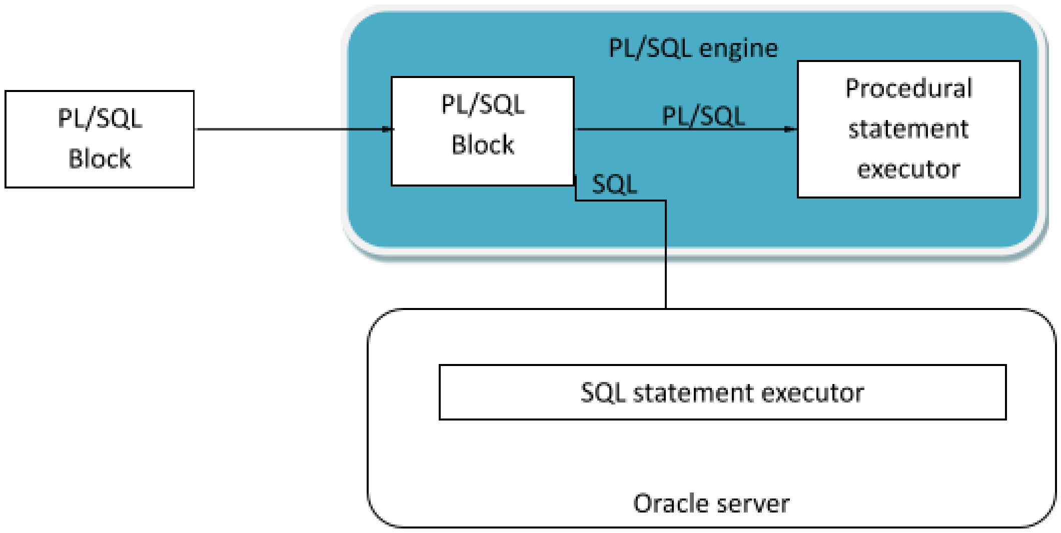
INTRODUCTION AND HISTORY OF PL/SQL: -

PL/SQL (Procedural Language / SQL) is a procedural extension of Oracle-SQL that offers language constructs similar to those in imperative programming language. PL/SQL allows users and designers to develop complex database applications that require the usage of control structures and procedural elements such as procedures, functions, and modules.

PL/SQL Environment:-



PL/SQL is not an Oracle product in its own right; it is a technology used by the Oracle server and by certain Oracle tools. Blocks of PL/SQL are passed to and processed by a PL/SQL engine which may reside within the tool or within the Oracle server. The engine that is used depends on where the PL/SQL block is being invoked from.

Advantages of PL/SQL:

- Allows to write methods for direct types.
- Supports the declaration and manipulation of Object types and collections.
- Allows the calling of external functions and procedures
- Comes with new libraries of built in packages
- Supporting for SQL : PL/SQL allows us to use all SQL data manipulation commands, transaction control commands, SQL functions (except group function), operators and pseudo column thus allowing us to manipulate data values in a table more flexible and effectively.
- Better Performance: Without PL/SQL, oracle must process SQL statements one at a time. With PL/SQL and entire block statements can be processed in a single command line statement. This reduces the time taken to communicate between the application and the Oracle server. Thus it helps in improving performance.
- Portability: Application written in PL/SQL is portable to any operating system or platform on which oracle version runs.
- Integration with Oracle: Both PL/SQL and Oracle have the foundations in SQL. PL/SQL supports all the SQL data types and integrates PL/SQL with Oracle data dictionary.

PL/SQL Block Structure:-

PL/SQL is a block-structured language, meaning that programs can be divided into logical blocks. Each block builds a program unit, and blocks can be nested. Blocks that build a procedure, a function or package must be named. A PL/SQL block consist of up to three declarative (optional), executable (required), and exception handling (optional).

[<Block header>]

DECLARE (Optional)

Constants, Variables, Cursors, user-defined exceptions

BEGIN (Mandatory)

- SQL statements
- PL/SQL statements

EXCEPTION (Optional) Actions to perform when errors occur

END; (Mandatory)

The block header specifies whether the PL/SQL block is a procedure, a function, or a package. If no header is specified, the block is said to be an anonymous PL/SQL block. Each PL/SQL block again builds a PL/SQL statement. Thus blocks can be nested like blocks in conventional programming languages. The scope of declared variables is analogous to the scope of variables in programming languages such as C or Pascal.

Declarations: Constants, variables, cursors, and exceptions used in a PL/SQL block must be declared in the declare section of that block. Variables and constants can be declared as follows:

<variable-name> [constant] <data type> [not null] [:= <expression>];

Example:

```
Declare
Hire-date date;                    /* implicit initialization with null*/
Job-title varchar2(80) := 'salesman';/* implicit initialization with null*/
Emp-found Boolean;
Salary-incr constant number(3,2) := 1.5; /* constant*/
…………..
Begin
……….
End;
```

Instead of specifying a data type, one can also refer to the data type of a table column (so-called anchored declaration). For example, the data type DEPT%ROWTYPE specifies a record suitable to store all attribute values of a complete row from the table DEPT. such records are typically used in combination with a cursor. A field in a record can be accessed using <record name>, <column name>, for example DEPT.Deptno%TYPE.

Executables: Executable block contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block.

Exception Handling: Exception Handling specifies the actions to perform when errors and abnormal conditions arise in the executable section.

PL/SQL Control Structures:

In addition to SQL commands, PL/SQL can also process data using flow of control statements. The flow of control statements can be classified under the following categories:

* Conditional Control
* Iterative Control

Conditional control: Sequence of statements can be executed based on some condition using the IF statement. There are three forms of IF statements, namely, IF…..THEN, IF…..THEN…..ELSE, IF….THEN…..ELSIF. The simple form of an if statement is the IF….THEN statement.

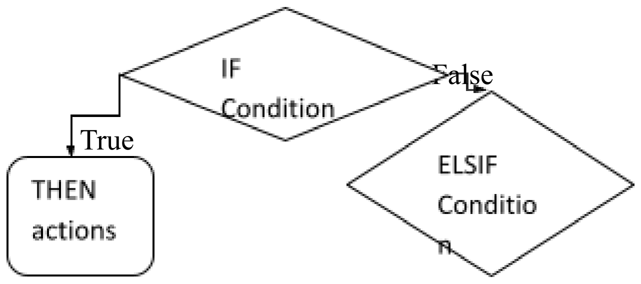
Syntax:

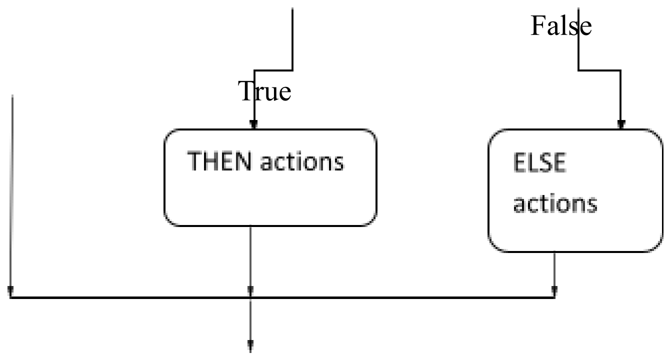
```
IF <condition> THEN

   Sequence of statements;

ELSIF

   Sequence of statements;

ELSE

  Sequence of statements;

END IF;
```

Example:

```
Declare
e_sal emp.sal%type;
Begin
  Select sal into e_sal from emp where ename='smith';
If e_sal>5000 then
  Update emp set sal=sal+(sal*10)/100 where ename='smith';
Else
  Update emp set sal=sal+(sal*5)/100 where ename='smith';
End if;
End;
```

IF – THEN – ELSIF Statement Execution Flow





Iterative Control

A sequence of statement can be executed any number of times using loop constructs. Loops can be broadly classified as:

- Simple Loop
- While Loop
- For Loop

Simple Loop – The keyword LOOP should be placed before the first statement in the sequence and the keyword END LOOP after the last statement in the sequence.

Syntax:

```
LOOP
   Sequence of statements;
   EXIT WHEN <condition>;
END LOOP;
```

Example:

```
Declare
   A number: =100;
Begin
   Loop
   A: =A+10;
   Dbms_output.put_line('--------------->' | | A);
   Exit when A>200;
   End loop;
```

```
End;
```

While Loop – The WHILE LOOP statements includes a condition associated with a sequence of statements. If the condition evaluates to true, then the sequence of statements will be executed, and again control resumes at the top of the loop. If the condition evaluates to false, then the loop is bypassed and the control passes to the next statement.

Syntax:

```
WHILE <condition> LOOP

  Sequence of statement;

END LOOP;
```

Example:

```
Declare

  I number(5) : =0;

Begin

  While I < = 100 loop

  I: = I+1;

  Dbmas_output.put_line('---------->' | | I);

  End loop;

End;
```

For Loop – The number of iterations for a while loop is unknown until the loop terminates, whereas the number of iteration in a FOR loop is known before the loop gets executed. The FOR loop statement specifies a range of integers, to execute the sequence of statement once for each integer.

Syntax:

```
FOR counter IN[REVERSE] lowerbound…upperbound loop

    Sequence of statements;

END LOOP;
```

By default, iteration proceeds from lowerbound to upperbound. If we use the optional keyword REVERSE, then, iteration proceeds downwards from upperbound to lowerbound.

Example:

```
Declare

  X number(5);

Begin
```

```
  For x in 65…90 loop

  Dbms_output.put_line('-------->' | | chr(x));

  End loop;

End;
```

Cursors and Cursor Operations

A cursor is an SQL object is associated with a specific table expression. The RDBMS uses cursors to navigate through a set of rows returned by an embedded SQL SELECT statement. A cursor can be compared to a pointer. The programmer declares a cursor and defines the SQL statement for the cursor. After that we can use the cursor like a sequential file. The cursor is opened, rows are fetched from the cursor is closed.

Cursor operations: - The four operations that must be performed for the successful working of the cursor are:

DECLARE: - This statement defines the cursor, gives it a name to it and assigns an SQL statement to it.

OPEN: - This makes the cursor ready for row retrieval. OPEN is an executable statement. It reads the SQL search fields, executes the SQL statement and sometimes builds the result table.

FETCH: - This statement returns data from the result table one row at a time to the host variable. If the result table is not built at the OPEN time it is built during FETCH.

CLOSE: - Releases all resources used by the cursor.

When the cursors are used to process multiple rows, the cursor is declared and opened and the fetch statement is coded in a loop that reads and processes each row. At the end of the processing, that is when there are no more rows to be fetched. The cursor is then closed.

Guidelines for coding CURSORS:

When coding embedded SQL statement using cursors, the following guidelines will improve the performance and maintainability of the program:

- Declare as many cursors as needed. There is no limit on the number of cursors that can be used in program.
- Open cursors before fetching.
- Initialize host variables before opening the cursor.
- Explicitly close the cursors. Even though the RDBMS closes all open cursors at the end of the program explicitly close the cursors using the close statement, otherwise we will be holding resources, which will affect the performance.
- Avoid using certain cursors for modification. A cursor cannot be used for updates or deletes if the DECLARE CURSOR statement includes a UNION, DISTINCT, group by, order by, having clauses, joins, sub queries or tables in read only mode etc.
- Include only the columns that are being updated.
- Use where current of to delete single rows using a cursor. Use where current of clause on update and delete statement that are meant to modify only a single row. Failure in doing this will result in the modification or delete of all the rows that are being processed.

Cursor positions: - When a cursor is open, designates a certain collection of rows and certain ordering for that collection. It's also designates a certain position with respect to that ordering. The possible positions are:

- ✔ On some specific row ('ON' state)
- ✔ Before some specific row ('BEFORE' state)
- ✔ After some specific row ('AFTER' state)

Cursor state is affected by a variety of operations open positions the cursor before the first row. FETCH NEXT positions the cursor on the next row or if there is no next row, after the last row. FETCH PRIOR positions the cursor on the prior row or before the first row, if there is no prior row. There are other FETCH formats like FIRST, LAST, ABSOLUTE n, RELATIVE n etc. if the cursor is on some row and that row is deleted by the cursor, the cursor is positioned before the next row or after the last row.

The 'ABSOLUTE n' refers to the nth row in the ordered table that the cursor is associated with. A negative value for 'n' means n rows, backward from the end of the table. 'RELATIVE n' refers to the nth row in the table relative to the row on which the cursor is currently positioned.

All cursors are in the closed state at transaction initiation and are forced state at transaction termination.

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 attributes such as **%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 −

S.No	Attribute & Description
1	**%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.
2	**%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.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%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 will update the table and increase the 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 the 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 includes the following steps −

Declaring the cursor for initializing the memory
Opening the cursor for allocating the memory
Fetching the cursor for retrieving the data
Closing the cursor to release the 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 the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the 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 the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_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 the SQL prompt, it produces the following result –

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP


PL/SQL procedure successfully completed.
```

Triggers: (What is Trigger? Explain with syntax? And how to replace, drop the triggers)

A trigger defines an action the database should take when some database related event occurs. For any event that causes a change in the contents users to make the database respond actively to changes within or outside the database.

Syntax:

CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE | AFTER] [DELETE | INSERT |UPDATE [OF column_name] ON [table name] [FOR EACH ROW] [WHEN <condition>] [PL/SQL block];

CREATE TRIGGER emp_trig – The trigger is created and named.

BEFORE update on emp – This trigger applies on the ‘emp’ table. It will be executed before update transaction have been committed to that database.

FOR EACH ROW – because for each row clause is used, the trigger will apply to each row in the transaction. If the clause is not used, then the trigger will execute only at the statement level.

WHEN (condition) - The when clause add further restrictions to the triggering condition.

BEGIN – Makes the beginning of the block

END – Marks the end of the PL/SQL block.

Replacing Triggers

The body of a trigger cannot be altered. Only the status can be altered. To alter the body the trigger must be re-created or replaced. When replacing a trigger, we should use CREATE OR REPLACE TRIGGER command. Using OR REPLACE option will maintain any grants made for the original version of the trigger. The alternative solution is dropping and re-creating the trigger, but it will drop all the grants made for the alternative.

Dropping Triggers

Triggers may be dropped using DROP TRIGGER command.

Syntax: DROP TRIGGER trigger_name;

Example: DROP TRIGGER emp_be_up_trig;

Types of Triggers: (What are the types of Triggers? Explain)

A trigger's type is defined by the type of triggering transaction and by the level at which the trigger is executed. The following describe the classifications

- ✔ Row level Triggers
- ✔ Statement level Triggers
- ✔ BEFORE and AFTER Triggers

Row level Triggers:-Row level trigger, trigger once for each row in a transaction. These types of trigger are very useful in cases like audit trails, where we want to track the modification made to the data in a table. Different RDBMS have implemented the row level triggers in different ways.

Statement level Triggers: - Statement level triggers execute once for each transaction. Statement level triggers are not often used for data related activities. They are normally used to enforce additional security measures on types if transactions that may default type of triggers created using create trigger command.

BEFORE and AFTER Triggers: - Since triggers occur because of events they may set to occur immediately before or after those events. Since the events that execute triggers are database transactions, triggers can be executed immediately before or after insert, update and delete.

Within a trigger, we will be able to reference the old and new values involved in the transaction. The access required for the old and new data may determine which type of trigger need. OLD refers to the data, as it existed prior to the transaction. Updates and deletes usually reference old values. New values are the data values that the transaction creates. They are referred by the keyword new.

Enabling and disabling Triggers

A trigger is enable when it is created. However there are situations in which we may wish to disable a trigger. The two most common reasons involve data loads. During large data loads, we may wish to disable a trigger that would execute during the load. Disabling the trigger during data loads will dramatically improve the performance of the data load. Once the data has been loaded, we will need to manually perform the data manipulation that the trigger would have performed had it been enabled during the data load.

The second data load related reason for disabling a trigger occurs when a data load fails and has to be performed a second time. In such a case, it is likely that the data load practically subsequent data load, the same trigger will be fired twice for the same transaction. Depending on the nature of the transaction and triggers, this may not be desirable. If the trigger was enabled during the failed load, then it manually perform the data manipulation that the trigger would have performed.

To enable a trigger, use the ALTER TRIGGER command with the ENABLE ALL triggers clause. We cannot enable specific triggers with this command we can disable a trigger using the same AFTER TRIGGER command with any of the clause DISABLE or DISABLE ALL Triggers.

Syntax:

- ALTER TRIGGER trigger_name ENABLE | DISABLE;
- ALTER TABLE table_name ENABLE | DISABLE ALL TRIGGERS;

Advantages and Limitations of Triggers

Trigger can be used to make the RDBMS take some action, when a database related event occurs. The following are the advantages and limitations of triggers.

The most important advantage of a trigger is that business rules can be stored in the database and enforced consistently with each and every update operation. This can be dramatically reducing the complexity of the application programs that access the database. But triggers have some disadvantages also.

When the business rules are moved into the database with the help of triggers, setting up the database becomes a more complex task. Also, with triggers, the rules are hidden in the database and the application programs, which appear deceptively simple and straight forward, can cause and enormous amount of database activity. The programmer no longer has total control over what happens to the database, because a program initiated database action may cause many other actions.

Combining Triggers types and setting inserted values:

Triggers for multiple INSERT, UPDATE and DELETE commands on a table can be combined into a single trigger, provided they are all at the same level (row or statement level).

Example: CREATE OR REPLACE TRIGGER emp_trig BEFORE INSERT OR UPDATE OF sal ON emp FOR EACH ROW

BEGIN

IF INSERTING THEN

Statements;

ELSE

Statement;

END IF;

END;

The above example shows a trigger that is executed whenever an insert or on update occurs. The update portion of the trigger only occurs when the sal column is updated and if clause is used in within PL/SQL block to determine which of the two commands executed the trigger.

Setting Inserted Values:

We may use triggers to set values during inserts and updates. For example, we may have partially de-normalized our table to include derived data. Sometimes the derived columns may not be in synchronize with the base table column. To avoid this synchronization problem, we may use a database trigger. Put a before insert and before update trigger on the table. They will act at the row level.

Procedures (what is procedure? Explain)

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

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

CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter 1[mode1] datatype1, parameter2 [mode2] datatype2 …)] IS | AS PL/SQL Block;

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

PL/SQL blocks start with either BEGIN or the declaration of local variables and end with END or END procedure_name.

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

We cannot restrict the size of the data type in the parameters. We can transfer values to and from the calling environment through parameters. Select one of the three modes for each parameter: IN, OUT, or IN OUT.

<u>Example</u>

CREATE OR REPLACE PROCEDURE raise_salary (p_id IN emp.empno%TYPE) IS

BEGIN

UPDATE emp SET sal=sal*1.10 WHERE empno=p_id;

END raise_salary;

Calling procedure

SQL> execute raise_salary(7566);

Removing Procedures

When a stored procedure is no longer required, we can use a SQL statement to drop it. To remove a server-side procedure by using iSQL*Plus, execute the SQL command DROP PROCEDURE.

- DROP PROCEDURE procedure_name;

Functions (What is Function? Explain with syntax)

A function is named PL/SQL block that can be accept parameters and be invoked. Generally use a function to computer a value. Functions and procedures are structured alike. A function must return a value to the calling environment; a function has a header, a declarative part, an executable part, and an optional exception-handling part. A function must be a RETURN clause in the header and at least one RETURN statement in the executable section.

CREATE [OR REPLACE] [FUNCTION function-name [(parameter1 [mode1] datatype1, parameter2 [mode2] datatype2……..)] RETURN datatype IS|AS PL/SQL Block;

A function is a PL/SQL block that returns a value. Create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

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

PL/SQL blocks start with either END or END function_name. There must be at least one RETURN (expression) statement.

The RETURN data type must not include a size specification.

Benefits of Stored Procedures and Functions

In addition to modularizing application development, stored procedures and functions have the following benefits:

- ❖ Improved performance
  - A void reparsing for multiple users by exploiting the shared SQL area
  - Avoid PL/SQL parsing at runtime by parsing at compile time
  - Reduce the number of calls to the database and decrease network traffic by bundling commands.
- ❖ Easy maintenance
  - Modify routines online without interfering with other users
  - Modify one routines to affect multiple applications
  - Modify one routine to eliminate duplicate testing
- ❖ Improved data security and integrity
  - Control indirect access to database objects from non privileged users with security privileges
- ❖ Improved code clarity: By using appropriate identifier names to describe the actions of the routine, we reduce the need for comments and enhance clarity.

Packages (What is package? How to develop a package? Explain)

Packages bundle related PL/SQL types, items, and subprograms into one container. A package usually has a specification and a body, stored separately in the database. The specification is the interface to our applications. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use.

Creating Package Specification

In the package specification, we declare public variables, public procedures, and public functions. The public procedures or functions are routines that can be invoked repeatedly by other constructs in the same package or from outside the package.

Syntax:

CREATE [OR REPLACE] PACKAGE package_name IS|AS public type and item declarations subprogram specifications

END package_name;

<u>Example:</u>

CREATE OR REPLACE PACKAGE comm_package IS g_comm NUMBER :=0.10;

PROCEDURE reset_comm (p_comm IN NUMBER);

END comm._package;

Creating the Package Body

To create packages, define all public and private constructs within the package body. Specify the REPLACE option when the package body already exists. The order in which subprograms are defined within the package body is important.

<u>Syntax:</u>

CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS

Private type and item declarations

Subprogram bodies

END package_name;

<u>Example:</u>

CREATE OR REPLACE PACKAGE BODY comm_package IS PROCEDURE reset_comm(p_comm IN NUMBER) IS

BEGIN

dbms_output.put_line(p_comm);

END reset_comm;

END comm._package;

Invoking Package Constructs

Invoke a package procedure from SQLPlus.

EXECUTE comm._package.reset_comm(0.15);

Removing a Package

When a package is no longer required, we can use a SQL statement in SQLPlus to drop it. A package has two parts, so we can drop the whole package or just the package body and retain the package specification.

To remove the package specification and the body, use the following syntax

DROP PACKAGE package_name;

DROP PACKAGE BODY package_name;