## PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

# Prerequisites

Before proceeding with this, you should have a basic understanding of software basic concepts like what is database, source code, text editor and execution of programs, etc. If you already have understanding on SQL and other computer programming language then it will be an added advantage to proceed.

## Overview of PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are notable facts about PL/SQL:

* PL/SQL is a completely portable, high-performance transaction-processing language.
* PL/SQL provides a built-in interpreted and OS independent programming environment.
* PL/SQL can also directly be called from the command-line SQL\*Plus interface.
* Direct call can also be made from external programming language calls to database.
* PL/SQL's general syntax is based on that of ADA and Pascal programming language.
* Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

## Features of PL/SQL

PL/SQL has the following features:

* PL/SQL is tightly integrated with SQL.
* It offers extensive error checking.
* It offers numerous data types.
* It offers a variety of programming structures.
* It supports structured programming through functions and procedures.
* It supports object-oriented programming.
* It supports developing web applications and server pages.

## Advantages of PL/SQL

PL/SQL has the following advantages:

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

## Environment setup of PL/SQL

PL/SQL is not a stand-alone programming language; it is a tool within the Oracle programming environment. SQL\* Plus is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt. These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen.

To run PL/SQL programs, you should have Oracle RBDMS Server installed in your machine which will take care of executing SQL commands. Most recent version of Oracle RDBMS is 11g. You can download a trial version of Oracle 11g

You will have to download either 32bit or 64 bit version of the installation as per your operating system. Usually there are two files, as I have downloaded for 64 bit Windows7. You will also use similar steps on your operating system, does not matter if it is Linux or Solaris.

## Text Editor

Running large programs from command prompt may land you in inadvertently losing some of the work. So a better option is to use command files. To use the command files:

* Type your code in a text editor, like Notepad, Notepad+, or EditPlus, etc.
* Save the file with the .sql extension in the home directory.
* Launch SQL\*Plus command prompt from the directory where you created your PL/SQL file.
* Type @file\_name at the SQL\*Plus command prompt to execute your program.

If you are not using a file to execute PL/SQL scripts, then simply copy your PL/SQL code and then right click on the black window having SQL prompt and use **paste** option to paste complete code at the command prompt. Finally, just press enter to execute the code, if it is not already executed.

## Basic Syntax

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:

|  |  |
| --- | --- |
|  |  |
|  | **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. |
|  | **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. |
|  | **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.

**NOTE: The backward slash '/' in the above program indicates to execute the above PL/SQL Block.**

DBMS\_OUTPUT is a package that enable you to display the output from PL/SQL block or subprograms,and PUT\_LINE is procedure which is define inside DBMS\_OUTPUT package.

PUT\_LINE procedure basically load the out put  to the SGA(System Global Area).so you have to setting SET SERVEROUTPUT ON  to See the output.

## 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.put\_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 Variables

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

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

PL/SQL programming language allows to define various types of variables, 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:

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

**PL/SQL 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.

## 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 |

## Conditions

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

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](http://www.tutorialspoint.com/plsql/plsql_if_then.htm) | 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](http://www.tutorialspoint.com/plsql/plsql_if_then_else.htm) | **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](http://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm) | It allows you to choose between several alternatives. |
| [Case statement](http://www.tutorialspoint.com/plsql/plsql_case_statement.htm) | 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](http://www.tutorialspoint.com/plsql/plsql_searched_case.htm) | The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values. |
| [nested IF-THEN-ELSE](http://www.tutorialspoint.com/plsql/plsql_nested_if.htm) | You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s). |

## IF-THEN:

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:

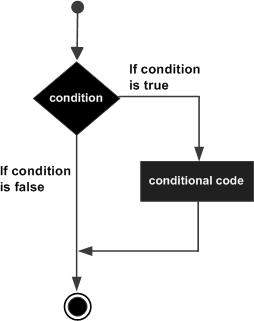
IF (a <= 20) THEN

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.

## IF-THEN-ELSE:

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')

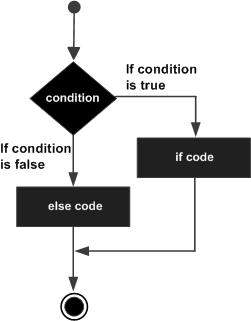
ELSE

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:

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;

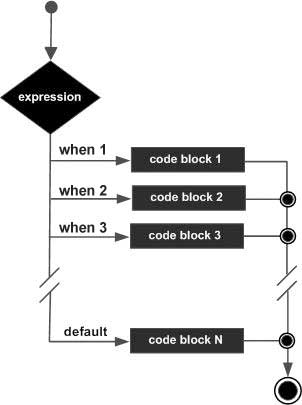
WHEN 'value3' THEN S3;

...

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.

## 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;

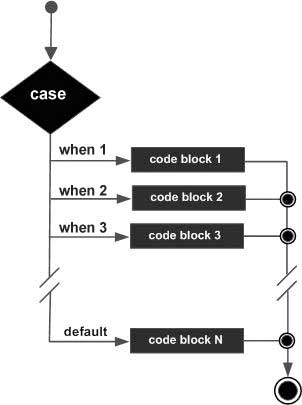
WHEN selector = 'value3' THEN S3;

...

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.

## PL/SQL LOOP

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](http://www.tutorialspoint.com/plsql/plsql_basic_loop.htm) | 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](http://www.tutorialspoint.com/plsql/plsql_while_loop.htm) | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [PL/SQL FOR LOOP](http://www.tutorialspoint.com/plsql/plsql_for_loop.htm) | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [Nested loops in PL/SQL](http://www.tutorialspoint.com/plsql/plsql_nested_loops.htm) | You can use one or more loop inside any another basic loop, while or for loop. |

## 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.

## 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.

## 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 Loop

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 details.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| [EXIT statement](http://www.tutorialspoint.com/plsql/plsql_exit_statement.htm) | The Exit statement completes the loop and control passes to the statement immediately after END LOOP |
| [CONTINUE statement](http://www.tutorialspoint.com/plsql/plsql_continue_statement.htm) | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| [GOTO statement](http://www.tutorialspoint.com/plsql/plsql_goto_statement.htm) | Transfers control to the labeled statement. Though it is not advised to use GOTO statement in your program. |

## EXIT

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;

## 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

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;

## 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

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;

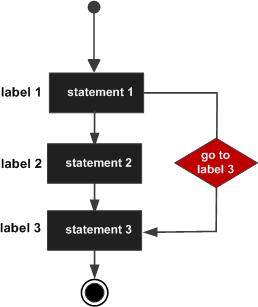
..

..

<< 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.

## PL/SQL Cursors

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

* Implicit cursors
* Explicit cursors

## Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK\_ROWCOUNT and %BULK\_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

## Example:

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

The following program would update the table and increase salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected:

DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 500;

IF sql%notfound THEN

dbms\_output.put\_line('no customers selected');

ELSIF sql%found THEN

total\_rows := sql%rowcount;

dbms\_output.put\_line( total\_rows || ' customers selected ');

END IF;

END;

/

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

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated:

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |

| 2 | Khilan | 25 | Delhi | 2000.00 |

| 3 | kaushik | 23 | Kota | 2500.00 |

| 4 | Chaitali | 25 | Mumbai | 7000.00 |

| 5 | Hardik | 27 | Bhopal | 9000.00 |

| 6 | Komal | 22 | MP | 5000.00 |

+----+----------+-----+-----------+----------+

## Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is :

CURSOR cursor\_name IS select\_statement;

Working with an explicit cursor involves four steps:

* Declaring the cursor for initializing in the memory
* Opening the cursor for allocating memory
* Fetching the cursor for retrieving data
* Closing the cursor to release allocated memory

## Declaring the Cursor

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

CURSOR c\_customers IS

SELECT id, name, address FROM customers;

## Opening the Cursor

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

OPEN c\_customers;

## Fetching the Cursor

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

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

## Closing the Cursor

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

CLOSE c\_customers;

## Example:

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

DECLARE

c\_id customers.id%type;

c\_name customers.name%type;

c\_addr customers.address%type;

CURSOR c\_customers is

SELECT id, name, address FROM customers;

BEGIN

OPEN c\_customers;

LOOP

FETCH c\_customers into c\_id, c\_name, c\_addr;

EXIT WHEN c\_customers%notfound;

dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);

END LOOP;

CLOSE c\_customers;

END;

/

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

1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

PL/SQL procedure successfully completed.

## PL/SQL Procedures

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

* **Functions**: these subprograms return a single value, mainly used to compute and return a value.
* **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure** and we will cover **PL/SQL function** in next chapter.

## Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may have a parameter list. Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

|  |  |
| --- | --- |
|  |  |
|  | **Declarative Part**  It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution. |
|  | **Executable Part**  This is a mandatory part and contains statements that perform the designated action. |
|  | **Exception-handling**  This is again an optional part. It contains the code that handles run-time errors. |

## Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

**Where,**

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows modifying an existing procedure.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* *procedure-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone procedure.

## Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

When above code is executed using SQL prompt, it will produce the following result:

Procedure created.

## Executing a Standalone Procedure

A standalone procedure can be called in two ways:

* Using the EXECUTE keyword
* Calling the name of the procedure from a PL/SQL block

The above procedure named 'greetings' can be called with the EXECUTE keyword as:

EXECUTE greetings;

The above call would display:

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block:

BEGIN

greetings;

END;

/

The above call would display:

Hello World

PL/SQL procedure successfully completed.

## Deleting a Standalone Procedure

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is:

DROP PROCEDURE procedure-name;

So you can drop *greetings* procedure by using the following statement:

DROP PROCEDURE greetings;

## Parameter Modes in PL/SQL Subprograms

|  |  |
| --- | --- |
|  |  |
|  | **IN**  An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing.** |
|  | **OUT**  An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable.** |
|  | **IN OUT**  An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read.  The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. |

## IN/OUT Mode Example 1

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;

BEGIN

a:= 23;

b:= 45;

findMin(a, b, c);

dbms\_output.put\_line(' Minimum of (23, 45) : ' || c);

END;

/

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

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

## IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

DECLARE

a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

x := x \* x;

END;

BEGIN

a:= 23;

squareNum(a);

dbms\_output.put\_line(' Square of (23): ' || a);

END;

/

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

Square of (23): 529

PL/SQL procedure successfully completed.

## Methods for Passing Parameters

Actual parameters could be passed in three ways:

* Positional notation
* Named notation
* Mixed notation

### POSITIONAL NOTATION

In positional notation, you can call the procedure as:

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

### NAMED NOTATION

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol ( => ). So the procedure call would look like:

findMin(x=>a, y=>b, z=>c, m=>d);

### MIXED NOTATION

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal:

findMin(a, b, c, m=>d);

But this is not legal:

findMin(x=>a, b, c, d);

## PL/SQL Functions

A PL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

## Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

**Where,**

* *function-name* specifies the name of the function.
* [OR REPLACE] option allows modifying an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* *RETURN* clause specifies that data type you are going to return from the function.
* *function-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

## Example:

The following example illustrates creating and calling a standalone function. This function returns the total number of CUSTOMERS in the customers table. We will use the CUSTOMERS table, which we had created in [PL/SQL Variables](http://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter:

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM customers;

RETURN total;

END;

/

When above code is executed using SQL prompt, it will produce the following result:

Function created.

## Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function.

A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms\_output.put\_line('Total no. of Customers: ' || c);

END;

/

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

Total no. of Customers: 6

PL/SQL procedure successfully completed.

## Example:

The following is one more example which demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

/

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

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

## PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

n! = n\*(n-1)!

= n\*(n-1)\*(n-2)!

...

= n\*(n-1)\*(n-2)\*(n-3)... 1

The following program calculates the factorial of a given number by calling itself recursively:

DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number

IS

f number;

BEGIN

IF x=0 THEN

f := 1;

ELSE

f := x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:= 6;

factorial := fact(num);

dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);

END;

/

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

Factorial 6 is 720

PL/SQL procedure successfully completed.

## PL/SQL Records

A PL/SQL **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 like, 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

## Table-Based Records

The %ROWTYPE attribute enables a programmer to create **table-based** and **cursor-based** records.

The following example would illustrate the concept of **table-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

customer\_rec customers%rowtype;

BEGIN

SELECT \* into customer\_rec

FROM customers

WHERE id = 5;

dbms\_output.put\_line('Customer ID: ' || customer\_rec.id);

dbms\_output.put\_line('Customer Name: ' || customer\_rec.name);

dbms\_output.put\_line('Customer Address: ' || customer\_rec.address);

dbms\_output.put\_line('Customer Salary: ' || customer\_rec.salary);

END;

/

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

Customer ID: 5

Customer Name: Hardik

Customer Address: Bhopal

Customer Salary: 9000

PL/SQL procedure successfully completed.

## Cursor-Based Records

The following example would illustrate the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

CURSOR customer\_cur is

SELECT id, name, address

FROM customers;

customer\_rec customer\_cur%rowtype;

BEGIN

OPEN customer\_cur;

LOOP

FETCH customer\_cur into customer\_rec;

EXIT WHEN customer\_cur%notfound;

DBMS\_OUTPUT.put\_line(customer\_rec.id || ' ' || customer\_rec.name);

END LOOP;

END;

/

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

1 Ramesh

2 Khilan

3 kaushik

4 Chaitali

5 Hardik

6 Komal

PL/SQL procedure successfully completed.

## User-Defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures.

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;

Here is the way you would declare the Book record:

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 the example to explain 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 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 in very similar way as you pass any other variable. You would access the record fields in the similar way as you have 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 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.

## EXCEPTIONS

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

* System-defined exceptions
* User-defined exceptions

## Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

exception3-handling-statements

END;

## Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

c\_id customers.id%type := 8;

c\_name customers.name%type;

c\_addr customers.address%type;

BEGIN

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

EXCEPTION

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND**, which is captured in **EXCEPTION** block.

## Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

DECLARE

exception\_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

EXCEPTION

WHEN exception\_name THEN

statement;

END;

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

## User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

The syntax for declaring an exception is:

DECLARE

my-exception EXCEPTION;

## Example:

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid\_id is raised.

DECLARE

c\_id customers.id%type := &cc\_id;

c\_name customers.name%type;

c\_addr customers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id <= 0 THEN

RAISE ex\_invalid\_id;

ELSE

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

dbms\_output.put\_line('ID must be greater than zero!');

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

Enter value for cc\_id: -6 (let's enter a value -6)

old 2: c\_id customers.id%type := &cc\_id;

new 2: c\_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

## Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQLCODE** | **Description** |
| ACCESS\_INTO\_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE\_NOT\_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DENIED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGGED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when s SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

# PL/SQL - Data Types

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.