** | Component selector |
| **(,)** | Expression or list delimiter |
| **:** | Host variable indicator |
| **,** | Item separator |
| **"** | Quoted identifier delimiter |
| **=** | Relational operator |
| **@** | Remote access indicator |
| **;** | Statement terminator |
| **:=** | Assignment operator |
| **=>** | Association operator |
| **||** | Concatenation operator |
| **\*\*** | Exponentiation operator |
| **<<, >>** | Label delimiter (begin and end) |
| **/\*, \*/** | Multi-line comment delimiter (begin and end) |
| **--** | Single-line comment indicator |
| **..** | Range operator |
| **<, >, <=, >=** | Relational operators |
| **<>, '=, ~=, ^=** | Different versions of NOT EQUAL |

### The PL/SQL Comments

Program comments are explanatory statements that you can include in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow for some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by PL/SQL compiler. The PL/SQL single-line comments start with the delimiter **--**(double hyphen) and multi-line comments are enclosed by /\* and \*/.

DECLARE

-- variable declaration

message varchar2(20):= 'Hello, World!';

BEGIN

/\*

\* PL/SQL executable statement(s)

\*/

dbms\_output.put\_line(message);

END;

/

When the above code is executed at SQL prompt, it produces the following result:

Hello World

PL/SQL procedure successfully completed.

## Data Types in PL/SQL

Most data types are obviously similar, but each implementation has unique storage and internal-processing requirements. When writing PL/SQL blocks, you will be declaring variables, which must be valid data types. The following subsections briefly describe the data types available in PL/SQL. In PL/SQL Oracle provides subtypes of data types. For example, the data type **NUMBER** has a subtype called **INTEGER**. You can use subtypes in your PL/SQL program to make the data types compatible with data types in other programs, such as a COBOL program, particularly if you are embedding PL/SQL code in another program. Subtypes are simply alternative names for Oracle data types and therefore must follow the rules of their associated data type.

**NOTE:** As in most implementations of SQL, case sensitivity is not a factor in the syntax of a statement. PL/SQL allows either uppercase or lowercase with its commands.

### Character String Data Types

Character string data types in PL/SQL, as you might expect, are data types generally defined as having alpha numeric values. Examples of character strings are names, codes,descriptions, and serial numbers that include characters.

**CHAR** stores fixed-length character strings. The maximum length of **CHAR** is 32,767 bytes, although it is hard to imagine a set of fixed-length values in a table being so long.

**SYNTAX:**

CHAR ( max\_length )

Subtype: **CHARACTER**

**VARCHAR2** stores variable-length character strings. You would normally use **VARCHAR2** instead of **CHAR** to store variable-length data, such as an individual's name. The maximum length of **VARCHAR2** is also 32,767 bytes.

**SYNTAX:**

VARCHAR2 ( max\_length )

Subtypes: **VARCHAR**, **STRING**

**LONG** also stores variable-length character strings, having a maximum length of 32,760 bytes. **LONG** is typically used to store lengthy text such as remarks, although **VARCHAR2** may be used as well.

### Numeric Data Types

**NUMBER** stores any type of number in an Oracle database.

**SYNTAX:**

NUMBER ( max\_length )

You may specify a **NUMBER**'s data precision with the following syntax:

NUMBER (precision, scale)

Subtypes: **DEC**, **DECIMAL**, **DOUBLE PRECISION**, **INTEGER**, **INT**, **NUMERIC**, **REAL**, **SMALLINT**, **FLOAT**,**PLS\_INTEGER** defines columns that may contained integers with a sign, such as negative numbers.

### Binary Data Types

Binary data types store data that is in a binary format, such as graphics or photographs. These data types include **RAW** and **LONGRAW**.

### The DATE Data Type

**DATE** is the valid Oracle data type in which to store dates. When you define a column as a **DATE**, you do not specify a length, as the length of a **DATE** field is implied.

The format of an Oracle date is, for example, 01-OCT-97.

### BOOLEAN

**BOOLEAN** stores the following values: **TRUE**, **FALSE**, and **NULL**. Like **DATE**, **BOOLEAN** requires no parameters when defining it as a column's or variable's data type.

### ROWID

**ROWID** is a pseudo column that exists in every table in an Oracle database. The **ROWID** is stored in binary format and identifies each row in a table. Indexes use **ROWID**s as pointers to data.

## Variable Assignment

Variables are values that are subject to change within a PL/SQL block. PL/SQL variables must be assigned a valid data type upon declaration and can be initialized if necessary. The following example defines a set of variables in the **DECLARE** portion of a block:

DECLARE

owner char(10);

tablename char(30);

bytes number(10);

today date;

**ANALYSIS:**

The **DECLARE** portion of a block cannot be executed by itself. The **DECLARE** section starts with the **DECLARE** statement. Then individual variables are defined on separate lines. Notice that each variable declaration ends with a semicolon. Variables may also be initialized in the **DECLARE** section. For example:

DECLARE

customer char(30);

fiscal\_year number(2) := '97';

You can use the symbol **:=** to initialize, or assign an initial value, to variables in the **DECLARE** section. You must initialize a variable that is defined as **NOT NULL**.

DECLARE

customer char(30);

fiscal\_year number(2) NOT NULL := '97';

**ANALYSIS:**

The **NOT NULL** clause in the definition of **fiscal\_year** resembles a column definition in a **CREATE TABLE** statement.

## Constant Assignment

Constants are defined the same way that variables are, but constant values are static; they do not change. In the previous example, **fiscal\_year** is probably a constant.

**NOTE:** You must end each variable declaration with a semicolon.

# Cursors

A cursor is another type of variable in PL/SQL. Usually when you think of a variable, a single value comes to mind. A cursor is a variable that points to a row of data from the results of a query. In a multiple-row result set, you need a way to scroll through each record to analyze the data. A cursor is just that. When the PL/SQL block looks at the results of a query within the block, it uses a cursor to point to each returned row. Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

* Implicit cursors
* Explicit cursors

## Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK\_ROWCOUNT and %BULK\_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement |

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

Example:

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

The following program would update the table and increase salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected:

DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 500;

IF sql%notfound THEN

dbms\_output.put\_line('no customers selected');

ELSIF sql%found THEN

total\_rows := sql%rowcount;

dbms\_output.put\_line( total\_rows || ' customers selected ');

END IF;

END;

/

When the above code is executed at SQL prompt, it produces the following result:

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated:

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |

| 2 | Khilan | 25 | Delhi | 2000.00 |

| 3 | kaushik | 23 | Kota | 2500.00 |

| 4 | Chaitali | 25 | Mumbai | 7000.00 |

| 5 | Hardik | 27 | Bhopal | 9000.00 |

| 6 | Komal | 22 | MP | 5000.00 |

+----+----------+-----+-----------+----------+

### The %TYPE Attribute

**%TYPE** is a variable attribute that returns the value of a given column of a table. Instead of hard-coding the data type in your PL/SQL block, you can use **%TYPE** to maintain data type consistency within your blocks of code.

**INPUT:**

**DECLARE**

**cursor employee\_cursor is**

**select \* from employees;**

A cursor is similar to a view. With the use of a loop in the **PROCEDURE** section, you can scroll a cursor. This technique is covered shortly.

**INPUT:**

**DECLARE**

**cursor employee\_cursor is**

**select emp\_id, emp\_name from employees;**

**id\_num employees.emp\_id%TYPE;**

**name employees.emp\_name%TYPE;**

**ANALYSIS:**

The variable **id\_num** is declared to have the same data type as **emp\_id** in the **EMPLOYEES** table. **%TYPE** declares the variable **name** to have the same data type as the column **emp\_name** in the **EMPLOYEES** table.

### The %ROWTYPE Attribute

Variables are not limited to single values. If you declare a variable that is associated with a defined cursor, you can use the **%ROWTYPE** attribute to declare the data type of that variable to be the same as each column in one entire row of data from the cursor. In Oracle's lexicon the **%ROWTYPE** attribute creates a record variable.

**INPUT:**

**DECLARE**

**cursor employee\_cursor is**

**select emp\_id, emp\_name from employees;**

**employee\_record employee\_cursor%ROWTYPE;**

**ANALYSIS:**

This example declares a variable called **employee\_record.** The **%ROWTYPE** attribute defines this variable as having the same data type as an entire row of data in the **employee\_cursor**. Variables declared using the **%ROWTYPE** attribute are also called aggregate variables.

### The %ROWCOUNT Attribute

The PL/SQL **%ROWCOUNT** attribute maintains a count of rows that the SQL statements in the particular block have accessed in a cursor.

**INPUT:**

**DECLARE**

**cursor employee\_cursor is**

**select emp\_id, emp\_name from employees;**

**records\_processed := employee\_cursor%ROWCOUNT;**

**ANALYSIS:**

In this example the variable **records\_processed** represents the current number of rows that the PL/SQL block has accessed in the **employee\_cursor**.

**WARNING:** Beware of naming conflicts with table names when declaring variables. For instance, if you declare a variable that has the same name as a table that you are trying to access with the PL/SQL code, the local variable will take precedence over the table name.

## Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is

CURSOR cursor\_name IS select\_statement;

Working with an explicit cursor involves four steps:

* Declaring the cursor for initializing in the memory
* Opening the cursor for allocating memory
* Fetching the cursor for retrieving data
* Closing the cursor to release allocated memory

## The PROCEDURE Section

The **PROCEDURE** section is the only mandatory part of a PL/SQL block. This part of the block calls variables and uses cursors to manipulate data in the database. The **PROCEDURE** section is the main part of a block, containing conditional statements and SQL commands.

**BEGIN...END**

In a block, the **BEGIN** statement denotes the beginning of a procedure. Similarly, the **END** statement marks the end of a procedure. The following example shows the basic structure of the **PROCEDURE** section:

**SYNTAX:**

BEGIN

open a cursor;

condition1;

statement1;

condition2;

statement2;

condition3;

statement3;

.

.

close the cursor;

END

# Output Statements

## Displaying Output to the User

Particularly when handling exceptions, you may want to display output to keep users informed about what is taking place. You can display output to convey information, and you can display your own customized error messages, which will probably make more sense to the user than an error number. Perhaps you want the user to contact the database administrator if an error occurs during processing, rather than to see the exact message.

PL/SQL does not provide a direct method for displaying output as a part of its syntax, but it does allow you to call a package that serves this function from within the block. The package is called **DBMS\_OUTPUT**.

EXCEPTION

WHEN zero\_divide THEN

DBMS\_OUTPUT.put\_line('ERROR: DIVISOR IS ZERO. SEE YOUR DBA.');

**ANALYSIS:**

**ZERO\_DIVIDE** is an Oracle predefined exception. Most of the common errors that occur during program processing will be predefined as exceptions and are raised implicitly (which means that you don't have to raise the error in the PROCEDURE section of the block).If this exception is encountered during block processing, the user will see:

**INPUT:**

SQL> **@block1**

ERROR: DIVISOR IS ZERO. SEE YOUR DBA.

PL/SQL procedure successfully completed.

Doesn't that message look friendly than:

**INPUT/OUTPUT:**

SQL> **@block1**

**begin**

**\***

ERROR at line 1:

ORA-01476: divisor is equal to zero

ORA-06512: at line 20

# Control Statements

## Cursor Control Commands

Now that you have learned how to define cursors in a PL/SQL block, you need to know how to access the defined cursors. This section explains the basic cursor control commands: **DECLARE**, **OPEN**, **FETCH**, and **CLOSE**.

### DECLARE

Earlier today you learned how to define a cursor in the **DECLARE** section of a block. The **DECLARE** statement belongs in the list of cursor control commands.

### OPEN

Now that you have defined your cursor, how do you use it? You cannot use this book unless you open it. Likewise, you cannot use a cursor until you have opened it with the **OPEN** command. For example:

**SYNTAX:**

BEGIN

open employee\_cursor;

statement1;

statement2;

.

.

.

END

### FETCH

**FETCH** populates a variable with values from a cursor. Here are two examples using **FETCH**: One populates an aggregate variable, and the other populates individual variables.

**INPUT:**

DECLARE

cursor employee\_cursor is

select emp\_id, emp\_name from employees;

employee\_record employee\_cursor%ROWTYPE;

BEGIN

open employee\_cursor;

loop

fetch employee\_cursor into employee\_record;

end loop;

close employee\_cursor;

END

**ANALYSIS:**

The preceding example fetches the current row of the cursor into the aggregate variable **employee\_record**. It uses a loop to scroll the cursor. Of course, the block is not actually accomplishing anything.

DECLARE

cursor employee\_cursor is

select emp\_id, emp\_name from employees;

id\_num employees.emp\_id%TYPE;

name employees.emp\_name%TYPE;

BEGIN

open employee\_cursor;

loop

fetch employee\_cursor into id\_num, name;

end loop;

close employee\_cursor;

END

**ANALYSIS:**

This example fetches the current row of the cursor into the variables **id\_num** and **name**, which was defined in the **DECLARE** section.

### CLOSE

When you have finished using a cursor in a block, you should close the cursor, as you normally close a book when you have finished reading it. The command you use is **CLOSE**.

**SYNTAX:**

BEGIN

open employee\_cursor;

statement1;

statement2;

.

.

close employee\_cursor;

END

**ANALYSIS:**

After a cursor is closed, the result set of the query no longer exists. You must reopen the cursor to access the associated set of data.

## Transactional Control commands in PL/SQL

The transactional control commands allow the programmer to control when transactions are actually written to the database, how often, and when they should be undone. They are **COMMIT**, **ROLLBACK**, and **SAVEPOINT**.

### COMMIT

### ROLLBACK

### SAVEPOINT

**SYNTAX:**

BEGIN

DECLARE

...

BEGIN

statements...

IF condition THEN

COMMIT;

ELSE

ROLLBACK;

END IF;

...

EXCEPTION

...

END;

END;

## Statements Control Commands

### Conditional Statements

Now we are getting to the good stuff--the conditional statements that give you control over how your SQL statements are processed. The conditional statements in PL/SQL resemble those in most third-generation languages.

### IF...THEN

The **IF...THEN** statement is probably the most familiar conditional statement to most programmers. The **IF...THEN** statement dictates the performance of certain actions if certain conditions are met. The structure of an **IF...THEN** statement is as follows:

**SYNTAX:**

IF condition1 THEN

statement1;

END IF;

If you are checking for two conditions, you can write your statement as follows:

**SYNTAX:**

IF condition1 THEN

statement1;

ELSE

statement2;

END IF;

If you are checking for more than two conditions, you can write your statement as follows:

**SYNTAX:**

IF condition1 THEN

statement1;

ELSIF condition2 THEN

statement2;

ELSE

statement3;

END IF;

**ANALYSIS:**

The final example states: If **condition1** is met, then perform **statement1**; if **condition2** is met, then perform **statement2**; otherwise, perform **statement3**. **IF...THEN** statements may also be nested within other statements and/or loops.

### LOOPS

Loops in a PL/SQL block allow statements in the block to be processed continuously for as long as the specified condition exists. There are three types of loops. **LOOP** is an infinite loop, most often used to scroll a cursor. To terminate this type of loop, you must specify when to exit. For example, in scrolling a cursor you would exit the loop after the last row in a cursor has been processed:

**INPUT:**

BEGIN

open employee\_cursor;

LOOP

FETCH employee\_cursor into employee\_record;

EXIT WHEN employee\_cursor%NOTFOUND;

statement1;

.

.

.

END LOOP;

close employee\_cursor;

END;

%NOTFOUNDis a cursor attribute that identifies when no more data is found in the cursor. The preceding example exits the loop when no more data is found. If you omit this statement from the loop, then the loop will continue forever.

#### WHILE-LOOP

The **WHILE-LOOP** executes commands while a specified condition is **TRUE**. When the condition is no longer true, the loop returns control to the next statement.

**INPUT:**

DECLARE

cursor payment\_cursor is

select cust\_id, payment, total\_due from payment\_table;

cust\_id payment\_table.cust\_id%TYPE;

payment payment\_table.payment%TYPE;

total\_due payment\_table.total\_due%TYPE;

BEGIN

open payment\_cursor;

WHILE payment < total\_due LOOP

FETCH payment\_cursor into cust\_id, payment, total\_due;

EXIT WHEN payment\_cursor%NOTFOUND;

insert into underpay\_table values (cust\_id, 'STILL OWES');

END LOOP;

close payment\_cursor;

END;

**ANALYSIS:**

The preceding example uses the **WHILE-LOOP** to scroll the cursor and to execute the commands within the loop as long as the condition **payment < total\_due** is met.

#### FOR-LOOP

You can use the **FOR-LOOP** in the previous block to implicitly fetch the current row of the cursor into the defined variables.

**INPUT:**

DECLARE

cursor payment\_cursor is

select cust\_id, payment, total\_due from payment\_table;

cust\_id payment\_table.cust\_id%TYPE;

payment payment\_table.payment%TYPE;

total\_due payment\_table.total\_due%TYPE;

BEGIN

open payment\_cursor;

FOR pay\_rec IN payment\_cursor LOOP

IF pay\_rec.payment < pay\_rec.total\_due THEN

insert into underpay\_table values (pay\_rec.cust\_id, 'STILL OWES');

END IF;

END LOOP;

close payment\_cursor;

END;

**ANALYSIS:**

This example uses the **FOR-LOOP** to scroll the cursor. The **FOR-LOOP** is performing an implicit **FETCH**, which is omitted this time. Also, notice that the **%NOTFOUND** attribute has been omitted. This attribute is implied with the **FOR-LOOP**; therefore, this and the previous example yield the same basic results.

## The EXCEPTION Section

The **EXCEPTION** section is an optional part of any PL/SQL block. If this section is omitted and errors are encountered, the block will be terminated. Some errors that are encountered may not justify the immediate termination of a block, so the **EXCEPTION** section can be used to handle specified errors or user-defined exceptions in an orderly manner. Exceptions can be user-defined, although many exceptions are predefined by Oracle.

### Raising Exceptions

Exceptions are raised in a block by using the command **RAISE**. Exceptions can be raised explicitly by the programmer, whereas internal database errors are automatically, or implicitly, raised by the database server.

**SYNTAX:**

BEGIN

DECLARE

exception\_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

EXCEPTION

WHEN exception\_name THEN

statement;

END;

END;

**ANALYSIS:**

This block shows the fundamentals of explicitly raising an exception. First **exception\_name** is declared using the **EXCEPTION** statement. In the **PROCEDURE** section, the exception is raised using **RAISE** if a given condition is met. The **RAISE** then references the **EXCEPTION** section of the block, where the appropriate action is taken.

### Handling Exceptions

The preceding example handled an exception in the **EXCEPTION** section of the block. Errors are easily handled in PL/SQL, and by using exceptions, the PL/SQL block can continue to run with errors or terminate gracefully.

**SYNTAX:**

EXCEPTION

WHEN exception1 THEN

statement1;

WHEN exception2 THEN

statement2;

WHEN OTHERS THEN

statement3;

**ANALYSIS:**

This example shows how the **EXCEPTION** section might look if you have more than one exception. This example expects two exceptions (**exception1** and **exception2**) when running this block. **WHEN OTHERS** tells **statement3** to execute if any other exceptions occur while the block is being processed. **WHEN OTHERS** gives you control over any errors that may occur within the block.

# Procedures

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

* At schema level
* Inside a package
* Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

## Parameter types and passing technics

We can pass input values to a procedure or function as a parameter. Parameters can be of three types

1. Input Parameters
2. Output Parameters and
3. InOut Parameters.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

* **Functions**: these subprograms return a single value, mainly used to compute and return a value.
* **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

## IN only parameter:

An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference.**

## OUT only parameter

An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.

## IN OUT Parameter

An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter

#### IN & OUT Mode Example 1

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

CREATE OR REPLACE PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

DECLARE

a number;

b number;

c number;

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;

BEGIN

a:= 23;

b:= 45;

findMin(a, b, c);

dbms\_output.put\_line(' Minimum of (23, 45) : ' || c);

END;

/

When the above code is executed

Minimum of (23, 45): 23

PL/SQL procedure successfully completed.

#### IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

CREATE PROCEDURE squareNum(x IN OUT number) IS

DECLARE

a number;

BEGIN

x := x \* x;

END;

BEGIN

a:= 23;

squareNum(a);

dbms\_output.put\_line(' Square of (23): ' || a);

END;

/

When the above code is executed at SQL prompt, it produces the following result:

Square of (23): 529

PL/SQL procedure successfully completed.

When the above code is executed at SQL prompt, it produces the following result:

Square of (23): 529

PL/SQL procedure successfully completed.

## Methods for Passing Parameters

Actual parameters could be passed in three ways:

* Positional notation
* Named notation
* Mixed notation

### POSITIONAL NOTATION

In positional notation, you can call the procedure as:

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

### NAMED NOTATION

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol ( => ). So the procedure call would look like:

findMin(x=>a, y=>b, z=>c, m=>d);

### MIXED NOTATION

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation. The following call is legal:

findMin(a, b, c, m=>d);

But this is not legal:

findMin(x=>a, b, c, d);

## Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

Where,

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows modifying an existing procedure.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* *procedure-body* contains the executable part.

The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

When above code is executed using SQL prompt, it will produce the following result:

Procedure created.

# Deleting a Standalone Procedure

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is:

DROP PROCEDURE procedure-name;

So you can drop *greetings* procedure by using the following statement:

BEGIN

DROP PROCEDURE greetings;

END;

/

Parameter Modes In PL/SQL Subprograms

# Functions

A PL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

## Library Functions

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Function** | **Purpose** |
| 1 | **ASCII(x)** | Returns the ASCII value of the character x. |
| 2 | **CHR(x)** | Returns the character with the ASCII value of x. |
| 3 | **CONCAT(x, y)** | Concatenates the strings x and y and return the appended string. |
| 4 | **INITCAP(x)** | Converts the initial letter of each word in x to uppercase and returns that string. |
| 5 | **INSTR(x, find\_string [, start] [, occurrence])** | Searches for find\_string in x and returns the position at which it occurs. |
| 6 | **INSTRB(x)** | Returns the location of a string within another string, but returns the value in bytes. |
| 7 | **LENGTH(x)** | Returns the number of characters in x. |
| 8 | **LENGTHB(x)** | Returns the length of a character string in bytes for single byte character set. |
| 9 | **LOWER(x)** | Converts the letters in x to lowercase and returns that string. |
| 10 | **LPAD(x, width [, pad\_string])** | Pads x with spaces to left, to bring the total length of the string up to width characters. |
| 11 | **LTRIM(x [, trim\_string])** | Trims characters from the left of x. |
| 12 | **NANVL(x, value)** | Returns value if x matches the NaN special value (not a number), otherwise x is returned. |
| 13 | **NLS\_INITCAP(x)** | Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT. |
| 14 | **NLS\_LOWER(x)** | Same as the LOWER function except that it can use a different sort method as specified by NLSSORT. |
| 15 | **NLS\_UPPER(x)** | Same as the UPPER function except that it can use a different sort method as specified by NLSSORT. |
| 16 | **NLSSORT(x)** | Changes the method of sorting the characters. Must be specified before any NLS function, otherwise, the default sort will be used. |
| 17 | **NVL(x, value)** | Returns value if x is null, otherwise, x is returned. |
| 18 | **NVL2(x, value1, value2)** | Returns value1 if x is not null, if x is null, value2 is returned. |
| 19 | **REPLACE(x, search\_string, replace\_string)** | Searches x for search\_string and replaces it with replace\_string. |
| 20 | **RPAD(x, width [, pad\_string])** | Pads x to the right. |
| 21 | **RTRIM(x [, trim\_string])** | Trims x from the right. |
| 22 | **SOUNDEX(x)** | Returns a string containing the phonetic representation of x. |
| 23 | **SUBSTR(x, start [, length])** | Returns a substring of x that begins at the position specified by start. An optional length for the substring may be supplied. |
| 24 | **SUBSTRB(x)** | Same as SUBSTR except the parameters are expressed in bytes instead of characters for the single-byte character systems. |
| 25 | **TRIM([trim\_char FROM) x)** | Trims characters from the left and right of x. |
| 26 | **UPPER(x)** | Converts the letters in x to uppercase and returns that string. |
| 27 | **EXISTS(n)** | otherwise returns FALSE. |
| 28 | **COUNT Returns the number of elements that a collection currently contains.** | Returns TRUE if the nth element in a collection exists |
| 29 | **LIMIT** | Checks the Maximum Size of a Collection. |
| 30 | **FIRST** | Returns the first (smallest) index numbers in a collection that uses integer subscripts. |
| 31 | **LAST** | Returns the last (largest) index numbers in a collection that uses integer subscripts. |
| 32 | **PRIOR(n)** | Returns the index number that precedes index n in a collection. |
| 33 | **NEXT(n)** | Returns the index number that succeeds index n. |
| 34 | **EXTEND** | Appends one null element to a collection. |
| 35 | **EXTEND(n)** | Appends n null elements to a collection. |
| 36 | **EXTEND(n,i)** | Appends n copies of the ith element to a collection. |
| 37 | **TRIM** | Removes one element from the end of a collection. |
| 38 | **TRIM(n)** | Removes n elements from the end of a collection. |
| 39 | **DELETE** | Removes all elements from a collection, setting COUNT to 0. |
| 40 | **DELETE(n)** | Removes the nth element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, DELETE(n) does nothing. |
| 41 | **DELETE(m,n)** | Removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing. |
| 42 | **ADD\_MONTHS(x, y)** | Adds y months to x. |
| 43 | **LAST\_DAY(x)** | Returns the last day of the month. |
| 44 | **MONTHS\_BETWEEN(x, y)** | Returns the number of months between x and y. |
| 45 | **NEXT\_DAY(x, day)** | Returns the datetime of the next day after x. |
| 46 | **NEW\_TIME** | Returns the time/day value from a time zone specified by the user. |
| 47 | **ROUND(x [, unit])** | Rounds x |
| 48 | **SYSDATE()** | Returns the current datetime. |
| 49 | **TRUNC(x [, unit])** | Truncates x. |
| 50 | **CURRENT\_TIMESTAMP()** | Returns a TIMESTAMP WITH TIME ZONE containing the current session time along with the session time zone. |
| 51 | **EXTRACT({ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND } | { TIMEZONE\_HOUR | TIMEZONE\_MINUTE } | { TIMEZONE\_REGION | } TIMEZONE\_ABBR ) FROM x)** | Extracts and returns a year, month, day, hour, minute, second, or time zone from x |
| 52 | **FROM\_TZ(x, time\_zone)** | Converts the TIMESTAMP x and time zone specified by time\_zone to a TIMESTAMP WITH TIMEZONE. |
| 53 | **LOCALTIMESTAMP()** | Returns a TIMESTAMP containing the local time in the session time zone. |
| 54 | **SYSTIMESTAMP()** | Returns a TIMESTAMP WITH TIME ZONE containing the current database time along with the database time zone. |
| 55 | **SYS\_EXTRACT\_UTC(x)** | Converts the TIMESTAMP WITH TIMEZONE x to a TIMESTAMP containing the date and time in UTC. |
| 56 | **TO\_TIMESTAMP(x, [format])** | Converts the string x to a TIMESTAMP. |
| 57 | **TO\_TIMESTAMP\_TZ(x, [format])** | Converts the string x to a TIMESTAMP WITH TIMEZONE. |
| 58 | **NUMTOYMINTERVAL(x, interval\_unit)** | Converts the number x to an INTERVAL YEAR TO MONTH. |
| 59 | **TO\_DSINTERVAL(x)** | Converts the string x to an INTERVAL DAY TO SECOND. |
| 60 | **TO\_YMINTERVAL(x)** | Converts the string x to an INTERVAL YEAR TO MONTH. |
| 61 | **NUMTODSINTERVAL(x, interval\_unit)** | Converts the number x to an INTERVAL DAY TO SECOND. |

## Stored Functions (User Defined)

### Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

Where,

* *function-name* specifies the name of the function.
* [OR REPLACE] option allows modifying an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* *RETURN* clause specifies that data type you are going to return from the function.
* *function-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

#### Example

The following example illustrates creating and calling a standalone function. This function returns the total number of CUSTOMERS in the customers table. We will use the CUSTOMERS table, which we had created in PL/SQL Variables chapter:

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total FROM customers;

RETURN total;

END;

/

When above code is executed using SQL prompt, it will produce the following result:

Function created.

### Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function.

A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms\_output.put\_line('Total no. of Customers: ' || c);

END;

/

Total no. of Customers: 6

PL/SQL procedure successfully completed.

#### Example:

The following is one more example which demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

/

When the above code is executed at SQL prompt, it produces the following result:

Maximum of (23,45): 78

PL/SQL procedure successfully completed.

### PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

n! = n\*(n-1)!

= n\*(n-1)\*(n-2)!

...

= n\*(n-1)\*(n-2)\*(n-3)... 1

DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number

IS

f number;

BEGIN

IF x=0 THEN

f := 1;

ELSE

f := x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:= 6;

factorial := fact(num);

dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);

END;

/

When the above code is executed at SQL prompt, it produces the following result:

# Factorial 6 is 720

# Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers.

Triggers can be written for the following purposes:

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations

## Creating Triggers

The syntax for creating a trigger is:

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Where,

* CREATE [OR REPLACE] TRIGGER trigger\_name: Creates or replaces an existing trigger with the *trigger\_name*.
* {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
* [OF col\_name]: This specifies the column name that would be updated.
* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

#### Example:

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters:

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

The following program creates a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

dbms\_output.put\_line('Old salary: ' || :OLD.salary);

dbms\_output.put\_line('New salary: ' || :NEW.salary);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

Trigger created.

* OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
* Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

## Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (7, 'Kriti', 22, 'HP', 7500.00 );

When a record is created in CUSTOMERS table, above create trigger display\_salary\_changes will be fired and it will display the following result:

Old salary:

New salary: 7500

Salary difference:

Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

UPDATE customers

SET salary = salary + 500

WHERE id = 2;

When a record is updated in CUSTOMERS table, above create trigger display\_salary\_changes will be fired and it will display the following result:

Old salary: 1500

New salary: 2000

Salary difference: 500

# Packages

#### What is a Package?

A **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

A package always has a **specification**, which declares the **public items** that can be referenced from outside the package.

If the public items include cursors or subprograms, then the package must also have a **body**. The body must define queries for public cursors and code for public subprograms. The body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an **initialization part**, whose statements initialize variables and do other one-time setup steps, and an exception-handling part. You can change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box.

In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using a **call specification**, which maps the external subprogram name, parameter types, and return type to their SQL counterparts.

The AUTHID **clause** of the package specification determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.

The ACCESSIBLE BY **clause** of the package specification lets you specify a white list of PL/SQL units that can access the package. You use this clause in situations like these:

* You implement a PL/SQL application as several packages—one package that provides the application programming interface (API) and helper packages to do the work. You want clients to have access to the API, but not to the helper packages. Therefore, you omit the ACCESSIBLE BY clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.
* You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict use of the package to the intended units, you list them in the ACCESSIBLE BY clause in the package specification.

### Reasons to Use Packages

Packages support the development and maintenance of reliable, reusable code with the following features:

* **Modularity**

Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules. You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined. This practice aids application development.

* **Easier Application Design**

When designing an application, all you need initially is the interface information in the package specifications. You can code and compile specifications without their bodies. Next, you can compile standalone subprograms that reference the packages. You need not fully define the package bodies until you are ready to complete the application.

* **Hidden Implementation Details**

Packages let you share your interface information in the package specification, and hide the implementation details in the package body. Hiding the implementation details in the body has these advantages:

* You can change the implementation details without affecting the application interface.
* Application users cannot develop code that depends on implementation details that you might want to change.
* **Added Functionality**

Package public variables and cursors can persist for the life of a session. They can be shared by all subprograms that run in the environment. They let you maintain data across transactions without storing it in the database. (For the situations in which package public variables and cursors do not persist for the life of a session, see "Package State".)

* **Better Performance**

The first time you invoke a package subprogram, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O.

Packages prevent cascading dependencies and unnecessary recompiling. For example, if you change the body of a package function, Oracle Database does not recompile other subprograms that invoke the function, because these subprograms depend only on the parameters and return value that are declared in the specification.

* **Easier to Grant Roles**

You can grant roles on the package, instead of granting roles on each object in the package.

### Package Specification

A **package specification** declares **public items**. The scope of a public item is the schema of the package. A public item is visible everywhere in the schema. To reference a public item that is in scope but not visible, qualify it with the package name. Each public item declaration has all information needed to use the item. For example, suppose that a package specification declares the function factorial this way:

FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!

The declaration shows that factorial needs one argument of type INTEGER and returns a value of type INTEGER, which is invokers must know to invoke factorial. Invokers need not know how factorial is implemented (for example, whether it is iterative or recursive).

#### Appropriate Public Items

Appropriate public items are:

* Types, variables, constants, subprograms, cursors, and exceptions used by multiple subprograms

A type defined in a package specification is either a PL/SQL user-defined subtype

* Associative array types of standalone subprogram parameters

You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone subprogram, you must declare the type of that variable in a package specification. Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and to the invoking subprogram or anonymous block.

* Variables that must remain available between subprogram invocations in the same session
* Subprograms that read and write public variables ("get" and "set" subprograms)

Provide these subprograms to discourage package users from reading and writing public variables directly.

* Subprograms that invoke each other

You need not worry about compilation order for package subprograms, as you must for standalone subprograms that invoke each other.

* Overloaded subprograms

Overloaded subprograms are variations of the same subprogram. That is, they have the same name but different formal parameters. For more information about them, see "Overloaded Subprograms".

### Creating Package Specifications

To create a package specification, use the "CREATE PACKAGE Statement". Because the package specifications do not declare cursors or subprograms, the packages trans\_data and aa\_pkg do not need bodies.

***Example 10-1 Simple Package Specification***

In this example, the specification for the package trans\_data declares two public types and three public variables.

CREATE OR REPLACE PACKAGE trans\_data AUTHID DEFINER AS

TYPE TimeRec IS RECORD (

minutes SMALLINT,

hours SMALLINT);

TYPE TransRec IS RECORD (

category VARCHAR2(10),

account INT,

amount REAL,

time\_of TimeRec);

minimum\_balance CONSTANT REAL := 10.00;

number\_processed INT;

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -4097);

END trans\_data;

/

***Example 10-2 Passing Associative Array to Standalone Subprogram***

In this example, the specification for the package aa\_pkg declares an associative array type, aa\_type. Then, the standalone procedure print\_aa declares a formal parameter of type aa\_type. Next, the anonymous block declares a variable of type aa\_type, populates it, and passes it to the procedure print\_aa, which prints it.

CREATE OR REPLACE PACKAGE **aa\_pkg** AUTHID DEFINER IS

TYPE **aa\_type** IS TABLE OF INTEGER INDEX BY VARCHAR2(15);

END;

/

CREATE OR REPLACE PROCEDURE print\_aa (

**aa aa\_pkg.aa\_type**

) AUTHID DEFINER IS

i VARCHAR2(15);

BEGIN

i := aa.FIRST;

WHILE i IS NOT NULL LOOP

DBMS\_OUTPUT.PUT\_LINE (aa(i) || ' ' || i);

i := aa.NEXT(i);

END LOOP;

END;

/

DECLARE

**aa\_var aa\_pkg.aa\_type**;

BEGIN

aa\_var('zero') := 0;

aa\_var('one') := 1;

aa\_var('two') := 2;

**print\_aa(aa\_var)**;

END;

/

Result:

1 one

2 two

0 zero

### Package Body

If a package specification declares cursors or subprograms, then a package body is required; otherwise, it is optional. The package body and package specification must be in the same schema.

Every cursor or subprogram declaration in the package specification must have a corresponding definition in the package body. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

To create a package body, use the "CREATE PACKAGE BODY Statement".

The headings of the corresponding subprogram declaration and definition do not match word for word; therefore, PL/SQL raises an exception, even though employees.hire\_date%TYPE is DATE.

The cursors and subprograms declared in the package specification and defined in the package body are public items that can be referenced from outside the package. The package body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package.

Finally, the body can have an **initialization part**, whose statements initialize public variables and do other one-time setup steps. The initialization part runs only the first time the package is referenced. The initialization part can include an exception handler.

You can change the package body without changing the specification or the references to the public items.

***Example 10-3 Matching Package Specification and Body***

CREATE PACKAGE emp\_bonus AS

PROCEDURE calc\_bonus (date\_hired **employees.hire\_date%TYPE**);

END emp\_bonus;

/

CREATE PACKAGE BODY emp\_bonus AS

-- DATE does not match employees.hire\_date%TYPE

PROCEDURE calc\_bonus (date\_hired **DATE**) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE

('Employees hired on ' || date\_hired || ' get bonus.');

END;

END emp\_bonus;

/

Result:

Warning: Package Body created with compilation errors.

Show errors (in SQL\*Plus):

SHOW ERRORS

Result:

Errors for PACKAGE BODY EMP\_BONUS:

LINE/COL ERROR

-------- -----------------------------------------------------------------

2/13 PLS-00323: subprogram or cursor 'CALC\_BONUS' is declared in a

package specification and must be defined in the package body

Correct problem:

CREATE OR REPLACE PACKAGE BODY emp\_bonus AS

PROCEDURE calc\_bonus

(**date\_hired employees.hire\_date%TYPE**) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE

('Employees hired on ' || date\_hired || ' get bonus.');

END;

END emp\_bonus;

/

Result:

Package body created.

### Package Instantiation and Initialization

When a session references a package item, Oracle Database instantiates the package for that session. Every session that references a package has its own instantiation of that package.

When Oracle Database instantiates a package, it initializes it. Initialization includes whichever of the following are applicable:

* Assigning initial values to public constants
* Assigning initial values to public variables whose declarations specify them
* Executing the initialization part of the package body

#### Package State

The values of the variables, constants, and cursors that a package declares (in either its specification or body) comprise its **package state**.

If a PL/SQL package declares at least one variable, constant, or cursor, then the package is **stateful**; otherwise, it is **stateless**.

Each session that references a package item has its own instantiation of that package. If the package is state ful, the instantiation includes its state.

The package state persists for the life of a session, except in these situations:

* The package is SERIALLY\_REUSABLE.
* The package body is recompiled.

If the body of an instantiated, stateful package is recompiled (either explicitly, with the "ALTER PACKAGE Statement", or implicitly), the next invocation of a subprogram in the package causes Oracle Database to discard the existing package state and raise the exception ORA-04068.

After PL/SQL raises the exception, a reference to the package causes Oracle Database to re-instantiate the package, which re-initializes it. Therefore, previous changes to the package state are lost.

* Any of the session's instantiated packages are invalidated and revalidated.

All of a session's package instantiations (including package states) can be lost if any of the session's instantiated packages are invalidated and revalidated.

Oracle Database treats a package as stateless if its state is constant for the life of a session (or longer). This is the case for a package whose items are all compile-time constants.

A **compile-time constant** is a constant whose value the PL/SQL compiler can determine at compilation time. A constant whose initial value is a literal is always a compile-time constant. A constant whose initial value is not a literal, but which the optimizer reduces to a literal, is also a compile-time constant. Whether the PL/SQL optimizer can reduce a nonliteral expression to a literal depends on optimization level. Therefore, a package that is stateless when compiled at one optimization level might be stateful when compiled at a different optimization level.

### SERIALLY\_REUSABLE Packages

SERIALLY\_REUSABLE packages let you design applications that manage memory better for scalability.

If a package is not SERIALLY\_REUSABLE, its package state is stored in the user global area (UGA) for each user. Therefore, the amount of UGA memory needed increases linearly with the number of users, limiting scalability. The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days.

If a package is SERIALLY\_REUSABLE, its package state is stored in a work area in a small pool in the system global area (SGA). The package state persists only for the life of a server call. After the server call, the work area returns to the pool. If a subsequent server call references the package, then Oracle Database reuses an instantiation from the pool. Reusing an instantiation re-initializes it; therefore, changes made to the package state in previous server calls are invisible.