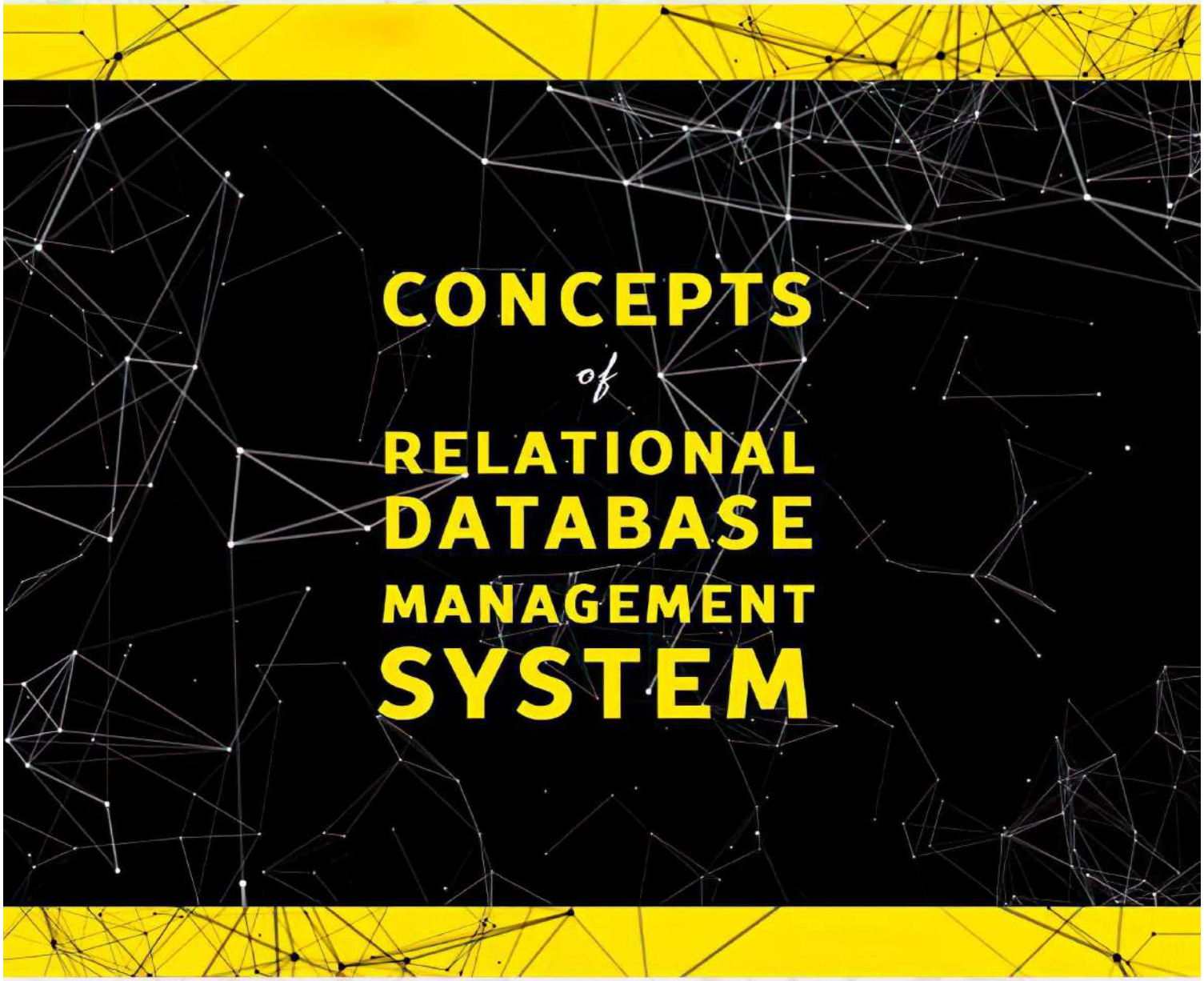
Jump2Learn
PUBLICATION

www.jump2learn.com



CONCEPTS of RELATIONAL DATABASE MANAGEMENT SYSTEM

Jump2Learn - The Online Learning Place

Dr. RajeshKumar R. Savaliya | Ms. Sonal B. Shah | Mr. Vaibhav D. Desai

Unit - 3

PL/SQL and Conditional Statements

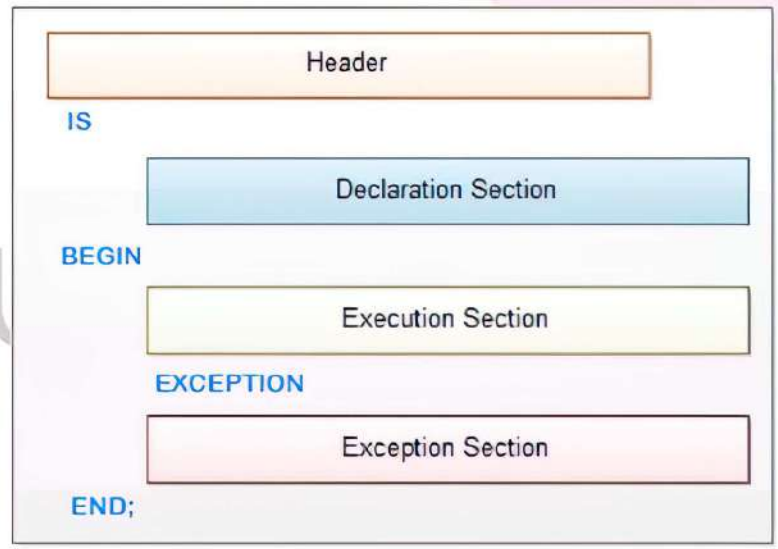
3.1 Introduction to PL/SQL (Definition & Block Structure)

PL/SQL program units organize the code into blocks. A block without a name is known as an anonymous block. The anonymous block is the simplest unit in PL/SQL. It is called anonymous block because it is not saved in the Oracle database.

An anonymous block is an only one-time use and useful in certain situations such as creating test units. The following illustrates anonymous block syntax:

```
[DECLARE]
  Declaration statements;
BEGIN
  Execution statements;
[EXCEPTION]
   Exception handling statements;
END;
/
```

Let's examine the PL/SQL block structure in greater detail.



PL/SQL Block Structure

The anonymous block has three basic sections that are the declaration, execution, and exception handling. Only the execution section is mandatory and the others are optional.

- The declaration section allows you to define data types, structures, and variables. You often declare variables in the declaration section by giving them names, data types, and initial values.
- The execution section is required in a block structure and it must have at least one statement. The execution section is the place where you put the execution code or business logic code. You can use both procedural and SQL statements inside the execution section.
- The exception handling section is starting with the EXCEPTION keyword. The exception section is the place that you put the code to handle exceptions. You can either catch or handle exceptions in the exception section.

PL/SQL block structure example:

Let's take a look at the simplest PL/SQL block that does nothing.

```
BEGIN
    NULL;
END;
```

If you execute the above anonymous block in *SQL*Plus* you will see that it issues a message saying:

PL/SQL procedure successfully completed.

Because the NULL statement does nothing.

To display database's output on the screen, you need to:

- First, use the SET SERVEROUTPUT ON command to instruct SQL*Plus to echo database's output after executing the PL/SQL block. The SET SERVEROUTPUT ONis SQL*Plus command, which is not related to PL/SQL.
- Second, use the DBMS_OUTPUT.PUT_LINE procedure to output a string on the screen.

The following example displays a message Hello PL/SQL on a screen using SQL*Plus:

```
SET SERVEROUTPUT ON SIZE 1000000
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello PL/SQL');
END; /
```

3.2 Variables, Constants and Data Type

PL/SQL - Data Types

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB** data types in this chapter. The other two data types will be covered in other chapters.

S.No	Category & Description
1	**Scalar** Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN.**
2	**Large Object (LOB)** Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	**Composite** Data items that have internal components that can be accessed individually. For example, collections and records.
4	**Reference** Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories –

S.No	Date Type & Description
1	**Numeric** Numeric values on which arithmetic operations are performed.
2	**Character** Alphanumeric values that represent single characters or strings of characters.
3	**Boolean** Logical values on which logical operations are performed.
4	**Datetime** Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types –

S.No	Data Type & Description
1	**PLS_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	**BINARY_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	**BINARY_FLOAT** Single-precision IEEE 754-format floating-point number
4	**BINARY_DOUBLE** Double-precision IEEE 754-format floating-point number
5	**NUMBER(prec, scale)** Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
6	**DEC(prec, scale)** ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	**DECIMAL(prec, scale)** IBM specific fixed-point type with maximum precision of 38 decimal digits
8	**NUMERIC(pre, secale)** Floating type with maximum precision of 38 decimal digits
9	**DOUBLE PRECISION** ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

10	**FLOAT** ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	**INT** ANSI specific integer type with maximum precision of 38 decimal digits
12	**INTEGER** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	**SMALLINT** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	**REAL** Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration –

```
DECLARE
  num1 INTEGER;
  num2 REAL;
  num3 DOUBLE PRECISION;
BEGIN
  null;
END;
/
```

When the above code is compiled and executed, it produces the following result –

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

PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types –

S.No	Data Type & Description
1	**CHAR** Fixed-length character string with maximum size of 32,767 bytes
2	**VARCHAR2** Variable-length character string with maximum size of 32,767 bytes
3	**RAW** Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	**NCHAR** Fixed-length national character string with maximum size of 32,767 bytes
5	**NVARCHAR2** Variable-length national character string with maximum size of 32,767 bytes
6	**LONG** Variable-length character string with maximum size of 32,760 bytes
7	**LONG RAW** Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	**ROWID** Physical row identifier, the address of a row in an ordinary table
9	**UROWID** Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in –

- SQL statements
- Built-in SQL functions (such as **TO_CHAR**)
- PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

The **DATE** datatype is used to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field –

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59

SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types –

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

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);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype –

```
DECLARE
  SUBTYPE name IS char(20);
  SUBTYPE message IS varchar2(100);
  salutation name;
  greetings message;
BEGIN
  salutation := 'Reader ';
  greetings := 'Welcome to the World of PL/SQL';
  dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```


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

```
Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.
```

NULLs in PL/SQL

PL/SQL NULL values represent **missing** or **unknown data** and they are not an integer, a character, or any other specific data type. Note that **NULL** is not the same as an empty data string or the null character value **'\0'**. A null can be assigned but it cannot be equated with anything, including itself.

3.3 Assigning Values to Variables

Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS –

```
CREATE TABLE CUSTOMERS
(
  ID   INT NOT NULL,
  NAME VARCHAR (20) NOT NULL,
  AGE INT NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2),
  PRIMARY KEY (ID)
);
Table Created
```

Let us now insert some values in the table –

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Aarohi', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Ishani', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Ronit', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Jiya', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Krish', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Aditi', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL –

```
DECLARE
  c_id customers.id%type := 1;
  c_name  customers.name%type;
  c_addr customers.address%type;
```

```
  c_sal  customers.salary%type;
BEGIN
  SELECT name, address, salary INTO c_name, c_addr, c_sal
  FROM customers
  WHERE id = c_id;
  dbms_output.put_line
  ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

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

Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully

3.4 User Defined Record

A **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book, such as Title, Author, Subject, Book ID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records –

- Table-based
- Cursor-based records
- User-defined records

User-Defined Records

PL/SQL provides a user-defined record type that allows you to define the different record structures. These records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining a Record

The record type is defined as –

```
TYPE
type_name IS RECORD
 ( field_name1  datatype1  [NOT NULL]  [:= DEFAULT EXPRESSION],
  field_name2   datatype2   [NOT NULL]  [:= DEFAULT EXPRESSION],
  ...
  field_nameN  datatypeN  [NOT NULL]  [:= DEFAULT EXPRESSION);
record-name  type_name;
```

The Book record is declared in the following way –

```
DECLARE
TYPE books IS RECORD
(title  varchar(50),
  author  varchar(50),
  subject varchar(100),
  book_id   number);
book1 books;
book2 books;
```

Accessing Fields

To access any field of a record, we use the dot **(.)** operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is an example to explain the usage of record –

```
DECLARE
  type books is record
    (title varchar(50),
    author varchar(50),
    subject varchar(100),
    book_id number);
  book1 books;
  book2 books;
BEGIN
  -- Book 1 specification
  book1.title  := 'C Programming';
  book1.author := 'Nuha Ali ';
  book1.subject := 'C Programming Tutorial';
  book1.book_id := 6495407;
 -- Book 2 specification
```

```
  book2.title := 'Telecom Billing';
  book2.author := 'Zara Ali';
  book2.subject := 'Telecom Billing Tutorial';
  book2.book_id := 6495700;

 -- Print book 1 record
  dbms_output.put_line('Book 1 title : '|| book1.title);
  dbms_output.put_line('Book 1 author : '|| book1.author);
  dbms_output.put_line('Book 1 subject : '|| book1.subject);
  dbms_output.put_line('Book 1 book_id : ' || book1.book_id);

  -- Print book 2 record
  dbms_output.put_line('Book 2 title : '|| book2.title);
  dbms_output.put_line('Book 2 author : '|| book2.author);
  dbms_output.put_line('Book 2 subject : '|| book2.subject);
  dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

PL/SQL procedure successfully completed.
```

Records as Subprogram Parameters

You can pass a record as a subprogram parameter just as you pass any other variable. You can also access the record fields in the same way as you accessed in the above example –

```
DECLARE
  type books is record
    (title  varchar(50),
    author  varchar(50),
    subject varchar(100),
    book_id   number);
  book1 books;
  book2 books;
PROCEDURE printbook (book books) IS
BEGIN
  dbms_output.put_line ('Book  title :  ' || book.title);
  dbms_output.put_line('Book  author : ' || book.author);
  dbms_output.put_line( 'Book  subject : ' || book.subject);
  dbms_output.put_line( 'Book book_id : ' || book.book_id);
END;

BEGIN
  -- Book 1 specification
  book1.title  := 'C Programming';
  book1.author := 'Nuha Ali ';
  book1.subject := 'C Programming Tutorial';
  book1.book_id := 6495407;

  -- Book 2 specification
  book2.title := 'Telecom Billing';
  book2.author := 'Zara Ali';
  book2.subject := 'Telecom Billing Tutorial';
  book2.book_id := 6495700;

  -- Use procedure to print book info
  printbook(book1);
  printbook(book2);
END;
/
```

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

```
Book  title : C Programming
```

Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700

PL/SQL procedure successfully completed.

3.5 Conditional Statements

3.5.1 IF...THEN Statement

The IF-THEN statement is mainly used to execute a particular section of codes only when the condition is satisfied.

The condition should yield Boolean (True/False). It is a basic conditional statement which will allow the ORACLE to execute/skip a particular piece of code based on the pre-defined conditions.

Syntax for IF THEN Statements:

IF <condition: returns Boolean>

THEN

-executed only if the condition returns TRUE

<action_block>

END if;

- In the above syntax, keyword 'IF' will be followed by a condition which evaluates to 'TRUE'/'FALSE'.
- The control will execute the <action_block> only if the condition returns <TRUE>.
- In the case of condition evaluates to <FALSE> then, SQL will skip the <action_block>, and it will start executing the code next to 'END IF' block.

Note: Whenever condition evaluated to 'NULL', then SQL will treat 'NULL' as 'FALSE'.

Example 1: In this example, we are going to print a message when the number is greater than 100. For that, we will execute the following code

To print a message when a number has value more than 100, we execute the following code.

```
DECLARE
a NUMBER :=10;
BEGIN
dbms_output.put_line('Program started.' );
IF( a > 100 ) THEN
dbms_output.put_line('a is greater than 100');
END IF;
dbms_output.put_line('Program completed.');
END;
/
```

Code Output:

```
Program started.
Program completed.
```

3.5.2 IF..Else statement

- The IF-THEN-ELSE statement is mainly used to select between two alternatives based on the condition.
- Below is the syntax representation of IF-THEN-ELSE statement.

Syntax for IF-THEN-ELSE Statements:

IF <condition: returns Boolean>
THEN
    -executed only if the condition returns TRUE
    <action_block1>
ELSE
    -execute if the condition failed (returns FALSE)
    <action_block2>
END if;

- In the above syntax, keyword 'IF' will be followed by a condition which evaluates to 'TRUE'/'FALSE'.
- The control will execute the <action_block1> only if the condition returns <TRUE>.
- In case of condition evaluates to <FALSE> then, SQL will execute <action_block2>.
- In any case, one of the two action blocks will be executed.

Note: Whenever condition evaluates to 'NULL', then SQL will treat 'NULL' as 'FALSE'.

Example 1: In this example, we are going to print message whether the given number is odd or even.

```
DECLARE
a NUMBER:=11;
BEGIN
dbms_output.put_line ('Program started');
IF( mod(a,2)=0) THEN
dbms_output.put_line('a is even number' );
ELSE
dbms_output.put_line('a is odd number1);
END IF;
dbms_output.put_line ('Program completed.');
END;
/
```

Code Output:

```
Program started.
a is odd number
Program completed.
```

3.5.3 Multiple conditions

IF-THEN-ELSIF Statement

- The IF-THEN-ELSIF statement is mainly used where one alternative should be chosen from a set of alternatives, where each alternative has its own conditions to be satisfied.
- The first conditions that return <TRUE> will be executed, and the remaining conditions will be skipped.
- The IF-THEN-ELSIF statement may contain 'ELSE' block in it. This 'ELSE' block will be executed if none of the conditions is satisfied.

Note: ELSE block is optional in this conditional statement. If there is no ELSE block, and none of the condition satisfied, then the controller will skip all the action block and start executing the remaining part of the code.

Syntax for IF-THEN-ELSIF Statements:

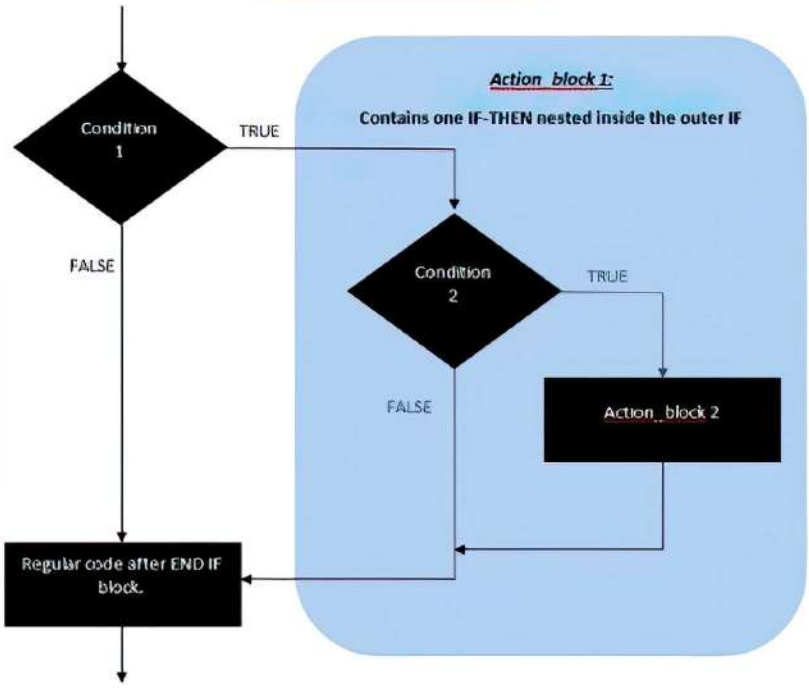
```
IF <condition1: returns Boolean>
THEN
  -executed only if the condition returns TRUE <action_block1>
ELSIF <condition2 returns Boolean> <action_block2>
ELSIF <condition3:returns Boolean> <action_block3>
ELSE —optional <action_block_else>
END if;
```

- In the above syntax, the control will execute the <action_block1> only if the condition1 returns <TRUE>.
- If condition1 is not satisfied, then the controller will check for condition2.
- The controller will exit from the IF-statement in the following two cases.
  - When the controller found any condition that returns <TRUE>. In this case, the corresponding action_block will be executed and the controller will exit this IF-statement block and will start executing the remaining code.
  - When none of the conditions satisfied, the then controller will execute ELSE block if present, then will exit from the IF-statement.

Note: Whenever condition evaluates to 'NULL', then SQL will treat 'NULL' as 'FALSE'.

3.5.4 Nested IF statement

- The NESTED-IF statement is basically allowed programmers to place one or more 'IF' condition inside another 'IF' condition's <action_block> other than normal statements.
- Each 'IF' condition should have a separate 'END IF' statement which marks the end-of-scope of that particular <action_block>.
- The 'IF' statement will consider the nearest 'END IF' statement as an endpoint for that particular condition.
- The pictorial representation for NESTED-IF is shown below diagram.



Syntax:

```
IF <condition1: returns Boolean>
THEN
   --executed only if the condition returns TRUE
   <action_block1 starts>
      IF <condition2: returns Boolean>
      THEN
         <action_block2>
      END IF;    --END IF corresponds to condition2
   <action_block1 ends>
END IF;    --END IF corresponds to condition1
```

Outer IF condition

Inner IF condition

```
IF <conditional: returns Boolean>
THEN
        —executed only if the condition returns TRUE
        <action block1 starts>
      IF <condition2: returns Boolean>
      THEN
        <action_block2>
      END IF; —END IF corresponds to condition2
<action_block1 ends>
END IF; —END IF corresponds to condition1
```

Syntax Explanation:

- In the above syntax, the outer IF contains one more IF statement in its action block.
- The condition1 returns <TRUE>, then control will be executing <action_block1> and checks the condition2.
- If condition2 also returns <TRUE>, then <action_block2> will also be executed.
- In case of condition2 evaluates to <FALSE> then, SQL will skip the <action_block2>.

Here we are going to see an example of Nested If –

Example of Nested- If Statement: Greatest of three number

In this example, we are going to print the greatest of three numbers by using Nested-If statement. The numbers will be assigned in the declare part, as you can see in the code below, i.e Number= 10,15 and 20 and the maximum number will be fetched using nested-if statements.

```
DECLARE
a NUMBER :=10;
b NUMBER :=15;
c NUMBER :=20;
BEGIN
dbms_output.put_line('Program started.' );
IF( a > b)THEN
/*Nested-if I */
        dbms_output.put_line('Checking Nested-IF 1');
        IF( a > c ) THEN
        dbms_output.put_line('A is greatest');
        ELSE
```

```
        dbms_output.put_line('C is greatest');
        END IF;
ELSE
/*Nested-if2 */
        dbms_output.put_line('Checking Nested-IF 2' );
        IF( b > c ) THEN
        dbms_output.put_line('B is greatest' );
        ELSE
        dbms_output.put_line('C is greatest' );
        END IF;
END IF;
dbms_output.put_line('Program completed.' );
END;
/
```

Output of code:

```
Program started.
Checking Nested-IF 2
C is greatest
Program completed.
```

3.5.5 CASE statement

A CASE statement is similar to IF-THEN-ELSIF statement that selects one alternative based on the condition from the available options.

- CASE statement uses "selector" rather than a Boolean expression to choose the sequence.
- The value of the expression in the CASE statement will be treated as a selector.
- The expression could be of any type (arithmetic, variables, etc.)
- Each alternative is assigned with a certain pre-defined value (selector), and the alternative with selector value that matches the conditional expression value will get executed.
- Unlike IF-THEN-ELSIF, the CASE statement can also be used in SQL statements.
- ELSE block in CASE statement holds the sequence that needs to be executed when none of the alternatives got selected.

Syntax:

```
CASE (expression)
 WHEN <value1> THEN action_block1;
 WHEN <value2> THEN action_block2;
 WHEN <value3> THEN action_block3;
 ELSE action_block_default;
END CASE;
```

- In the above syntax, the expression will return a value that could be of any type (variable, number, etc.).
- Each 'WHEN' clause is treated as an alternatives which have <value> and <action_block>.
- The 'WHEN' clause which matches the value as that of the expression will be selected, and the corresponding <action_block> will be executed.
- 'ELSE' block is optional which hold the <action_block_default> that needs to be executed when none of the alternatives match the expression value.
- The 'END' marks the end of the CASE statement, and it is a mandatory part of the CASE.

Example 1: Arithmetic Calculation using Case

In this example, we are going to do arithmetic calculation between two numbers 55 and 5.

```
DECLARE
a NUMBER :=55;
b NUMBER :=5;
arth_operation VARCHAR2(20) :='MULTIPLY';
BEGIN
dbms_output.put_line('Program started.' );
CASE (arth_operation)
WHEN 'ADD' THEN dbms_output.put_line('Addition of the numbers are: '|| a+b );
WHEN 'SUBTRACT' THEN dbms_output.put_line('Subtraction of the numbers are: '||a-b );
WHEN 'MULTIPLY' THEN dbms_output.put_line('Multiplication of the numbers are: '|| a*b
```

```
);
WHEN 'DIVIDE' THEN dbms_output.put_line('Division of the numbers are:'|| a/b);
ELSE dbms_output.put_line('No operation action defined. Invalid operation');
END CASE;
dbms_output.put_line('Program completed.' );
END;
/
```

Code Output:

```
Program started.
Multiplication of the numbers are: 275
Program completed.
```

SEARCHED CASE Statement

The SEARCHED CASE statement is similar to the CASE statement, rather than using the selector to select the alternative, SEARCHED CASE will directly have the expression defined in the WHEN clause.

- The first WHEN clause that satisfies the condition will be executed, and the controller will skip the remaining alternatives.

Syntax:

```
CASE
WHEN <expression1> THEN action_block1;
WHEN <expression2> THEN action_block2;
WHEN <expression3> THEN action_block3;
ELSE action_block_default;
END CASE;
```


- In the above syntax, each WHEN clause has the separate <expression> and <action_block>.

- The WHEN clause for which the expression returns TRUE will be executed.
- 'ELSE' block is optional which hold the <action_block_default> that needs to be executed when none of the alternatives satisfies.
- The 'END' marks the end of the CASE statement and, it is a mandatory part of CASE.

Example 1: Arithmetic Calculation using Searched Case

In this example, we are going to do arithmetic calculation between two numbers 55 and 5.

```
DECLARE a NUMBER :=55;
b NUMBER :=5;
arth_operation VARCHAR2(20) :='DIVIDE';
BEGIN
dbms_output.put_line('Program started.' );
CASE
WHEN arth_operation = 'ADD'
THEN dbms_output.put_line('Addition of the numbers are: '||a+b );
WHEN arth_operation = 'SUBTRACT'
THEN dbms_output.put_line('Subtraction of the numbers are: '|| a-b);
WHEN arth_operation = 'MULTIPLY'
THEN dbms_output.put_line('Multiplication of the numbers are: '|| a*b );
WHEN arth_operation = 'DIVIDE'
THEN dbms_output.put_line('Division of the numbers are: '|| a/b ):
ELSE dbms_output.put_line('No operation action defined. Invalid operation');
END CASE;
dbms_output.put_line('Program completed.' );
END;
/
```

****