What is PL/SQL?

PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL). SQL is a powerful language for both querying and updating data in relational databases.

Oracle created PL/SQL that extends some limitations of SQL to provide a more comprehensive solution for building mission-critical applications running on Oracle database.

Why Use PL/SQL?

An application that uses Oracle Database is worthless unless only correct and complete data is persisted. The time-honored way to ensure this is to expose the database only via an interface that hides the implementation details -- the tables and the SQL statements that operate on these. This approach is generally called the thick database paradigm, because PL/SQL subprograms inside the database issue the SQL statements from code that implements the surrounding business logic; and because the data can be changed and viewed only through a PL/SQL interface.

PL/SQL – Overview

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. Following are notable facts about 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.
- Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

Features of PL/SQL

PL/SQL has the following features:

- 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.
- It supports developing web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages:

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. Dynamic SQL is SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for Developing Web Applications and Server Pages.

PL/SQL - Basic Syntax

S.N.	Sections & Description
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.

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Here is the basic structure of a PL/SQL block:

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
```

```
EXCEPTION
<exception handling>
END;
```

The 'Hello World' Example:

```
DECLARE
message  varchar2(20):='Hello, World!';
BEGIN
dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from SQL command line, you may need to type **/** at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces the following result:

```
HelloWorld


PL/SQL procedure successfully completed.
```

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

PL/SQL Program Units

A PL/SQL unit is any one of the following:

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

PL/SQL – Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is:

```
variable_name[CONSTANT]datatype[NOT NULL][:=| DEFAULT initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in last chapter. Some valid variable declarations along with their definition are shown below:

```
sales number(10,2);

pi CONSTANT double precision :=3.1415;

name varchar2(25);

address varchar2(100);
```

PL/SQL - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators:

- Arithmetic operators
- Relational operators

- Comparison operators
- Logical operators
- String operators

Declaring a Constant

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed. For example:

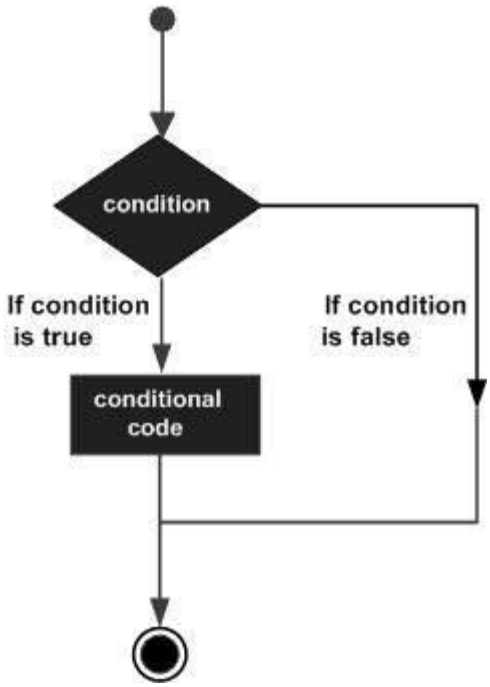
```
PI CONSTANT NUMBER :=3.141592654;
DECLARE
--constant declaration
pi constant number :=3.141592654;
--other declarations
radius number(5,2);
dia number(5,2);
circumference number(7,2);
area number (10,2);
BEGIN
-- processing
radius:=9.5;
dia:= radius *2;
circumference:=2.0* pi * radius;
area:= pi * radius * radius;
--output
dbms_output.put_line('Radius: '|| radius);
dbms_output.put_line('Diameter: '||dia);
dbms_output.put_line('Circumference: '|| circumference);
dbms_output.put_line('Area: '|| area);
END;
/
```

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

```
Radius:9.5

Diameter:19

Circumference:59.69

Area:283.53


Pl/SQL procedure successfully completed.
```

PL/SQL – Conditions

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



Statement	Description

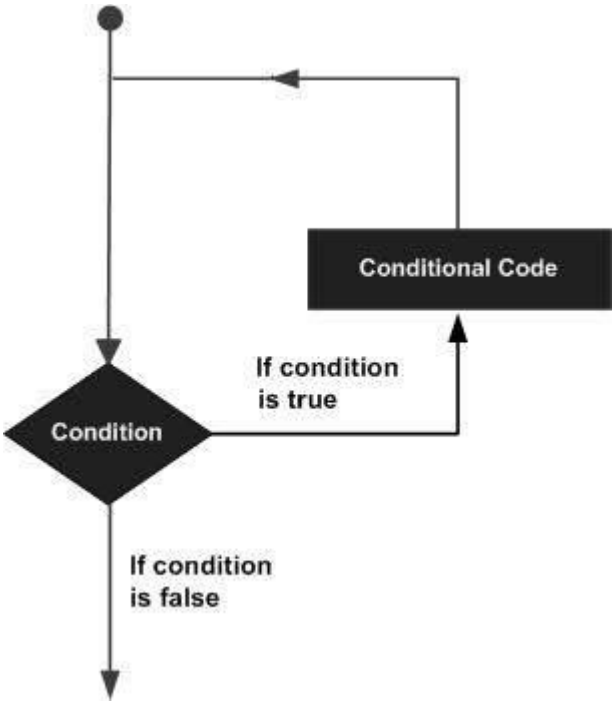
IF - THEN statement	The **IF statement** associates a condition with a sequence of statements enclosed by the keywords**THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
IF-THEN-ELSE statement	**IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL , then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
IF-THEN-ELSIF statement	It allows you to choose between several alternatives.
Case statement	Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
Searched CASE statement	The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
nested IF-THEN-ELSE	You can use one **IF-THEN** or **IF-THEN-ELSIF**statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

PL/SQL - Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Loop Type	Description
PL/SQL Basic LOOP	In this loop structure, sequence of statements is enclosed between the LOOP and END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
PL/SQL WHILE LOOP	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
PL/SQL FOR LOOP	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
Nested loops in PL/SQL	You can use one or more loop inside any another basic loop, while or for loop.

PL/SQL – 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 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.

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>

ENDprocedure_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.
```

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 Variableschapter:

```
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.
```

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.

```
DECLARE
c number(2);
BEGIN
c:=totalCustomers();
dbms_output.put_line('Total no. of Customers: '|| c);
```

```
END;
/
```

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

```
Totalno. 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):45


PL/SQL procedure successfully completed.
```

PL/SQL - Cursors

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.

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.

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|
```

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

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id,c_name,c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

```
CLOSE c_customers;
```

Example:

Following is a complete example to illustrate the concepts of explicit cursors:

```
DECLARE
c_idcustomers.id%type;
c_namecustomers.name%type;
c_addrcustomers.address%type;
   CURSOR c_customersis
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customersintoc_id,c_name,c_addr;
      EXIT WHEN c_customers%notfound;
dbms_output.put_line(c_id||' '||c_name||' '||c_addr);
END LOOP;
   CLOSE c_customers;
```

```
END;
/
```

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

```
1RameshAhmedabad
2KhilanDelhi
3kaushikKota
4ChaitaliMumbai
5HardikBhopal
6Komal MP


PL/SQL procedure successfully completed.
```

subprogram

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.

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.

Procedures

A **procedure** is a group of PL/SQL statements that you can call by name. A call specification (sometimes called call spec) declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells **Oracle** Database which Java method to invoke when a call is made.

Stored Procedures

What is a Stored Procedure?

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.
1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

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.

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 Variableschapter:

```
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;
/
```

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

```
Totalno. 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;
/
```

Packages

L/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a**private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS

   PROCEDURE find_sal(c_idcustomers.id%type);

ENDcust_sal;

/
```

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

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust_sal*package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in PL/SQL - Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

   PROCEDURE find_sal(c_idcustomers.id%TYPE) IS

c_salcustomers.salary%TYPE;

BEGIN

      SELECT salary INTO c_sal

      FROM customers

      WHERE id =c_id;
```

```
dbms_output.put_line('Salary: '||c_sal);
ENDfind_sal;
ENDcust_sal;
/
```

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

```
Package body created.
```

VIEW

An Oracle VIEW, in essence, is a virtual table that does not physically exist. Rather, it is created by a query joining one or more tables.

Create VIEW

Syntax

The syntax for the CREATE VIEW Statement in Oracle/PLSQL is:

```
CREATE VIEW view_name AS
  SELECT columns
  FROM tables
  [WHERE conditions];
```

view_name

The name of the Oracle VIEW that you wish to create.

WHERE conditions

Optional. The conditions that must be met for the records to be included in the VIEW.

Example

Here is an example of how to use the Oracle CREATE VIEW:

```
CREATE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
  INNER JOIN orders
  ON suppliers.supplier_id = orders.supplier_id
  WHERE suppliers.supplier_name = 'Microsoft';
```

This Oracle CREATE VIEW example would create a virtual table based on the result set of the SELECT statement. You can now query the Oracle VIEW as follows:

```
SELECT *
FROM sup_orders;
```

Update VIEW

You can modify the definition of an Oracle VIEW without dropping it by using the Oracle CREATE OR REPLACE VIEW Statement.

Syntax

The syntax for the CREATE OR REPLACE VIEW Statement in Oracle/PLSQL is:

```
CREATE OR REPLACE VIEW view_name AS
  SELECT columns
  FROM table
  WHERE conditions;
```

view_name

The name of the Oracle VIEW that you wish to create or replace.

Example

Here is an example of how you would use the Oracle CREATE OR REPLACE VIEW Statement:

```
CREATE or REPLACE VIEW sup_orders AS
```

```
SELECT suppliers.supplier_id, orders.quantity, orders.price
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id
WHERE suppliers.supplier_name = 'Apple';
```

This Oracle CREATE OR REPLACE VIEW example would update the definition of the Oracle VIEW called *sup_orders* without dropping it. If the Oracle VIEW did not yet exist, the VIEW would merely be created for the first time.

Drop VIEW

Once an Oracle VIEW has been created, you can drop it with the Oracle DROP VIEW Statement.

Syntax

The syntax for the DROP VIEW Statement in Oracle/PLSQL is:

```
DROP VIEW view_name;
```

view_name

The name of the view that you wish to drop.

Example

Here is an example of how to use the Oracle DROP VIEW Statement:

```
DROP VIEW sup_orders;
```

This Oracle DROP VIEW example would drop/delete the Oracle VIEW called *sup_orders*.

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
- Preventing invalid transactions

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;
```