



PL/SQL

pl/sql programming

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

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 an intermediate level of expertise from where you can take yourself to a 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 an understanding on SQL and other computer programming language, then it will be an added advantage to proceed.

Copyright & Disclaimer

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	i
Audience	i
Prerequisites	i
Copyright & Disclaimer.....	i
Table of Contents	ii
1. PL/SQL — OVERVIEW	1
Features of PL/SQL	1
Advantages of PL/SQL	1
2. PL/SQL — ENVIRONMENT SETUP	3
Text Editor.....	14
3. PL/SQL — BASIC SYNTAX	15
4. PL/SQL — DATA TYPES.....	19
PL/SQL Scalar Data Types and Subtypes	19
PL/SQL Numeric Data Types and Subtypes	20
PL/SQL Character Data Types and Subtypes	21
PL/SQL Boolean Data Types	22
PL/SQL Datetime and Interval Types	22
PL/SQL Large Object (LOB) Data Types	23
PL/SQL User-Defined Subtypes.....	24
NULLs in PL/SQL	25
5. PL/SQL — VARIABLES	26
Variable Declaration in PL/SQL.....	26
Initializing Variables in PL/SQL	27
Variable Scope in PL/SQL	28

Assigning SQL Query Results to PL/SQL Variables	29
6. PL/SQL — CONSTANTS AND LITERALS	31
Declaring a Constant	31
The PL/SQL Literals.....	32
7. PL/SQL — OPERATORS	34
Arithmetic Operators	34
Relational Operators	35
[Comparison Operators.....	37
Logical Operators	41
PL/SQL Operator Precedence	42
8. PL/SQL — CONDITIONS	44
IF-THEN Statement.....	45
IF-THEN-ELSE Statement.....	48
IF-THEN-ELSIF Statement.....	50
CASE Statement	51
Searched CASE Statement	53
Nested IF-THEN-ELSE Statements	54
9. PL/SQL — LOOPS.....	56
Basic Loop Statement.....	57
WHILE LOOP Statement	59
FOR LOOP Statement	60
Reverse FOR LOOP Statement	61
Nested Loops	62
Labeling a PL/SQL Loop	64
The Loop Control Statements.....	65

EXIT Statement	65
The EXIT WHEN Statement.....	67
CONTINUE Statement.....	68
GOTO Statement	69
10. PL/SQL — STRINGS	72
Declaring String Variables	72
PL/SQL String Functions and Operators.....	73
11. PL/SQL — ARRAYS	78
Creating a Varray Type	78
12. PL/SQL — PROCEDURES	82
Parts of a PL/SQL Subprogram.....	82
Creating a Procedure.....	83
Executing a Standalone Procedure	84
Deleting a Standalone Procedure	84
Parameter Modes in PL/SQL Subprograms.....	85
Methods for Passing Parameters.....	87
13. PL/SQL — FUNCTIONS.....	88
Creating a Function	88
Calling a Function	89
PL/SQL Recursive Functions	91
14. PL/SQL — CURSORS.....	93
Implicit Cursors	93
Explicit Cursors.....	95
Declaring the Cursor.....	96
Opening the Cursor	96

Fetching the Cursor	96
Closing the Cursor	96
15. PL/SQL — RECORDS.....	98
Table-Based Records	98
Cursor-Based Records	99
User-Defined Records.....	100
16. PL/SQL — EXCEPTIONS	104
Syntax for Exception Handling.....	104
Raising Exceptions.....	105
User-defined Exceptions	106
Pre-defined Exceptions	107
17. PL/SQL — TRIGGERS.....	110
Creating Triggers	110
Triggering a Trigger	112
18. PL/SQL — PACKAGES.....	114
Package Specification	114
Package Body	114
Using the Package Elements.....	115
19. PL/SQL — COLLECTIONS.....	120
Index-By Table	121
Nested Tables.....	123
Collection Methods	126
Collection Exceptions	127

20. PL/SQL — TRANSACTIONS	129
Starting and Ending a Transaction	129
Committing a Transaction	129
Rolling Back Transactions	130
Automatic Transaction Control	131
21. PL/SQL — DATE & TIME.....	132
Field Values for Datetime and Interval Data Types.....	132
The Datetime Data Types and Functions	133
The Interval Data Types and Functions.....	137
22. PL/SQL — DBMS OUTPUT.....	138
DBMS_OUTPUT Subprograms	138
23. PL/SQL — OBJECT-ORIENTED	141
Instantiating an Object.....	142
Member Methods	142
Using Map method.....	143
Using Order method.....	145
Inheritance for PL/SQL Objects	147
Abstract Objects in PL/SQL.....	149

1. 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 certain notable facts about PL/SQL:

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

Features of PL/SQL

PL/SQL has the following features:

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of 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. In Dynamic SQL, 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.

2. PL/SQL — Environment Setup

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







To run PL/SQL programs, you should have the Oracle RDBMS Server installed in your machine. This will take care of the execution of the SQL commands. The most recent version of Oracle RDBMS is 11g. You can download a trial version of Oracle 11g from the following link:

[Download Oracle 11g Express Edition](#)

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

- **win64_11gR2_database_1of2.zip**
- **win64_11gR2_database_2of2.zip**

After downloading the above two files, you will need to unzip them in a single directory **database** and under that you will find the following sub-directories:

	doc	3/24/2010 12:15 AM	File folder	
	install	3/30/2010 8:05 AM	File folder	
	response	3/30/2010 9:31 AM	File folder	
	stage	3/30/2010 9:31 AM	File folder	
	setup	3/12/2010 1:11 AM	Application	334 KB
	welcome	3/16/2010 1:42 PM	HTML Document	6 KB

Step 1

Let us now launch the Oracle Database Installer using the setup file. Following is the first screen. You can provide your email ID and check the checkbox as shown in the following screenshot. Click the **Next** button.



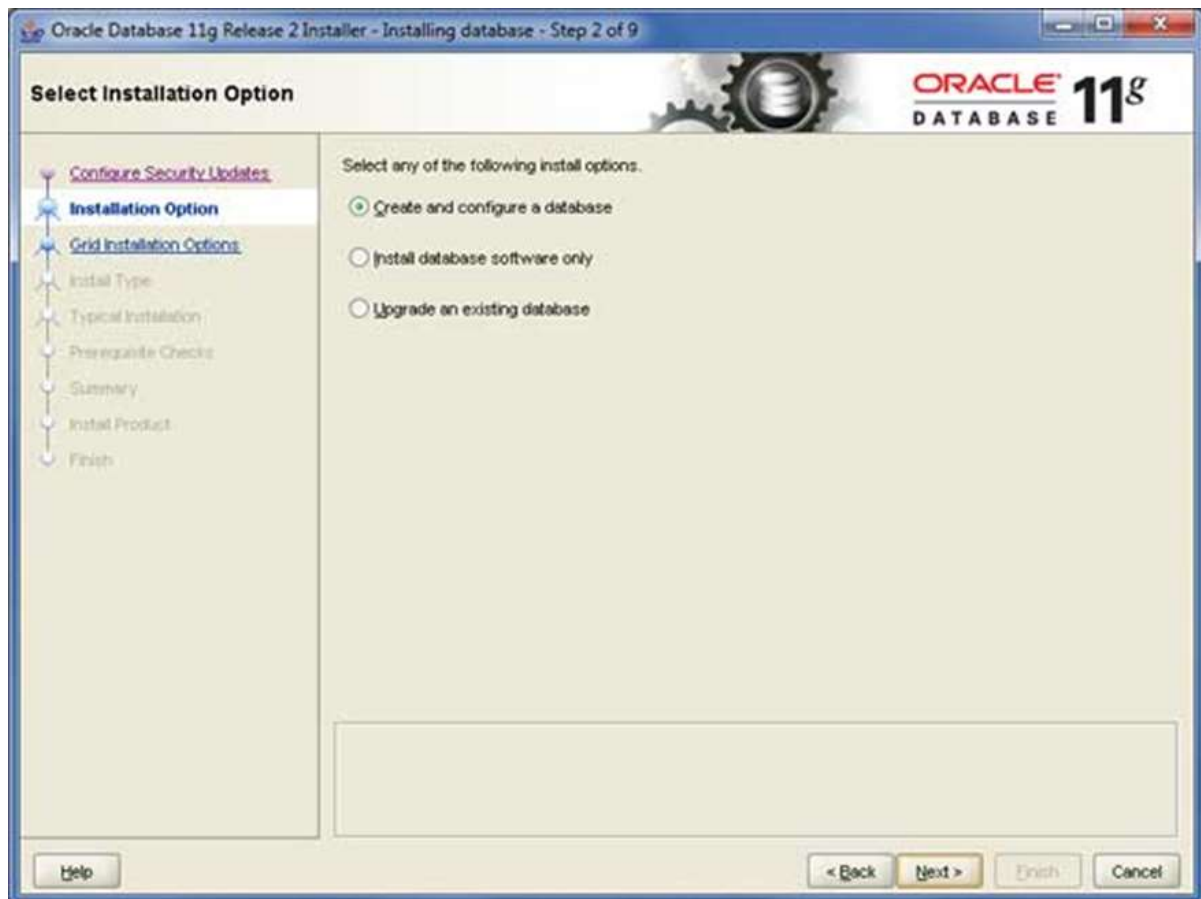
Step 2

You will be directed to the following screen; uncheck the checkbox and click the **Continue** button to proceed.



Step 3

Just select the first option **Create and Configure Database** using the radio button and click the **Next** button to proceed.



Step 4

We assume you are installing Oracle for the basic purpose of learning and that you are installing it on your PC or Laptop. Thus, select the **Desktop Class** option and click the **Next** button to proceed.



Step 5

Provide a location, where you will install the Oracle Server. Just modify the **Oracle Base** and the other locations will set automatically. You will also have to provide a password; this will be used by the system DBA. Once you provide the required information, click the **Next** button to proceed.

Oracle Database 11g Release 2 Installer - Installing database - Step 4 of 8

Typical Install Configuration

Perform full Database installation with basic configuration.

Oracle base: C:\app\ZARA

Software location: C:\app\ZARA\product\11.2.0\vdhome_1

Database file location: C:\app\ZARA\oradata

Database edition: Enterprise Edition (3.34GB)

Character Set: Default (AL32UTF8)

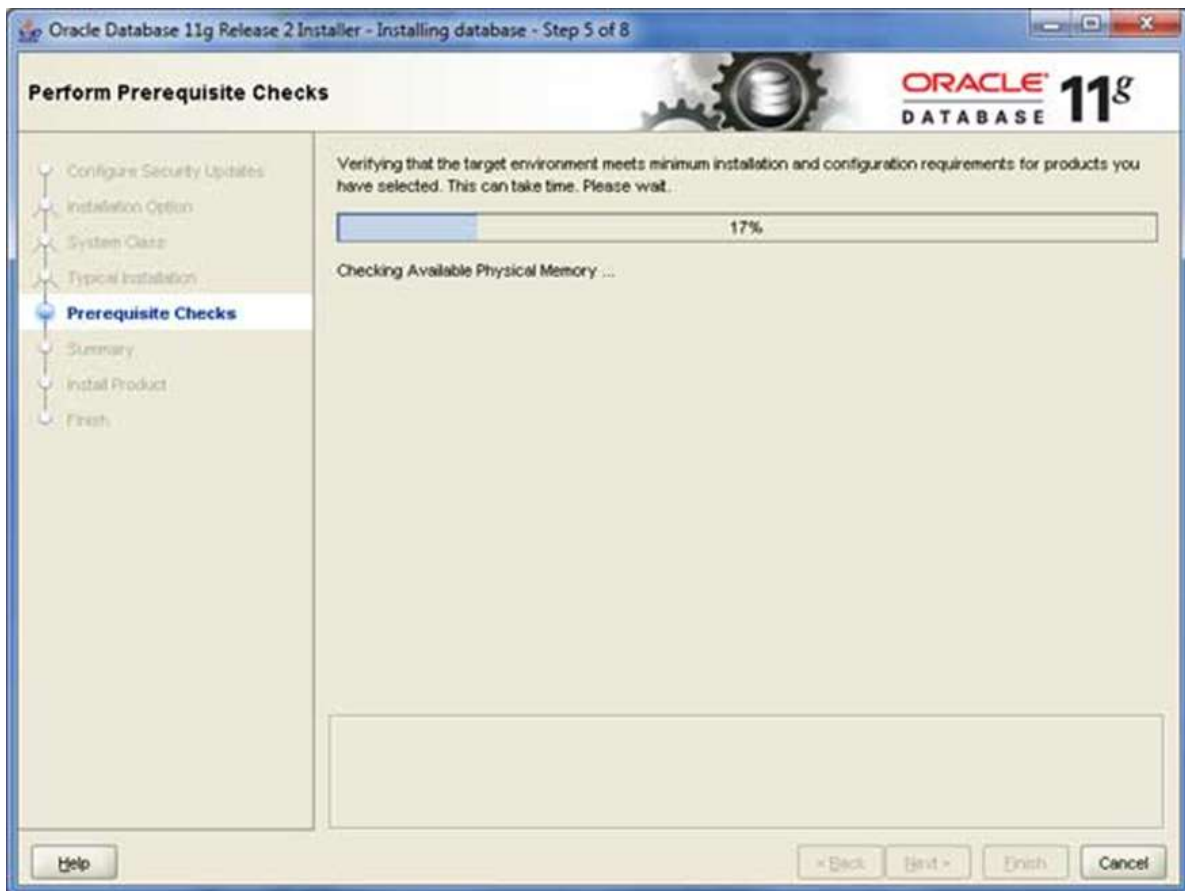
Global database name: orcl

Administrative password:

Confirm Password:

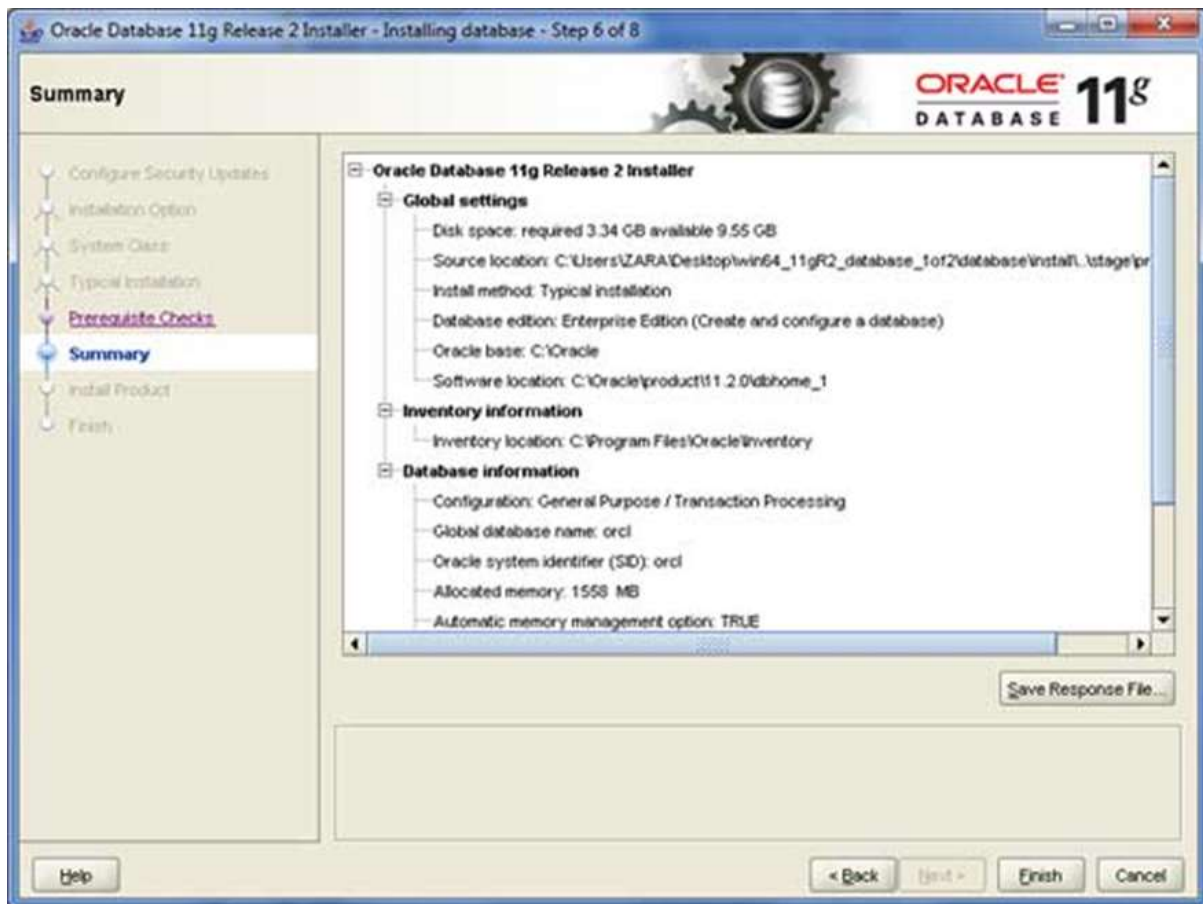
Step 6

Again, click the **Next** button to proceed.



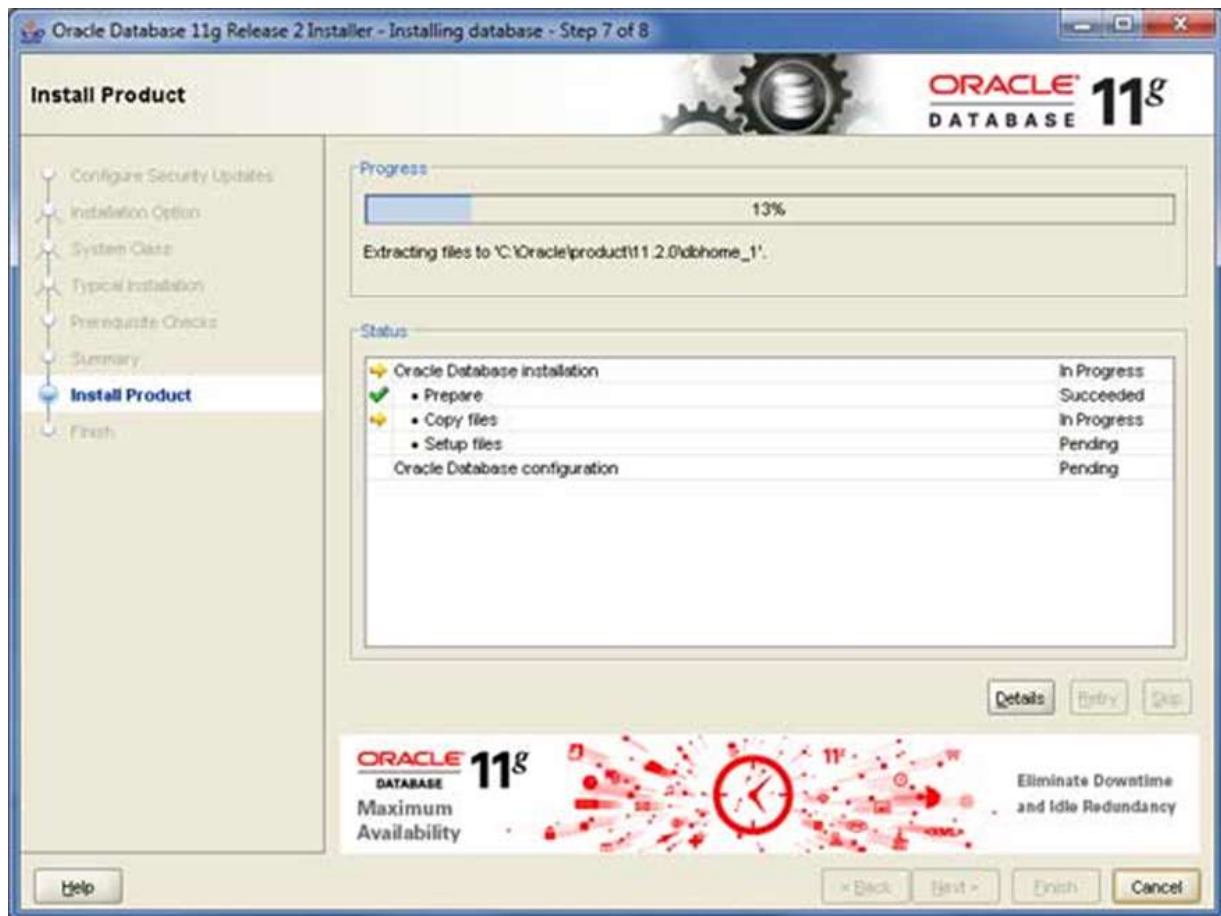
Step 7

Click the **Finish** button to proceed; this will start the actual server installation.



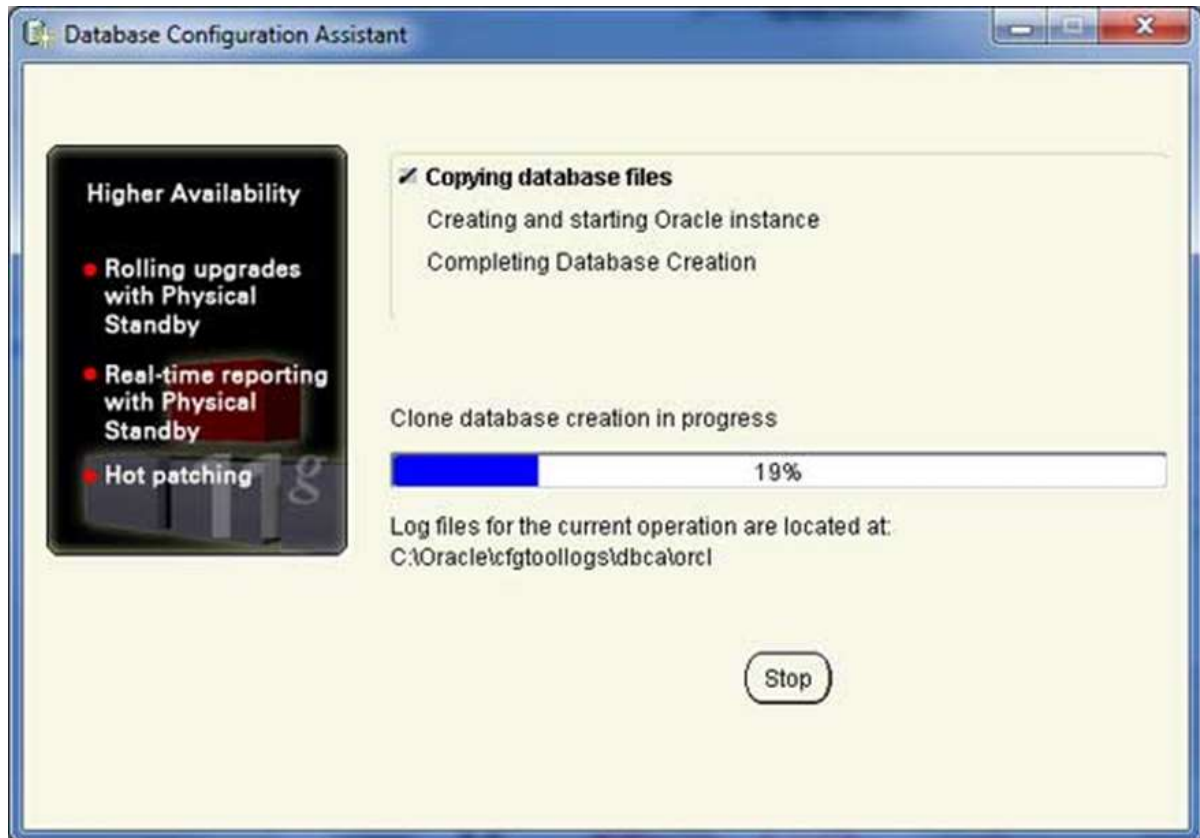
Step 8

This will take a few moments, until Oracle starts performing the required configuration.



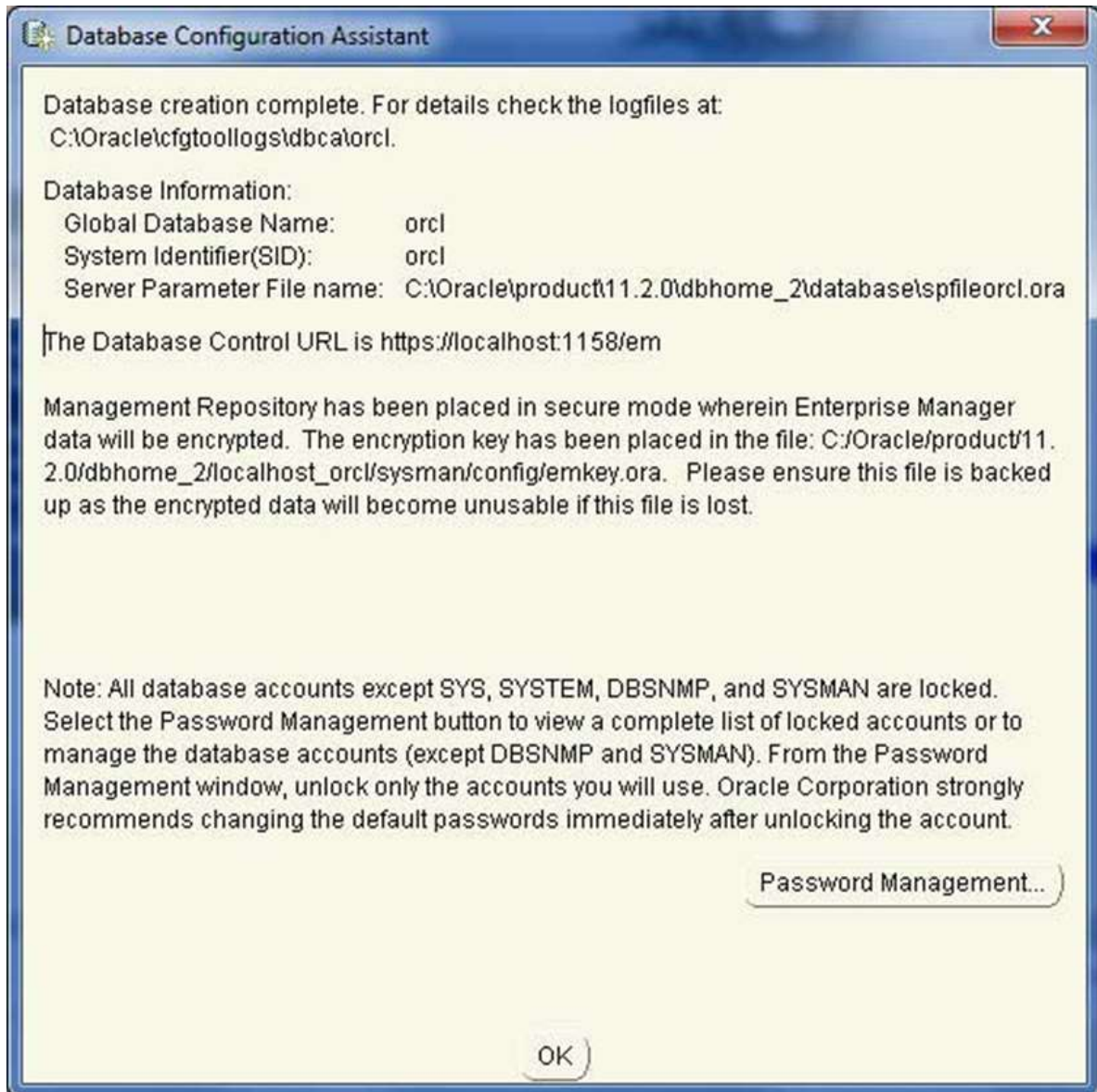
Step 9

Here, Oracle installation will copy the required configuration files. This should take a moment:



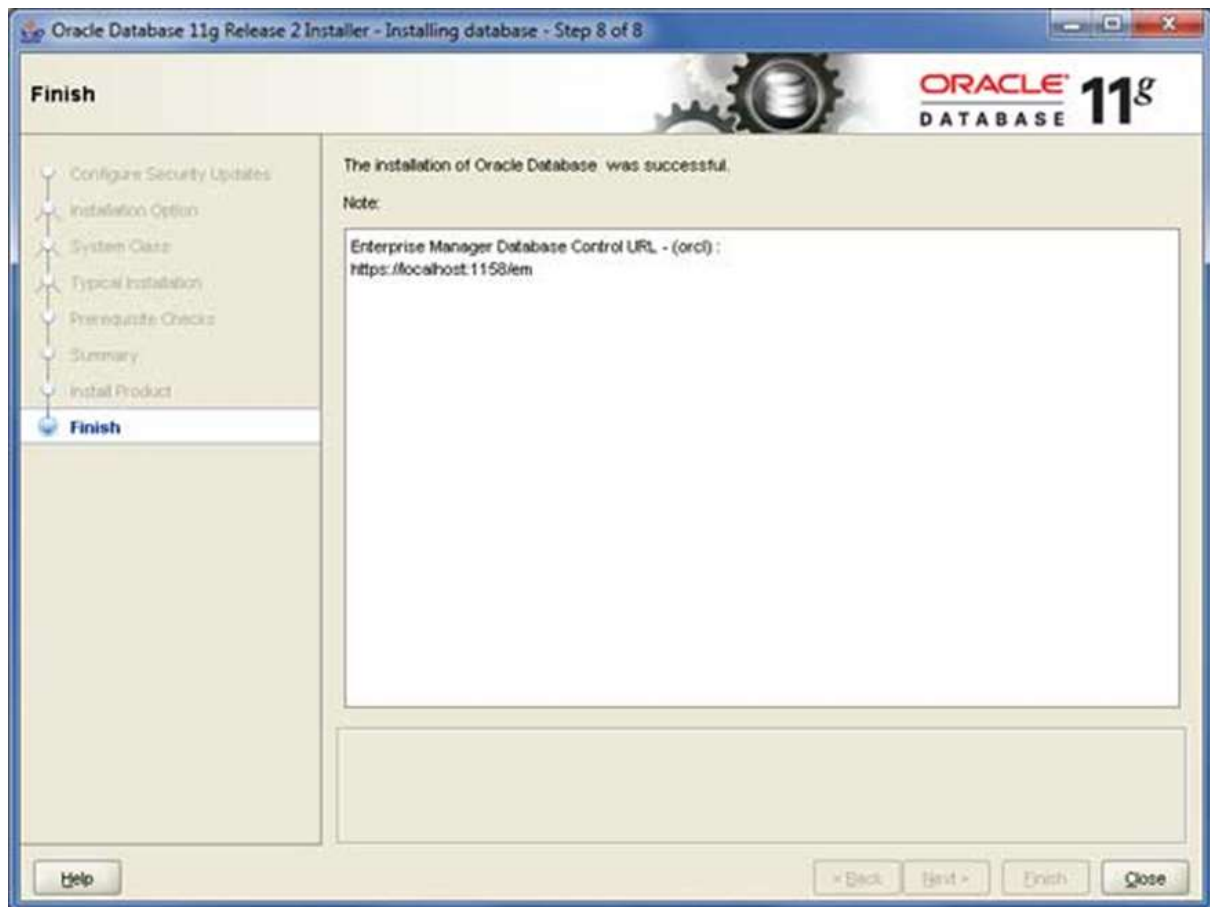
Step 10

Once the database files are copied, you will have the following dialogue box. Just click the **OK** button and come out.



Step 11

Upon installation, you will have the following final window.




Final Step

It is now time to verify your installation. At the command prompt, use the following command if you are using Windows:

```
sqlplus "/ as sysdba"
```

You should have the SQL prompt where you will write your PL/SQL commands and scripts:



```
Administrator: C:\Windows\system32\cmd.exe - sqlplus "/ as sysdba"
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\ZARA>sqlplus "/ as sysdba"

SQL*Plus: Release 11.2.0.1.0 Production on Fri Aug 31 19:15:57 2012
Copyright (c) 1982, 2010, Oracle. All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing opti
SQL>
```

Text Editor

Running large programs from the command prompt may land you in inadvertently losing some of the work. It is always recommended to use the command files. To use the command files:

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

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

3. PL/SQL — Basic Syntax

In this chapter, we will discuss the Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

Sr. No.	Sections & Description
1	Declarations This section starts with the keyword DECLARE . It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	Executable Commands This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
3	Exception Handling This section starts with the keyword EXCEPTION . This optional section 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**. Following 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 the 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 the SQL prompt, it produces the following result:

```
Hello World

PL/SQL procedure successfully completed.
```

The PL/SQL Identifiers

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

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

The PL/SQL Delimiters

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

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter

:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, !=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

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

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the 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 the 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

Each of these units will be discussed in the following chapters.

4. PL/SQL — Data Types

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

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

PL/SQL Scalar Data Types and Subtypes

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

Date Type	Description
Numeric	Numeric values on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values on which logical operations are performed.
Datetime	Dates and times.

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

PL/SQL Numeric Data Types and Subtypes

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

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER(prec, scale)	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits
NUMERIC(pre, scale)	Floating type with maximum precision of 38 decimal digits
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits

REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)
-------------	--

Following is a valid declaration:

```

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

```

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

```

PL/SQL procedure successfully completed

```

PL/SQL Character Data Types and Subtypes

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

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes

LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

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

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

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

PL/SQL Datetime and Interval Types

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

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

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

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11

DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

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

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

CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

PL/SQL User-Defined Subtypes

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

PL/SQL predefines several subtypes in package **STANDARD**. For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

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

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

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

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

PL/SQL procedure successfully completed.
```

NULLs in PL/SQL

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

5. PL/SQL — Variables

In this chapter, we will discuss Variables in PL/SQL. 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 the 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, such as date time data types, records, collections, etc. which we will cover in subsequent chapters. 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 the last chapter. Some valid variable declarations along with their definition are shown below:

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

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

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

Initializing Variables in PL/SQL

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

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

For example:

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

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

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

```
DECLARE  
    a integer := 10;  
    b integer := 20;  
    c integer;  
    f real;  
BEGIN  
    c := a + b;  
    dbms_output.put_line('Value of c: ' || c);  
    f := 70.0/3.0;  
    dbms_output.put_line('Value of f: ' || f);  
END;  
/
```

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

```
Value of c: 30  
Value of f: 23.333333333333333333  
  
PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

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

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

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

```
DECLARE
    -- Global variables
    num1 number := 95;
    num2 number := 85;
BEGIN
    dbms_output.put_line('Outer Variable num1: ' || num1);
    dbms_output.put_line('Outer Variable num2: ' || num2);
    DECLARE
        -- Local variables
        num1 number := 195;
        num2 number := 185;
    BEGIN
        dbms_output.put_line('Inner Variable num1: ' || num1);
        dbms_output.put_line('Inner Variable num2: ' || num2);
    END;
END;
/
```

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

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

PL/SQL procedure successfully completed.
```

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 refer to the [SQL tutorial](#))

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

Table Created

Let us now 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.No.ame%type;
    c_addr customers.address%type;
    c_sal  customers.salary%type;
BEGIN
    SELECT name, address, salary INTO c_name, c_addr, c_sal
    FROM customers
    WHERE id = c_id;

    dbms_output.put_line
    ('Customer ' || c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

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

```
Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully
```

6. PL/SQL — Constants and Literals

In this chapter, we will discuss **constants** and **literals** in PL/SQL. A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the **NOT NULL constraint**.

Declaring a Constant

A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example:

```
PI CONSTANT NUMBER := 3.141592654;

DECLARE
    -- constant declaration
    pi constant number := 3.141592654;
    -- other declarations
    radius number(5,2);
    dia number(5,2);
    circumference number(7, 2);
    area number (10, 2);
BEGIN
    -- processing
    radius := 9.5;
    dia := radius * 2;
    circumference := 2.0 * pi * radius;
    area := pi * radius * radius;
    -- output
    dbms_output.put_line('Radius: ' || radius);
    dbms_output.put_line('Diameter: ' || dia);
    dbms_output.put_line('Circumference: ' || circumference);
    dbms_output.put_line('Area: ' || area);
END;
/
```

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

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

PL/SQL procedure successfully completed.
```

The PL/SQL Literals

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

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

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

Literal Type	Example:
Numeric Literals	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	'Hello, world!' 'Tutorials Point' '19-NOV-12'
BOOLEAN Literals	TRUE, FALSE, and NULL
Date and Time Literals	DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown in the following program:

```
DECLARE
    message  varchar2(30):= 'That''s tutorialspoint.com!';
BEGIN
    dbms_output.put_line(message);
END;
/
```

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

```
That's tutorialspoint.com!
```

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


7. PL/SQL — Operators

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

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

Here, we will understand the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed in a later chapter: **PL/SQL - Strings**.

Arithmetic Operators

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

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

Arithmetic Operator - Example

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

```

    dbms_output.put_line( 10 ** 5);
END;
/

```

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

```

15
5
50
2
100000

PL/SQL procedure successfully completed.

```

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us 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.
----	--	-------------------

Relational Operators - Example

```

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

    IF (a < b) then
        dbms_output.put_line('Line 2 - a is less than b');
    ELSE
        dbms_output.put_line('Line 2 - a is not less than b');
    END IF;

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

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

```

```

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

END;
/

```

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

```

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

PL/SQL procedure successfully completed

```

Comparison Operators

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

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

Comparison Operators - Example

LIKE Operator

This program tests the LIKE operator. Here, we will use a small ***procedure()*** to show the functionality of the LIKE operator:

```

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

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

```

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

```

True
False

PL/SQL procedure successfully completed.

```

BETWEEN Operator

The following program shows the usage of the BETWEEN operator:

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

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

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

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

```
True
True
False

PL/SQL procedure successfully completed.
```

IN and IS NULL Operators

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

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

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

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

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

```
False
True
False

PL/SQL procedure successfully completed.
```

Logical Operators

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

Operator	Description	Example
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the 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.

Logical Operators – Example

```

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

```



```
END;
/
```

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

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

PL/SQL procedure successfully completed.
```

PL/SQL Operator Precedence

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

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

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

The precedence of operators goes as follows: $=$, $<$, $>$, $<=$, $>=$, $<>$, $!=$, $\sim=$, $\wedge=$, IS NULL, LIKE, BETWEEN, IN.

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

OR	inclusion
-----------	-----------

Operators Precedence - Example

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

```

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

    e := ((a + b) * c) / d;   -- (30 * 15 ) / 5
    dbms_output.put_line('Value of ((a + b) * c) / d is : ' || e );

    e := (a + b) * (c / d);   -- (30) * (15/5)
    dbms_output.put_line('Value of (a + b) * (c / d) is : ' || e );

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

```

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

```

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

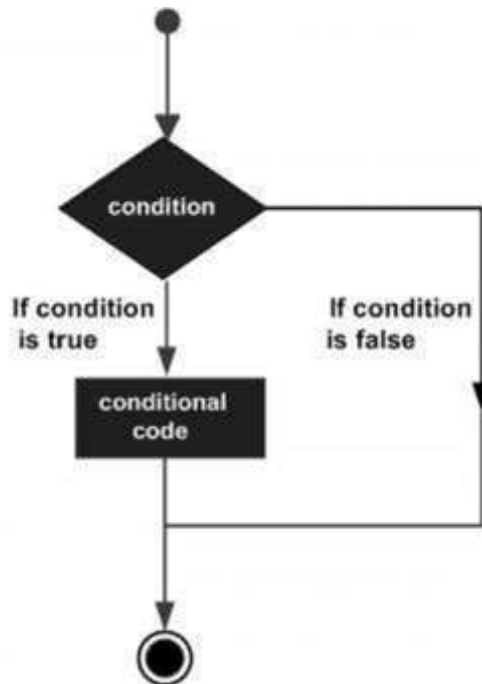
PL/SQL procedure successfully completed.

```

8. PL/SQL — Conditions

In this chapter, we will discuss conditions in PL/SQL. 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 form of a typical conditional (i.e., decision making) structure found in most of the programming languages:



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

Statement	Description
IF - THEN statement	The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF . If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
IF-THEN-ELSE statement	IF statement adds the keyword ELSE followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.

IF-THEN-ELSIF statement	It allows you to choose between several alternatives.
Case statement	Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
Searched CASE statement	The searched CASE statement has no selector , and it's WHEN clauses contain search conditions that yield Boolean values.
nested IF-THEN-ELSE	You can use one IF-THEN or IF-THEN-ELSIF statement inside another IF-THEN or IF-THEN-ELSIF statement(s).

IF-THEN Statement

It is the simplest form of the **IF** control statement, frequently used in decision-making and changing the control flow of the program execution.

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL**, then the **IF** statement does nothing.

Syntax

Syntax for **IF-THEN** statement is:

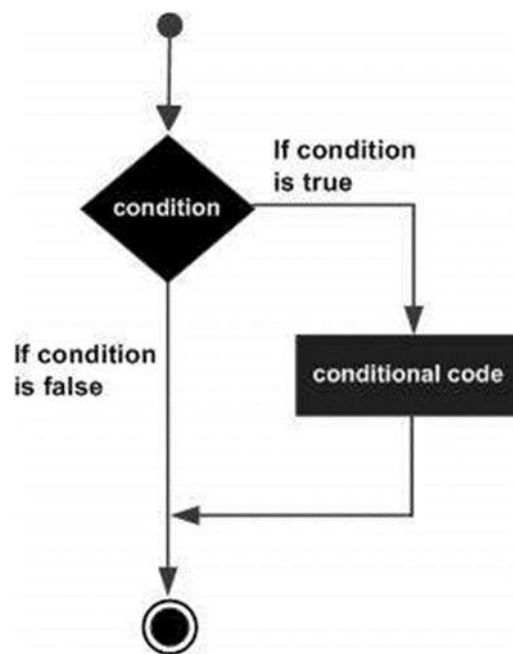
```
IF condition THEN
    S;
END IF;
```

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

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

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

Flow Diagram



Example 1

Let us try an example that will help you understand the concept:

```

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

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

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

PL/SQL procedure successfully completed.
```

Example 2

Consider we have a table and few records in the table as we had created in [PL/SQL Variable Types](#).

```
DECLARE
    c_id customers.id%type := 1;
    c_sal customers.salary%type;
BEGIN
    SELECT salary
    INTO c_sal
    FROM customers
    WHERE id = c_id;
    IF (c_sal <= 2000) THEN
        UPDATE customers
        SET salary = salary + 1000
        WHERE id = c_id;
        dbms_output.put_line ('Salary updated');
    END IF;
END;
/
```

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

```
Salary updated

PL/SQL procedure successfully completed.
```

IF-THEN-ELSE Statement

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

Syntax

Syntax for the IF-THEN-ELSE statement is:

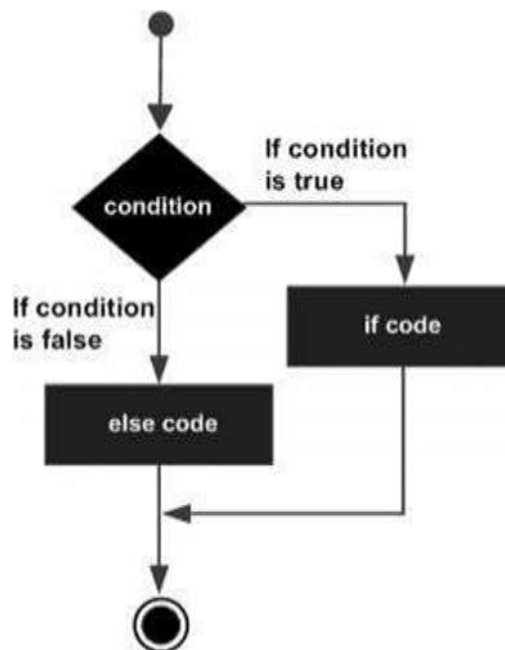
```
IF condition THEN
    S1;
ELSE
    S2;
END IF;
```

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

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

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

Flow Diagram



Example

Let us try an example that will help you understand the concept:

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


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

```
a is not less than 20
value of a is : 100

PL/SQL procedure successfully completed.
```

IF-THEN-ELSIF Statement

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements there are a few points to keep in mind.

- It's ELSIF, not ELSEIF.
- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax

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

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

Example

```
DECLARE
    a number(3) := 100;
BEGIN
    IF ( a = 10 ) THEN
```

```

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

```

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

```

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

PL/SQL procedure successfully completed.

```

CASE Statement

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

Syntax

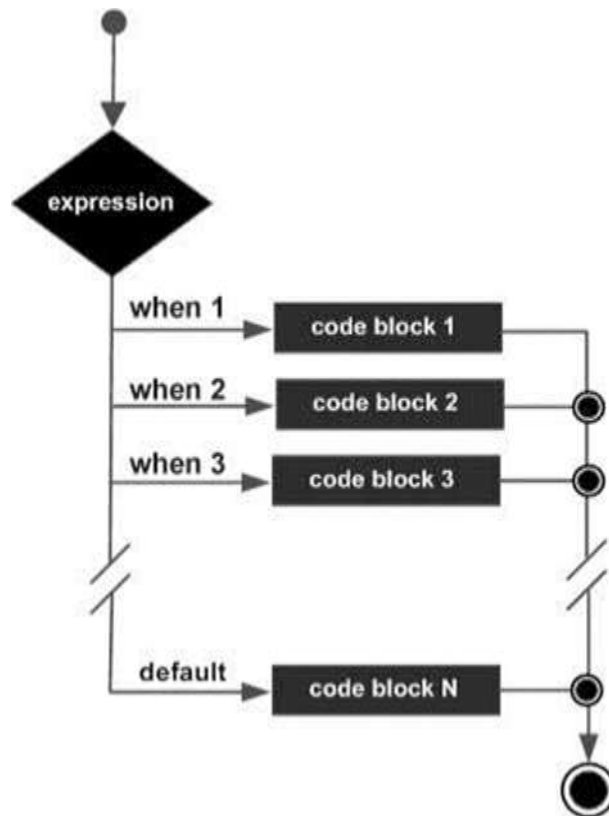
The syntax for the case statement in PL/SQL is:

```

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

```

Flow Diagram



Example

```

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

```

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

Excellent

PL/SQL procedure successfully completed.

Searched CASE Statement

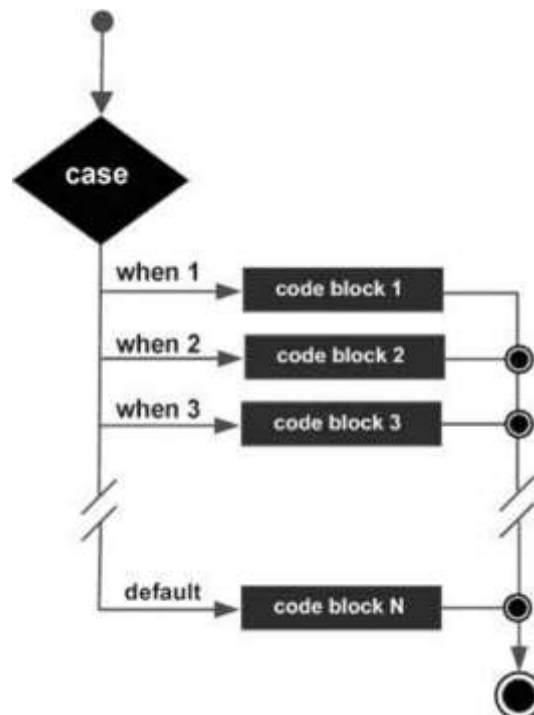
The searched **CASE** statement has no selector and the **WHEN** clauses of the statement contain search conditions that give Boolean values.

Syntax

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

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

Flow Diagram



Example

```

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

```

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

```

Very good

PL/SQL procedure successfully completed.

```

Nested IF-THEN-ELSE Statements

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

Syntax

```

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

```

Example

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

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

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

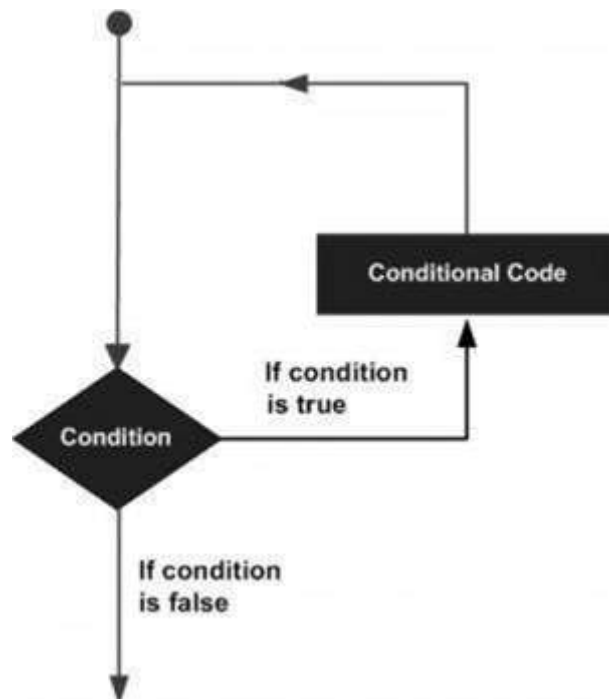
PL/SQL procedure successfully completed.
```

9. PL/SQL — Loops

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

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

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



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

Loop Type	Description
PL/SQL Basic LOOP	In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
PL/SQL WHILE LOOP	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

PL/SQL FOR LOOP	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
Nested loops in PL/SQL	You can use one or more loop inside any another basic loop, while, or for loop.

Basic Loop Statement

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

Syntax

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

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

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

Example

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

    dbms_output.put_line('After Exit x is: ' || x);
END;
/
```


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

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

PL/SQL procedure successfully completed.
```

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

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

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

```
10
20
30
40

50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

WHILE LOOP Statement

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

Syntax

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

Example

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

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

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

PL/SQL procedure successfully completed.

FOR LOOP Statement

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

Syntax

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

Following is the flow of control in a **For Loop**:

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

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

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

Example

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

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

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

PL/SQL procedure successfully completed.

Reverse FOR LOOP Statement

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

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

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

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

```
value of a: 20
value of a: 19
value of a: 18
```

```
value of a: 17
value of a: 16
```

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

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

Nested Loops

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

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

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

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

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

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

```
WHILE condition1 LOOP
    sequence_of_statements1
    WHILE condition2 LOOP
```

```

        sequence_of_statements2
    END LOOP;

END LOOP;

```

Example

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

```

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

```

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

```

2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime

```

```
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
```

PL/SQL procedure successfully completed.

Labeling a PL/SQL Loop

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

The following program illustrates the concept:

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

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

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

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

PL/SQL procedure successfully completed.

The Loop Control Statements

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

PL/SQL supports the following control statements. Labeling loops also help 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 the 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 the GOTO statement in your program.

EXIT Statement

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

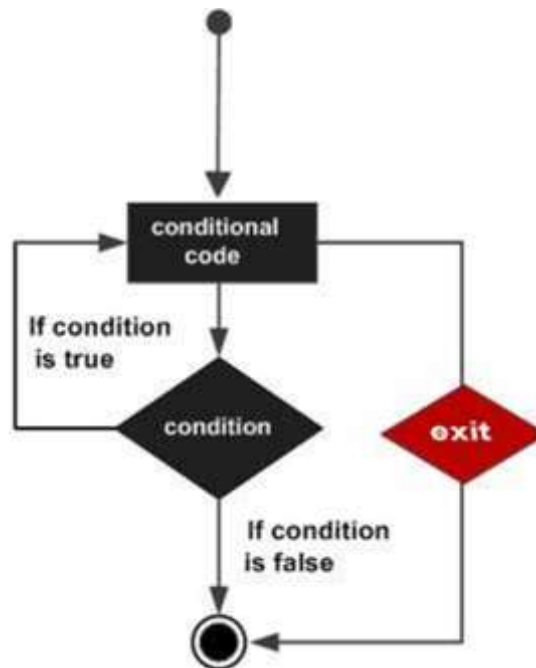
- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- If you are using nested loops (i.e., one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

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

```
EXIT;
```


Flow Diagram



Example

```

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

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

```

value of a: 10
value of a: 11
value of a: 12
  
```

```
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.
```

The EXIT WHEN Statement

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

Following are the two important aspects for the EXIT WHEN statement:

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

Syntax

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

```
EXIT WHEN condition;
```

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

Example

```
DECLARE
    a number(2) := 10;
BEGIN
    -- while loop execution
    WHILE a < 20 LOOP
        dbms_output.put_line ('value of a: ' || a);

        a := a + 1;
        -- terminate the loop using the exit when statement
    EXIT WHEN a > 15;
    END LOOP;
END;
/
```

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

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

PL/SQL procedure successfully completed.

CONTINUE Statement

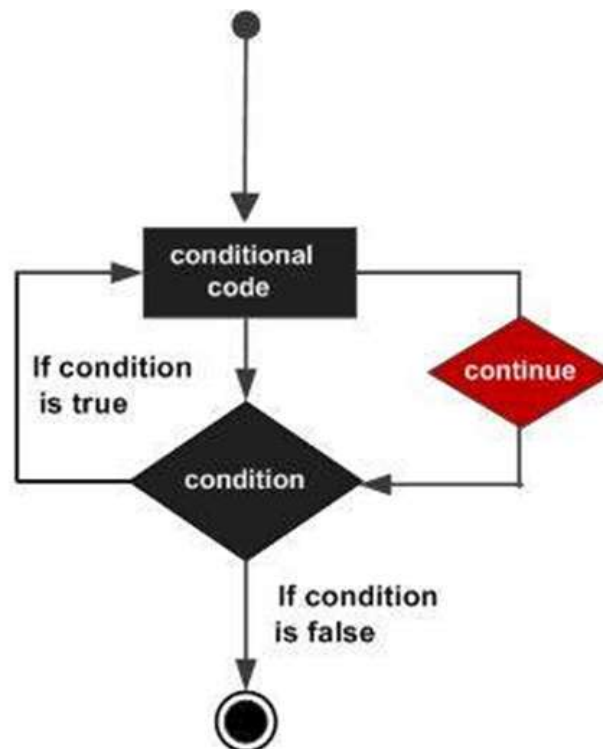
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

Flow Diagram



Example

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

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

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

PL/SQL procedure successfully completed.

GOTO Statement

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

NOTE: The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program

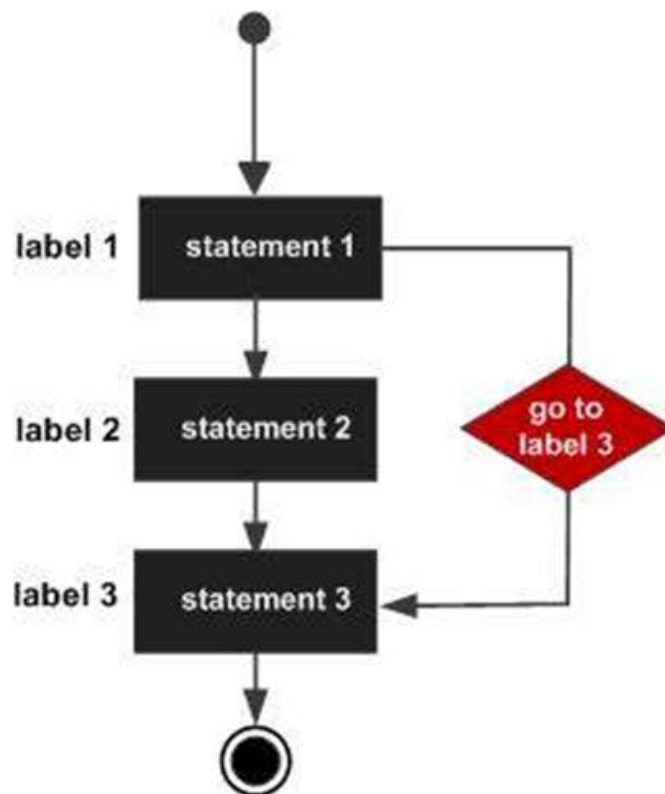
hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax

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

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

Flow Diagram



Example

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

```
        dbms_output.put_line ('value of a: ' || a);
        a := a + 1;
    IF a = 15 THEN
        a := a + 1;
        GOTO loopstart;
    END IF;
END LOOP;
END;
/
```

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

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

PL/SQL procedure successfully completed.
```

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions:

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block (i.e., an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

10. 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. For example,

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

Declaring String Variables

Oracle database provides numerous string datatypes, such as 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;
/

```

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

```

John Smith
Infotech Corporation
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed

```

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. The following two declarations 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:

Sr. No.	Function & Purpose
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 returns 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 the 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 that 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.

Let us now work out on a few examples to understand the concept:

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;
/

```

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

```
HELLO WORLD
hello world
Hello World
h
d
World
ello World
2

PL/SQL procedure successfully completed.
```

Example 2

```
DECLARE
    greetings varchar2(30) := '.....Hello World.....';
BEGIN
    dbms_output.put_line(RTRIM(greetings, '.'));
    dbms_output.put_line(LTRIM(greetings, '.'));
    dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

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

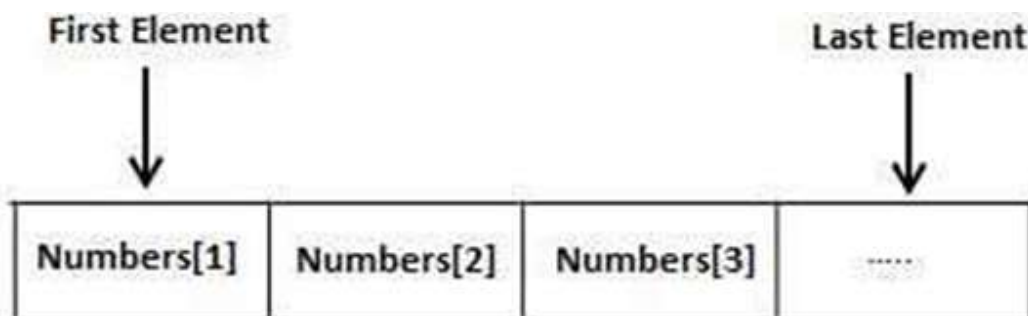
```
.....Hello World
Hello World.....
Hello World

PL/SQL procedure successfully completed.
```

11. PL/SQL — Arrays

In this chapter, we will discuss arrays in PL/SQL. The 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, however it is often better 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 VARRAY 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 VARRAY 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;
```

Let us work out on a few examples to understand the concept.

Example 1

The following program illustrates the use of 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;
/
```

When the above code is executed at the 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.
```

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.

Example 2

Elements of a varray 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

Following example makes the use of **cursor**, which you will study in detail in a separate chapter.

```
DECLARE
    CURSOR c_customers is
    SELECT name FROM customers;
    type c_list is varray (6) of customerS.No.ame%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 the 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.
```


12. PL/SQL — Procedures

In this chapter, we will discuss Procedures in PL/SQL. 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 the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the 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**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

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

Sr. No.	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 the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and **OUT** represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

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

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
    dbms_output.put_line('Hello World!');
END;
```

```
/
```

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

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways:

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

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

```
EXECUTE greetings;
```

The above call will 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 will 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;
```

You can drop the *greetings* procedure by using the following statement:

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms:

Sr. No.	Parameter Mode & Description
1	<p>IN</p> <p>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.</p>
2	<p>OUT</p> <p>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.</p>
3	<p>IN OUT</p> <p>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 the value can be read.</p> <p>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.</p>

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the 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 the SQL prompt, it produces the following result:

```

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

```

IN & OUT Mode Example 2

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

```

DECLARE
    a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
    x := x * x;
END;

BEGIN
    a:= 23;
    squareNum(a);
    dbms_output.put_line(' Square of (23): ' || a);
END;

```

/

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

```
Square of (23): 529
PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters can 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 (=>)**. The procedure call will be like the following:

```
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);
```

However, this is not legal:

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

13. PL/SQL — Functions

In this chapter, we will discuss the functions in PL/SQL. A 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 the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the 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 how to create and call 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 the [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 the above code is executed using the SQL prompt, it will produce the following result:

```
Function created.
```

Calling a Function

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

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

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

```
DECLARE
    c number(2);
BEGIN
    c := totalCustomers();
    dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

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

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example

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

```
DECLARE
    a number;
    b number;
    c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
    IF x > y THEN
        z:= x;
    ELSE
        Z:= y;
    END IF;

    RETURN z;
END;
```

```

BEGIN
    a:= 23;
    b:= 45;

    c := findMax(a, b);
    dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/

```

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

```

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

```

PL/SQL Recursive Functions

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

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

```

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

```

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

```

DECLARE
    num number;
    factorial number;

FUNCTION fact(x number)
RETURN number
IS
    f number;

```

```
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;

BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial ' || num || ' is ' || factorial);
END;
/
```

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

```
Factorial 6 is 720
```

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

14. PL/SQL — Cursors

In this chapter, we will discuss the cursors in PL/SQL. Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the 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 attributes such as **%FOUND**, **%ISOPEN**, **%NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes:

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

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

Example

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

```
Select * from customers;
```

```

+----+-----+-----+-----+-----+
| ID | NAME   | AGE | ADDRESS | SALARY |
+----+-----+-----+-----+-----+
|  1 | Ramesh |  32 | Ahmedabad | 2000.00 |
|  2 | Khilan |  25 | Delhi     | 1500.00 |
|  3 | kaushik | 23 | Kota      | 2000.00 |
|  4 | Chaitali | 25 | Mumbai   | 6500.00 |
|  5 | Hardik | 27 | Bhopal    | 8500.00 |
|  6 | Komal | 22 | MP        | 4500.00 |
+----+-----+-----+-----+-----+

```

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

```

DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers
    SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
    
```

```

        dbms_output.put_line( total_rows || ' customers selected ');
    END IF;
END;
/

```

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

```

6 customers selected

PL/SQL procedure successfully completed.

```

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

```

Select * from customers;

+----+-----+-----+-----+-----+
| ID | NAME    | AGE | ADDRESS  | SALARY |
+----+-----+-----+-----+-----+
|  1 | Ramesh  |  32 | Ahmedabad | 2500.00 |
|  2 | Khilan  |  25 | Delhi     | 2000.00 |
|  3 | kaushik |  23 | Kota      | 2500.00 |
|  4 | Chaitali | 25 | Mumbai   | 7000.00 |
|  5 | Hardik  |  27 | Bhopal    | 9000.00 |
|  6 | Komal   |  22 | MP        | 5000.00 |
+----+-----+-----+-----+-----+

```

Explicit Cursors

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

The syntax for creating an explicit cursor is:

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps:

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the 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 the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the 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 the 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.No.ame%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 the 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.

15. PL/SQL — Records

In this chapter, we will discuss Records in PL/SQL. A **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

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

PL/SQL can handle the following types of records:

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

Table-Based Records

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

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

```
DECLARE
    customer_rec customers%rowtype;
BEGIN
    SELECT * into customer_rec
    FROM customers
    WHERE id = 5;

    dbms_output.put_line('Customer ID: ' || customer_rec.id);
    dbms_output.put_line('Customer Name: ' || customer_rec.name);
    dbms_output.put_line('Customer Address: ' || customer_rec.address);
    dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```

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

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

PL/SQL procedure successfully completed.
```

Cursor-Based Records

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

```
DECLARE
    CURSOR customer_cur is
        SELECT id, name, address
        FROM customers;
    customer_rec customer_cur%rowtype;
BEGIN
    OPEN customer_cur;
    LOOP
        FETCH customer_cur into customer_rec;
        EXIT WHEN customer_cur%notfound;
        DBMS_OUTPUT.put_line(customer_rec.id || ' ' || customer_rec.name);
    END LOOP;
END;
/
```

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

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal

PL/SQL procedure successfully completed.
```

User-Defined Records

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

- Title
- Author
- Subject
- Book ID

Defining a Record

The record type is defined as:

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

The Book record is declared in the following way:

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

Accessing Fields

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

```
DECLARE
    type books is record
```

```

        (title varchar(50),
         author varchar(50),

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

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

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

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

```

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

```

Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial

```

```

Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

PL/SQL procedure successfully completed.

```

Records as Subprogram Parameters

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

```

DECLARE
    type books is record
        (title  varchar(50),
         author  varchar(50),
         subject varchar(100),
         book_id number);
    book1 books;
    book2 books;

PROCEDURE printbook (book books) IS
BEGIN
    dbms_output.put_line ('Book title : ' || book.title);
    dbms_output.put_line('Book author : ' || book.author);
    dbms_output.put_line( 'Book subject : ' || book.subject);
    dbms_output.put_line( 'Book book_id : ' || book.book_id);
END;

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

```

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

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

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

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

PL/SQL procedure successfully completed.
```

16. PL/SQL — Exceptions

In this chapter, we will discuss Exceptions in PL/SQL. An exception is an error condition during a program execution. 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 exceptions as you can 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 a 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.No.ame%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 the 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 the **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 for 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 the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

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.No.ame%type;
    c_addr  customers.address%type;

    -- user defined exception
    ex_invalid_id  EXCEPTION;
BEGIN
    IF c_id <= 0 THEN
        RAISE ex_invalid_id;
    ELSE
        SELECT  name, address INTO  c_name, c_addr
        FROM  customers
        WHERE  id = c_id;
```

```

        DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);

        DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
    END IF;
EXCEPTION
    WHEN ex_invalid_id THEN
        dbms_output.put_line('ID must be greater than zero!');
    WHEN no_data_found THEN
        dbms_output.put_line('No such customer!');
    WHEN others THEN
        dbms_output.put_line('Error!');
END;
/

```

When the above code is executed at the 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 clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a 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 a 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.

17. PL/SQL — Triggers

In this chapter, we will discuss Triggers in PL/SQL. 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 can 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 will 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 will 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, such as INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW]:** This specifies a row-level trigger, i.e., the trigger will 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 the SQL prompt, it produces the following result:

```
Trigger created.
```

The following points need to be considered here:

- 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.
- The 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 the 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 the CUSTOMERS table, the 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, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table:

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

When a record is updated in the CUSTOMERS table, the 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
```


18. PL/SQL — Packages

In this chapter, we will discuss the Packages in 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 the 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 the 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 the [PL/SQL - 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 the 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 the 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 the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the 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.No.ame%type,
    c_age customers.age%type,
    c_addr customers.address%type,
    c_sal customers.salary%type);

  -- Removes a customer
  PROCEDURE delCustomer(c_id customers.id%TYPE);

  --Lists all customers
  PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at the 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.No.ame%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.No.ame%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;

```

/

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

When the above code is executed at the 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 the 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

19. PL/SQL — Collections

In this chapter, we will discuss the Collections in PL/SQL. 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 the PL/SQL tables.

Both types of PL/SQL tables, i.e., the index-by tables and the 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**, the keys of which will be of the *subscript_type* and associated values will be of the *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;

    -- 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 the SQL prompt, it produces the following result:

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

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.No.ame%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 the 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 the declaration of an **index-by** table, but there is no **INDEX BY** clause.

A nested table can be stored in a database column. It can further 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 the 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** can 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.No.ame%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 the 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:

Sr. No.	Method Name & Purpose
1	EXISTS(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 the integer subscripts.
5	LAST Returns the last (largest) index numbers in a collection that uses the 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 Exception	Raised in Situations
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.

20. PL/SQL — Transactions

In this chapter, we will discuss the transactions in PL/SQL. 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, such as **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;
```


For 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 >;
```

For 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;
```

ROLLBACK TO sav1 – This statement rolls back all the changes up to the point, where you had marked savepoint sav1.

After that, the new changes that you make 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 the auto commit mode using the following command:

```
SET AUTOCOMMIT OFF;
```

21. PL/SQL — Date & Time

In this chapter, we will discuss the Date and Time in PL/SQL. There are two classes of date and time related data types in PL/SQL:

- 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 data type. 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
HOURL	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 data type. 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 data type 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 the datetime value):

Sr. No.	Function Name & Description
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 <i>day</i> 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):

Sr. No.	Function Name & Description
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 the 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:

Example 1

```
SELECT SYSDATE FROM DUAL;
```

Output:

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

Example 2

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

Output:

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

Example 3

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

Output:

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

Example 4

```
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:

Sr. No.	Function Name & Description
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.

22. PL/SQL — DBMS Output

In this chapter, we will discuss the DBMS Output in PL/SQL. The **DBMS_OUTPUT** is a built-in package that enables you to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers. We have already used this package throughout our tutorial.

Let us look at a small code snippet that will 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:

Sr. No.	Subprogram & Purpose
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 the 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.
```

23. PL/SQL — Object-Oriented

In this chapter, we will discuss Object-Oriented PL/SQL. PL/SQL allows defining an object type, which helps in designing object-oriented database in Oracle. An object type allows you to create composite types. Using objects allow you to implement 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 behavior.

Objects are created using the CREATE [OR REPLACE] TYPE statement. Following 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 the 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 the 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 the 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 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. So the relationship between these two objects would depend upon the value of code.

Order method

The **Order method** implements 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 the 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 the 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 the 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 the 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 the 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);
    END IF;
  END measure;
END;
```



```

        ELSE
            return(-1);
        END IF;
    END measure;
END;
/

```

When the above code is executed at the 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 the 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 the existing base objects. To implement inheritance, the base objects should be declared as **NOT FINAL**. The default is **FINAL**.

The following programs illustrate the inheritance in PL/SQL Objects. Let us create another object named **TableTop**, this is inherited from the Rectangle object. For this, we need to create 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 the 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 the 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 the 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 the 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 the 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 the SQL prompt, it produces the following result:

```
Type created.
```