PL/SQL Tutorial

Originaal asub siin.

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.

This tutorial will give you great understanding on PL/SQL to proceed with Oracle database and other advanced RDBMS concepts.

Audience

This tutorial is designed for Software Professionals, who are willing to learn PL/SQL Programming Language in simple and easy steps. This tutorial will give you great understanding on PL/SQL Programming concepts, and after completing this tutorial, you will be at intermediate level of expertise from where you can take yourself to higher level of expertise.

Prerequisites

Before proceeding with this tutorial, 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.

PL/SQL - Overview

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:

- 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.

PL/SQL - 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:

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 sepparator
“	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator

Delimiter	Description
⇒	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 Program Units

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

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

- Type
- Type body

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:

Data 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:

- 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;
/
```

PL/SQL Character Data Types and Subtypes

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

- 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:

- 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.

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;
```

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;
```

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)
);
```

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;
```

PL/SQL - Constants and Literals

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 consant

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;
/
```

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.

```
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
```

```
'Hello, world!'
'Tutorials Point'
'19-NOV-12'
```

Literal type	Example
Numeric Literals	
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	
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(30):= ''That''s tutorialspoint.com!'';
BEGIN
   dbms_output.put_line(message);
END;
/
```

PL/SQL - Operators

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** (LINK).

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
-	Substracts 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

Operator	Description	Example
* *	Exponentiation operator, raises one operand to the power of other	* * B will give 100000

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.

Comparison operator

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 and x ⇐ b.	If x = 10 then, x between 5 and 20 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.

Operator	Description	Example
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.

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.

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.

=, <, >, ⇐, >=, <>, !=, ~=, ^=,

IS NULL, LIKE, BETWEEN, IN

Operator	Operation
* *	exponentiation
+,-	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

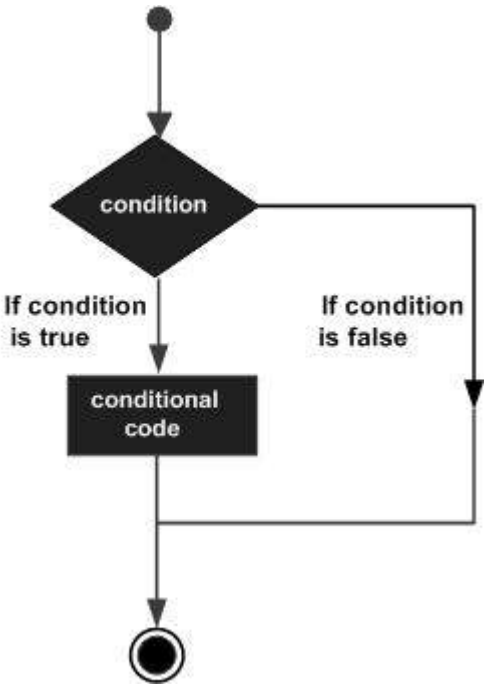
Example:

```
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;
/
```

PL/SQL - 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.

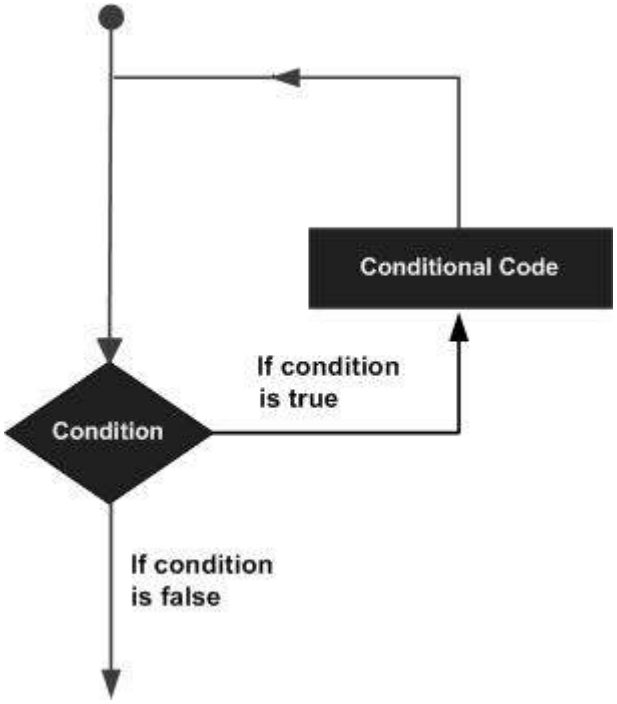
Statement	Description
IF - THEN statement	The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
IF-THEN-ELSE statement	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).

PL/SQL - Loops

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:



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:

Loop Type	Description
LOOP Basic	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.
LOOP WHILE	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
LOOP FOR	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.

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;
/
```

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	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.

PL/SQL - Strings

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings:

- Fixed-length strings: In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
- Variable-length strings: In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
- Character large objects (CLOBs): These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

```
'This is a string literal.' Or 'hello world'
```

To include a single quote inside a string literal, you need to type two single quotes next to one another, like:

```
'this isn''t what it looks like'
```

Declaring String Variables

Oracle database provides numerous string datatypes , like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are 'national character set' datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables:

```
DECLARE
   name varchar2(20);
   company varchar2(30);
   introduction clob;
   choice char(1);
BEGIN
   name := 'John Smith';
   company := 'Infotech';
```

```
   introduction := ' Hello! I''m John Smith from Infotech.';
   choice := 'y';
   IF choice = 'y' THEN
      dbms_output.put_line(name);
      dbms_output.put_line(company);
      dbms_output.put_line(introduction);
   END IF;
END;
/
```

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

```
red_flag CHAR(1) := 'Y';
red_flag CHAR    := 'Y';
```

PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator (||) for joining two strings. The following table provides the string functions provided by PL/SQL:

1. ASCII(x); Returns the ASCII value of the character x.
2. CHR(x); Returns the character with the ASCII value of x.
3. CONCAT(x, y); Concatenates the strings x and y and return the appended string.
4. INITCAP(x); Converts the initial letter of each word in x to uppercase and returns that string.
5. INSTR(x, find_string [, start] [, occurrence]); Searches for find_string in x and returns the position at which it occurs.
6. INSTRB(x); Returns the location of a string within another string, but returns the value in bytes.
7. LENGTH(x); Returns the number of characters in x.
8. LENGTHB(x); Returns the length of a character string in bytes for single byte character set.
9. LOWER(x); Converts the letters in x to lowercase and returns that string.
10. LPAD(x, width [, pad_string]); Pads x with spaces to left, to bring the total length of the string up to width characters.
11. LTRIM(x [, trim_string]); Trims characters from the left of x.
12. NANVL(x, value); Returns value if x matches the NaN special value (not a number), otherwise x is returned.
13. NLS_INITCAP(x); Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT.
14. NLS_LOWER(x); Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15. NLS_UPPER(x); Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.
16. NLSSORT(x); Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17. NVL(x, value); Returns value if x is null; otherwise, x is returned.
18. NVL2(x, value1, value2); Returns value1 if x is not null; if x is null, value2 is returned.
19. REPLACE(x, search_string, replace_string); Searches x for search_string and replaces it with replace_string.

20. RPAD(x, width [, pad_string]); Pads x to the right.
21. RTRIM(x [, trim_string]); Trims x from the right.
22. SOUNDEX(x); Returns a string containing the phonetic representation of x.
23. SUBSTR(x, start [, length]); Returns a substring of x that begins at the position specified by start. An optional length for the substring may be supplied.
24. SUBSTRB(x); Same as SUBSTR except the parameters are expressed in bytes instead of characters for the single-byte character systems
25. TRIM([trim_char FROM) x); Trims characters from the left and right of x.
26. UPPER(x); Converts the letters in x to uppercase and returns that string.

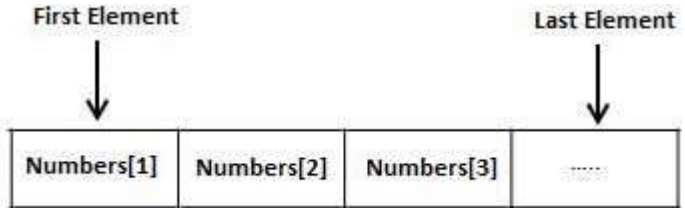
Example 1

```
DECLARE
   greetings varchar2(11) := 'hello world';
BEGIN
   dbms_output.put_line(UPPER(greetings));

   dbms_output.put_line(LOWER(greetings));

   dbms_output.put_line(INITCAP(greetings));

   /* retrieve the first character in the string */
   dbms_output.put_line ( SUBSTR (greetings, 1, 1));

   /* retrieve the last character in the string */
   dbms_output.put_line ( SUBSTR (greetings, -1, 1));

   /* retrieve five characters,
      starting from the seventh position. */
   dbms_output.put_line ( SUBSTR (greetings, 7, 5));

   /* retrieve the remainder of the string,
      starting from the second position. */
   dbms_output.put_line ( SUBSTR (greetings, 2));

   /* find the location of the first "e" */
   dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

PL/SQL - Arrays

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter 'PL/SQL Collections'.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VRRAY type at the schema level is:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,

- varray_type_name is a valid attribute name,
- n is the number of elements (maximum) in the varray,
- element_type is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/
Type created.
```

The basic syntax for creating a VRRAY type within a PL/SQL block is:

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

For example:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

Example 1 The following program illustrates using varrays:

```
DECLARE
   type namesarray IS VARRAY(5) OF VARCHAR2(10);
   type grades IS VARRAY(5) OF INTEGER;
   names namesarray;
```

```
   marks grades;
   total integer;
BEGIN
   names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i in 1 .. total LOOP
      dbms_output.put_line('Student: ' || names(i) || '
      Marks: ' || marks(i));
   END LOOP;
END;
/
```

Please note:

- In oracle environment, the starting index for varrays is always 1.
- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
- Varrays are one-dimensional arrays.
- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

PL/SQL - Procedures

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a standalone subprogram. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

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:

S.N.	Parts & Description
1	**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.
2	**Executable Part** - This is a mandatory part and contains statements that perform the designated action.
3	**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;
```

```
/
```

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

S.N.	Parameter Mode & Description
1	**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. Parameters are passed by reference.
2	**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 and it is passed by value.
3	**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. Actual parameter is passed by 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.
```

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

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 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:

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;
/
```

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;
```

```
/
```

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;
/
```

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:

- **%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;
/
```

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 - 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;
/
```

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;
/
```

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;
```

1. - Book 2 specification

book2.title := 'Telecom Billing';

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

1. - 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);
```

1. - 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;
/
```

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;
```

1. - Book 2 specification

book2.title := 'Telecom Billing';

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

1. - Use procedure to print book info

printbook(book1);

```
   printbook(book2);
END;
/
```

PL/SQL - 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;
```

1. - 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.

Exception	Oracle error	SQLCODE	Description
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 - Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Example

To start with, 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 creates a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

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

```
Trigger created.
```

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

PL/SQL - Packages

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called public objects. Any subprogram not in the package specification but coded in the package body is called a private object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

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

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the cust_sal package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

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

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the find_sal method of the cust_sal package:

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

```
Enter value for cc_id: 1
Salary: 3000
PL/SQL procedure successfully completed.
```

Example

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records:

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  3000.00 |
|  2 | Khilan   |  25 | Delhi     |  3000.00 |
|  3 | kaushik  |  23 | Kota      |  3000.00 |
|  4 | Chaitali |  25 | Mumbai    |  7500.00 |
|  5 | Hardik   |  27 | Bhopal    |  9500.00 |
|  6 | Komal    |  22 | MP        |  5500.00 |
+----+----------+-----+-----------+----------+
```

THE PACKAGE SPECIFICATION:

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name  customers.name%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);
```

1. - Removes a customer

PROCEDURE delCustomer(c_id customers.id%TYPE);

1. -Lists all customers

PROCEDURE listCustomer;

```
END c_package;
/
```

When the above code is executed at SQL prompt, it creates the above package and displays the following result:

```
Package created.
```

CREATING THE PACKAGE BODY:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
   PROCEDURE addCustomer(c_id  customers.id%type,
      c_name customers.name%type,
      c_age  customers.age%type,
      c_addr  customers.address%type,
```

```
      c_sal   customers.salary%type)
   IS
   BEGIN
      INSERT INTO customers (id,name,age,address,salary)
         VALUES(c_id, c_name, c_age, c_addr, c_sal);
   END addCustomer;
```

```
   PROCEDURE delCustomer(c_id   customers.id%type) IS
   BEGIN
       DELETE FROM customers
         WHERE id = c_id;
   END delCustomer;
```

```
   PROCEDURE listCustomer IS
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list is TABLE OF customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
   BEGIN
      FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
      END LOOP;
   END listCustomer;
END c_package;
/
```

Above example makes use of nested table which we will discuss in the next chapter. When the above code is executed at SQL prompt, it produces the following result:

```
Package body created.
```

USING THE PACKAGE:

The following program uses the methods declared and defined in the package c_package.

```
DECLARE
   code customers.id%type:= 8;
BEGIN
      c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
      c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
      c_package.listcustomer;
      c_package.delcustomer(code);
      c_package.listcustomer;
END;
/
```

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

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
```

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

PL/SQL - Collections

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections:

Collection type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

We have already discussed varray in the chapter 'PL/SQL arrays'. In this chapter, we will discuss PL/SQL tables.

Both types of PL/SQL tables, i.e., index-by tables and nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-By Table

An index-by table (also called an associative array) is a set of key-value pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an index-by table named table_name whose keys will be of subscript_type and associated values will be of element_type

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;
table_name type_name;
```

Example

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
   TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
   salary_list salary;
   name   VARCHAR2(20);
BEGIN
   -- adding elements to the table
   salary_list('Rajnish')  := 62000;
   salary_list('Minakshi')  := 75000;
   salary_list('Martin') := 100000;
   salary_list('James') := 78000;
```

1. - printing the table

name := salary_list.FIRST;

```
   WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
   END LOOP;
END;
/
```

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

```
Salary of Rajnish is 62000
Salary of Minakshi is 75000
Salary of Martin is 100000
Salary of James is 78000
```

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

Example

Elements of an index-by table could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
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 |
+----+----------+-----+-----------+----------+
```

```
DECLARE
   CURSOR c_customers is
      select  name from customers;
   TYPE c_list IS TABLE of customers.name%type INDEX BY binary_integer;
   name_list c_list;
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter|| '):'||name_list(counter));
  END LOOP;
END;
/
```

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

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
```

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

Nested Tables

A nested table is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested

table can increase dynamically.
- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
```

```
table_name type_name;
```

This declaration is similar to declaration of an index-by table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example

The following examples illustrate the use of nested table:

```
DECLARE
   TYPE names_table IS TABLE OF VARCHAR2(10);
   TYPE grades IS TABLE OF INTEGER;
```

```
   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i IN 1 .. total LOOP
      dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
   end loop;
END;
/
```

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

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92
```

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

Example

Elements of a nested table could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
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 |
+----+----------+-----+-----------+----------+

  DECLARE
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list IS TABLE of customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
   END LOOP;
END;
/
```

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

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
```

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

Collection Methods

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose:

1. Ordered List ItemEXISTS(n) - Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.
2. COUNT - Returns the number of elements that a collection currently contains.
3. LIMIT - Checks the Maximum Size of a Collection.
4. FIRST - Returns the first (smallest) index numbers in a collection that uses integer subscripts.
5. LAST - Returns the last (largest) index numbers in a collection that uses integer subscripts.
6. PRIOR(n) - Returns the index number that precedes index n in a collection.
7. NEXT(n) - Returns the index number that succeeds index n.
8. EXTEND - Appends one null element to a collection.
9. EXTEND(n) - Appends n null elements to a collection.
10. EXTEND(n,i) - Appends n copies of the ith element to a collection.
11. TRIM - Removes one element from the end of a collection.
12. TRIM(n) - Removes n elements from the end of a collection.
13. DELETE - Removes all elements from a collection, setting COUNT to 0.
14. DELETE(n) - Removes the nth element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, DELETE(n) does nothing.
15. DELETE(m,n) - Removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing.

Collection Exceptions

The following table provides the collection exceptions and when they are raised:

- COLLECTION_IS_NULL - You try to operate on an atomically null collection.
- NO_DATA_FOUND - A subscript designates an element that was deleted, or a nonexistent element of an associative array.
- SUBSCRIPT_BEYOND_COUNT - A subscript exceeds the number of elements in a collection.
- SUBSCRIPT_OUTSIDE_LIMIT - A subscript is outside the allowed range.
- VALUE_ERROR - A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS_INTEGER range, and the subscript is outside this range.

PL/SQL - Transactions

A database transaction is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting and Ending a Transaction

A transaction has a beginning and an end. A transaction starts when one of the following events take place:

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place:

- A COMMIT or a ROLLBACK statement is issued.
- A DDL statement, like CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
- A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from SQL*PLUS by issuing the EXIT command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a ROLLBACK is automatically performed.
- A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is:

```
COMMIT;
```

Example

```
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 );
COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is:

```
ROLLBACK [TO SAVEPOINT < savepoint_name>];
```

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using savepoint, then simply

use the following statement to rollback all the changes:

```
ROLLBACK;
```

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command.

The general syntax for the SAVEPOINT command is:

```
SAVEPOINT < savepoint_name >;
```

Example

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );
SAVEPOINT sav1;
```

```
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000;
ROLLBACK TO sav1;
```

```
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;
COMMIT;
```

Here,ROLLBACK TO sav1; statement rolls back the changes up to the point, where you had marked savepoint sav1 and after that new changes will start.

Automatic Transaction Control

To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as:

```
SET AUTOCOMMIT ON;
```

You can turn-off auto commit mode using the following command:

```
SET AUTOCOMMIT OFF;
```

PL/SQL - Date & Time

PL/SQL provides two classes of date and time related data types:

- Datetime data types
- Interval data types

The Datetime data types are:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

The Interval data types are:

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Field Values for Datetime and Interval Data Types

Both datetime and interval data types consist of fields. The values of these fields determine the value of the datatype. The following table lists the fields and their possible values for datetimes and intervals.

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 The 9(n) portion is not applicable for DATE.	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 for DATE or TIMESTAMP.	Not applicable
TIMEZONE_MINUTE	00 to 59 Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_REGION	Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_ABBR	Not applicable for DATE or TIMESTAMP.	Not applicable

The Datetime Data Types and Functions

Following are the Datetime data types:

- DATE - it stores date and time information in both character and number datatypes. It is made of information on century, year, month, date, hour, minute, and second. It is specified as:
- TIMESTAMP - it is an extension of the DATE datatype. It stores the year, month, and day of the

DATE datatype, along with hour, minute, and second values. It is useful for storing precise time values.
- TIMESTAMP WITH TIME ZONE - it is a variant of TIMESTAMP that includes a time zone region name or a time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC. This datatype is useful for collecting and evaluating date information across geographic regions.
- TIMESTAMP WITH LOCAL TIME ZONE - it is another variant of TIMESTAMP that includes a time zone offset in its value.

Following table provides the Datetime functions (where, x has datetime value):

1. ADD_MONTHS(x, y); Adds y months to x.
2. LAST_DAY(x); Returns the last day of the month.
3. MONTHS_BETWEEN(x, y); Returns the number of months between x and y.
4. NEXT_DAY(x, day); Returns the datetime of the next day after x.
5. NEW_TIME; Returns the time/day value from a time zone specified by the user.
6. ROUND(x [, unit]); Rounds x;
7. SYSDATE(); Returns the current datetime.
8. TRUNC(x [, unit]); Truncates x.

Timestamp functions (where, x has a timestamp value):

1. CURRENT_TIMESTAMP(); Returns a TIMESTAMP WITH TIME ZONE containing the current session time along with the session time zone.
2. EXTRACT({ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND } | { TIMEZONE_HOUR | TIMEZONE_MINUTE } | { TIMEZONE_REGION | } TIMEZONE_ABBR) FROM x); Extracts and returns a year, month, day, hour, minute, second, or time zone from x;
3. FROM_TZ(x, time_zone); Converts the TIMESTAMP x and time zone specified by time_zone to a TIMESTAMP WITH TIMEZONE.
4. LOCALTIMESTAMP(); Returns a TIMESTAMP containing the local time in the session time zone.
5. SYSTIMESTAMP(); Returns a TIMESTAMP WITH TIME ZONE containing the current database time along with the database time zone.
6. SYS_EXTRACT_UTC(x); Converts the TIMESTAMP WITH TIMEZONE x to a TIMESTAMP containing the date and time in UTC.
7. TO_TIMESTAMP(x, [format]); Converts the string x to a TIMESTAMP.
8. TO_TIMESTAMP_TZ(x, [format]); Converts the string x to a TIMESTAMP WITH TIMEZONE.

Examples

The following code snippets illustrate the use of the above functions:

```
SELECT SYSDATE FROM DUAL;
```

Output:

```
08/31/2012 5:25:34 PM
```

```
SELECT TO_CHAR(CURRENT_DATE, 'DD-MM-YYYY HH:MI:SS') FROM DUAL;
```

Output:

```
31-08-2012 05:26:14
```

```
SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

Output:

```
01/31/2013 5:26:31 PM
```

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

Output:

```
8/31/2012 5:26:55.347000 PM
```

The Interval Data Types and Functions

Following are the Interval data types:

- INTERVAL YEAR TO MONTH - it stores a period of time using the YEAR and MONTH datetime fields.
- INTERVAL DAY TO SECOND - it stores a period of time in terms of days, hours, minutes, and seconds.

Interval functions:

1. NUMTODSINTERVAL(x, interval_unit); Converts the number x to an INTERVAL DAY TO SECOND.
2. NUMTOYMINTERVAL(x, interval_unit); Converts the number x to an INTERVAL YEAR TO MONTH.
3. TO_DSINTERVAL(x); Converts the string x to an INTERVAL DAY TO SECOND.
4. TO_YMINTERVAL(x); Converts the string x to an INTERVAL YEAR TO MONTH.

PL/SQL - DBMS Output

The DBMS_OUTPUT is a built-in package that enables you to display output, display debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers. We have already used this package all throughout our tutorial.

Let us look at a small code snippet that would display all the user tables in the database. Try it in your database to list down all the table names:

```
BEGIN
   dbms_output.put_line  (user || ' Tables in the database:');
   FOR t IN (SELECT table_name FROM user_tables)
   LOOP
      dbms_output.put_line(t.table_name);
   END LOOP;
END;
/
```

DBMS_OUTPUT Subprograms ====

The DBMS_OUTPUT package has the following subprograms:

1. DBMS_OUTPUT.DISABLE; Disables message output
2. DBMS_OUTPUT.ENABLE(buffer_size IN INTEGER DEFAULT 20000); Enables message output. A NULL value of buffer_size represents unlimited buffer size.
3. DBMS_OUTPUT.GET_LINE (line OUT VARCHAR2, status OUT INTEGER); Retrieves a single line of buffered information.
4. DBMS_OUTPUT.GET_LINES (lines OUT CHARARR, numlines IN OUT INTEGER); Retrieves an array of lines from the buffer.
5. DBMS_OUTPUT.NEW_LINE; Puts an end-of-line marker
6. DBMS_OUTPUT.PUT(item IN VARCHAR2); Places a partial line in the buffer.
7. DBMS_OUTPUT.PUT_LINE(item IN VARCHAR2); Places a line in the buffer.

Example

```
DECLARE
   lines dbms_output.chararr;
   num_lines number;
BEGIN
   -- enable the buffer with default size 20000
   dbms_output.enable;
```

```
   dbms_output.put_line('Hello Reader!');
   dbms_output.put_line('Hope you have enjoyed the tutorials!');
   dbms_output.put_line('Have a great time exploring pl/sql!');
```

```
   num_lines := 3;
```

```
   dbms_output.get_lines(lines, num_lines);
```

```
   FOR i IN 1..num_lines LOOP
      dbms_output.put_line(lines(i));
   END LOOP;
END;
/
```

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

```
Hello Reader!
Hope you have enjoyed the tutorials!
Have a great time exploring pl/sql!
```

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

PL/SQL - Object Oriented

PL/SQL allows defining an object type, which helps in designing object-oriented database in Oracle. An

object type allows you to crate composite types. Using objects allow you implementing real world objects with specific structure of data and methods for operating it. Objects have attributes and methods. Attributes are properties of an object and are used for storing an object's state; and methods are used for modeling its behaviors.

Objects are created using the CREATE [OR REPLACE] TYPE statement. Below is an example to create a simple address object consisting of few attributes:

```
CREATE OR REPLACE TYPE address AS OBJECT
(house_no varchar2(10),
 street varchar2(30),
 city varchar2(20),
 state varchar2(10),
 pincode varchar2(10)
);
/
```

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

```
Type created.
```

Let's create one more object customer where we will wrap attributes and methods together to have object oriented feeling:

```
CREATE OR REPLACE TYPE customer AS OBJECT
(code number(5),
 name varchar2(30),
 contact_no varchar2(12),
 addr address,
 member procedure display
);
/
```

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

```
Type created.
```

Instantiating an Object

Defining an object type provides a blueprint for the object. To use this object, you need to create instances of this object. You can access the attributes and methods of the object using the instance name and the access operator (.) as follows:

```
DECLARE
   residence address;
BEGIN
   residence := address('103A', 'M.G.Road', 'Jaipur', 'Rajasthan','201301');
   dbms_output.put_line('House No: '|| residence.house_no);
   dbms_output.put_line('Street: '|| residence.street);
```

```
   dbms_output.put_line('City: '|| residence.city);
   dbms_output.put_line('State: '|| residence.state);
   dbms_output.put_line('Pincode: '|| residence.pincode);
END;
/
```

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

```
House No: 103A
Street: M.G.Road
City: Jaipur
State: Rajasthan
Pincode: 201301
```

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

Member MethodsHouse No: 103A Street: M.G.Road City: Jaipur State: Rajasthan Pincode: 201301

PL/SQL procedure successfully completed.

Member Methods

Member methods are used for manipulating the attributes of the object. You provide the declaration of a member method while declaring the object type. The object body defines the code for the member methods. The object body is created using the CREATE TYPE BODY statement.

Constructors are functions that return a new object as its value. Every object has a system defined constructor method. The name of the constructor is same as the object type. For example:

```
residence := address('103A', 'M.G.Road', 'Jaipur', 'Rajasthan','201301');
```

The comparison methods are used for comparing objects. There are two ways to compare objects:

- Map method: The Map method is a function implemented in such a way that its value depends upon the value of the attributes. For example, for a customer object, if the customer code is same for two customers, both customers could be the same and one. So the relationship between these two objects would depend upon the value of code.
- Order method: The Order methods implement some internal logic for comparing two objects. For example, for a rectangle object, a rectangle is bigger than another rectangle if both its sides are bigger.

Using Map method

Let us try to understand above concepts using the following rectangle object:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 member function enlarge( inc number) return rectangle,
```

```
 member procedure display,
 map member function measure return number
);
/
```

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

```
Type created.
```

Creating the type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
   MEMBER FUNCTION enlarge(inc number) return rectangle IS
   BEGIN
      return rectangle(self.length + inc, self.width + inc);
   END enlarge;
```

```
   MEMBER PROCEDURE display IS
   BEGIN
      dbms_output.put_line('Length: '|| length);
      dbms_output.put_line('Width: '|| width);
   END display;
```

```
   MAP MEMBER FUNCTION measure return number IS
   BEGIN
      return (sqrt(length*length + width*width));
   END measure;
END;
/
```

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

```
Type body created.
```

Now using the rectangle object and its member functions:

```
DECLARE
   r1 rectangle;
   r2 rectangle;
   r3 rectangle;
   inc_factor number := 5;
BEGIN
   r1 := rectangle(3, 4);
   r2 := rectangle(5, 7);
   r3 := r1.enlarge(inc_factor);
   r3.display;
```

```
   IF (r1 > r2) THEN -- calling measure function
      r1.display;
   ELSE
```

```
      r2.display;
   END IF;
END;
/
```

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

```
Length: 8
Width: 9
Length: 5
Width: 7
```

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

Using Order method

Now, the same effect could be achieved using an order method. Let us recreate the rectangle object using an order method:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 member procedure display,
 order member function measure(r rectangle) return number
);
/
```

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

```
Type created.
```

Creating the type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
   MEMBER PROCEDURE display IS
   BEGIN
      dbms_output.put_line('Length: '|| length);
      dbms_output.put_line('Width: '|| width);
   END display;
```

```
   ORDER MEMBER FUNCTION measure(r rectangle) return number IS
   BEGIN
      IF(sqrt(self.length*self.length + self.width*self.width)>
sqrt(r.length*r.length + r.width*r.width)) then
         return(1);
      ELSE
         return(-1);
      END IF;
   END measure;
```

```
END;
/
```

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

```
Type body created.
```

Using the rectangle object and its member functions:

```
DECLARE
   r1 rectangle;
   r2 rectangle;
BEGIN
   r1 := rectangle(23, 44);
   r2 := rectangle(15, 17);
   r1.display;
   r2.display;
   IF (r1 > r2) THEN -- calling measure function
      r1.display;
   ELSE
      r2.display;
   END IF;
END;
/
```

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

```
Length: 23
Width: 44
Length: 15
Width: 17
Length: 23
Width: 44
```

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

Inheritance for PL/SQL Objects:

PL/SQL allows creating object from existing base objects. To implement inheritance, the base objects should be declared as NOT FINAL. The default is FINAL.

The following programs illustrate inheritance in PL/SQL Objects. Let us create another object named TableTop, which is inheriting from the Rectangle object. Creating the base rectangle object:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 member function enlarge( inc number) return rectangle,
 NOT FINAL member procedure display) NOT FINAL
```

```
/
```

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

```
Type created.
```

Creating the base type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
   MEMBER FUNCTION enlarge(inc number) return rectangle IS
   BEGIN
      return rectangle(self.length + inc, self.width + inc);
   END enlarge;
```

```
   MEMBER PROCEDURE display IS
   BEGIN
      dbms_output.put_line('Length: '|| length);
      dbms_output.put_line('Width: '|| width);
   END display;
END;
/
```

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

```
Type body created.
```

Creating the child object tabletop:

```
CREATE OR REPLACE TYPE tabletop UNDER rectangle
(
   material varchar2(20);
   OVERRIDING member procedure display
)
/
```

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

```
Type created.
```

Creating the type body for the child object tabletop:

```
CREATE OR REPLACE TYPE BODY tabletop AS
OVERRIDING MEMBER PROCEDURE display IS
BEGIN
   dbms_output.put_line('Length: '|| length);
   dbms_output.put_line('Width: '|| width);
   dbms_output.put_line('Material: '|| material);
END display;
/
```

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

```
Type body created.
```

Using the tabletop object and its member functions:

```
DECLARE
   t1 tabletop;
   t2 tabletop;
BEGIN
   t1:= tabletop(20, 10, 'Wood');
   t2 := tabletop(50, 30, 'Steel');
   t1.display;
   t2.display;
END;
/
```

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

```
Length: 20
Width: 10
Material: Wood
Length: 50
Width: 30
Material: Steel
```

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

Abstract Objects in PL/SQL

The NOT INSTANTIABLE clause allows you to declare an abstract object. You cannot use an abstract object as it is; you will have to create a subtype or child type of such objects to use its functionalities.

For example

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 NOT INSTANTIABLE NOT FINAL MEMBER PROCEDURE display)
 NOT INSTANTIABLE NOT FINAL
/
```

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

```
Type created.
```