# Unit 2 Overview to PL/SQL

## 2.1 Introduction

PL/SQL is an extension of Structured Query Language (SQL) that is used in Oracle. Unlike SQL, PL/SQL allows the programmer to write code in a procedural format. Full form of PL/SQL is "Procedural Language extensions to SQL".

It combines the data manipulation power of SQL with the processing power of procedural language to create super powerful SQL queries.

PL/SQL means instructing the compiler 'what to do' through SQL and 'how to do' through its procedural way.

Similar to other database languages, it gives more control to the programmers by the use of loops, conditions and object-oriented concepts.

**Disadvantages of SQL:**

- SQL doesn't provide the programmers with a technique of condition checking, looping and branching.
- SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
- SQL has no facility of error checking during manipulation of data.

**Features of PL/SQL:**

1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

**Differences between SQL and PL/SQL:**

| **SQL** | **PL/SQL** |
|---|---|
| SQL is a single query that is used to perform DML and DDL operations. | PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc. |
| It is declarative, that defines what needs to be done, rather than how things need to be done. | PL/SQL is procedural that defines how the things needs to be done. |
| Execute as a single statement. | Execute as a whole block. |
| Mainly used to manipulate data. | Mainly used to create an application. |
| Cannot contain PL/SQL code in it. | It is an extension of SQL, so it can contain SQL inside it. |

**Architecture of PL/SQL**

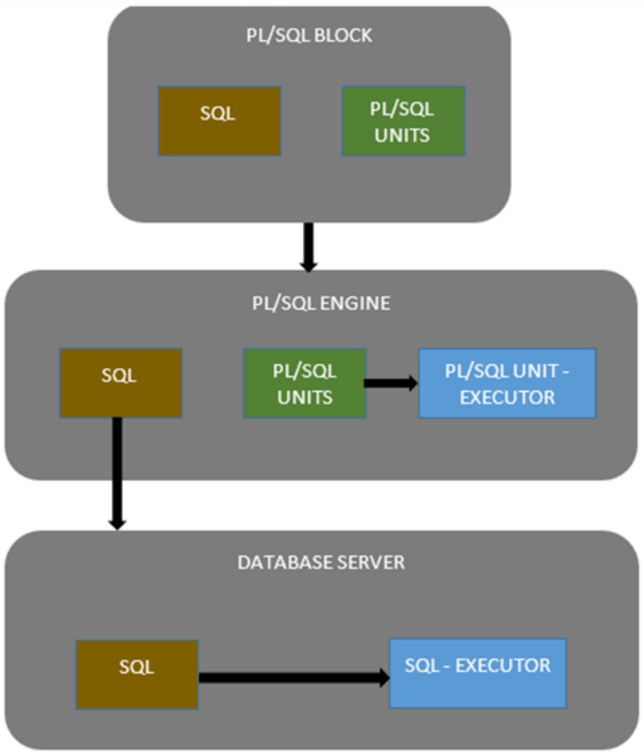
The PL/SQL architecture mainly consists of following three components:

1. PL/SQL block
2. PL/SQL Engine
3. Database Server

**PL/SQL block:**

- This is the component which has the actual PL/SQL code.
- This consists of different sections to divide the code logically (declarative section for declaring purpose, execution section for processing statements, exception handling section for handling errors)

- It also contains the SQL instruction that used to interact with the database server.
- All the PL/SQL units are treated as PL/SQL blocks, and this is the starting stage of the architecture which serves as the primary input.
- Following are the different type of PL/SQL units.
    - Anonymous Block
    - Function
    - Library
    - Procedure
    - Package Body
    - Package Specification
    - Trigger
    - Type
    - Type Body



PL/SQL Architecture Diagram

**PL/SQL Engine**

PL/SQL engine is the component where the actual processing of the codes takes place. PL/SQL engine separates PL/SQL units and SQL part in the input (as shown in the image below). The separated PL/SQL units will be handled by the PL/SQL engine itself. The SQL part will be sent to database server where the actual interaction with database takes place. It can be installed in both database server and in the application server.

**Database Server:**

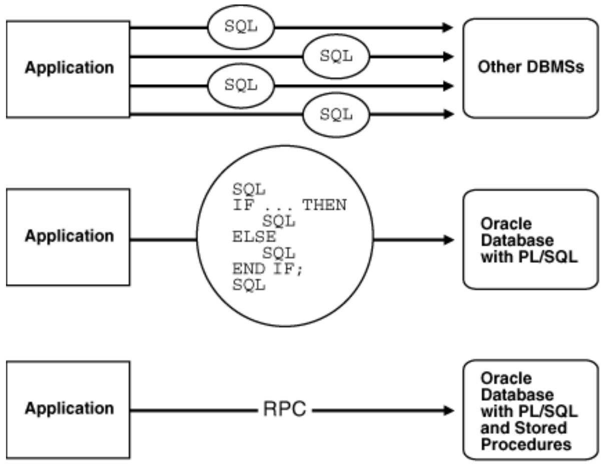
- This is the most important component of Pl/SQL unit which stores the data.
- The PL/SQL engine uses the SQL from PL/SQL units to interact with the database server.
- It consists of SQL executor which parses the input SQL statements and execute the same.

## 2.2 Advantage of Using PL/SQL

1. **Better performance**

   With PL/SQL, an entire block of statements can be sent to the database at one time. This can drastically reduce network traffic between the database and an application. As Figure 1-1 shows, you can use PL/SQL blocks and subprograms (procedures and functions) to group SQL statements before sending them to the database for execution. PL/SQL also has language features to further speed up SQL statements that are issued inside a loop.

   PL/SQL stored subprograms are compiled once and stored in executable form, so subprogram calls are efficient. Because stored subprograms execute in the database server, a single call over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and call overhead.



2. **High Productivity**

   PL/SQL lets you write very compact code for manipulating data. In the same way that scripting languages such as PERL can read, transform, and write data from files, PL/SQL can query, transform, and update data in a database. PL/SQL saves time on design and debugging by offering a full range of software-engineering features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

   PL/SQL extends tools such as Oracle Forms. With PL/SQL in these tools, you can use familiar language constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger, instead of multiple trigger steps, macros, or user exits. PL/SQL is the same in all environments. After you learn PL/SQL with one Oracle tool, you can transfer your knowledge to other tools.

3. **Tight integration with SQL**

   SQL has become the standard database language because it is flexible, powerful, and easy to learn. A few English-like statements such as SELECT, INSERT, UPDATE, and DELETE make it easy to manipulate the data stored in a relational database.

   PL/SQL is tightly integrated with SQL. With PL/SQL, you can use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, and operators.

   PL/SQL fully supports SQL data types. You need not convert between PL/SQL and SQL data types. For example, if your PL/SQL program retrieves a value from a database column of the SQL type VARCHAR2, it can store that value in a PL/SQL variable of the type VARCHAR2. Special PL/SQL language features let you work with table columns and rows without specifying the data types, saving on maintenance work when the table definitions change.

4. **Full Portability**

   Applications written in PL/SQL can run on any operating system and platform where the database runs. With PL/SQL, you can write portable program libraries and reuse them in different environments.

5. **Tight Security**

   PL/SQL stored subprograms move application code from the client to the server, where you can protect it from tampering, hide the internal details, and restrict who has access. For example, you can grant users access to a subprogram that updates a table, but not grant them access to the table itself or to the text of the UPDATE statement. Triggers written in PL/SQL can control or record changes to data, making sure that all changes obey your business rules.

6. **Support Object Oriented Programming concepts.**

   Object types are an ideal object-oriented modeling tool, which you can use to reduce the cost and time required to build complex applications. Besides enabling you to create software components that are modular, maintainable, and reusable, object types allow different teams of programmers to develop software components concurrently.

   By encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods. Also, object types hide implementation details, so that you can change the details without affecting client programs.

   In addition, object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. This direct mapping helps your programs better reflect the world they are trying to simulate.

## 2.3 Understanding the main feature of PL/SQL

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages. When a problem can be solved using SQL, you can issue SQL statements from your PL/SQL programs, without learning new APIs. Like other procedural programming languages, PL/SQL lets you declare constants and variables, control program flow, define subprograms, and trap run-time errors. You can break complex problems into easily understandable subprograms, which you can reuse in multiple applications.

### 2.3.1 PL/SQL Block Structure

**PL/SQL block**

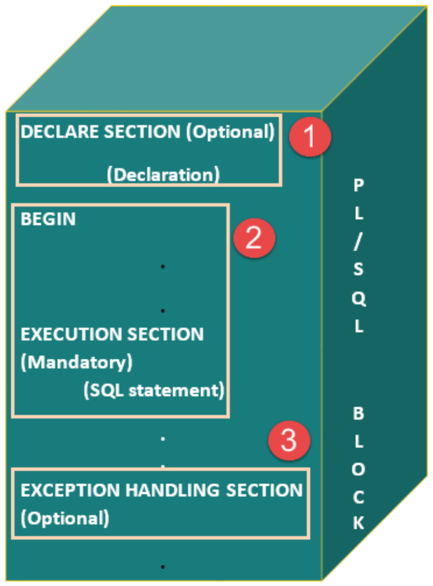
In PL/SQL, the code is not executed in single line format, but it is always executed by grouping the code into a single element called Blocks. Blocks contain both PL/SQL as well as SQL instruction. All these instruction will be executed as a whole rather than executing a single instruction at a time.

**Block Structure**

PL/SQL blocks have a pre-defined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks.

1. Declaration section
2. Execution section
3. Exception-Handling section

The below picture illustrates the different PL/SQL block and their section order.



**Declaration Section**

This is the first section of the PL/SQL blocks. This section is an optional part. This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared. Below are few more characteristics of this part.

- This particular section is optional and can be skipped if no declarations are needed.
- This should be the first section in a PL/SQL block, if present.
- This section starts with the keyword 'DECLARE' for triggers and anonymous block. For other subprograms, this keyword will not be present. Instead, the part after the subprogram name definition marks the declaration section.
- This section should always be followed by execution section.

**Execution Section**

Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it. Below are few more characteristics of this part.

- This can contain both PL/SQL code and SQL code.
- This can contain one or many blocks inside it as a nested block.
- This section starts with the keyword 'BEGIN'.
- This section should be followed either by 'END' or Exception-Handling section (if present)

**Exception-Handling Section:**

The exception is unavoidable in the program which occurs at run-time and to handle this Oracle has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

- This is the section where the exception raised in the execution block is handled.
- This section is the last part of the PL/SQL block.
- Control from this section can never return to the execution block.
- This section starts with the keyword 'EXCEPTION'.
- This section should always be followed by the keyword 'END'.

The Keyword 'END' marks the end of PL/SQL block.

**PL/SQL Block Syntax**

Below is the syntax of the PL/SQL block structure.

```
DECLARE --optional
   <declarations>

BEGIN   --mandatory
   <executable statements. At least one executable statement is mandatory>

EXCEPTION --optional
   <exception handles>

END;   --mandatory
/
```

**Types of PL/SQL block**

PL/SQL blocks are of mainly two types.

1. Anonymous blocks
2. Named Blocks

**Anonymous blocks:**

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them. They need to be created and used in the same session because they will not be stored in the server as database objects.

Since they need not store in the database, they need no compilation steps. They are written and executed directly, and compilation and execution happen in a single process.

Below are few more characteristics of Anonymous blocks.

- These blocks don't have any reference name specified for them.
- These blocks start with the keyword 'DECLARE' or 'BEGIN'.
- Since these blocks do not have any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
- They can call the other named blocks, but call to anonymous block is not possible as it is not having any reference.
- It can have nested block in it which can be named or anonymous. It can also be nested in any blocks.
- These blocks can have all three sections of the block, in which execution section is mandatory, the other two sections are optional.

**Named blocks:**

Named blocks have a specific and unique name for them. They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present on the server. The compilation process for named blocks happens separately while creating them as a database objects.

Below are few more characteristics of Named blocks.

- These blocks can be called from other blocks.
- The block structure is same as an anonymous block, except it will never start with the keyword 'DECLARE'. Instead, it will start with the keyword 'CREATE' which instruct the compiler to create it as a database object.
- These blocks can be nested within other blocks. It can also contain nested blocks.
- Named blocks are basically of two types:
    1. Procedure
    2. Function

**2.3.2 PL/SQL Execution Environment**

The execution environment of PL/SQL block is shown below.

The PL/SQL engine resides in the Oracle engine, the Oracle engine can process not only single SQL statements but also entire PL/SQL blocks.

These blocks are sent to the PL/SQL engine, where procedural statements are executed and SQL statements are sent to the SQL executor in the Oracle engine. Since the PL/SQL engine resides in the Oracle engine, this is an efficient and fast operation.

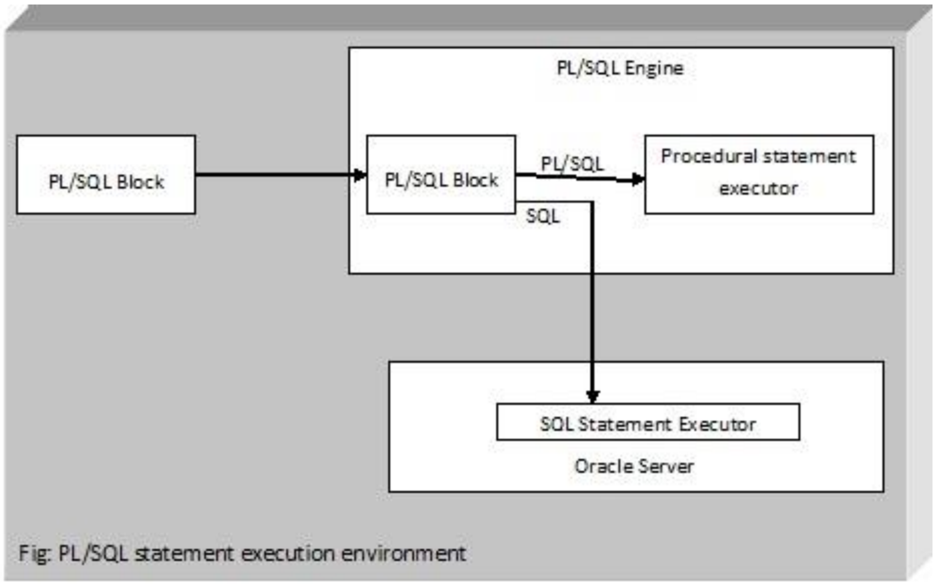


Fig: PL/SQL statement execution environment

The call to the Oracle engine needs to be made only once to execute any number of SQL statements, if these SQL statements are bundled inside a PL/SQL block.

Since the oracle engine is called only once for each block, the speed of SQL statement execution is vastly enhanced, when compared to the Oracle engine being called once for each SQL sentence.

### 2.3.3 Data Types

PL/SQL datatypes are grouped into composite, LOB, reference, and scalar type categories.

- A composite type has internal components that can be manipulated individually, such as the elements of an array, record, or table.
- A LOB type holds values, called lob locators that specify the location of large objects, such as text blocks or graphic images that are stored separately from other database data. LOB types include BFILE, BLOB, CLOB, and NCLOB.
- A reference type holds values, called pointers that designate other program items. These types include REF CURSORS and REFs to object types.
- A scalar type has no internal components. It holds a single value, such as a number or character string. The scalar types fall into four families, which store number, character, Boolean, and date/time data. The scalar families with their datatypes are:

**1. PL/SQL Number Types**

BINARY_DOUBLE, BINARY_FLOAT, BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NATURAL, NATURALN, NUMBER, NUMERIC, PLS_INTEGER, POSITIVE, POSITIVEN, REAL, SIGNTYPE, SMALLINT

**2. PL/SQL Character and String Types and PL/SQL National Character Types**

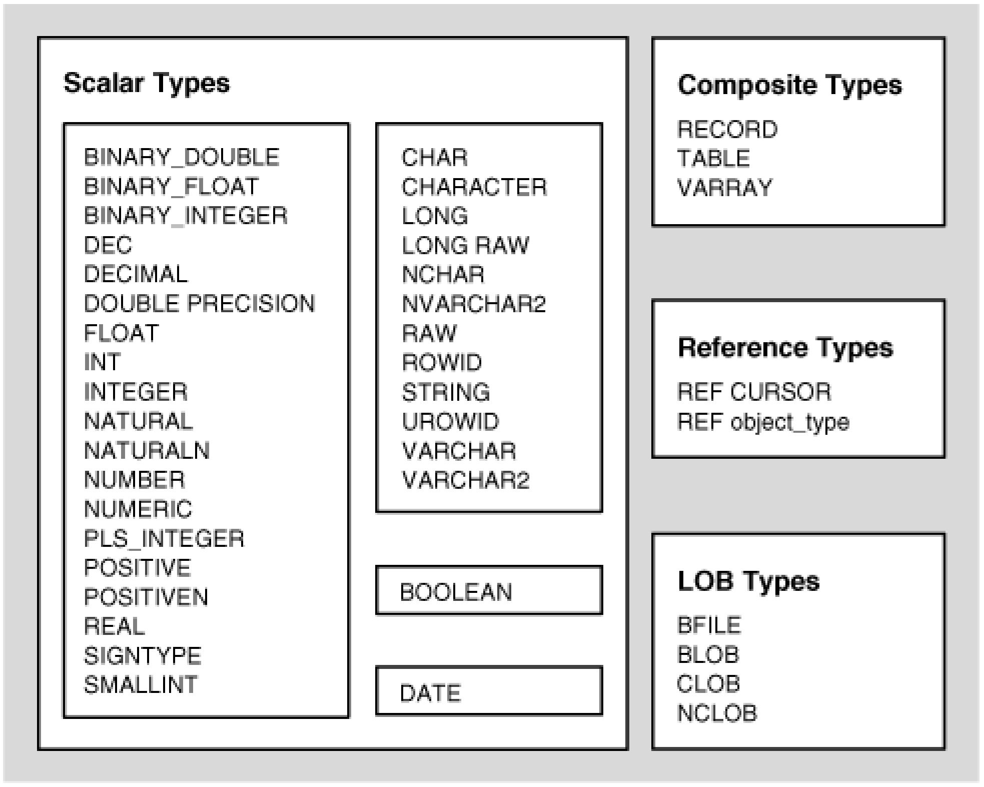
CHAR, CHARACTER, LONG, LONG RAW, NCHAR, NVARCHAR2, RAW, ROWID, STRING, UROWID, VARCHAR, VARCHAR2 Note that the LONG and LONG RAW datatypes are supported only for backward compatibility Information.

**3. PL/SQL Boolean Types**

BOOLEAN

**4. PL/SQL Date, Time, and Interval Types**

DATE, TIMESTAMP, TIMESTAMP WITH TIMEZONE, TIMESTAMP WITH LOCAL TIMEZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND



**PL/SQL Subtypes**

Each **PL/SQL base type** specifies a set of values and a set of operations applicable to items of that type. Subtypes specify the same set of operations as their base type, but only a subset of its values. A subtype does not introduce a new type; rather, it places an optional constraint on its base type.

**Subtypes** can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables. PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

```
SUBTYPE CHARACTER IS CHAR;

SUBTYPE INTEGER IS NUMBER(38,0); -- allows only whole numbers
```

The subtype CHARACTER specifies the same set of values as its base type CHAR, so CHARACTER is an unconstrained subtype. But, the subtype INTEGER specifies only a subset of the values of its base type NUMBER, so INTEGER is a constrained subtype.

- **Defining Subtypes**

You can define your own subtypes in the declarative part of any PL/SQL block, subprogram, or package using the syntax

```
SUBTYPE subtype_name IS base_type[(constraint)] [NOT NULL];
```

where subtype_name is a type specifier used in subsequent declarations, base_type is any scalar or user-defined PL/SQL datatype, and constraint applies only to base types that can specify precision and scale or a maximum size.

- **Using Subtypes**

After you define a subtype, you can declare items of that type. In the following example, you declare a variable of type Counter. Notice how the subtype name indicates the intended use of the variable.

```
DECLARE

SUBTYPE Counter IS NATURAL;

rows Counter;
```

## 2.3.4 Variables and Constants

### Variables

Variable is the basic identifier which is used more frequently and the most important of all. Variable is nothing but a placeholder where the user can store the value. This variable needs to be associated with some valid PL/SQL datatype before using them. The datatype will define the storage and processing method for these variables.

#### 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 the 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);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example −

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

## Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

- The **DEFAULT** keyword
- The **assignment** operator

For example −

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables −

```
DECLARE

  a integer := 10;
```

```
   b integer := 20;

   c integer;

   f real;

BEGIN

   c := a + b;

   dbms_output.put_line('Value of c: ' || c);

   f := 70.0/3.0;

   dbms_output.put_line('Value of f: ' || f);

END;

/
```

When the above code is executed, it produces the following result −

```
Value of c: 30
Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.
```

## Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form −

```
DECLARE

   -- Global variables

   num1 number := 95;

   num2 number := 85;
```

```
BEGIN

   dbms_output.put_line('Outer Variable num1: ' || num1);

   dbms_output.put_line('Outer Variable num2: ' || num2);

   DECLARE

      -- Local variables

      num1 number := 195;

      num2 number := 185;

   BEGIN

      dbms_output.put_line('Inner Variable num1: ' || num1);

      dbms_output.put_line('Inner Variable num2: ' || num2);

   END;

END;

/
```

When the above code is executed, it produces the following result −

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

**Constants**

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the **NOT NULL constraint**.

**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;

/
```

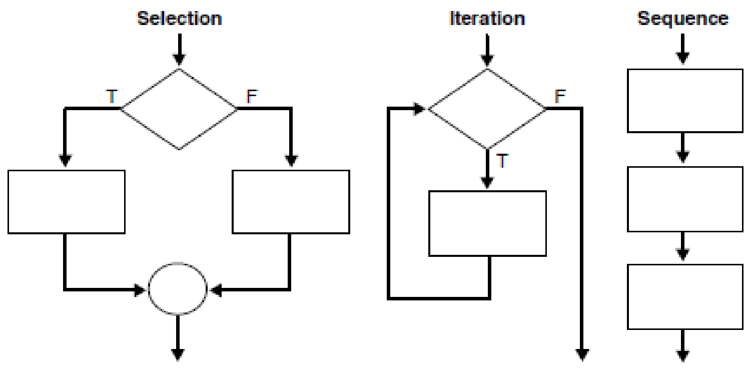
When the above code is executed at the SQL prompt, it produces the following result −

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.
```

## 2.4 Condition Control in PL/SQL

Procedural computer programs use the basic control structures.



- The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a BOOLEAN value (TRUE or FALSE).
- The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.
- The sequence structure simply executes a sequence of statements in the order in which they occur.

### Using the IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF) as illustrated in Example.

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

### Example Simple IF-THEN Statement

```
SQL> DECLARE
  2    sales  NUMBER(8,2) := 10100;
  3    quota  NUMBER(8,2) := 10000;
  4    bonus  NUMBER(6,2);
  5    emp_id NUMBER(6) := 120;
  6  BEGIN
```

```
 7   IF sales > (quota + 200) THEN

 8     bonus := (sales - quota)/4;

 9

10      UPDATE employees SET salary =

11        salary + bonus

12          WHERE employee_id = emp_id;

13   END IF;

14  END;

15  /

PL/SQL procedure successfully completed.
```

**Using the IF-THEN-ELSE Statement**

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as shown in Example.

The statements in the ELSE clause are executed only if the condition is FALSE or NULL. The IF-THEN-ELSE statement ensures that one or the other sequence of statements is executed.

**Example Using a Simple IF-THEN-ELSE Statement**

```
SQL> DECLARE

 2   sales  NUMBER(8,2) := 12100;

 3   quota  NUMBER(8,2) := 10000;

 4   bonus  NUMBER(6,2);

 5   emp_id NUMBER(6) := 120;

 6  BEGIN

 7   IF sales > (quota + 200) THEN

 8    bonus := (sales - quota)/4;

 9   ELSE
 10     bonus := 50;
 11   END IF;
 12
 13   UPDATE employees
 14     SET salary = salary + bonus
 15       WHERE employee_id = emp_id;
 16  END;
 17  /
PL/SQL procedure successfully completed
```

IF statements can be nested. Example shows nested IF-THEN-ELSE statements.

**Example Nested IF-THEN-ELSE Statements**

```
SQL> DECLARE
  2   sales  NUMBER(8,2) := 12100;
  3   quota  NUMBER(8,2) := 10000;
  4   bonus  NUMBER(6,2);
  5   emp_id NUMBER(6)   := 120;
  6  BEGIN
  7   IF sales > (quota + 200) THEN
  8     bonus := (sales - quota)/4;
  9   ELSE
 10     IF sales > quota THEN
 11       bonus := 50;
 12     ELSE
 13       bonus := 0;
```

```
    END IF;
  END IF;

  UPDATE employees
    SET salary = salary + bonus
      WHERE employee_id = emp_id;
END;
/
PL/SQL procedure successfully completed.
```

**Using the IF-THEN-ELSIF Statement**

Sometimes you want to choose between several alternatives. You can use the keyword ELSIF (not ELSIF or ELSE IF) to introduce additional conditions, as shown in Example.

If the first condition is FALSE or NULL, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is TRUE, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or NULL, the sequence in the ELSE clause is executed, as shown in Example.

**Example Using the IF-THEN-ELSIF Statement**

```
SQL> DECLARE
  sales  NUMBER(8,2) := 20000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6)   := 120;
BEGIN
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
  9    bonus := 500;
 10   ELSE
 11     bonus := 100;
 12   END IF;
 13
 14   UPDATE employees
 15     SET salary = salary + bonus
 16       WHERE employee_id = emp_id;
 17  END;
 18  /
```

If the value of sales is larger than 50000, the first and second conditions are TRUE. Nevertheless, bonus is assigned the proper value of 1500 because the second condition is never tested. When the first condition is TRUE, its associated statement is executed and control passes to the UPDATEstatement.

**Using the Simple 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.

You can rewrite the code in Example using the CASE statement, as shown in Example.

**Example Simple CASE Statement**

```
SQL> DECLARE
  2   grade CHAR(1);
  3  BEGIN
  4   grade := 'B';
  5
  6   CASE grade
  7     WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
```

```
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
Very Good

PL/SQL procedure successfully completed.
```

The CASE statement is more readable and more efficient. When possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.

The CASE statement begins with the keyword CASE. The keyword is followed by a selector, which is the variable grade in the last example. The selector expression can be arbitrarily complex. For example, it can contain function calls. Usually, however, it consists of a single variable. The selector expression is evaluated only once. The value it yields can have any PL/SQL data type other than BLOB, BFILE, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed. For example, in the last example, if gradeequals 'C', the program outputs 'Good'. Execution never falls through; if any WHEN clause is executed, control passes to the next statement.

The ELSE clause works similarly to the ELSE clause in an IF statement. In the last example, if the grade is not one of the choices covered by a WHENclause, the ELSE clause is selected, and the phrase 'No such grade' is output. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

There is always a default action, even when you omit the ELSE clause. If the CASE statement does not match any of the WHEN clauses and you omit the ELSE clause, PL/SQL raises the predefined exception CASE_NOT_FOUND.

The keywords END CASE terminate the CASE statement. These two keywords must be separated by a space

## 2.5 Iterative Control in PL/SQL

### 2.5.1 FOR LOOP

This Oracle tutorial explains how to use the **FOR LOOP** in Oracle with syntax and examples.

**Description**

In Oracle, the FOR LOOP allows you to execute code repeatedly for a fixed number of times.

**Syntax**

The syntax for the FOR Loop in Oracle/PLSQL is:

```
FOR loop_counter IN [REVERSE] lowest_number..highest_number
                    LOOP
                      {...statements...}
                    END LOOP;
```

**Parameters or Arguments**

**loop_counter**

The loop counter variable.

**REVERSE**

Optional. If specified, the loop counter will count in reverse.

**lowest_number**

The starting value for *loop_counter*.

**highest_number**

The ending value for *loop_counter*.

**statements**

The statements of code to execute each pass through the loop.

**Note**

- You would use a FOR LOOP when you want to execute the loop body a fixed number of times.
- If REVERSE is specified, then the *highest_number* will be the starting value for *loop_counter* and *lowest_number* will be the ending value for *loop_counter*.

Example

```
SQL> BEGIN
  2    FOR i IN 1..3 LOOP
  3      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  4    END LOOP;
  5  END;
  6  /
1
2
3
PL/SQL procedure successfully completed.
```

By default, iteration proceeds upward from the lower bound to the higher bound. If you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. You still write the range bounds in ascending (not descending) order.

**Example Reverse FOR-LOOP Statement**

```
SQL> BEGIN
  2    FOR i IN REVERSE 1..3 LOOP
  3      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  4    END LOOP;
  5  END;
  6  /
3
2
1
PL/SQL procedure successfully completed.
```

**WHILE LOOP**

This Oracle tutorial explains how to use the **WHILE LOOP** in Oracle with syntax and examples.

**Description**

In Oracle, you use a WHILE LOOP when you are not sure how many times you will execute the loop body and the loop body may not execute even once.

**Syntax**

The syntax for the WHILE Loop in Oracle/PLSQL is:

```
WHILE condition

LOOP

   {...statements...}

END LOOP;
```

**Parameters or Arguments**

**condition**

The condition is tested each pass through the loop. If *condition* evaluates to TRUE, the loop body is executed. If *condition*evaluates to FALSE, the loop is terminated.

**statements**

The statements of code to execute each pass through the loop.

**Note**

- You would use a WHILE LOOP statement when you are unsure of how many times you want the loop body to execute.
- Since the WHILE condition is evaluated before entering the loop, it is possible that the loop body may **not** execute even once.

**Example**

```
DECLARE
a NUMBER :=1;
BEGIN
dbms_output.put_line('Program started');
WHILE (a < 5)
LOOP
dbms_output.put_line(a);
a:=a+l;
END LOOP;
dbms_output.put_line(‘Program completed' );
```

```
END:
/
```

**Exit Statement**

This Oracle tutorial explains how to use the **EXIT statement** in Oracle with syntax and examples

**Description**

In Oracle, the EXIT statement is most commonly used to terminate LOOP statements.

**Syntax**

The syntax for the EXIT statement in Oracle/PLSQL is:

EXIT [WHEN *boolean_condition*];

**Parameters or Arguments**

**boolean_condition**

Optional. It is the condition to terminate the LOOP.

**Example EXIT Statement**

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop:  x = ' || TO_CHAR(x));

    x := x + 1;

    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
```

```
 14    -- After EXIT, control resumes here

 15

 16    DBMS_OUTPUT.PUT_LINE

 17      (' After loop:  x = ' || TO_CHAR(x));

 18  END;

 19  /

Inside loop:  x = 0

Inside loop:  x = 1

Inside loop:  x = 2

Inside loop:  x = 3

After loop:  x = 4

PL/SQL procedure successfully completed.
```

**Example Using an EXIT-WHEN Statement**

```
SQL> DECLARE

  2    x NUMBER := 0;

  3  BEGIN

  4    LOOP

  5      DBMS_OUTPUT.PUT_LINE

  6        ('Inside loop:  x = ' || TO_CHAR(x));

  7

  8      x := x + 1;

  9

 10      EXIT WHEN x > 3;

 11    END LOOP;
```

```
-- After EXIT statement, control resumes here
DBMS_OUTPUT.PUT_LINE
  ('After loop:  x = ' || TO_CHAR(x));
END;
/
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
PL/SQL procedure successfully completed.
```

**GOTO Statement**

This Oracle tutorial explains how to use the **GOTO statement** in Oracle with syntax and examples.

**Description**

The GOTO statement causes the code to branch to the label after the GOTO statement.

**Syntax**

The syntax for the GOTO statement in Oracle/PLSQL consists of two parts - the GOTO statement and the Label Declaration:

GOTO statement

The GOTO statement consists of the GOTO keyword, followed by a *label_name*.

```
GOTO label_name;
```

**Label Declaration**

The Label Declaration consists of the *label_name* encapsulated in << >>, followed by at least one statement to execute.

```
<<label_name>>
    {...statements...}
```

**Note**

- *label_name* must be unique within the scope of the code.
- There must be at least one statement to execute after the Label Declaration.

**Example**

Let's look at an Oracle example that uses the GOTO statement.

```
CREATE OR REPLACE Function FindCourse
   ( name_in IN varchar2 )
   RETURN number
IS
   cnumber number;

   CURSOR c1
   IS
     SELECT MAX(course_number)
     FROM courses_tbl
     WHERE course_name = name_in;

BEGIN


   open c1;
   fetch c1 into cnumber;


   IF c1%notfound then
      GOTO default_number;


   ELSE
      GOTO increment_number;
```

```
  END IF;


<<default_number>>
  cnumber := 0;


<<increment_number>>
  cnumber := cnumber + 1;


  close c1;


RETURN cnumber;


END;
```

In this GOTO example, we have created two GOTO statements. The first one is called *default_number* and the second one is called *increment_number*.