CHAPTER 1

Basic Syntax

This chapter describes the basic syntax followed:

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

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

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

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

The 'Hello World' Example:

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

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

```
Hello World


PL/SQL procedure successfully completed.
```

The PL/SQL Identifiers

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

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

The PL/SQL Delimiters

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

Delimiter	**Description**
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)

/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

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

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

```
DECLARE
   -- variable declaration
   message  varchar2(20):= 'Hello, World!';
BEGIN
   /*
     * PL/SQL
     executable
     statement(
     s) */
   dbms_
output.p
ut_line(
message)
; END;
/
```

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

```
Hello World


PL/SQL procedure successfully completed.
```

PL/SQL Program Units

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

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

CHAPTER 2

Data Types

This chapter describes the Data Types used under PL/SQL:

PL/SQL variables, constants and parameters must have a valid data type which specifies a storage format, constraints, and valid range of values. This tutorial will take you through **SCALAR** and **LOB** data types available in PL/SQL and other two data types will be covered in other chapters.

Category	Description
Scalar	Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
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.
Composite	Data items that have internal components that can be accessed individually. For example, collections and records.
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:

Date Type	Description
Numeric	Numeric values on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values on which logical operations are performed.
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 subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

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

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
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.
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits.
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits.
NUMERIC(pre, secale)	Floating type with maximum precision of 38 decimal digits.
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
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:

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
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 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 large to 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 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.

CHAPTER 3

Variables

This chapter describes the variables used:

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

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

PL/SQL programming language allows to define various types of variables,s which we will cover in subsequent chapters like date time data types, records, collections, etc. For this chapter, let us study only basic variable types.

Variable Declaration in PL/SQL

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

The syntax for declaring a variable is:

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

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

```
sales number(10, 2);
pi CONSTANT double precision :=
3.1415; name varchar2(25);
address varchar2(100);
```

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 program would produce unexpected result. 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.
```

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:

(For SQL statements please look at the SQL tutorial)

```
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
```

Next, let us insert some values in the table:

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

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

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

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

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

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 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
```

CHAPTER 4

Constants

This chapter shows the usage of 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 SQL prompt, it produces the following result:

```
Radius: 9.5
Diameter: 19
```

```
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.
```

The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals:

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

The following table provides examples from all these categories of literal values.

Literal Type	**Example:**
Numeric Literals	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	'Hello, world!' 'Tutorials Point' '19-NOV-12'
BOOLEAN Literals	TRUE, FALSE, and NULL.

Date and Time Literals	DATE '1978-12-25';
	TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown below:

```
DECLARE
   message  varchar2(20):= ''That''s tutorialspoint.com!'';
BEGIN
   dbms_output.put_line(message)
   ;
END;
/
```

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

```
That's tutorialspoint.com!

PL/SQL procedure successfully completed.
```

CHAPTER 5

Operators

This chapter describes the different operators used under PL/SQL:

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

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

This tutorial will explain the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed under the chapter: **PL/SQL - Strings**.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5, then:

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Example:

```
BEGIN
   dbms_output.put_line(  10  +  5);
   dbms_output.put_line(  10  -  5);
   dbms_output.put_line(  10  *  5);
   dbms_output.put_line(  10  /  5);
   dbms_output.put_line( 10 ** 5);
END;
/
```

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

```
15
5
50
2
100000


PL/SQL procedure successfully completed.
```

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example:

```
DECLARE
   a number (2) := 21;
   b number (2) := 10;
BEGIN
   IF (a = b) then
      dbms_output.put_line('Line 1 - a is equal to b');
   ELSE
      dbms_output.put_line('Line 1 - a is not equal to b');
   END IF;
   IF (a < b) then
      dbms_output.put_line('Line 2 - a is less than b');
   ELSE
      dbms_output.put_line('Line 2 - a is not less than b');
   END IF;

   IF ( a > b ) THEN
      dbms_output.put_line('Line 3 - a is greater than b');
   ELSE
      dbms_output.put_line('Line 3 - a is not greater than b');
   END IF;

   -- Lets change value of a and
   b a := 5;
   b := 20;
   IF ( a <= b ) THEN
      dbms_output.put_line('Line 4 - a is either equal or less than b');
   END IF;

   IF ( b >= a ) THEN
      dbms_output.put_line('Line 5 - b is either equal or greater than a');
   END IF;

   IF ( a <> b ) THEN
      dbms_output.put_line('Line 6 - a is not equal to b');
   ELSE
      dbms_output.put_line('Line 6 - a is equal to b');
   END IF;

END;
/
```

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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either equal or less than b
Line 5 - b is either equal or greater than a
Line 6 - a is not equal to b

PL/SQL procedure successfully completed
```

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE OR NULL.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a	If x = 10 then, x between 5 and 20

	and x <= b.	returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

LIKE Operator:

This program tests the LIKE operator, though you will learn how to write procedure in PL/SQL, but I'm going to use a small *procedure()* to show the functionality of LIKE operator:

```
DECLARE
PROCEDURE compare (value  varchar2,  pattern varchar2 ) is
BEGIN
   IF value LIKE pattern THEN
      dbms_output.put_line ('True');
   ELSE
      dbms_output.put_line ('False');
   END
IF; END;

BEGIN
   compare('Zara Ali', 'Z%A_i');
   compare('Nuha Ali', 'Z%A_i');
END;
/
```

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

```
True
False

PL/SQL procedure successfully completed.
```

BETWEEN Operator:

The following program shows the usage of the BETWEEN operator:

```
DECLARE
   x number(2) := 10;
BEGIN
   IF (x between 5 and 20) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False')
      ;
   END IF;

   IF (x BETWEEN 5 AND 10) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False')
      ;
   END IF;

   IF (x BETWEEN 11 AND 20) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False')
      ;
   END
IF; END;
/
```

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

```
True
True
False

PL/SQL procedure successfully completed.
```

IN and IS NULL Operators:

The following program shows the usage of IN and IS NULL operators:

```
DECLARE
   letter varchar2(1) := 'm';
BEGIN
   IF (letter in ('a', 'b', 'c')) THEN
      dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False')
      ;
   END IF;

   IF (letter in ('m', 'n', 'o')) THEN
       dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False')
      ;
   END IF;

   IF (letter is null) THEN
    dbms_output.put_line('True');
   ELSE
      dbms_output.put_line('False')
      ;
   END
IF; END;
/
```

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

```
False
True
False

PL/SQL procedure successfully completed.
```

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false, then:

Operator	Description	Example
and	Called logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

Example:

```
DECLARE
   a boolean := true; b
   boolean := false;
BEGIN
   IF (a AND b) THEN
      dbms_output.put_line('Line 1 - Condition is true');
   END IF;
   IF (a OR b) THEN
      dbms_output.put_line('Line 2 - Condition is true');
   END IF;
   IF (NOT a) THEN
      dbms_output.put_line('Line 3 - a is not true');
   ELSE
      dbms_output.put_line('Line 3 - a is true');
   END IF;
   IF (NOT b) THEN
      dbms_output.put_line('Line 4 - b is not true');
   ELSE
      dbms_output.put_line('Line 4 - b is true');
   END IF;
END;
/
```

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

```
Line 2 - Condition is true
Line 3 - a is true
Line 4 - b is not true


PL/SQL procedure successfully completed.
```

PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operator	**Operation**
**	Exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	logical negation
AND	Conjunction
OR	Inclusion

Example:

Try the following example to understand the operator precedence available in PL/SQL:

```
DECLARE
   a number(2) := 20;
   b number(2) := 10;
   c number(2) := 15;
   d number(2) := 5;
   e number(2) ;
BEGIN
   e := (a + b) * c / d;      -- ( 30 * 15 ) / 5
   dbms_output.put_line('Value of (a + b) * c / d is : '|| e );

   e := ((a + b) * c) / d;   -- (30 * 15 ) / 5
   dbms output.put line('Value of ((a + b) * c) / d is  : ' ||  e );

   e := (a + b) * (c / d);   -- (30) * (15/5)
   dbms output.put line('Value of (a + b) * (c / d) is  : '||  e );

   e := a + (b * c) / d;     --  20 + (150/5)
   dbms_output.put_line('Value of a + (b * c) / d is   : ' || e );
END;
/
```

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

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is   : 90
Value of (a + b) * (c / d) is   : 90
Value of a + (b * c) / d is  : 50
```

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

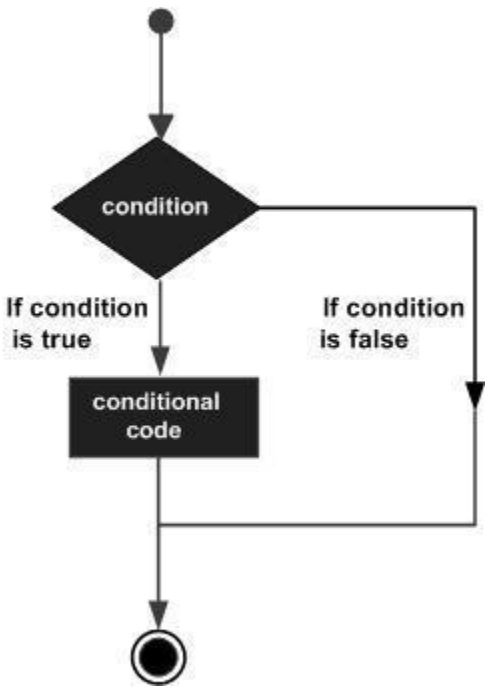
CHAPTER 7

Conditions

This chapter describes the Decision Making Structure:

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

Following is the general from of a typical conditional (i.e., decision making) structure found in most of the programming languages:



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

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

IF - THEN statement

It is the simplest form of **IF** control statement, frequently used in decision making and changing the control flow of the program execution.
The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL,** then the **IF** statement does nothing.

Syntax:

Syntax for IF-THEN statement is:

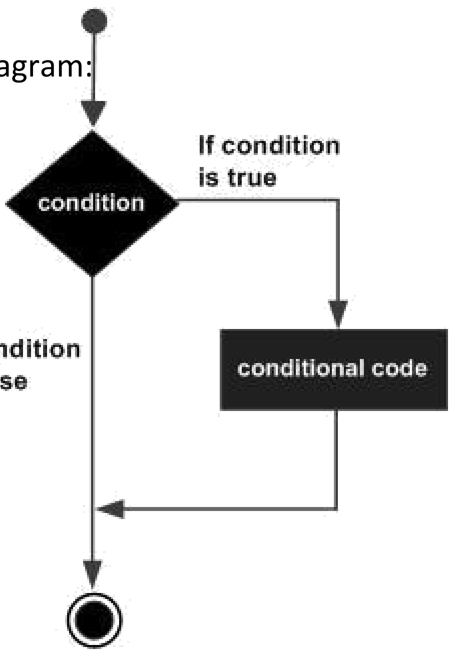
```
IF condition THEN
   S;
END IF;
```

Where *condition* is a Boolean or relational condition and *S* is a simple or compound statement. Example of an IF-THEN statement is:

```
IF (a <= 20)
   THEN c:=
   c+1;
END IF;
```

If the Boolean expression *condition* evaluates to true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end if) will be executed.

Flow Diagram:



Example 1:

Let us try a complete example that would illustrate the concept:

```
DECLARE
   a number(2) := 10;
BEGIN
   a:= 10;
  -- check the boolean condition using if
   statement IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

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

```
a is less than 20
value of a is : 10

PL/SQL procedure successfully completed.
```

Example 2:

Consider we have a table and few records in the table as we had created in PL/SQL Variable Types

```
DECLARE
   c_id customers.id%type := 1;
   c_sal customers.salary%type;
```

BEGIN

```
   SELECT salary
   INTO c_sal FROM
   customers WHERE
   id = c_id;
   IF (c_sal <= 2000) THEN
      UPDATE customers
      SET salary = salary + 1000
         WHERE id = c_id;
      dbms_output.put_line ('Salary updated');
   END
IF; END;
/
```

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

```
Salary updated


PL/SQL procedure successfully completed.
```

IF-THEN-ELSE statement

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which execute when the condition is **FALSE**.

Syntax:

Syntax for the IF-THEN-ELSE statement is:

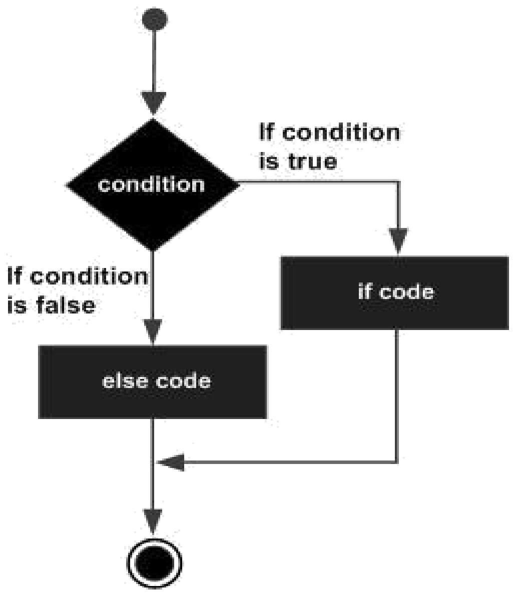
```
IF condition THEN
   S1;
ELSE
   S2;
END IF;
```

Where, *S1* and *S2* are different sequence of statements. In the IF-THEN-ELSE statements, when the test *condition* is TRUE, the statement *S1* is executed and *S2* is skipped; when the test *condition* is FALSE, then *S1* is bypassed and statement *S2* is executed. For example:

```
IF color = red THEN
  dbms_output.put_line('You have chosen a red car')
ELSE
  dbms_output.put_line('Please choose a color for your car');
END IF;
```

If the Boolean expression *condition* evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

Flow Diagram:



Example:

Let us try a complete example that would illustrate the concept:

```
DECLARE
   a number(3) := 100;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   ELSE
      dbms_output.put_line('a is not less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' ||
a); END;
/
```

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

```
a is not less than 20

value of a is : 100


PL/SQL procedure successfully completed.
```

IF-THEN-ELSIF statement

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.
When using **IF-THEN-ELSIF** statements, there are few points to keep in mind.

- It's ELSIF, not ELSEIF

- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax:

The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is:

```
IF(boolean_expression 1)THEN
   S1; -- Executes when the boolean expression 1 is
true ELSIF( boolean_expression 2) THEN
   S2; -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
   S3; -- Executes when the boolean expression 3 is true
ELSE
   S4; -- executes when the none of the above condition is true
END IF;
```

Example:

```
DECLARE
   a number(3) := 100;
BEGIN
   IF ( a = 10 ) THEN
      dbms_output.put_line('Value of a is 10' );
   ELSIF ( a = 20 ) THEN
      dbms_output.put_line('Value of a is 20' );
   ELSIF ( a = 30 ) THEN
      dbms_output.put_line('Value of a is 30' );
   ELSE
       dbms_output.put_line('None of the values is matching');
   END IF;
   dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

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

```
None of the values is matching
Exact value of a is: 100

PL/SQL procedure successfully completed.
```

Case statement

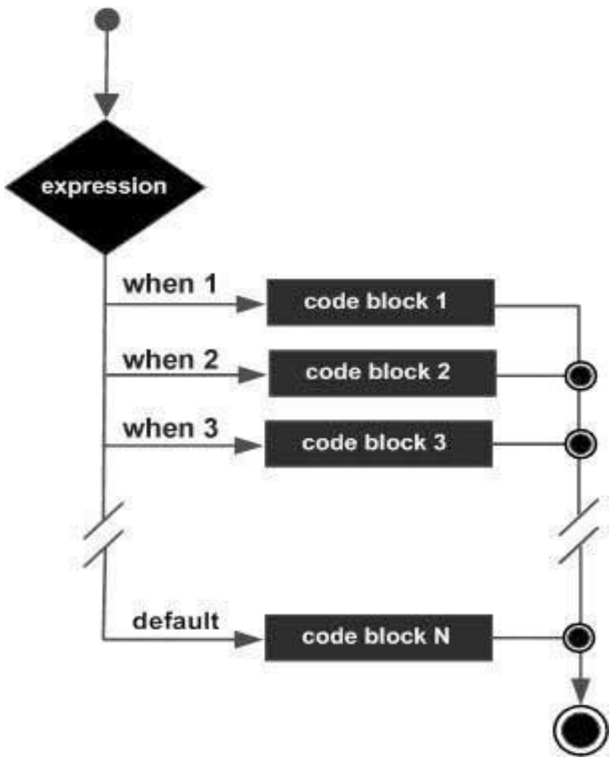
Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, whose value is used to select one of several alternatives.

Syntax:

The syntax for case statement in PL/SQL is:

```
CASE selector
    WHEN 'value1' THEN S1;
    WHEN 'value2' THEN S2;
    WHEN 'value3' THEN S3;
    ...
    ELSE Sn;  -- default case
END CASE;
```

Flow Diagram:



Example:

```
DECLARE
   grade char(1) := 'A';
BEGIN
   CASE grade
      when 'A' then dbms_output.put_line('Excellent');
      when 'B' then dbms_output.put_line('Very good');
      when 'C' then dbms_output.put_line('Well done');
      when 'D' then dbms_output.put_line('You passed');
      when 'F' then dbms_output.put_line('Better try
      again'); else dbms_output.put_line('No such grade');
   END
CASE; END;
/
```

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

```
Excellent

PL/SQL procedure successfully completed.
```

Searched CASE statement

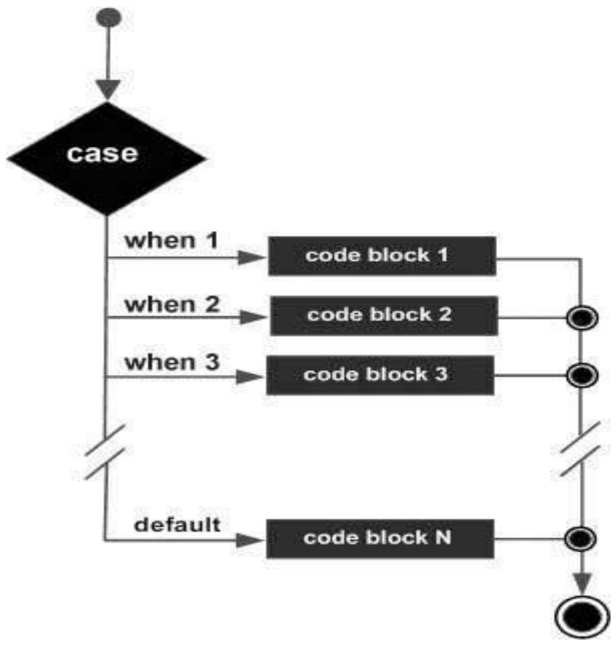
The searched **CASE** statement has no selector and its **WHEN** clauses contain search conditions that give Boolean values.

Syntax:

The syntax for searched case statement in PL/SQL is:

```
CASE
    WHEN  selector  =  'value1'  THEN  S1;
    WHEN  selector  =  'value2'  THEN  S2;
    WHEN selector = 'value3' THEN S3;
    ...
    ELSE Sn;  -- default case
END CASE;
```

Flow Diagram:



Example:

```
DECLARE
   grade char(1) := 'B';
BEGIN
   case
      when grade = 'A' then dbms_output.put_line('Excellent');
      when grade = 'B' then dbms_output.put_line('Very good');
      when grade = 'C' then dbms_output.put_line('Well done');
      when grade = 'D' then dbms_output.put_line('You passed');
      when grade = 'F' then dbms_output.put_line('Better try
      again'); else dbms_output.put_line('No such grade');
   end case;
```

```
END;
/
```

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

```
Very good

PL/SQL procedure successfully completed.
```

Nested IF-THEN-ELSE

It is always legal in PL/SQL programming to nest **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

Syntax:

```
IF( boolean_expression 1)THEN
    -- executes when the boolean expression 1 is true
    IF(boolean_expression 2) THEN
      -- executes when the boolean expression 2 is true
      sequence-of-statements;
   END IF;
ELSE
   -- executes when the boolean expression 1 is not true
  else-statements;
END IF;
```

Example:

```
DECLARE
   a number(3) := 100;
   b number(3) := 200;
BEGIN
   --     check     the     boolean
   condition IF( a = 100 ) THEN
   --   if   condition   is   true   then   check   the
      following IF( b = 200 ) THEN
      -- if condition is true then print the following
         dbms_output.put_line('Value of a is 100 and b is 200' );
      END IF;
   END IF;
   dbms_output.put_line('Exact value of a is : ' || a );
   dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200

PL/SQL procedure successfully completed.
```

CHAPTER

10

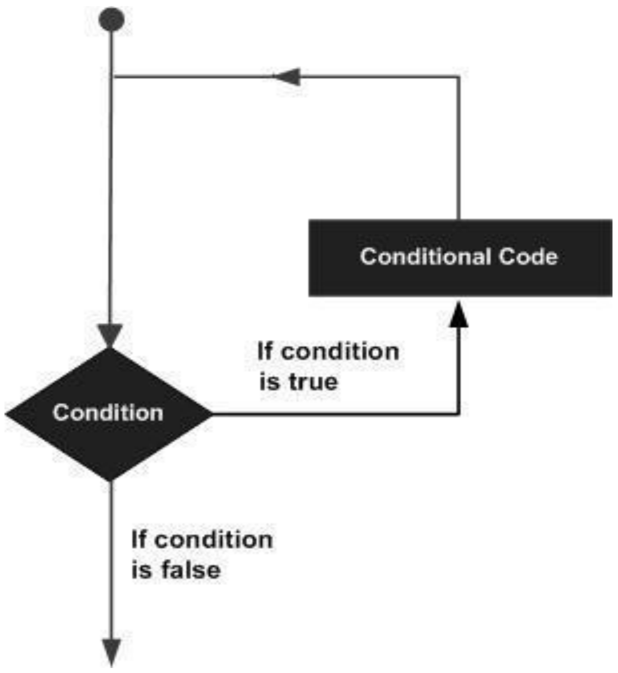
Loops

This chapter describes the various loops used under PL/SQL:

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

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

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



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

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

PL/SQL Basic LOOP

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax:

The syntax of a basic loop in PL/SQL programming language is:

```
LOOP
  Sequence of statements;
END LOOP;
```

Here, sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

Example:

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      IF x > 50 THEN
         exit;
      END IF;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

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

```
10
20
30
40
50
After Exit x is: 60
```

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

You can use the **EXIT WHEN** statement instead of the **EXIT** statement:

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      exit WHEN x > 50;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

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

```
10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.
```

PL/SQL WHILE LOOP

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP dbms_output.put_line('value
      of a: ' || a); a := a + 1;

   END
LOOP; END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

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

PL/SQL FOR LOOP

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
FOR counter IN initial_value .. final_value
   LOOP sequence_of_statements;
END LOOP;
```

Here is the flow of control in a for loop:

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition ,i.e., *initial_value .. final_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the value of the *counter* variable is increased or decreased.
- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

Following are some special characteristics of PL/SQL for loop:

- The *initial_value* and *final_value* of the loop variable or *counter* can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The *initial_value* need not to be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows determine the loop range dynamically at run time.

Example:

```
DECLARE
   a number(2);
BEGIN
```

```
   FOR a in 10 .. 20 LOOP
       dbms_output.put_line('value of a: ' || a);
  END
LOOP; END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this:

```
DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
   END
LOOP; END;
/
```

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

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10

PL/SQL procedure successfully completed.
```

Nested loops in PL/SQL

PL/SQL allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows:

```
LOOP
   Sequence of statements1
   LOOP
      Sequence of statements2
   END LOOP;
END LOOP;
```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows:

```
FOR counter1 IN initial_value1 .. final_value1
   LOOP sequence_of_statements1
   FOR counter2 IN initial_value2 .. final_value2
      LOOP sequence_of_statements2
   END LOOP;
END LOOP;
```

The syntax for a nested WHILE LOOP statement in Pascal is as follows:

```
WHILE condition1 LOOP
   sequence_of_statements1
   WHILE condition2 LOOP
      sequence_of_statements2
   END LOOP;
END LOOP;
```

Example:

The following program uses a nested basic loop to find the prime numbers from 2 to 100:

```
DECLARE
   i number(3);
   j number(3);
BEGIN
   i := 2;
   LOOP
      j:= 2;
      LOOP
         exit WHEN ((mod(i, j) = 0) or (j = i));
         j := j +1;
      END LOOP;
   IF (j = i ) THEN
      dbms_output.put_line(i || ' is prime');
   END IF;
   i := i + 1;
   exit WHEN i = 50;
   END
LOOP; END;
/
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime

PL/SQL procedure successfully completed.
```

Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept:

```
DECLARE
   i number(1);
   j number(1);
BEGIN
   << outer_loop >> FOR
   i IN 1..3 LOOP
      << inner_loop >> FOR
      j IN 1..3 LOOP
         dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
      END loop inner_loop;
   END loop outer_loop;
END;
/
```

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

```
i is: 1 and j is: 1 i
is:  1  and  j  is:  2  i
is:  1  and  j  is:  3  i
is:  2  and  j  is:  1  i
is:  2  and  j  is:  2  i
is:  2  and  j  is:  3  i
is:  3  and  j  is:  1  i
is:  3  and  j  is:  2  i
is: 3 and j is: 3

PL/SQL procedure successfully completed.
```

The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also helps in taking the control outside a loop. Click the following links to check their detail.

Control Statement	Description
EXIT statement	The Exit statement completes the loop and control passes to the statement immediately after END LOOP
CONTINUE statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
GOTO statement	Transfers control to the labeled statement. Though it is not advised to use GOTO statement in your program.

Hace que el bucle omita el resto de su cuerpo e inmediatamente vuelva a probar su condición antes de reiterar.

Transfiere el control a la declaración etiquetada. Aunque no se recomienda
use la declaración GOTO en su programa.

EXIT statement

The **EXIT** statement in PL/SQL programming language has following two usages:

- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

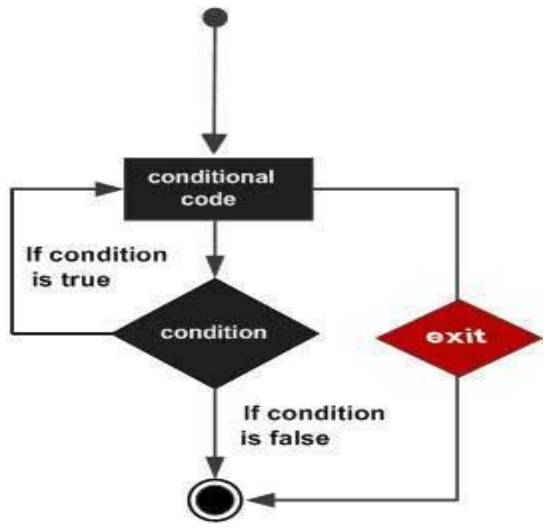
- If you are using nested loops (i.e. one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for an EXIT statement in PL/SQL is as follows:

```
EXIT;
```

Flow Diagram:



Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a > 15 THEN
         -- terminate the loop using the exit statement
         EXIT;
      END IF;
   END
LOOP; END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.
```

The EXIT WHEN Statement

The **EXIT-WHEN** statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after END LOOP.

Following are two important aspects for the EXIT WHEN statement:

- Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.
- A statement inside the loop must change the value of the condition.

Syntax:

The syntax for an EXIT WHEN statement in PL/SQL is as follows:

```
EXIT WHEN condition;
```

The EXIT WHEN statement **replaces a conditional statement like if-then** used with the EXIT statement.

Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
```

```
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      -- terminate the loop using the exit when
   statement EXIT WHEN a > 15;
   END
LOOP; END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15


PL/SQL procedure successfully completed.
```

CONTINUE statement

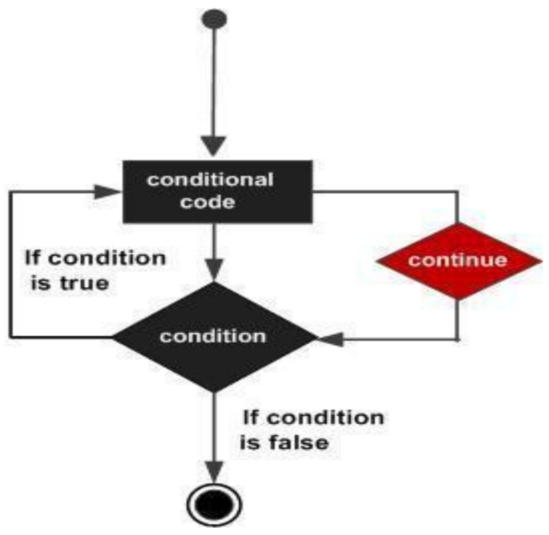
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax:

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

Flow Diagram:



Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' ||
      a); a := a + 1;
      IF a = 15 THEN
         -- skip the loop using the CONTINUE
         statement a := a + 1;
         CONTINUE;
      END IF;
   END
LOOP; END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

GOTO statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

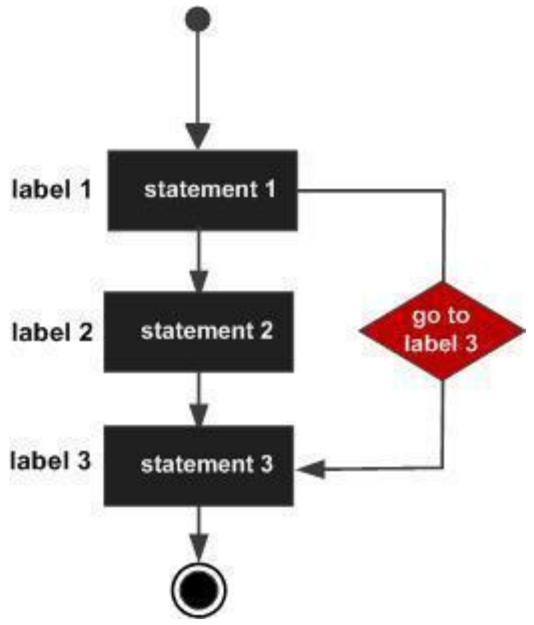
NOTE: Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax:

The syntax for a GOTO statement in PL/SQL is as follows:

```
GOTO label;
..
..
<< label >>
statement;
```

Flow Diagram:



Example:

```
DECLARE
   a number(2) := 10;
BEGIN
   <<loopstart>>
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' ||
      a); a := a + 1;
      IF a = 15 THEN
         a := a + 1;
         GOTO loopstart;
      END IF;
   END
LOOP; END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions:

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.

- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.

- A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).

- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.

- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.